1.Left view :

Recursive:

```
void leftview(node* root,int level,int &max_level){
	if(root==NULL)return;
	if(max_level<level){
		cout<<root->value<<" ";
		max_level=level;
	}
	leftview(root->left,level+1,max_level);
	leftview(root->right,level+1,max_level);
}
```

Iterative:

```
void leftview(node* root){
	if(root==NULL)return;
	queue<node*> q;
	q.push(root);
	while(!q.empty()){
		int n=q.size();
		for(int i=0;i<n;i++){
			node* t=q.front();
			q.pop();
			if(i==0)cout<<t->value<<" ";
			if(t->left)q.push(t->left);
			if(t->right)q.push(t->right);
		}
	}
}
```

2.Right view:

Recursive:

```
void rightview(node* root,int level,int &max_level)
{
	if(root==NULL)return;
	if(max_level<level){
		cout<<root->value<<" ";
		max_level=level;
	}
	rightview(root->right,level+1,max_level);
	rightview(root->left,level+1,max_level);
}
```

Iterative:

```
void leftview(node* root){
	if(root==NULL)return;
	queue<node*> q;
	q.push(root);
	while(!q.empty()){
		int n=q.size();
		for(int i=0;i<n;i++){
			node* t=q.front();
			q.pop();
			if(i==n-1)cout<<t->value<<" ";
			if(t->left)q.push(t->left);
			if(t->right)q.push(t->right);
		}
```

```
	}
}
```

3.Preorder:

Iterative:

```
void iterative_preorder(Node* root){
	stack<Node*> s;
	s.push(root);
	while(!s.empty()){
		Node* curr=s.top();
		s.pop();
		cout<<curr->val<<" ";
		if(curr->right)s.push(curr->right);
		if(curr->left)s.push(curr->left);
	}
	return;
}
```

4:Postorder:

Iterative:

```
void iterative_postorder(Node* root){
	stack<Node*> s;
	s.push(root);
	stack<int> s1;
	while(!s.empty()){
		Node* curr=s.top();
```

```
        s.pop();
        s1.push(curr->val);
        if(curr->left)s.push(curr->left);
        if(curr->right)s.push(curr->right);

    }
    while(!s1.empty()){
        cout<<s1.top()<<" ";
        s1.pop();
    }
    return;
}
```

5.Inorder:

Iterative:

```
void iterative_inorder(Node* root){
    if(root==NULL)return;
    stack<Node*> s;
    Node* curr=root;
    while(!s.empty() || curr){
        if(curr){
            s.push(curr);
            curr=curr->left;
        }
        else{
            curr=s.top();
            s.pop();
            cout<<curr->val<<" ";
```

```
            curr=curr->right;
        }

    }
}
```

6:Diameter:

Method1:

```
int diameter(node* root){
    if(root==NULL)return 0;
    int d1=diameter(root->left);
    int d2=diameter(root->right);
    int h=height_tree(root->left)+height_tree(root->right)+1;
    return(max(max(d1,d2),h));
}
```

complexity=O(n^2).
Method2:
Return optimal_diameter_tree(root,height)-1;

```
int optimal_diameter_tree(node* root,int &height){
    if(root==NULL)return 0;
    int lh=0,rh=0;
    int ld=optimal_diameter_tree(root->left,lh);
    int rd=optimal_diameter_tree(root->right,rh);
    int currdiameter=lh+rh+1;
    height=max(lh,rh)+1;
    return max(max(ld,rd),currdiameter);
```

```
}
complexity=O(n)
```

8.Level order traversal:

```
void levelorder_traversal(node* root){
	queue<node*> q;
	q.push(root);
	while(!q.empty()){
		node* t=q.front();
		q.pop();
		cout<<t->value<<" ";
		if(t->left)q.push(t->left);
		if(t->right)q.push(t->right);
	}
	return ;
}
```

9.Reverse of level order traversal:

```
void reverse_levelorder_traversal(node* root){
	queue<node*> q;
	stack<int> s;
	q.push(root);
	while(!q.empty()){
		node* t=q.front();
		q.pop();
		s.push(t->value);
		if(t->right)q.push(t->right);
		if(t->left)q.push(t->left);
```

```
    }
    while(!s.empty()){
        cout<<s.top()<<" ";
        s.pop();
    }
    return;
}
```

10:Mirror image

```
void mirror_image(node* root)
{
    if(root==NULL)return;
    node* t=root->right;
    root->right=root->left;
    root->left=t;
    mirror_image(root->left);
    mirror_image(root->right);
}
```

11.zigzag traversal:

```
void zigzag_traversal(node* root){
    stack<node*> q1;
    stack<node*> q2;
    q1.push(root);
    bool left_to_right=1;
    while(!q1.empty()){
        node* t=q1.top();
        q1.pop();
```

```
            cout<<t->value<<" ";
            if(t){
            if(left_to_right){
                    if(t->left)q2.push(t->left);
                    if(t->right)q2.push(t->right);
            }
            else {
                    if(t->right)q2.push(t->right);
                    if(t->left)q2.push(t->left);
            }}
            if(q1.empty()){
            left_to_right=!left_to_right;
            swap(q1,q2);
        }
        }
}
```

Complexity:Time=O(n)--->if we use stl swap function
Time=O(height*N)--->if we use our func

12.Check Balanced tress:

```
bool check_balanced(node* root,int &height)
{
        if(root==0)return true;
        int lh=0,rh=0;
        bool l1=check_balanced(root->left,lh);
        bool l2=check_balanced(root->right,rh);
        height=max(lh,rh)+1;
        return (l1 && l2 && abs(lh-rh)<=1 );
```

```
}
```

13:Tree to DLL

```
Node* bToDLL(Node *root)
  {
     Node* head=NULL;
     Node* tail=NULL;
      bt(root,&head,&tail);
      return head;
  }
  void bt(Node* root,Node **head,Node **tail){
     if(root==NULL)return;
     //bt(root->left,head);
     bt(root->left,head,tail);
     if(*head==NULL){
        *head=root;
     }
        root->left=*tail;
        if(*tail)
        (*tail)->right=root;
     *tail=root;
      bt(root->right,head,tail);
  }
```

2nd way of same comp:

```
class Solution
{
   public:
   Node* prev=NULL;
   Node* head=NULL;

   Node * bToDLL(Node *root)
   {
      if(!root)return NULL;
      bToDLL(root->left);
      if(prev==NULL)head=root;
      else{
         root->left=prev;
         prev->right=root;
      }
      prev=root;
      bToDLL(root->right);
      return head;
   }
```

14:Sum Tree:

```
int sumtree(node* root)
{
      if(root==NULL)return 0;
      int prev=root->value;
```

```
        root->value=sumtree(root->left)+sumtree(root->right);
        return root->value+prev;
}
```

15.Build tree_inorder_&_preorder:

```
Int search(int pre[],int s,int e,int value){
for(int i=s;i<=e;i++){
if(pre[i]==value)return i;
}
}
node* Build tree_inorder_&_preorder(int in[],pre[],int s,int e)
{
if(s>e) return;
Static Int index=0;
node* tnode = new node(pre[index++]);
if(s==e)return tnode;
Int search_index=search(pre,s,e,tnode->value);
tnode->left=
Build tree_inorder_&_preorder(in,pre,s,search_index-1);
tnode->right=
Build tree_inorder_&_preorder(in,pre,search_index+1,e);
Return tnode;
}
```

Complexity :O(N)^2.

Efficient approach:Using maps.

```
class Solution{
    public:
    int idx=0;
    unordered_map<int,int> m;
    Node* bt(int in[],int pre[],int lb,int ub){
        if(lb>ub)return NULL;
        Node* node=new Node(pre[idx++]);
        if(lb==ub)return node;
        int mid=m[node->data];
        node->left= bt(in,pre,lb,mid-1);
        node->right= bt(in,pre,mid+1,ub);

        return node;
    }
    Node* buildTree(int in[],int pre[], int n)
    {
        for(int i=0;i<n;i++)m[in[i]]=i;
        return  bt(in,pre,0,n-1);
    }
};
```

Complexity:Time=O(N) because search in maps is in O(1)

16:Check whether a tree is sum tree or not:

```
class Solution
{
   public:
   bool f=1;
   int check_sumtree(Node* root)
   {
      if(!root)return 0;
      if(!root->left && !root->right)return root->data;
      int a=check_sumtree(root->left);
      int b=check_sumtree(root->right);
      if(f==0)return 0;
      if(a+b!=root->data)f=0;
      return a+b+root->data;
   }
   bool isSumTree(Node* root)
   {
       check_sumtree(root);
       return f;
   }
```

17.Top view tree:

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 given horizontal distance is seen or not.

```
vector<int> topView(Node *root)
   {
     vector<int> v;
      if(!root)return v;
     map<int,int> m;
     queue<pair<Node*,int>> q;
     q.push({root,0});
     while(!q.empty())
     {
         Node* t=q.front().first;
         int hd=q.front().second;
         if(!m[hd])m[hd]=t->data;
         q.pop();
         if(t->left)q.push({t->left,hd-1});
         if(t->right)q.push({t->right,hd+1});
     }
     for(auto x:m)v.push_back(x.second);
     return v;
   }
```

18.Bottom View of a tree:

```
vector<int> bottomView(Node *root)
   {
     vector<int> v;
      if(!root)return v;
```

```
    map<int,int> m;
    queue<pair<Node*,int>> q;
    q.push({root,0});
    while(!q.empty())
    {
        Node* t=q.front().first;
        int hd=q.front().second;
        m[hd]=t->data;
        q.pop();
        if(t->left)q.push({t->left,hd-1});
        if(t->right)q.push({t->right,hd+1});
    }
    for(auto x:m)v.push_back(x.second);
    return v;
  }
```

19.Diagonal Traversal of a Binary tree:

Using queue to store left nodes;

```
vector<int> diagonal(Node *root)
{
   vector<int> v;
   if(!root)return v;
   queue<Node*> q;
   q.push(root);
   while(!q.empty())
   {
```

```
        Node* t=q.front();q.pop();
        while(t)
        {
        if(t->left)q.push(t->left);
        v.push_back(t->data);
         t=t->right;
        }
    }
    return v;
}
```

20.Boundary traversal of a Binary tree:

Here we are using 3 functions

```
class Solution {
public:
    void left_nodes(Node* root,vector<int>& v)
    {
        if(!root)return;
        if(root->left)
        {
            v.push_back(root->data);
            left_nodes(root->left,v);
        }
        else if(root->right)
        {
            v.push_back(root->data);
            left_nodes(root->right,v);
```

```
    }
}
void leaf_nodes(Node* root,vector<int>& v)
{
    if(!root)return;
    leaf_nodes(root->left,v);
    if(!root->left && !root->right)v.push_back(root->data);
    leaf_nodes(root->right,v);
}
 void right_nodes(Node* root,vector<int>& v)
{
    if(!root)return;
    if(root->right)
    {
        right_nodes(root->right,v);
        v.push_back(root->data);
    }
    else if(root->left)
    {
        right_nodes(root->left,v);
        v.push_back(root->data);
    }
}
vector <int> boundary(Node *root)
{
   vector<int> v;
   v.push_back(root->data);
   if(!root->left && !root->right)return v;
   left_nodes(root->left,v);
   leaf_nodes(root,v);
```

```
        right_nodes(root->right,v);
        return v;
    }
};
```

21.Construct Binary Tree from String with Bracket Representation:

```
#include<bits/stdc++.h>
using namespace std;
struct Node{
    int data;
    Node* left;
    Node* right;
    Node(int val)
    {
        data=val;
        left=right=NULL;
    }
};
int get_median(string s,int si,int ei)
{
    if(si>ei)return -1;
    stack<int> st;
    for(int i=si;i<=ei;i++)
    {
    if(s[i]=='(')st.push(s[i]);
```

```
    else if(s[i]==')')
            {
        if(st.top()=='(')st.pop();
        if(st.empty())return i;

            }
    }
}
Node* get_tree(string s,int si,int ei)
{
    if(si>ei)return NULL;

    Node* root=new Node(s[si]-'0');
    int index=-1;

    if(si+1<=ei && s[si+1]=='(')
    index=get_median(s,si+1,ei);
    if(index!=-1)
    {
        root->left=get_tree(s,si+2,index-1);
        root->right=get_tree(s,index+2,ei);
    }
    return root;
}
void preorder(Node* root){
    if(!root)return;
    cout<<root->data<<" ";
    preorder(root->left);
```

```
    preorder(root->right);

}
int main(){
    string s="4(2(3)(1))(6(5))";
    Node* root=get_tree(s,0,s.size()-1);
    preorder(root);
    return 0;
}
```

22.Convert Binary tree into Doubly Linked List:

```
class Solution
{
    public:
    Node* prev=NULL;
    Node* head=NULL;
    //Function to convert binary tree to doubly linked list
and return it.
    Node * bToDLL(Node *root)
    {
        if(!root)return NULL;
        bToDLL(root->left);
        if(prev==NULL)head=root;
        else{
            root->left=prev;
            prev->right=root;
```

```
        }
        prev=root;
        bToDLL(root->right);
        return head;
    }
};
```

23.Construct Binary tree from Inorder and preorder traversal:

Method 1:if we use linear search for finding index,then Complexity will be O(N^2)

Method 2:using map search time will be O(1)

```
class Solution{
    public:
    int idx=0;
    unordered_map<int,int> m;
    Node* bt(int in[],int pre[],int lb,int ub){
        if(lb>ub)return NULL;
        Node* node=new Node(pre[idx++]);
        if(lb==ub)return node;
        int mid=m[node->data];
        node->left= bt(in,pre,lb,mid-1);
        node->right= bt(in,pre,mid+1,ub);
```

```
        return node;
    }
    Node* buildTree(int in[],int pre[], int n)
    {
        for(int i=0;i<n;i++)m[in[i]]=i;
        return  bt(in,pre,0,n-1);
    }
};
```

Complexity,Time=O(N)   space=O(N)

23.Check if Leaf nodes are at same level :

```
class Solution{
  public:
    int ans;
      void check_leaf(Node* root,int h,int& m)
{
        if(!root)return;
        if(ans==0)return;
        if(!root->left && !root->right)
        {
            if(m==-1)m=h;
            else if(m!=h){ans=0;return;}
        }
        check_leaf(root->left,h+1,m);
        check_leaf(root->right,h+1,m);
    }

bool check(Node *root)
```

```
    {
        int h=0,m=-1;
        ans=1;
        check_leaf(root,h,m);
        return ans;
    }
};
```

24.Check if a Binary Tree contains duplicate subtrees of size 2 or more:

```
class Solution {
  public:
    unordered_map<string,int> m;
    string check_dupsub(Node* root)
    {
        if(!root)return "$";
        string s="";
        if(!root->left && !root->right)
        {
            s=to_string(root->data);
            return s;
         }
        s=s+to_string(root->data);
        s=s+check_dupsub(root->left);
        s=s+check_dupsub(root->right);
        m[s]++;
```

```
        return s;
    }
    int dupSub(Node *root)
    {
    string s=check_dupsub(root);
    int c=0;
    for(auto x:m)if(x.second>1)c++;
    return c;
    }
};
```

25.Check if 2 trees are mirror or not:

Here we are given Arrays with edges
n = 3, e = 2
A[] = {1, 2, 1, 3}
B[] = {1, 3, 1, 2}
Output:
1

```
int checkMirrorTree(int n, int e, int A[], int B[])
    {
        unordered_map<int, stack<int>>m;
       for(int i=0;i<2*e;i+=2){
          m[A[i]].push(A[i+1]);
       }
       for(int i=0;i<2*e;i+=2){
           int x=m[B[i]].top();
           m[B[i]].pop();
           if(x!=B[i+1])
```

```
        return 0;
      }
      return 1;
   }
```

26.Sum of Nodes on the Longest path from root to leaf node:

Here we are using a vector which contain only 2 elements
v[0]--->height
v[1]---->sum

```
class Solution
{
public:
   vector<int> sumLRLeaf(Node* root)
   {
      if(!root)return {0,0};
      vector<int> a=sumLRLeaf(root->left);
      vector<int> b=sumLRLeaf(root->right);

      if(a[0]>b[0])return {a[0]+1,a[1]+root->data};
      else if(a[0]<b[0])return {b[0]+1,b[1]+root->data};
      else return{a[0]+1,max(a[1],b[1])+root->data};
   }
   int sumOfLongRootToLeafPath(Node *root)
   {
      vector<int> v=sumLRLeaf(root);
      return v[1];
```

```
    }
};
```

27.Find Largest subtree sum in a tree:

```
int findLargestSubtreeSumUtil(Node* root,int & ans)
{
        if(!root)return 0;
        int a=findLargestSubtreeSumUtil(root->left,ans);
        Int b=findLargestSubtreeSumUtil(root->right,ans);
        return max(ans,a+b+root->data);
}
```

28.Maximum Sum of nodes in Binary tree such that no two are adjacent:

```
class Solution{
  public:
    unordered_map<Node*,int> m;
    int getMaxSum(Node *root)
    {
        if(!root)return 0;
```

```
        if(m[root])return m[root];
        int inc=root->data;
        if(root->left)
        {
            inc+=getMaxSum(root->left->left);
            inc+=getMaxSum(root->left->right);
        }
        if(root->right)
        {
            inc+=getMaxSum(root->right->left);
            inc+=getMaxSum(root->right->right);
        }
        int ex=getMaxSum(root->right)+getMaxSum(root->left);

        return m[root]=max(inc,ex);
    }
};
```

29.Lowest Common Ancestor in a Binary Tree :

Here there are 4 cases:

1.root->data==n1 or root->data==n2.
2.if n1 is left subtree and n2 right subtree.
3.both are in same subtree.
4.not found

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

```
    if(!root)return NULL;
    if(root->data==n1 || root->data==n2)return root;

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

    if(!left)return right;
    if(!right)return left;

    return root;

  }
```

30.Print all k-sum paths in a binary tree:

```
void ksum_paths(Node* root,vector<Node*>& path,int k)
{
if(!root)return;
path.push_back(root);
ksum_paths(root->left,path,k);
ksum_paths(root->right,path,k);
Int f=0;
for(int j=path.size()-1;j>=0;j–
{
f+=path[j];
if(f==k)
     {
          for(i=j;i<path.size();i++)cout<<path[i]<<” ”;
          cout<<endl;
```

```
    }
}
path.pop();
return;
}
```

31.Find distance between 2 nodes in a Binary tree:

```
class Solution{
  public:
  Node* LCA(Node* root,int a,int b)
  {
    if(!root)return NULL;
    if(root->data==a || root->data==b)return root;

    Node* left=LCA(root->left,a,b);
    Node* right=LCA(root->right,a,b);

    if(!left)return right;
    if(!right)return left;

    return root;
  }
  int height(Node* root,int a)
  {
    if(!root)return 0;
```

```
        if(root->data==a)return 1;

        int l=height(root->left,a);
        int r=height(root->right,a);

        if(!l && !r)return 0;
        return l+r+1;
    }

int findDist(Node* root, int a, int b) {
        if(!root)return 0;
        Node* lca=LCA(root,a,b);
        int x=height(lca,a);
        int y=height(lca,b);

        return x+y-2;
    }
};
```

32.Tree Isomorphism Problem:

```
bool isIsomorphic(Node *root1,Node *root2)
    {
    if(!root1 && !root2)return true;
    if(!root1 or !root2)return false;
    if(root1->data!=root2->data)return false;
    bool
a=(isIsomorphic(root1->left,root2->left)&&isIsomorphic(root1
->right,root2->right));//if both tree r same
```

```
    bool
b=(isIsomorphic(root1->left,root2->right)&&isIsomorphic(root1->right,root2->left));//if trees r swapped position
    return a or b;
    }
```

33.Construct Tree from preorder traversal:

Construct a binary tree of size N using two given arrays pre[] and preLN[]. Array pre[] represents preorder traversal of a binary tree. Array preLN[] has only two possible values L and N. The value L in preLN[] indicates that the corresponding node in Binary Tree is a leaf node and value N indicates that the corresponding node is a non-leaf node.

```
Node* construct_tree(int n,int &i,int pre[],char preLN[])
{
   if(i>=n)return NULL;
   Node* node=new Node(pre[i]);
   if(preLN[i]=='N')
   {
      node->left=construct_tree(n,++i,pre,preLN);
      node->right=construct_tree(n,++i,pre,preLN);
   }
   return node;
}
struct Node *constructTree(int n, int pre[], char preLN[])
{
   int i=0;
   return construct_tree(n,i,pre,preLN);
}
```