DATA STRUCTURE (BASICS)

[TREES]

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

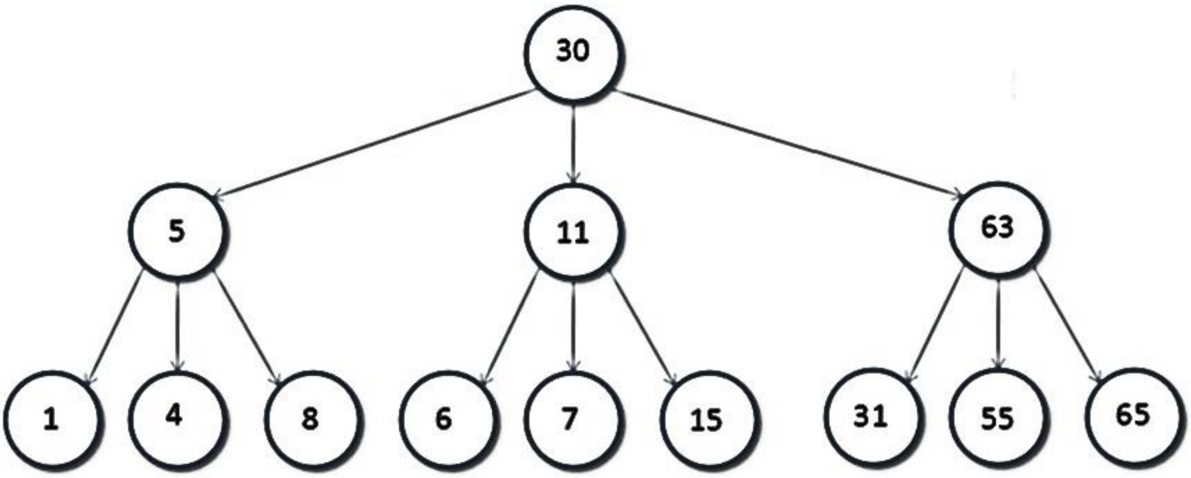
TOPICS COVERED-

- **Introduction to Trees**
- **Binary Tree Traversals**
- **Level Order Traversal of a Binary Tree**
- **Insertion in a Binary Tree**
- **Deletion in a Binary Tree**
- **Finding LCA in Binary Tree**
- **Diameter of a Binary Tree**
- **Left, Right, Top and Bottom View of a Binary Tree**
- **Threaded Binary Tree Sample**
- **Problems on Trees**

Introduction to Trees-

A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.

A Sample tree is as shown below:



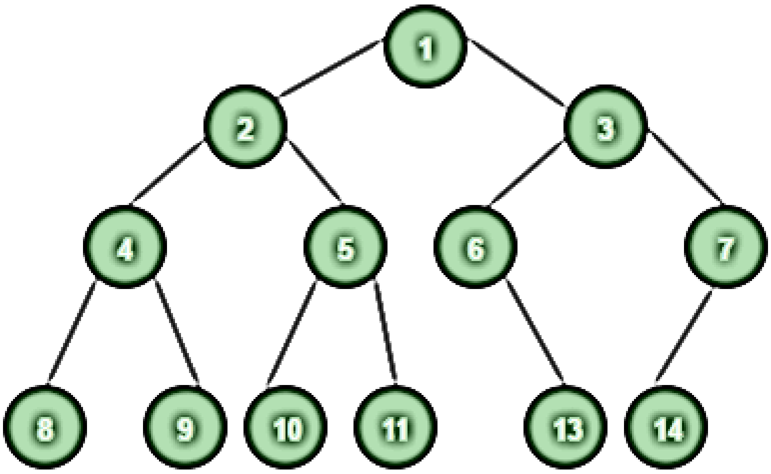
Basic Tree Terminologies:

- **Root**: The root of a tree is the first node of the tree. In the above image, the root node is the node 30.
- **Edge**: An edge is a link connecting any two nodes in the tree. For example, in the above image there is an edge between node **11** and **6**.
- **Siblings**: The children nodes of same parent are called siblings. That is, the nodes with same parent are called siblings. In the above tree, nodes *5, 11, and 63* are siblings.
- **Leaf Node**: A node is said to be the leaf node if it has no children. In the above tree, node **15** is one of the leaf nodes.
- **Height of a Tree**: Height of a tree is defined as the total number of levels in the tree or the length of the path from the root node to the node present at the last level. The above tree is of height **2**.

Binary Tree

A Tree is said to be a Binary Tree if all of its nodes have atmost 2 children. That is, all of its node can have either no child, 1 child, or 2 child nodes.

Below is a sample **Binary Tree**:



Properties of a Binary Tree:

1. ***The maximum number of nodes at level 'l' of a binary tree is (2^{l-1})***. Level of root is 1.

   This can be proved by induction.
   For root, l = 1, number of nodes = 2^{1-1} = 1
   Assume that the maximum number of nodes on level l is 2^{l-1}.
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 * 2^{l-1}.
2. ***Maximum number of nodes in a binary tree of height 'h' is (2^h – 1)***.
   Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1.
   This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2^h-1. This is a simple geometric series with h terms and sum of this series is 2^h – 1.
   In some books, the height of the root is considered as 0. In that convention, the above formula becomes 2^{h+1} – 1.
3. ***In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is Log_2(N+1).*** This can be directly derived from point 2 above. If we consider the convention where the height of a leaf node is considered 0, then above formula for minimum possible height becomes Log_2(N+1) – 1.

4. ***A Binary Tree with L leaves has at least ($Log_2L + 1$) levels.*** A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

```
   L   <=  2l-1  [From Point 1]
   l =  Log₂L + 1
   where l is the minimum number of levels.
```

5. ***In a Binary tree in which every node has 0 or 2 children, the number of leaf nodes is always one more than the nodes with two children.***

```
   L = T + 1
Where L = Number of leaf nodes
   T = Number of internal nodes with two children
```

Types of Binary Trees: Based on the structure and number of parents and children nodes, a Binary Tree is classified into the following common types:

- **Full Binary Tree**: A Binary Tree is full if every node has either 0 or 2 children. The following are examples of a full binary tree. We can also say that a full binary tree is a binary tree in which all nodes except leave nodes have two children.

```
       18
    /       \
   15         30
  /  \        /  \
40    50    100   40


       18
     /    \
    15     20
   /  \
  40    50
 /   \
30   50
```

```
     18
   /     \
 40       30
         /  \
       100   40
```

In a Full Binary, the number of leaf nodes is number of internal nodes plus 1.

- **Complete Binary Tree**: A Binary Tree is a complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

  Following are the examples of Complete Binary Trees:

```
        18
    /       \
   15         30
  /  \        /  \
40    50    100   40

          18
      /       \
     15         30
    /  \        /  \
  40    50    100   40
 /  \   /
8   7  9
```

- **Perfect Binary Tree**: A Binary tree is a Perfect Binary Tree when all internal nodes have two children and all the leave nodes are at the same level.

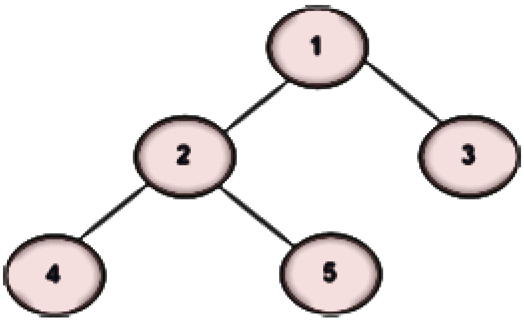
  Following are the examples of Perfect Binary Trees:

```
          18
       /       \
     15         30
    /  \        /  \
  40    50    100   40

       18
    /       \
  15         30
```

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.

Binary Tree Traversals-

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.), which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees:



- Inorder (Left, Root, Right) : 4 2 5 1 3
- Preorder (Root, Left, Right) : 1 2 4 5 3.
- Postorder (Left, Right, Root) : 4 5 2 3 1

Let's look at each of these tree traversal algorithms in details:

- **Inorder Traversal:** In Inorder traversal, a node is processed after processing all the nodes in its left subtree. The right subtree of the node is processed after processing the node itself.

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e.,
      call Inorder(left->subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e.,
      call Inorder(right->subtree)
```

Example: Inorder traversal for the above-given tree is 4 2 5 1 3.

- **Preorder Traversal:** In preorder traversal, a node is processed before processing any of the nodes in its subtree.

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e.,
      call Preorder(left-subtree)
   3. Traverse the right subtree, i.e.,
      call Preorder(right-subtree)
```

Example: Preorder traversal for the above-given tree is 1 2 4 5 3.

- **Postorder Traversal:** In post order traversal, a node is processed after processing all the nodes in its subtrees.

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e.,
      call Postorder(left-subtree)
   2. Traverse the right subtree, i.e.,
      call Postorder(right-subtree)
   3. Visit the root.
```

Example: Postorder traversal for the above-given Tree is 4 5 2 3 1.

C++ :-

```
// C++ program for different tree traversals
#include <iostream>
using namespace std;
/* A binary tree node has data, pointer to left child
  and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};
// Function to print the postorder traversal
// of a Binary Tree
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;
    // first recur on left subtree
    printPostorder(node->left);
    // then recur on right subtree
    printPostorder(node->right);
    // now deal with the node
    cout << node->data << " ";
}
// Function to print the Inorder traversal
// of a Binary Tree
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    /* first recur on left child */
    printInorder(node->left);
    /* then print the data of node */
    cout << node->data << " ";
    /* now recur on right child */
    printInorder(node->right);
}
// Function to print the PreOrder traversal
// of a Binary Tree
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;
    /* first print data of node */
    cout << node->data << " ";
    /* then recur on left sutree */
    printPreorder(node->left);
    /* now recur on right subtree */
    printPreorder(node->right);
}
// Driver Code
int main()
{
    // Contrust the Tree
    //      1
    //     / \
    //    2   3
    //   / \
    //  4   5
    struct Node *root = new Node(1);
    root->left    = new Node(2);
    root->right    = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    cout << "Preorder traversal of binary tree is \n";
    printPreorder(root);
    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);
    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);
    return 0;
}
```

Java :-

```
// Java program for different tree traversals
/* Class containing left and right child of current
  node and key value*/
class Node
{
   int key;
   Node left, right;
   public Node(int item)
   {
      key = item;
      left = right = null;
   }
}
class BinaryTree
{
   // Root of Binary Tree
   Node root;
   BinaryTree()
   {
      root = null;
   }
   // Method to print postorder traversal.
   void printPostorder(Node node)
   {
      if (node == null)
         return;
      // first recur on left subtree
      printPostorder(node.left);
      // then recur on right subtree
      printPostorder(node.right);

      // now deal with the node
      System.out.print(node.key + " ");
   }
   // Method to print inorder traversal
   void printInorder(Node node)
   {
      if (node == null)
         return;
      /* first recur on left child */
      printInorder(node.left);
      /* then print the data of node */
      System.out.print(node.key + " ");
      /* now recur on right child */
      printInorder(node.right);
   }
   // Method to print preorder traversal
   void printPreorder(Node node)
   {
      if (node == null)
         return;
      /* first print data of node */
      System.out.print(node.key + " ");
      /* then recur on left sutree */
      printPreorder(node.left);
      /* now recur on right subtree */
      printPreorder(node.right);
   }
   // Wrappers over above recursive functions
   void printPostorder() {     printPostorder(root); }
   void printInorder() {     printInorder(root); }
   void printPreorder() {     printPreorder(root); }
   // Driver method
   public static void main(String[] args)
   {
      BinaryTree tree = new BinaryTree();
      tree.root = new Node(1);
      tree.root.left = new Node(2);
      tree.root.right = new Node(3);
      tree.root.left.left = new Node(4);
      tree.root.left.right = new Node(5);
      System.out.println("Preorder traversal of binary
tree is ");
      tree.printPreorder();
```

```
        System.out.println("\nInorder traversal of binary tree is ");
        tree.printInorder();
        System.out.println("\nPostorder traversal of binary tree is ");
        tree.printPostorder();
    }
}
```

Output:

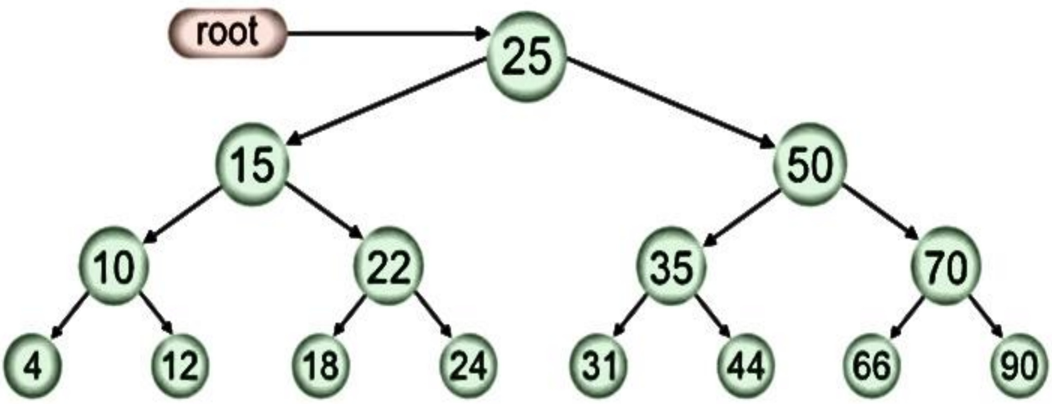
```
Preorder traversal of binary tree is

1 2 4 5 3

Inorder traversal of binary tree is

4 2 5 1 3

Postorder traversal of binary tree is

4 5 2 3 1
```

One more example:

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

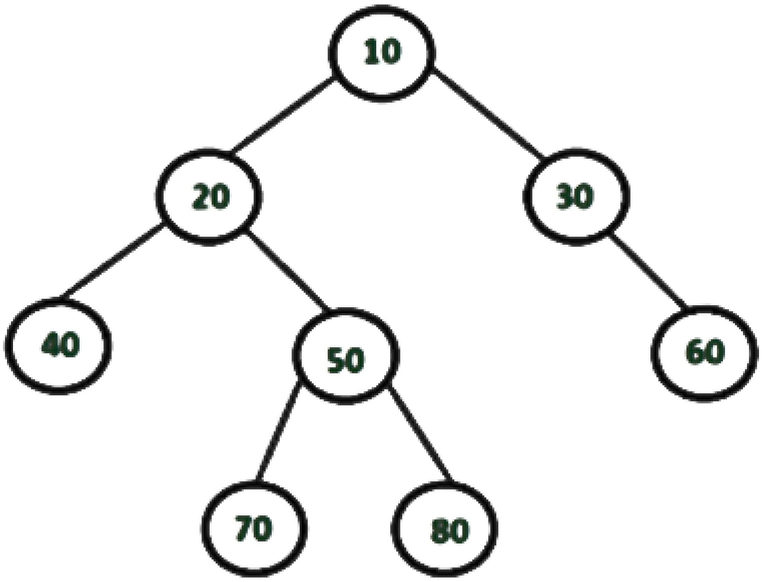


Level Order Traversal of a Binary Tree-

We have seen the three basic traversals(Preorder, postorder, and Inorder) of a Binary Tree. We can also traverse a Binary Tree using the *Level Order Traversal*.

In the Level Order Traversal, the binary tree is traversed level-wise starting from the first to last level sequentially.

Consider the below binary tree:



The Level Order Traversal of the above Binary Tree will be: **10 20 30 40 50 60 70 80**.

Algorithm: The Level Order Traversal can be implemented efficiently using a Queue.

1. Create an empty queue q.
2. Push the root node of tree to q. That is, q.push(root).
3. Loop while the queue is not empty:
    - Pop the top node from queue and print the node.
    - Enqueue node's children (first left then right children) to q
    - Repeat the process until queue is not empty.

Implementation:

C++ :-

```
// C++ program to print level order traversal
// of a Tree
#include <iostream>
#include <queue>
using namespace std;
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};
// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
// Function to print Level Order Traversal
// of the Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL) return;
    // Create an empty queue for
    // level order tarversal
    queue<Node *> q;
    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove
        // it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();
        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);
        /* Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}
// Driver Code
int main()
{
    // Create the following Binary Tree
    //      1
    //     / \
    //    2   3
    //   / \
    //  4   5
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << "Level Order traversal of binary
tree is \n";
    printLevelOrder(root);
    return 0; }
```

java :-

```
// Iterative Queue based Java program to do
// level order traversal of Binary Tree
import java.util.Queue;
import java.util.LinkedList;
/* Class to represent Tree node */
class Node {
    int data;
    Node left, right;
    public Node(int item) {
        data = item;
        left = null;
        right = null;
    }
}
/* Class to print Level Order Traversal */
class BinaryTree {
    Node root;
    /* Given a binary tree. Print its nodes in
       level order using array for implementing queue */
    void printLevelOrder()
    {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while (!queue.isEmpty())
        {
            Node tempNode = queue.poll();
            System.out.print(tempNode.data + " ");
            /* Enqueue left child */
            if (tempNode.left != null) {
                queue.add(tempNode.left);
            }
            /* Enqueue right child */
            if (tempNode.right != null) {
                queue.add(tempNode.right);
            }
        }
    }
    // Driver Code
    public static void main(String args[])
    {
        // Create the following Binary Tree
        //        1
        //       / \
        //      2   3
        //     / \
        //    4   5
        BinaryTree tree_level = new BinaryTree();
        tree_level.root = new Node(1);
        tree_level.root.left = new Node(2);
        tree_level.root.right = new Node(3);
        tree_level.root.left.left = new Node(4);
        tree_level.root.left.right = new Node(5);
        System.out.println("Level order traversal " +  "of
binary tree is - ");
        tree_level.printLevelOrder();
    }
}
```

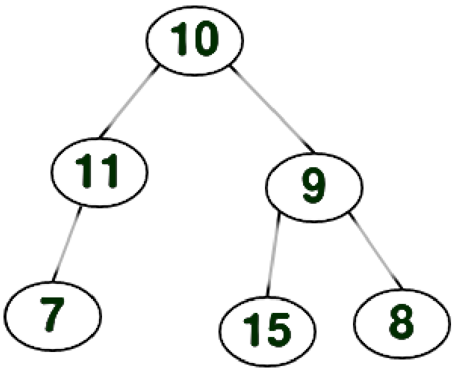
Output:

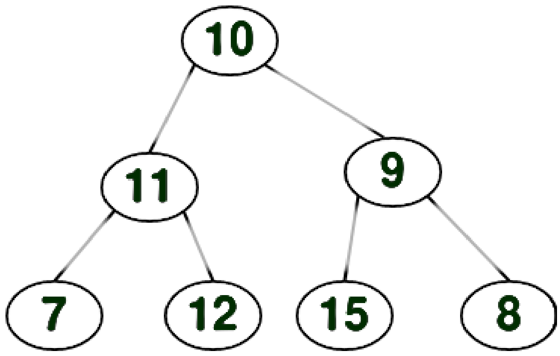
```
1 2 3 4 5
```

Time Complexity: O(N), where N is the number of nodes in the Tree.
Auxiliary Space: O(N)

Insertion in a Binary Tree-

Problem: Given a **Binary Tree** and a **Key**. The task is to insert the *key* into the binary tree at first position available in level order.





After inserting 12

The idea is to do iterative level order traversal of the given tree using a queue. If we find a node whose left child is empty, we make new key as the left child of the node. Else if we find a node whose right child is empty, we make new key as the right child of that node. We keep traversing the tree until we find a node whose either left or right child is empty.

Below is the implementation of this approach:

C++ :-

```
// C++ program to insert element in binary tree
#include <iostream>
#include <queue>
using namespace std;
// A binary tree node
struct Node {
    int key;
    struct Node* left, *right;
};
// Utility function to create a new Node
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
};
// Function to print InOrder traversal
// of a Binary Tree
void inorder(struct Node* temp)
{
    if (!temp)
        return;
    inorder(temp->left);
    cout << temp->key << " ";
    inorder(temp->right);
}
// Function to insert a new element in a Binary Tree
void insert(struct Node* temp, int key)
{
    queue<struct Node*> q;
    q.push(temp);
```

```
    // Do level order traversal until we find
    // an empty place.
    while (!q.empty()) {
        struct Node* temp = q.front();
        q.pop();
        if (!temp->left) {
            temp->left = newNode(key);
            break;
        } else
            q.push(temp->left);
        if (!temp->right) {
            temp->right = newNode(key);
            break;
        } else
            q.push(temp->right);
    }
}
// Driver code
int main()
{
    // Create the following Binary Tree
    //          10
    //         /  \
    //       11    9
    //      /       \
    //     7         8
    struct Node* root = newNode(10);
    root->left = newNode(11);
    root->left->left = newNode(7);
    root->right = newNode(9);
    root->right->left = newNode(15);
    root->right->right = newNode(8);
    cout << "Inorder traversal before insertion:";
    inorder(root);
    int key = 12;
    insert(root, key);
    cout << endl;
    cout << "Inorder traversal after insertion:";
    inorder(root);
    return 0;
}
```

Java :-

```
// Java program to insert element in binary tree
import java.util.LinkedList;
import java.util.Queue;
public class GFG {
    // Binary Tree Node
    static class Node {
        int key;
        Node left, right;
        // constructor
        Node(int key){
            this.key = key;
            left = null;
            right = null;
        }
    }
    static Node root;
    static Node temp = root;
    // Function to perform InOrder traversal
    // of the Binary Tree
    static void inorder(Node temp)
    {
        if (temp == null)
            return;
        inorder(temp.left);
        System.out.print(temp.key+" ");
        inorder(temp.right);
    }
    // Function to insert a new element in the
    // Binary Tree
    static void insert(Node temp, int key)
    {
```

```
        Queue<Node> q = new LinkedList<Node>();
        q.add(temp);
        // Do level order traversal until we find
        // an empty place.
        while (!q.isEmpty()) {
            temp = q.peek();
            q.remove();
            if (temp.left == null) {
                temp.left = new Node(key);
                break;
            } else
                q.add(temp.left);
            if (temp.right == null) {
                temp.right = new Node(key);
                break;
            } else
                q.add(temp.right);
        }
    }
    // Driver code
    public static void main(String args[])
    {
        // Create the following Binary Tree
        //         10
        //        / \
        //       11   9
        //      /     \
        //     7       8
        root = new Node(10);
        root.left = new Node(11);
        root.left.left = new Node(7);
        root.right = new Node(9);
        root.right.left = new Node(15);
        root.right.right = new Node(8);
        System.out.print( "Inorder traversal before insertion: ");
        inorder(root);
        int key = 12;
        insert(root, key);
        System.out.print("\nInorder traversal after insertion: ");
        inorder(root);
    }
}
```

Output:

```
Inorder traversal before insertion: 7 11 10 15 9 8
Inorder traversal after insertion: 7 11 12 10 15 9 8
```

Deletion in a Binary Tree-

Problem: Given a Binary Tree and a node to be deleted from this tree. The task is to delete the given node from it.

Examples:

```
Delete 10 in below tree
   10
 /    \
20     30
Output :
```

```
   30
  /
 20


Delete 20 in below tree
   10
  /    \
 20     30
          \
           40
Output :
   10
  /   \
 40    30
```

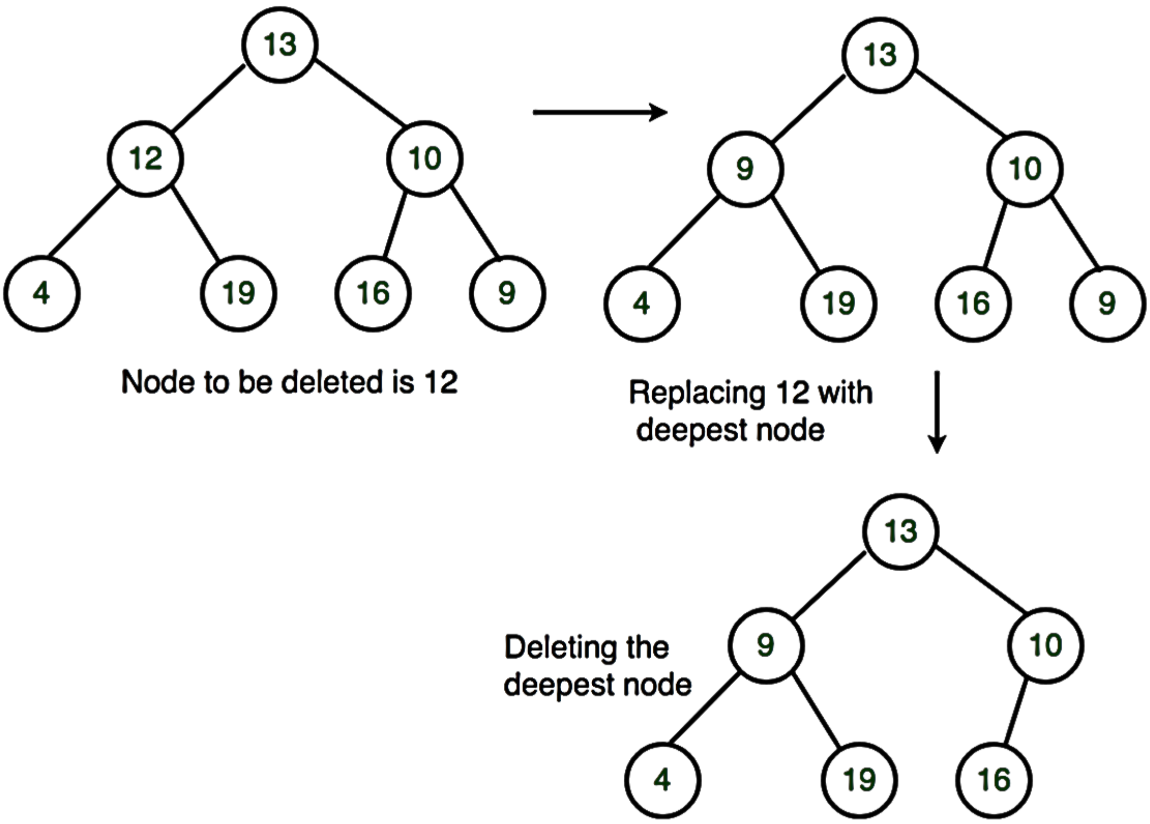
While performing the delete operation on binary trees, there arise a few cases:

1. The node to be deleted is a leaf node. That is it does not have any children.
2. The node to be deleted is a internal node. That is it have left or right child.
3. The node to be deleted is the root node.

In the first case 1, since the node to be deleted is a leaf node, we can simply delete the node without any overheads. But in the next 2 cases, we will have to take care of the children of the node to be deleted.

In order to handle all of the cases, one way to delete a node is to:

1. Starting at the root, find the deepest and rightmost node in binary tree and node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.



Below is the implementation of the above approach:

C++ :-

```
// C++ program to delete element in binary tree
#include <bits/stdc++.h>
using namespace std;
// Binary Tree Node
struct Node
{
    int key;
    struct Node* left, *right;
};
// Utility function to create a new
// Binary Tree Node
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
};
// Function to perform Inorder Traversal
void inorder(struct Node* temp)
{
    if (!temp)
        return;
    inorder(temp->left);
    cout << temp->key << " ";
    inorder(temp->right);
}
// Function to delete the given deepest node
// (d_node) in binary tree
void deletDeepest(struct Node *root, struct Node
*d_node)
{
    queue<struct Node*> q;
    q.push(root);
    // Do level order traversal until last node
```

```
    struct Node* temp;
    while(!q.empty())
    {
        temp = q.front();
        q.pop();
        if (temp->right)
        {
            if (temp->right == d_node)
            {
                temp->right = NULL;
                delete(d_node);
                return;
            }
            else
                q.push(temp->right);
        }
        if (temp->left)
        {
            if (temp->left == d_node)
            {
                temp->left=NULL;
                delete(d_node);
                return;
            }
            else
                q.push(temp->left);
        }
    }
}
// Function to delete element in binary tree
void deletion(struct Node* root, int key)
{
    queue<struct Node*> q;
    q.push(root);
    struct Node *temp;
    struct Node *key_node = NULL;
    // Do level order traversal to find deepest
    // node(temp) and node to be deleted (key_node)
    while (!q.empty())
    {
        temp = q.front();
        q.pop();
        if (temp->key == key)
            key_node = temp;
        if (temp->left)
            q.push(temp->left);
        if (temp->right)
            q.push(temp->right);
    }
    int x = temp->key;
    deletDeepest(root, temp);
    key_node->key = x;
}
// Driver code
int main()
{
    // Create the following Binary Tree
    //          10
    //        /    \
    //      11      9
    //     / \     / \
    //    7   12  15  8
    struct Node* root = newNode(10);
    root->left = newNode(11);
    root->left->left = newNode(7);
    root->left->right = newNode(12);
    root->right = newNode(9);
    root->right->left = newNode(15);
    root->right->right = newNode(8);
    cout << "Inorder traversal before deletion : ";
    inorder(root);
    int key = 11;
    deletion(root, key);
    cout << endl;
    cout << "Inorder traversal after deletion : ";
    inorder(root);
    return 0;
}
```

Java-

```
// Java program to delete element in binary tree
import java.util.LinkedList;
import java.util.Queue;
public class GFG {
    // Binary Tree Node
    static class Node {
        int key;
        Node left, right;
        // constructor
        Node(int key){
            this.key = key;
            left = null;
            right = null;
        }
    }
    static Node root;
    static Node temp = root;
    // Function to perform Inorder traversal
    // of a binary tree
    static void inorder(Node temp)
    {
        if (temp == null)
            return;
        inorder(temp.left);
        System.out.print(temp.key+" ");
        inorder(temp.right);
    }
    // Function to delete the given deepest node
    // (d_node) in binary tree
    static void deleteDeepest(Node root, Node d_node)
    {
        Queue<Node> q = new LinkedList<Node>();
        q.add(root);
        // Do level order traversal until last node
        Node temp;
        while(!q.isEmpty())
        {
            temp = q.peek();
            q.remove();
            if (temp.right!=null)
            {
                if (temp.right == d_node)
                {
                    temp.right = null;
                    d_node = null;
                    return;
                }
                else
                    q.add(temp.right);
            }
            if (temp.left!=null)
            {
                if (temp.left == d_node)
                {
                    temp.left=null;
                    d_node = null;
                    return;
                }
                else
                    q.add(temp.left);
            }
        }
    }
    // Function to delete element in binary tree
    static void deletion(Node root, int key)
    {
        Queue<Node> q = new LinkedList<Node>();
        q.add(root);
        Node temp = null;
        Node key_node = null;
        // Do level order traversal to find deepest
        // node(temp) and node to be deleted (key_node)
        while (!q.isEmpty())
        {
```

```
        temp = q.peek();
        q.remove();
        if (temp.key == key)
            key_node = temp;
        if (temp.left!=null)
            q.add(temp.left);
        if (temp.right!=null)
            q.add(temp.right);
    }
    int x = temp.key;
    deleteDeepest(root, temp);
    key_node.key = x;
}
// Driver code
public static void main(String args[])
{
    // Create the following Binary Tree
    //          10
    //        /    \
    //       11      9
    //      / \     / \
    //     7   12  15  8
    root = new Node(10);
    root.left = new Node(11);
    root.left.left = new Node(7);
    root.left.right = new Node(12);
    root.right = new Node(9);
    root.right.left = new Node(15);
    root.right.right = new Node(8);
    System.out.print( "Inorder traversal before
Deletion:");
    inorder(root);
    int key = 11;
    deletion(root, key);
    System.out.print("\nInorder traversal after
Deletion:");
    inorder(root);
  }
}
```

Output:

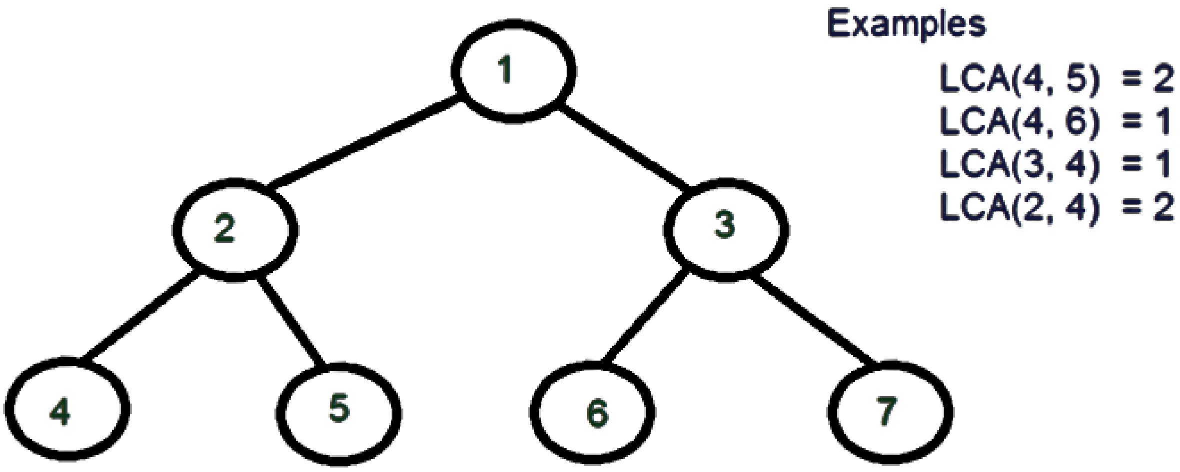
```
Inorder traversal before Deletion: 7 11 12 10 15 9 8
Inorder traversal after Deletion: 7 8 12 10 15 9
```

Finding LCA in Binary Tree-

Given a **Binary Tree** and the value of two nodes **n1** and **n2**. The task is to find the *lowest common ancestor* of the nodes n1 and n2 in the given Binary Tree.

The **LCA** *or* **Lowest Common Ancestor** *of any two nodes N1 and N2 is defined as the common ancestor of both the nodes which is closest to them. That is the distance of the common ancestor from the nodes N1 and N2 should be least possible.*

Below image represents a tree and LCA of different pair of nodes (N1, N2) in it:



Finding LCA

Method 1: The simplest method of finding LCA of two nodes in a Binary Tree is to observe that the LCA of the given nodes will be the last common node in the paths from the root node to the given nodes.

For Example: consider the above-given tree and nodes **4** and **5**.

- Path from root to node 4: [1, 2, 4]
- Path from root to node 5: [1, 2, 5].

The last common node is **2** *which will be the LCA*.

Algorithm:

1. Find the path from the **root** node to node **n1** and store it in a vector or array.
2. Find the path from the **root** node to node **n2** and store it in another vector or array.
3. Traverse both paths untill the values in arrays are same. Return the common element just before the mismatch.

Implementation:

C++ :-

```
// C++ Program for Lowest Common Ancestor
// in a Binary Tree
#include <iostream>
#include <vector>
using namespace std;
// A Binary Tree node
struct Node {
    int key;
    struct Node *left, *right;
};
// Utility function creates a new binary tree
// node with given key
Node* newNode(int k)
{
    Node* temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}
// Function to find the path from root node to
// given root of the tree, Stores the path in a
// vector path[], returns true if path exists
// otherwise false
bool findPath(Node* root, vector<int>& path, int k)
{
    // base case
    if (root == NULL)
        return false;
    // Store this node in path vector.
    // The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;
    // Check if k is found in left or right sub-tree
    if ((root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)))
        return true;
    // If not present in subtree rooted with root,
    // remove root from path[] and return false
    path.pop_back();
    return false;
}
// Function to return LCA if node n1, n2 are
// present in the given binary tree, otherwise
// return -1
int findLCA(Node* root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;
    // Find paths from root to n1 and root to n1.
    // If either n1 or n2 is not present, return -1
    if (!findPath(root, path1, n1) || !findPath(root, path2,
n2))
        return -1;
    // Compare the paths to get the first
    // different value
    int i;
    for (i = 0; i < path1.size() && i < path2.size(); i++)
        if (path1[i] != path2[i])
            break;
    return path1[i - 1];
}
// Driver Code
int main()
{
    // Let us create the Binary Tree
    // as shown in the above diagram
    Node* root = newNode(1);
```

```
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}
```

Java :-

```
// Java Program for Lowest Common Ancestor
// in a Binary Tree
import java.util.ArrayList;
import java.util.List;
// A Binary Tree node
class Node {
    int data;
    Node left, right;
    Node(int value)
    {
        data = value;
        left = right = null;
    }
}
public class FindLCA {
    Node root;
    private List<Integer> path1 = new ArrayList<>();
    private List<Integer> path2 = new ArrayList<>();
    // Finds the path from root node to given root of the
tree
    int findLCA(int n1, int n2)
    {
        path1.clear();
        path2.clear();
        return findLCAInternal(root, n1, n2);
    }
    private int findLCAInternal(Node root, int n1, int n2)
    {
        if (!findPath(root, n1, path1) || !findPath(root, n2,
path2)) {
            System.out.println((path1.size() > 0) ?
                 "n1 is present" : "n1 is missing");
            System.out.println((path2.size() > 0) ?
                 "n2 is present" : "n2 is missing");
            return -1;
        }
        int i;
        for (i = 0; i < path1.size() && i < path2.size(); i++) {
            if (!path1.get(i).equals(path2.get(i)))
                break;
        }
        return path1.get(i - 1);
    }
    // Finds the path from root node to given
    // root of the tree, Stores the path in a
    // vector path[], returns true if path
    // exists otherwise false
    private boolean findPath(Node root, int n,
                             List<Integer> path)
    {
        // base case
        if (root == null) {
            return false;
        }
        // Store this node. The node will be removed if
        // not in path from root to n.
        path.add(root.data);
        if (root.data == n) {
            return true;
        }
        if (root.left != null && findPath(root.left, n, path)) {
            return true;
        }
```

```
        if (root.right != null && findPath(root.right, n, path)) {
            return true;
        }
        // If not present in subtree rooted with root,
        // remove root from path[] and return false
        path.remove(path.size() - 1);
        return false;
    }

    // Driver code
    public static void main(String[] args)
    {
        FindLCA tree = new FindLCA();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        System.out.println("LCA(4, 5): " + tree.findLCA(4, 5));
        System.out.println("LCA(4, 6): " + tree.findLCA(4, 6));
        System.out.println("LCA(3, 4): " + tree.findLCA(3, 4));
        System.out.println("LCA(2, 4): " + tree.findLCA(2, 4));
    }
}
```

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Analysis: *The* **time complexity** *of the above solution is O(N) where N is the number of nodes in the given Tree and the above solution also takes O(N)* **extra space**.

Method 2: The method 1 finds LCA in O(N) time but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from the root node. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtrees. The node which has one key present in its left subtree and the other key present in the right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise, LCA lies in the right subtree.

Below is the implementation of the above approach:

C++ :-

```
// C++ Program to find LCA of n1 and n2 using
// one traversal of Binary Tree

#include <iostream>

using namespace std;

// A Binary Tree Node
struct Node {
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given
// values n1 and n2. This function assumes that
// n1 and n2 are present in Binary Tree
struct Node* findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL)
        return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes
LCA
    if (root->key == n1 || root->key == n2)
        return root;
// Look for keys in left and right subtrees
    Node* left_lca = findLCA(root->left, n1, n2);
    Node* right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL,
    // then one key is present in once subtree and
    // other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)
        return root;

    // Otherwise check if left subtree or
    // right subtree is LCA
    return (left_lca != NULL) ? left_lca : right_lca;
}

// Driver Code
int main()
{
    // Let us create binary tree given in the above example
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "nLCA(2, 4) = " << findLCA(root, 2, 4)->key;

    return 0;
}
```

Java :-

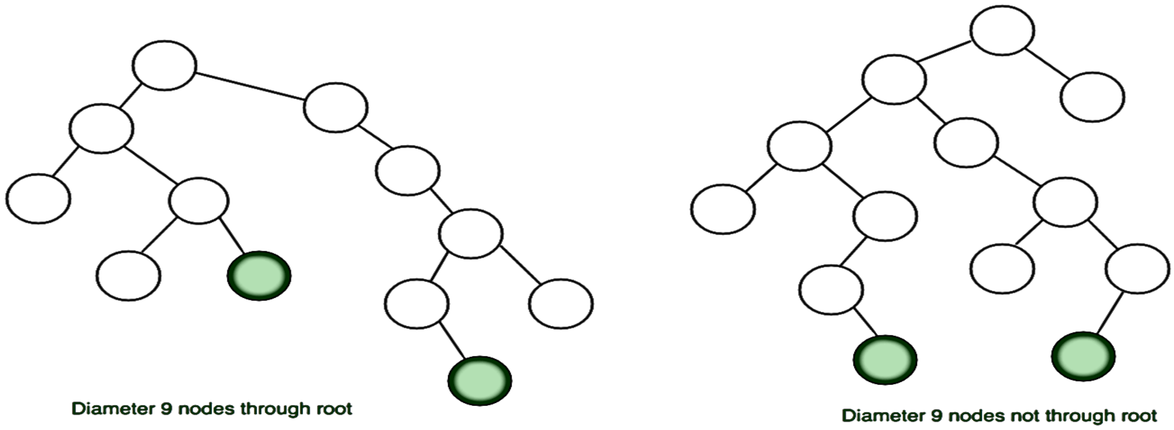
```
// Java implementation to find lowest common ancestor of

// n1 and n2 using one traversal of binary tree

// Class containing left and right child of current

// node and key value

class Node {

    int data;

    Node left, right;

    public Node(int item)

    {

        data = item;

        left = right = null;

    }

}

// Binary Tree Class

public class BinaryTree {

    // Root of the Binary Tree

    Node root;

    Node findLCA(int n1, int n2)

    {

        return findLCA(root, n1, n2);

    }

    // This function returns pointer to LCA of two given

    // values n1 and n2. This function assumes that n1 and

    // n2 are present in Binary Tree

    Node findLCA(Node node, int n1, int n2)

    {

        // Base case

        if (node == null)

            return null;

        // If either n1 or n2 matches with root's key, report

        // the presence by returning root (Note that if a key is

        // ancestor of other, then the ancestor key becomes LCA

        if (node.data == n1 || node.data == n2)

            return node;

        // Look for keys in left and right subtrees

        Node left_lca = findLCA(node.left, n1, n2);

        Node right_lca = findLCA(node.right, n1, n2);

        // If both of the above calls return Non-NULL, then one key

        // is present in once subtree and other is present in other,

        // So this node is the LCA

        if (left_lca != null && right_lca != null)

            return node;

        // Otherwise check if left subtree or right subtree is LCA

        return (left_lca != null) ? left_lca : right_lca;

    }

    // Driver Code

    public static void main(String args[])

    {

        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);

        tree.root.left = new Node(2);

        tree.root.right = new Node(3);

        tree.root.left.left = new Node(4);

        tree.root.left.right = new Node(5);

        tree.root.right.left = new Node(6);

        tree.root.right.right = new Node(7);

        System.out.println("LCA(4, 5) = " + tree.findLCA(4, 5).data);

        System.out.println("LCA(4, 6) = " + tree.findLCA(4, 6).data);

        System.out.println("LCA(3, 4) = " + tree.findLCA(3, 4).data);

        System.out.println("LCA(2, 4) = " + tree.findLCA(2, 4).data);

    }

}
```

Output:

```
LCA(4, 5) = 2nLCA(4, 6) = 1nLCA(3, 4) = 1nLCA(2, 4) = 2
```

Diameter of a Binary Tree-

The **diameter** of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Solution: The diameter of a tree T is the largest of the following quantities:

- The diameter of T's left subtree.
- The diameter of T's right subtree.
- The longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T).

The longest path between leaves that goes through a particular node say, ***nd*** can be calculated as:

1 + height of left subtree of nd + height of right subtree of nd

Therefore, final **Diameter** of a node can be calculated as:

Diameter = maximum(lDiameter, rDiameter, 1 + lHeight + rHeight)
Where,
lDiameter = Diameter of left subtree
rDiameter = Diameter of right subtree
lHeight = Height of left subtree
rHeight = Height of right subtree

Implementation:

```
// C++ program to calculate Diameter of a Binary Tree
#include <bits/stdc++.h>
using namespace std;
// Binary Tree Node
struct node
{
    int data;
    struct node* left, *right;
};
// Function to create a new node of tree
// and returns pointer
struct node* newNode(int data);
// Function to Compute height of a tree
int height(struct node* node);
// Function to get diameter of a binary tree
int diameter(struct node * tree)
{
    /* base case where tree is empty */
    if (tree == NULL)
        return 0;
    /* get the height of left and right sub-trees */
    int lheight = height(tree->left);
    int rheight = height(tree->right);
    /* get the diameter of left and right sub-trees */
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);
    /* Return max of following three
    1) Diameter of left subtree
    2) Diameter of right subtree
    3) Height of left subtree + height of right subtree + 1
*/
    return max(lheight + rheight + 1, max(ldiameter,
rdiameter));
}
/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION
*/
/* The function Compute the "height" of a tree. Height
is the
    number f nodes along the longest path from the
root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if(node == NULL)
        return 0;
    /* If tree is not empty then height = 1 + max of left
        height and right heights */
    return 1 + max(height(node->left), height(node-
>right));
}
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                     malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}
// Driver Code
int main()
{

    /* Constructed binary tree is
          1
         / \
        2   3
       / \
      4   5
    */
    struct node *root = newNode(1);
    root->left     = newNode(2);
```

```
    root->right     = newNode(3);

    root->left->left = newNode(4);

    root->left->right = newNode(5);
```

Java :-

```
// Recursive optimized Java program to find the diameter of a
// Binary Tree
/* Class containing left and right child of current
 node and key value*/
class Node
{
    int data;
    Node left, right;
    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}
/* Class to print the Diameter */
class BinaryTree
{
    Node root;
    /* Method to calculate the diameter and return it to main */
    int diameter(Node root)
    {
        /* base case if tree is empty */
        if (root == null)
            return 0;
        /* get the height of left and right sub trees */
        int lheight = height(root.left);
        int rheight = height(root.right);
        /* get the diameter of left and right subtrees */
        int ldiameter = diameter(root.left);
        int rdiameter = diameter(root.right);
        /* Return max of following three
          1) Diameter of left subtree
         2) Diameter of right subtree
```

```
    cout<<"Diameter of the given binary tree is "<<diameter(root);
    return 0;
}
```

```
         3) Height of left subtree + height of right subtree + 1 */
        return Math.max(lheight + rheight + 1,
                        Math.max(ldiameter, rdiameter));
    }
    /* A wrapper over diameter(Node root) */
    int diameter()
    {
        return diameter(root);
   }
    /*The function Compute the "height" of a tree. Height is the
      number f nodes along the longest path from the root node
      down to the farthest leaf node.*/
    static int height(Node node)
    {
        /* base case tree is empty */
        if (node == null)
            return 0;
        /* If tree is not empty then height = 1 + max of left
           height and right heights */
        return (1 + Math.max(height(node.left), height(node.right)));
    }
    public static void main(String args[])
    {
        /* creating a binary tree and entering the nodes */
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        System.out.println("The diameter of given binary tree is : "
```

```
                        + tree.diameter());                              }
    }
```

Output:

```
Diameter of the given binary tree is 4
```

Time Complexity: $O(N^2)$, where N is the number of nodes in the binary tree.

Left, Right, Top and Bottom View of a Binary Tree-

Problem: Given a Binary Tree. The task is to print the nodes of the binary tree when viewed from different sides. That is, the left view of the binary tree will contain only those nodes which can be seen when the Binary tree is viewed from left.

Example:

```
Consider the Below Binary Tree:
       1
    /     \
   2       3
 /  \    / \
4    5  6   7

Left View of above Tree will be: 1, 2, 4
Right View of above Tree will be: 1, 3, 7
Top View of above Tree will be: 4, 2, 1, 3, 7
Bottom View of above Tree will be: 4, 5, 6, 7
```

Let us now look at each of the solutions in details:

- **Left View**: A simple solution is to notice that the nodes appearing in the left view of the binary tree are the first nodes at every level. So, the idea is to do a level order traversal of the binary tree using a marker to identify levels and print the first node at every level.

Below is the complete function to print left view:

```
// Function to print the left view of the binary tree
void leftViewUtil(Node root)
{
    // Declare a queue for Level order Traversal
    queue<Node*> q;
  if (root == NULL)
    return;
  // Push root
  q.push(root);
  // Delimiter
  q.push(NULL);
  while (!q.empty()) {
    Node* temp = q.front();
    if (temp) {
      // Prints first node
      // of each level
      print temp->data;
      // Push children of all nodes at
      // current level
      while (q.front() != NULL) {
        // If left child is present
        // push into queue
        if (temp->left)
          q.push(temp->left);
        // If right child is present
        // push into queue
        if (temp->right)
          q.push(temp->right);
        // Pop the current node
        q.pop();
        temp = q.front();
      }
      // Push delimiter
      // for the next level
      q.push(NULL);
    }
    // Pop the delimiter of
    // the previous level
    q.pop();
} }
```

- **Right View**: Printing Right View of the Binary Tree is similar to the above approach of printing the Left view of the tree. The idea is to print the last node present at every level. So, perform a level order traversal using a delimeter to identify levels and print the last node present at every level.

- **Top View**: Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. A node **x** is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to the horizontal distance of x minus 1, and that of right child is the horizontal distance of x plus 1.

  So, the idea is to do a level order traversal of the tree and calculate the horizontal distance of every node from the root node and print those nodes which are the first nodes of a particular horizontal distance.

Hashing can be used to keep a check on whether any node with a particular horizontal distance is encountered yet or not.

Below is the function implementing the above approach:

```
// Structure of binary tree
// Binary Tree node is modified to contain
// an extra parameter hd, which is the
// horizontal distance of the node from root node.
struct Node
{
   Node * left;
   Node* right;
   int hd;
   int data;
};
// Function to print the topView of
// the binary tree
void topview(Node* root)
{
   if(root==NULL)
      return;
   queue<Node*>q;
   map<int,int> m;
   int hd=0;
   root->hd = hd;
   // push node and horizontal distance to queue
   q.push(root);
print "The top view of the tree is : ";
while(q.size())
{
   hd = root->hd;
   // Check if any node with this horizontal distance
   // is encontered yet or not.
   // If not insert, current node's data to Map
   if(m.count(hd)==0)
      m[hd]=root->data;
   // Push the left child with its
   // horizontal distance to queue
   if(root->left)
   {
      root->left->hd=hd-1;
      q.push(root->left);
   }
   // Push the right child with its
   // horizontal distance to queue
   if(root->right)
   {
      root->right->hd=hd+1;
      q.push(root->right);
   }
   q.pop();
```

```
        root=q.front();
    }
    // Print the map as it stores the nodes
    // appearing in the Top View
    for(auto i=m.begin();i!=m.end();i++)
    {
        cout<<i->second<<" ";
    }
}
```

- **Bottom View**: The Bottom View of a binary tree can be printed using the similar approach as that of printing the Top View. A node **x** is there in output if x is the bottom most instead of the top most node at its horizontal distance.

  The process of printing the bottom view is almost the same as that of top view with a little modification that while storing the node's data along with a particular horizontal distance in the map, keep updating the node's data in the map for a particular horizontal distance so that the map contains the last node appearing with a particular horizontal distance instead of first.

Threaded Binary Tree Sample-

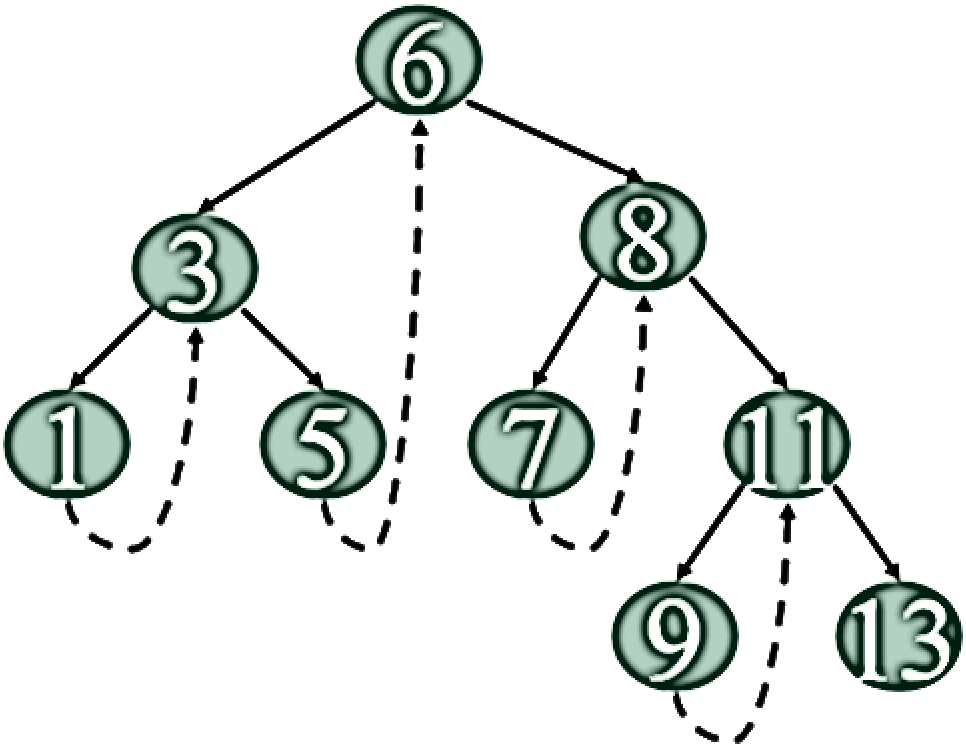
The *Inorder traversal of a Binary tree* can either be done using recursion or with the use of an auxiliary stack. **Threaded Binary Trees** are used to make the inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be *NULL* point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees:

1. ***Single Threaded:*** Where a NULL right pointers is made to point to the inorder successor (if successor exists).
2. ***Double Threaded:*** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Note: The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Representation of a Threaded Node:

C++ :-

```
struct Node

{

   int data;

   Node *left, *right;

   bool rightThread;

}
```

Java-

```
class Node

  {

    int data;

    Node left, right;

    boolean rightThread;

    public Node(int item)

    {

      data = item;

      left = right = null;

    }}
```

Since the right pointer in a Threaded Binary Tree is used for two purposes, the boolean variable *rightThread* is used to indicate whether the right pointer points to a right child or an inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

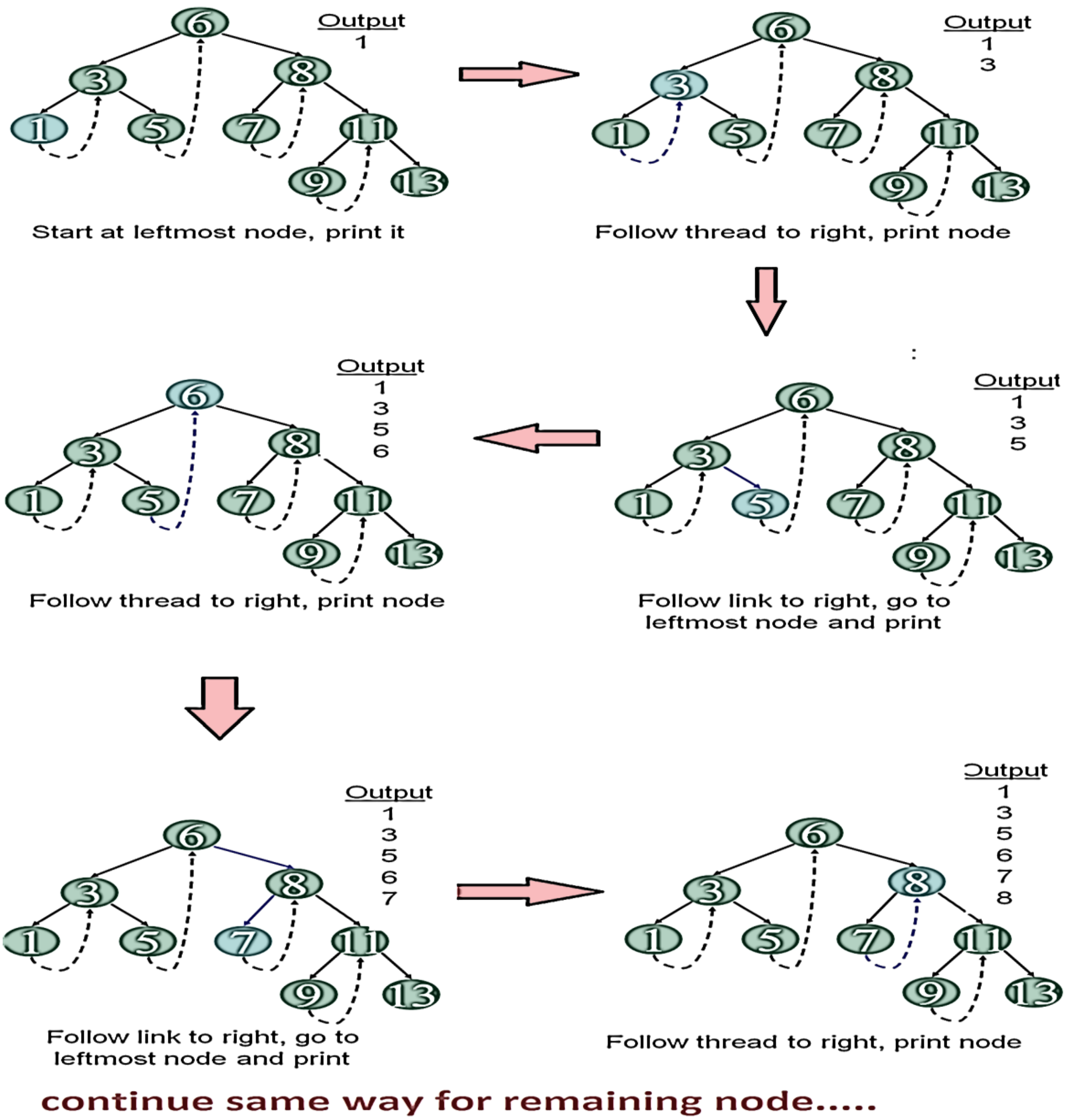
Inorder Traversal in a Threaded Binary Tree: Below is the algorithm to perform inorder traversal in a Threaded Binary Tree using threads:

1. Start from the root node, go to the leftmost node and print the node.

2. Check if there is a thread towards the right for the current node.
    - If Yes, then follow the thread to the node and print the data of node linked with this thread.
    - Otherwise follow the link to the right subtree, find the leftmost node in the right subtree and print the leftmost node.

Repeat the above process until the complete tree is traversed.

Following diagram demonstrates the inorder traversal in a Threaded Binary Tree:



Below functions implements the inorder traversal in a threaded binary tree:

```
// Utility function to find leftmost node
// in a tree rooted with N
Node* leftMost(Node *N)
{
   if (N == NULL)
      return NULL;

   while (N->left != NULL)
      N = N->left;

   return N;
}

// Function to do inorder traversal in a
// threaded binary tree
void inOrder(Node *root)
{
   // Find leftmost node of the root node
   Node *cur = leftmost(root);

   // Until the complete tree is traversed
   while (cur != NULL)
   {
      print cur->data;

      // If this node is a thread node, then go to
      // inorder successor
      if (cur->rightThread)
         cur = cur->right;
      // Else go to the leftmost child in
      // right subtree
      else
         cur = leftmost(cur->right);
   }
}
```

Problems on Trees-

#1 : Print Nodes in Top View of Binary Tree

Description - Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. The horizontal distance of left child of a node x is equal to a horizontal distance of x minus 1, and that of a right child is the horizontal distance of x plus 1.

```
       1
    /     \
   2       3
  /  \    / \
 4    5  6   7
Top view of the above binary tree is
4 2 1 3 7
```

```
     1
   /   \
 2       3
   \
     4
       \
         5
           \
             6
Top view of the above binary tree is
2 1 3 6
```

Solution - The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to put nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at a given horizontal distance is seen or not.

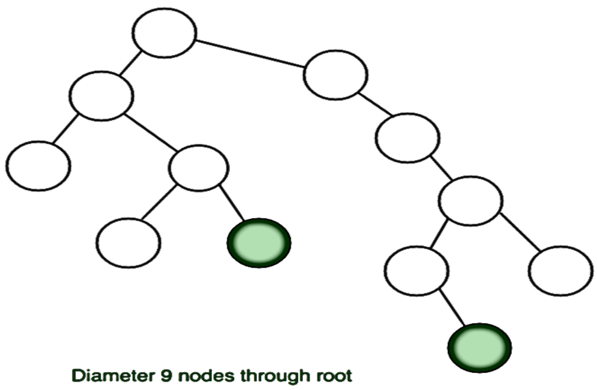
Pseudo Code

```
// function should print the topView of
// the binary tree
void topview(Node* root)
{
    if(root==NULL)
       return;
    queue < Node* > q
    map < int,int > m
    int hd=0
    root->hd=hd

     // push node and horizontal distance to queue
    q.push(root);
    while(!q.empty())
    {
        hd=root->hd
        // check whether node at hd distance seen or not
        if(m.find(hd)==false)
            m[hd]=root->data
        if(root->left)
        {
            root->left->hd=hd-1
            q.push(root->left)
        }
        if(root->right)
        {
            root->right->hd=hd+1
            q.push(root->right)
```

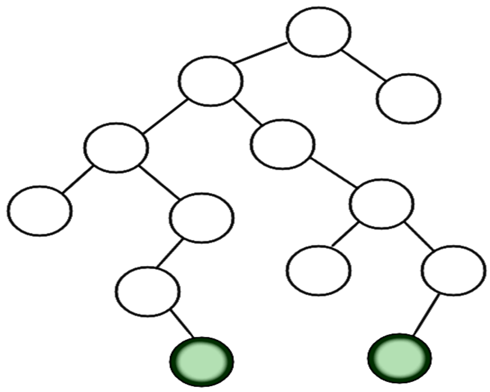
```
        }
        q.pop()
        root=q.front()
    }
    for(it=m.begin();it!=m.end();it++)
    {
        print(it->second)
    }
}
```

Problem #2 : Diameter of a Binary Tree

Description - The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Diameter 9 nodes through root



Diameter 9 nodes not through root

Solution - The diameter of a tree T is the largest of the following quantities:

1. the diameter of T's left subtree
2. the diameter of T's right subtree
3. the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

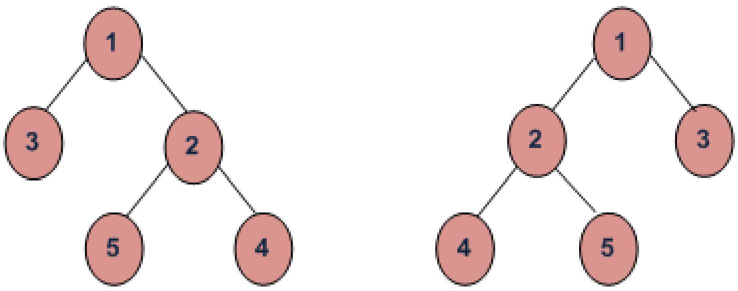
Pseudo Code

```
int diameter(struct node *root, int* height)
{
    /* lh --> Height of left subtree
        rh --> Height of right subtree */
    int lh = 0, rh = 0

    /* ldiameter  --> diameter of left subtree
        rdiameter  --> Diameter of right subtree */
    int ldiameter = 0, rdiameter = 0
```

```
    if(root == NULL)
    {
        *height = 0
        return 0 /* diameter is also 0 */
    }

    /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in ldiameter and ldiameter */
    ldiameter = diameter(root->left, &lh)
    rdiameter = diameter(root->right, &rh)

    /* Height of current node is max of heights of left and
        right subtrees plus 1*/
    *height = max(lh, rh) + 1

    return max(lh + rh + 1, max(ldiameter, rdiameter))
}
```

Problem #3 : Convert a Binary Tree into its Mirror Tree

Description - Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.Trees in the below figure are mirror of each other.



Mirror Trees

Solution - The idea is to recursively call for left and right subtrees of a given node. On each recursive call swap the pointers of the children nodes.

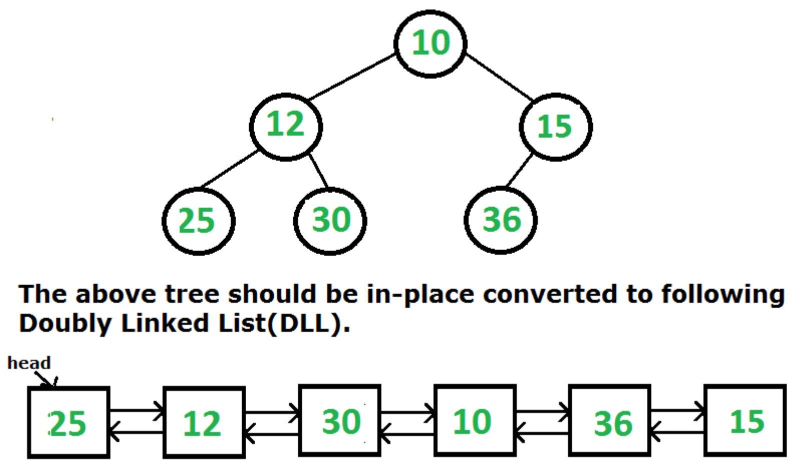
Pseudo Code

```
// function to convert binary tree to it's mirror
void mirror(struct Node* node)
{
    if (node == NULL)
        return
    else
    {
        struct Node* temp

        /* Recur for subtrees */
        mirror(node->left)
```

```
        mirror(node->right)

        /* swap the pointers in this node */
        temp     = node->left
        node->left = node->right
        node->right = temp
    }
}
```

Problem #4 : Convert a given Binary Tree to Doubly Linked List

Description - Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



Solution - The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say prev. For every visited node, make it next of prev and previous of this node as prev.

Pseudo Code

```
// A simple recursive function to convert a given Binary tree to Doubly
// Linked List
// root --> Root of Binary Tree
// head --> Pointer to head node of created doubly linked list
void BinaryTree2DoubleLinkedList(node *root, node **head)
{
    // Base case
    if (root == NULL)
        return

    // Initialize previously visited node as NULL. This is
    // static so that the same value is accessible in all recursive
    // calls
```

```
    static node* prev = NULL

    // Recursively convert left subtree
    BinaryTree2DoubleLinkedList(root->left, head)

    // Now convert this node
    if (prev == NULL)
        *head = root
    else
    {
        root->left = prev
        prev->right = root
    }
    prev = root

    // Finally convert right subtree
    BinaryTree2DoubleLinkedList(root->right, head)
}
```

**HIMANSHU KUMAR(LINKEDIN)**

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.