**Tree**

***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.***

**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.***

**Properties of a Binary Tree**:

1. ***The maximum number of nodes at level 'l' of a binary tree is (2l - 1)***. Level of root is 1.  
     
   This can be proved by induction.  
   For root, l = 1, number of nodes = 21-1 = 1  
   Assume that the maximum number of nodes on level l is 2l-1.  
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 \* 2l-1.
2. ***Maximum number of nodes in a binary tree of height 'h' is (2h – 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 + .. + 2h-1. This is a simple geometric series with h terms and sum of this series is 2h – 1.  
   In some books, the height of the root is considered as 0. In that convention, the above formula becomes 2h+1 – 1.
3. ***In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is Log2(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 Log2(N+1) – 1.
4. ***A Binary Tree with L leaves has at least (Log2L + 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.

1. ***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.

***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

**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.

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

applications of Trees:

* To represent hierarchical data.
* Binary Search Trees.
* Binary heap
* B and B+ Trees in DBMS
* Spanning and Shortest path in computer networks
* Parse Tree and Expression Tree in Compilers.

**Binary Trees Traversal**

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
* **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.

**Binary Tree**

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

}

**Implementation of Inorder Traversal**

#include <iostream>

using namespace std;

struct Node{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key =k;

        left=right=NULL;

    }

};

void inorder(Node \*root){

    if(root!=NULL){

        inorder(root->left);

        cout<<root->key<<" ";

        inorder(root->right);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    inorder(root);

}

**OUTPUT :**

**20 10 40 30 50**

**Implementation of preorder Traversal**

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void preorder(Node \*root){

    if(root!=NULL){

        cout<<root->key<<" ";

        preorder(root->left);

        preorder(root->right);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    preorder(root);

}

OUTPUT :

10 20 30 40 50

**Implementation of postorder Traversal**

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void postorder(Node \*root){

    if(root!=NULL){

        postorder(root->left);

        postorder(root->right);

        cout<<root->key<<" ";

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    postorder(root);

}

**OUTPUT :**

**20 40 50 30 10**

**Height of Binary tree**

Height of Binary Tree is the number of nodes between the longest path from root to leaf node(including the root and leaf node).  
In this video we discuss about a recursive function that takes root of the tree and returns the height of the Binary Tree

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int height(Node \*root){

    if(root==NULL)

        return 0;

    else

        return (1+max(height(root->left),height(root->right)));

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    cout<<height(root);

}

**OUTPUT :**

**3**

**Print Nodes at K distance**

Nodes at distance k from the root are basically the nodes at (k+1)th level of the Binary Tree.  
In this video, we discuss a function that takes root and k as a parameter, whose return type is void and is supposed to print the nodes at distance k from the root.

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void printKDis(Node \*root, int k){

    if(root==NULL)return;

    if(k==0){cout<<root->key<<" ";}

    else{

        printKDis(root->left,k-1);

        printKDis(root->right,k-1);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->right->right=new Node(70);

    root->right->right->right=new Node(80);

    int k=2;

    printKDis(root,k);

}

**OUTPUT :**

**40 50 70**

**Level Order Traversal**

Level order traversal of a tree is breadth first traversal of binary tree.  
In this video we will discuss about a function that takes root as a parameter, doesn’t returns anything and prints the level order traversal in a single line.we implement this function using queue datastructure.

#include <iostream>

#include<queue>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void printLevel(Node \*root){

    if(root==NULL) return;

    queue<Node \*>q;

    q.push(root);

    while (q.empty()==false)

    {

        Node \*curr=q.front();

        q.pop();

        cout<<curr->key<<" ";

        if(curr->left!=NULL)

            q.push(curr->left);

        if(curr->right);

            q.push(curr->right);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->right->left=new Node(60);

    root->right->right=new Node(70);

    printLevel(root);

} OUTPUT : 10 20 30 40 50 60 70

**Level Order Traversal Line By Line (Part 1)**

In Level Order Traversal Line by Line, we print the nodes at each level separately in a new line.  
In this video we discuss:  
A function that takes root as a parameter, doesn’t return anything and prints the level order traversal line by line by using method-1.  
In method-1, we implement this function using a single loop.

#include<iostream>

#include<queue>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void printLevel(Node \*root){

    if(root==NULL) return;

    queue<Node \*>q;

    q.push(root);

    q.push(NULL);

    while (q.size()>1)

    {

        Node \*curr=q.front();

        q.pop();

        if(curr==NULL){

            cout<<"\n";

            q.push(NULL);

            continue;

        }

        cout<<curr->key<<" ";

        if(curr->left!=NULL)

            q.push(curr->left);

        if(curr->right)

            q.push(curr->right);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->left=new Node(50);

    root->right->left=new Node(60);

    root->right->right=new Node(70);

    printLevel(root);

}

**OUTPUT :**

**10**

**20 30**

**50 60 70**

**Level Order Traversal Line By Line (Part 2)**

In Level Order Traversal Line by Line, we print the nodes at each level seperately in a new line.  
In this video we discuss:  
A function that takes root as a parameter, doesn’t return anything and prints the level order traversal line by line by using method-2.  
In method-2, we implement this function using nested loops.

#include<iostream>

#include<queue>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void printLevel(Node \*root){

    if(root==NULL) return;

    queue<Node \*>q;

    q.push(root);

    while (q.empty()==false)

    {

        int count=q.size();

        for(int i=0;i<count; i++){

            Node \*curr=q.front();

            q.pop();

            cout<<curr->key<<" ";

            if(curr->left!=NULL)

                q.push(curr->left);

            if(curr->right!=NULL)

                q.push(curr->right);

        }

        cout<<"\n";

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->right->left=new Node(60);

    root->right->right=new Node(70);

    printLevel(root);

}

**OUTPUT :**

**10**

**20 30**

**40 50 60 70**

**Size Of Binary Tree**

Size of Binary Tree is the total numbers of nodes present in that Tree.  
In this video, we discuss a recursive function that takes root as a parameter and is supposed to return the size of the Tree whose nodes are given.

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int getSize(Node \*root){

    if(root==NULL)

        return 0;

    else

        return 1+getSize(root->left)+getSize(root->right);

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->right->right=new Node(60);

    cout<<getSize(root);

}

**OUTPUT :**

**6**

**Maximum in Binary Tree**

Largest node(key) in a Tree is the maximum of the Tree.  
In this video, we discuss a recursive function that takes the root of a binary Tree and returns the maximum of the Tree.

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int getMax(Node \*root){

    if(root==NULL)

        return INT16\_MIN;

    else

        return max(root->key,max(getMax(root->left),getMax(root->right)));

}

int main(){

    Node \*root=new Node(20);

    root->left=new Node(80);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    cout<<getMax(root);

}

**OUTPUT :**

**80**

**Print Left View of Binary Tree**

To Print Left View of Binary Tree we need to print the leftmost node at every level of the Binary Tree.  
In this video we discuss two methods to print left view of a given Binary Tree.In Method-1 we use Recursive method whereas in Method-2 we use the Iterative method approach by using queue datastructure.

**Recursive :**

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int maxLevel=0;

void printLeft(Node \*root,int level){

    if(root==NULL)

        return;

    if(maxLevel<level){

        cout<<root->key<<" ";

        maxLevel=level;

    }

    printLeft(root->left,level+1);

    printLeft(root->right,level+1);

}

void printLeftView(Node \*root){

    printLeft(root,1);

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    printLeftView(root);

}OUTPUT : 10 20 40

**Iterative for print left view of binary tree**

#include <iostream>

#include <queue>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

void printLeft(Node \*root){

    if(root==NULL)

        return;

    queue<Node \*> q;

    q.push(root);

    while(q.empty()==false){

        int count=q.size();

        for(int i=0;i<count;i++){

            Node \*curr=q.front();

            q.pop();

            if(i==0)

                cout<<curr->key<<" ";

            if(curr->left!=NULL)

                q.push(curr->left);

            if(curr->right!=NULL)

                q.push(curr->right);

        }

    }

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    printLeft(root);

}

**OUTPUT : 10 20 40**

**Children Sum Property**

Children Sum Property is a property in which the sum of values of the left child and right child should be equal to the value of their node if both children are present. Else if only one child is present then the value of the child should be equal to its node value.  
In this video, we discuss a recursive function that takes the root node as a parameter and returns TRUE if the Tree follows C.S.P. and FALSE if the Tree does not follow C.S.P.

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

bool isCSum(Node \*root){

    if(root==NULL)

        return true;

    if(root->left==NULL && root->right==NULL)

        return true;

    int sum=0;

    if(root->left!=NULL) sum+=root->left->key;

    if(root->right!=NULL) sum+=root->right->key;

    return(root->key==sum && isCSum(root->left) && isCSum(root->right));

}

int main(){

    Node \*root=new Node(20);

    root->left=new Node(8);

    root->right=new Node(12);

    root->right->left=new Node(3);

    root->right->right=new Node(9);

    cout<<isCSum(root);

}

**OUTPUT :**

**1**

**Check for Balanced Binary Tree**

In a Balanced Binary Tree for every node, the difference between heights of left subtree and right subtree should not be more than one.  
In this video we discuss two solutions (one with time complexity of O(n^2) and another with time complexity of O(n) ) to check whether a Tree is Balanced or not.

**Naïve :**

#include <bits/stdc++.h>

using namespace std;

//In a Balanced Binary Tree for every node, the difference between heights of left subtree and right subtree should not be more than one.

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int height(Node \*root){

    if(root==NULL)

        return 0;

    else

        return (1+max(height(root->left),height(root->right)));

}

bool isBalanced(Node \*root){

    if(root==NULL)

        return true;

    int lh=height(root->left);

    int rh=height(root->right);

    return (abs(lh-rh)<=1 && isBalanced(root->left) && isBalanced(root->right));

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(5);

    root->right=new Node(30);

    root->right->left=new Node(15);

    root->right->right=new Node(20);

    cout<<isBalanced(root);

} OUTPUT : 1

**Efficient for check for balanced binary tree :**

#include <bits/stdc++.h>

using namespace std;

//In a Balanced Binary Tree for every node, the difference between heights of left subtree and right subtree should not be more than one.

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int isBalanced(Node \*root){

    if(root==NULL)

        return 0;

    int lh=isBalanced(root->left);

    if(lh==-1)return -1;

    int rh=isBalanced(root->right);

    if(rh==-1)return -1;

    if(abs(lh-rh)>1)

        return -1;

    else

        return max(lh,rh)+1;

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(5);

    root->right=new Node(30);

    root->right->left=new Node(15);

    root->right->right=new Node(20);

    if(isBalanced(root))

        cout<<"Balanced";

    else

        cout<<"Not Balanced";

}

**OUTPUT : Balanced**

**Maximum Width of Binary tree**

Maximum Width of Binary tree is the maximum number of nodes present in a level of the Tree.  
In this video, we discuss a function that takes the root of a Binary Tree and returns the maximum width of the Binary Tree.  
Hint:- we use level order traversal line by line logic to find maximum width of the Binary Tree.

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int maxWidth(Node \*root){

    if(root==NULL)return 0;

    queue<Node \*>q;

    q.push(root);

    int res=0;

    while(q.empty()==false){

        int count=q.size();

        res=max(res,count);

        for(int i=0;i<count;i++){

            Node \*curr=q.front();

            q.pop();

            if(curr->left!=NULL)

                q.push(curr->left);

            if(curr->right!=NULL)

                q.push(curr->right);

        }

    }

    return res;

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->right->left=new Node(60);

    cout<<maxWidth(root);

} OUTPUT : 3

**Convert Binary Tree to Doubly linked list**

Inorder conversion of Binary Tree to Doubly Linked List.  
A function that takes root of a Binary Tree and converts it into a Doubly linked list.  
Hint:- we need to do the inorder traversal of the Tree and while doing inorder traversal we can simply maintain a previous pointer or reference of the previously traversed node. And change right child of the previous node to current node and left child of current node as previous.

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

void printlist(Node \*head){

    Node \*curr=head;

    while(curr!=NULL){

        cout<<curr->key<<" ";

        curr=curr->right;

    }cout<<endl;

}

Node \*BTToDLL(Node \*root){

    if(root==NULL)return root;

    static Node\* prev = NULL;

    Node \*head=BTToDLL(root->left);

    if(prev==NULL){head=root;}

    else{

        root->left=prev;

        prev->right=root;

    }

    prev=root;

    BTToDLL(root->right);

    return head;

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(5);

    root->right=new Node(20);

    root->right->left=new Node(30);

    root->right->right=new Node(35);

    Node \*head=BTToDLL(root);

    printlist(head);

}

**OUTPUT :**

**5 10 30 20 35**

**Construct Binary tree from inorder and preorder**

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void inorder(Node \*root){

    if(root!=NULL){

    inorder(root->left);

    cout<<root->key<<" ";

    inorder(root->right);

    }

}

int preIndex=0;

Node \*cTree(int in[],int pre[],int is, int ie){

    if(is>ie) return NULL;

    Node \*root=new Node(pre[preIndex++]);

    int inIndex;

    for(int i=is;i<=ie;i++){

        if(in[i]==root->key){

            inIndex=i;

            break;

        }

    }

    root->left=cTree(in , pre, is, inIndex-1);

    root->right=cTree(in, pre , inIndex+1, ie);

    return root;

}

int main(){

    int in[]={20,10,40,30,50};

    int pre[]={10,20,30,40,50};

    int n=sizeof(in)/sizeof(in[0]);

    Node \*root=cTree(in, pre, 0, n-1);

    inorder(root);

}

**OUTPUT :**

**20 10 40 30 50**

**Tree Traversal in spiral form**

**Method 1:**

#include <iostream>

#include <queue>

#include<stack>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

void printSpiral(Node \*root){

    if(root==NULL)return;

    queue<Node \*>q;

    stack<int> s;

    bool reverse=false;

    q.push(root);

    while(q.empty()==false){

        int count=q.size();

        for(int i=0;i<count;i++){

            Node \*curr=q.front();

            q.pop();

            if(reverse)

                s.push(curr->key);

            else

                cout<<curr->key<<" ";

            if(curr->left!=NULL)

                q.push(curr->left);

            if(curr->right!=NULL)

                q.push(curr->right);

        }

        if(reverse){

            while(s.empty()==false){

                cout<<s.top()<<" ";

                s.pop();

            }

        }

        reverse=!reverse;

    }

}

int main() {

    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);

    root->right->left=new Node(6);

    root->right->right=new Node(7);

    printSpiral(root);

}

**OUTPUT :**

**1 3 2 4 5 6 7**

**Method 2:**

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

void printSpiral(Node \*root){

    if (root == NULL)

        return;

    stack<Node\*> s1;

    stack<Node\*> s2;

    s1.push(root);

    while (!s1.empty() || !s2.empty()) {

        while (!s1.empty()) {

            Node\* temp = s1.top();

            s1.pop();

            cout << temp->key << " ";

            if (temp->right)

                s2.push(temp->right);

            if (temp->left)

                s2.push(temp->left);

        }

        while (!s2.empty()) {

            Node\* temp = s2.top();

            s2.pop();

            cout << temp->key << " ";

            if (temp->left)

                s1.push(temp->left);

            if (temp->right)

                s1.push(temp->right);

        }

    }

}

int main() {

    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);

    root->right->left=new Node(6);

    root->right->right=new Node(7);

    printSpiral(root);

}

**OUTPUT :**

**1 2 3 7 6 5 4**

**Diameter of Binary tree**

**Naïve :**

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node (int k){

        key=k;

        left=right=NULL;

    }

};

int height(Node \*root){

    if(root==NULL)

        return 0;

    else

        return (1+max(height(root->left),height(root->right)));

}

int diameter(Node \*root){

    if(root==NULL) return 0;

    int d1=1+height(root->left)+height(root->right);

    int d2=diameter(root->left);

    int d3=diameter(root->right);

    return max(d1,max(d2,d3));

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(60);

    root->right->left->left=new Node(50);

    root->right->right->right=new Node(70);

    cout<<diameter(root);

}

**OUTPUT : 5**

**Efficient for diameter of binary tree :**

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int res=0;

int height(Node \*root){

    if(root==NULL)

        return 0;

    int lh=height(root->left);

    int rh=height(root->right);

    res=max(res,1+lh+rh);

    return 1+max(lh,rh);

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(60);

    root->right->left->left=new Node(50);

    root->right->right->right=new Node(70);

    cout<<"Height: "<<height(root)<<endl;

    cout<<"Diameter: "<<res;

}

**OUTPUT : Height: 4**

**Diameter: 5**

**LCA of Binary tree part 1**

LCA (Lowest Common Ancestor) problem and a O(n) solution to the problem.

#include <iostream>

#include<vector>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

bool findPath(Node \*root, vector<Node \*> &p,int n){

    if(root==NULL) return false;

    p.push\_back(root);

    if(root->key==n)return true;

    if(findPath(root->left,p,n) || findPath(root->right,p,n)) return true;

    p.pop\_back();

    return false;

}

Node \*lca(Node \*root,int n1, int n2){

    vector<Node \*>path1,path2;

    if(findPath(root,path1,n1)==false || findPath(root,path2,n2)==false)

        return NULL;

    for(int i=0;i<path1.size()-1 && i<path2.size()-1; i++)

        if(path1[i+1]!=path2[i+1])

            return path1[i];

    return NULL;

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    int n1=20,n2=50;

    Node \*ans=lca(root,n1,n2);

    cout<<"LCA: "<<ans->key;

}

**OUTPUT :**

**LCA: 10**

**Efficient for LCA of Binary tree part 2**

The solution does only one traversal of binary tree, but assumes that both keys exist in the binary tree.

#include <iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

Node \*lca(Node \*root, int n1, int n2){

    if(root==NULL)return NULL;

    if(root->key==n1 || root->key==n2)

        return root;

    Node \*lca1=lca(root->left,n1,n2);

    Node \*lca2=lca(root->right,n1,n2);

    if(lca1!=NULL && lca2!=NULL)

        return root;

    if(lca1!=NULL)

        return lca1;

    else

        return lca2;

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    int n1=20,n2=50;

    Node \*ans=lca(root,n1 , n2);

    cout<<"LCA: "<<ans->key;

}

**OUTPUT : LCA: 10**

**Burn a Binary tree from a leaf**

We are given a binary tree and a leaf node, we need to find time to burn the Binary Tree if we burn the given leaf at 0-th second.

#include<iostream>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int res=0;

int burnTime(Node \*root, int leaf,int &dist){

    if(root==NULL)return 0;

    if(root->key==leaf){dist=0;return 1;}

    int ldist=-1,rdist=-1;

    int lh=burnTime(root->left,leaf,ldist);

    int rh=burnTime(root->right,leaf,rdist);

    if(ldist!=-1){

        dist=ldist+1;

        res=max(res,dist+rh);

    }

    else if(rdist!=-1){

        dist=rdist+1;

        res=max(dist,dist+lh);

    }

    return max(lh,rh)+1;

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->left->left=new Node(40);

    root->left->right=new Node(50);

    root->left->left->left=new Node(60);

    root->left->left->left->left=new Node(70);

    int dist=-1;int leaf=50;

    burnTime(root,leaf,dist);

    cout<<res;

} OUTPUT : 4

**Count Node in Complete Binary tree**

Given a binary tree, our task is to count toal nodes.  Two methods are discussed here, naive method which is O(n).  And an efficient method which is O(Log n \* Log n)

**Naïve :**

#include<iostream>

using namespace std;

struct Node{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

int countNode(Node \*root){

    if(root==NULL)return 0;

    else

        return 1+countNode(root->left)+countNode(root->right);

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    cout<<countNode(root);

}

**OUTPUT :**

**5**

**Efficient for count Node in complete binary tree :**

#include <bits/stdc++.h>

using namespace std;

struct Node

{

  int key;

  struct Node \*left;

  struct Node \*right;

  Node(int k){

      key=k;

      left=right=NULL;

  }

};

int countNode(Node \*root){

    int lh=0,rh=0;

    Node \*curr=root;

    while(curr!=NULL){

        lh++;

        curr=curr->left;

    }

    curr=root;

    while(curr!=NULL){

        rh++;

        curr=curr->right;

    }

    if(lh==rh){

        return pow(2,lh)-1;

    }else{

        return 1+countNode(root->left)+countNode(root->right);

    }

}

int main() {

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    cout<<countNode(root);

}

**OUTPUT : 5**

**Iterative Inorder Traversal**

#include <iostream>

#include<stack>

using namespace std;

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void iterativeInorder(Node \*root){

    stack<Node \*>s;

    Node \*curr=root;

    while (curr!=NULL || s.empty()==false)

    {

        while (curr!=NULL)

        {

            s.push(curr);

            curr=curr->left;

        }

        curr=s.top();

        s.pop();

        cout<<curr->key<<" ";

        curr=curr->right;

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    iterativeInorder(root);

}

OUTPUT : 20 10 40 30 50

**Iterative Preorder Traversal (simple)**

#include<iostream>

#include<stack>

using namespace std;

//A O(n) extra space and O(n) time solution

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void iterativePreoder(Node \*root){

    if(root==NULL) return;

    stack<Node \*>s;

    s.push(root);

    while (s.empty()==false)

    {

        Node \*curr=s.top();

        cout<<(curr->key)<<" ";

        s.pop();

        if(curr->right!=NULL)

            s.push(curr->right);

        if(curr->left!=NULL)

            s.push(curr->left);

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    iterativePreoder(root);

}

**OUTPUT : 10 20 30 40 50**

**Iterative Preorder Traversal (space Optmized)**

#include <iostream>

#include<stack>

using namespace std;

//A O(h) extra space and O(n) time solution

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    Node(int k){

        key=k;

        left=right=NULL;

    }

};

void iterativePreoder(Node \*root)

{

    if(root==NULL) return;

    stack<Node \*> s;

    Node \*curr=root;

    while(curr!=NULL || s.empty()==false)

    {

        while (curr!=NULL)

        {

            cout<<curr->key<<" ";

            if(curr->right!=NULL)

                s.push(curr->right);

            curr=curr->left;

        }

        if(s.empty()==false)

        {

            curr=s.top();

            s.pop();

        }

    }

}

int main(){

    Node \*root=new Node(10);

    root->left=new Node(20);

    root->right=new Node(30);

    root->right->left=new Node(40);

    root->right->right=new Node(50);

    iterativePreoder(root);

}

OUTPUT : 10 20 30 40 50

**Serialize and deserialize a Binary tree**

serialize and deserialize a binary tree. It discusses preorder traversal based approach.

**Serialize :**