TREES

1. INORDER TRAVERSAL

Time: O(n) Aux. Space: O(h)

```
void inorder(Node *root){
    if(root!=NULL){
        inorder(root->left);
        cout<<root->key<<" ";
        inorder(root->right);
    }
}
```

2. PREORDER TRAVERSAL

Time: O(n) Aux. Space: O(h)

```
void preorder(Node *root){
    if(root!=NULL){
        cout<<root->key<<" ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

3. POSTORDER TRAVERSAL

Time: O(n) Aux. Space: O(h)

```
void postorder(Node *root){
    if(root!=NULL){
        postorder(root->left);
        postorder(root->right);
        cout<<root->key<<" ";
    }
```

```
}
```

4. HEIGHT

Time: O(n) Aux. Space: O(h)

```
int height(Node *root)
{
    if(root==NULL)
    {
        return 0;
    }
    return 1+max(height(root->left),height(root->right));
}
```

5. PRINT NODES AT K DISTANCE

Time: O(n) Aux. Space: O(h)

```
void kd(Node *root, int k)
{
    if(root==NULL)
    {
        return;
    }
    if(k==0)
    {
        cout<<root->key<<" ";
    }
    else
    {
        kd(root->left,k-1);
        kd(root->right,k-1);
    }
}
```

6. LEVEL ORDER TRAVERSAL Time:O(n) Aux. Space:O(n)

```
void level(Node *root)
{
    if(root==NULL)
    {
        return;
    }
    queue<Node *>q;
    q.push(root);
    while(!q.empty())
    {
        Node *curr=q.front();
        q.pop();
        cout<<curr->key<<" ";
        if(curr->left)
        {
            q.push(curr->left);
        }
        if(curr->right)
        {
            q.push(curr->right);
        }
    }
}
```

7. LEVEL ORDER LINE BY LINE Time:O(n) Aux. Space:O(n)

```
void level(Node *root)
{
    if(root==NULL)
    {
```

```
        return;
    }
    queue<Node *>q;
    q.push(root);
    while(!q.empty())
    {
        int count=q.size();
        for(int i=0;i<count;i++)
        {
            Node *curr=q.front();
            q.pop();
            cout<<curr->key<<" ";
            if(curr->left)
            {
                q.push(curr->left);
            }
            if(curr->right)
            {
                q.push(curr->right);
            }
        }
        cout<<endl;
    }
}
```

8. **SIZE OF BT** Time: O(n) Aux. Space: O(h)

```
int size(Node *root)
{
    if(root==NULL)
    {
        return 0;
    }
    return 1+size(root->left)+size(root->right);
}
```

9. MAXIMUM IN BT

Time: O(n) Aux. Space: O(h)

```
int mb(Node *root)
{
    if(root==NULL)
    {
        return INT_MIN;
    }
    return max(root->key,max(mb(root->left),mb(root->right)));
}
```

10. PRINT LEFT VIEW

Time: O(n) Aux. Space: O(n)

```
void left(Node *root)
{
    if(root==NULL)
    {
        return;
    }
    queue<Node *>q;
    q.push(root);
    while(!q.empty())
    {
        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)
            {
                q.push(curr->left);
            }
            if(curr->right)
            {
                q.push(curr->right);
            }
```

```
        }
    }
}
```

11. CHILDREN SUM PROPERTY Time: O(n) Aux. Space: O(h)

```
bool children(Node *root)
{
    if(root==NULL || (root->left==NULL && root->right==NULL))
    {
        return true;
    }
    int sum=0;
    if(root->left)
    {
        sum+=root->left->key;
    }
    if(root->right)
    {
        sum+=root->right->key;
    }
    return (sum==root->key && children(root->left) && children(root->right));
}
```

12. CHECK FOR BALANCED TREE

Time: O(n2) Aux. Space: O(h)

```
bool isBal(node *root)
{
    if(root==NULL)
    {
        return true;
    }
    int lh=height(root->left);
    int rh=height(root->right);
    return (abs(lh-rh)<=1 && isBal(root->left) && isBal(root->right));
}
```

13. MAXIMUM WIDTH OF BINARY TREE

Time: O(n) Aux. Space:O(n)

```
int mw(Node *root)
{
    if(root==NULL)
    {
        return INT_MIN;
    }
    int m=INT_MIN;
    queue<Node *>q;
    q.push(root);
    while(!q.empty())
    {
        int count=q.size();
        for(int i=0;i<count;i++)
        {
            Node *curr=q.front();
            q.pop();
            if(curr->left)
            {
                q.push(curr->left);
            }
            if(curr->right)
            {
                q.push(curr->right);
            }
        }
        m=max(m,count);
    }
    return m;
}
```

14. BINARY TREE TO DOUBLY LINKED LIST

Time: O(n) Aux. Space: O(h)

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

15. BINARY TREE FROM INORDER AND PREORDER

Time: O(n2) Aux. Space: O(h)

```
    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=0;
    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;
}
```

16. SPIRAL TRAVERSAL

Time: O(n) Aux. Space:O(n)

```
    void spiral(Node *root)
{
    if(root==NULL)
    {
        return;
    }
    stack<Node *>s1,s2;
    s1.push(root);
    while(s1.empty()==false || s2.empty()==false)
    {
        while(s1.empty()==false)
        {
            Node *curr=s1.top();
            s1.pop();
            cout<<curr->key<<" ";
            if(curr->left)
```

```
            {
                s2.push(curr->left);
            }
            if(curr->right)
            {
                s2.push(curr->right);
            }
        }
        while(s2.empty()==false)
        {
            Node *curr=s2.top();
            s2.pop();
            cout<<curr->key<<" ";
            if(curr->right)
            {
                s1.push(curr->right);
            }
            if(curr->left)
            {
                s1.push(curr->left);
            }
        }
    }
}
```

17. DIAMETER OF A BINARY TREE

Time: O(n2) Aux. Space:O(h)

```
int dia(Node *root)
{
    if(root==NULL)
    {
        return 0;
```

```
    }
    int lh=height(root->left);
    int rh=height(root->right);
    int d1=(lh+rh+1);
    int d2=dia(root->left);
    int d3=dia(root->right);
    return max(d1,max(d2,d3));
}
```

18.

19. LCA

Time:O(n) Aux. Space:O(h)

```
    Node *lca(Node *root,int n1,int n2)
{
    if(root==NULL)
    {
        return NULL;
    }
    if(n1==root->data || n2==root->data)
    {
        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;
    }
}
```

20. BURN A BINARY TREE FROM A LEAF

Time:O(n) Aux. Space:O(h)

```
 int res=0;
int burn(Node *root, int leaf,int &dist)
{
    if(root==NULL)
    {
        return 0;
    }
    if(root->data==leaf)
    {
        dist=0;
        return 1;
    }
    int ldist=-1,rdist=-1;
    int lh=burn(root->left,leaf,ldist);
    int rh=burn(root->right,leaf,rdist);
    if(ldist!=-1)
    {
        dist=ldist+1;
        res=max(res,dist+rh);
    }
    else if(rdist!=-1)
    {
        dist=rdist+1;
        res=max(res,dist+lh);
```

```
    }
    return max(lh,rh)+1;
}
```

21. COUNT NODES IN A COMPLETE BINARY TREE

Time:O(logn * logn) Aux.Space:O(h)

```
 int count(Node *root)
{
    if(root==NULL)
    {
        return 0;
    }
    Node *curr=root;
    int lh=0;
    while(curr)
    {
        lh++;
        curr=curr->left;
    }
    curr=root;
    int rh=0;
    while(curr)
    {
        rh++;
        curr=curr->right;
    }
    if(lh==rh)
    {
        return pow(2,lh)-1;
    }
    return count(root->left)+count(root->right)+1;
}
```

22. SERIALIZE AND DESERIALIZE A BINARY TREE

```
    const int EMPTY=-1;
void serialize(Node *root,vector<int>&v)
{
   if(root==NULL)
   {
      v.push_back(EMPTY);
      return;
   }
   v.push_back(root->data);
   serialize(root->left,v);
   serialize(root->right,v);
}
```

Serialization -> Time:O(n) Aux. Space:O(h)

```
int index=0;
Node* deserialize(vector<int>&v)
{
   if(index==v.size())
   {
      return NULL;
   }
   int val=v[index];
   index++;
   if(val==EMPTY)
   {
      return NULL;
   }
   Node *root=new Node(val);
   root->left=deserialize(v);
   root->right=deserialize(v);
```

```
    return root;
}
```
Deserialization-> Time:O(n) Aux. Space:O(h)

23. ITERATIVE PREORDER

Time: O(n) Aux. Space:O(n)

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

24. ITERATIVE INORDER

Time:O(n) Aux. Space:O(h)

```
 void inorder(Node *root)
{
   if(root==NULL)
   {
      return;
   }
   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->data<<" ";
      curr=curr->right;
   }
}
```

25. DETERMINE IF TWO TREES ARE IDENTICAL

```
   bool isIdentical(Node *r1, Node *r2)
   {
      //Your Code here
      if(r1==NULL && r2==NULL)
```

```
{
    return true;
}
else if(r1==NULL || r2==NULL)
{
    return false;
}
else
{
    queue<Node *>q1,q2;
    q1.push(r1);
    q2.push(r2);
    while(!q1.empty() || !q2.empty())
    {
        Node *curr1=q1.front();
        Node *curr2=q2.front();
        q1.pop();
        q2.pop();
        if(curr1->data!=curr2->data)
        {
            return false;
        }
        if(curr1->left)
        {
            q1.push(curr1->left);
        }
        if(curr1->right)
        {
            q1.push(curr1->right);
        }
        if(curr2->left)
        {
            q2.push(curr2->left);
        }
        if(curr2->right)
```

```
            {
                q2.push(curr2->right);
            }
        }
        return true;
    }
}
```

26. VERTICAL WIDTH OF A BINARY TREE

```
void c(Node *root,unordered_map<int,int>&mp,int hd)
{
    if(!root)
    {
        return;
    }
    mp[hd]+=root->data;
    c(root->left,mp,hd-1);
    c(root->right,mp,hd+1);
}

//Function to find the vertical width of a Binary Tree.
int verticalWidth(Node* root)
{
    // Code here
    unordered_map<int,int>mp;
    c(root,mp,0);
    return mp.size();
}
```

27. MIRROR TREE

```
void mirror(Node* node) {
        // code here
        if(node!=NULL)
```

```
        {
            mirror(node->left);
            mirror(node->right);
            swap(node->left,node->right);
        }
    }
```

28. CHECK IF S IS A SUBTREE OF T

```
bool same(Node *a,Node *b)
    {
        if(a==NULL && b==NULL)
        {
            return true;
        }
        else if(a==NULL || b==NULL)
        {
            return false;
        }
        return a->data==b->data && same(a->left,b->left) &&
same(a->right,b->right);
    }
    bool isSubTree(Node* T, Node* S)
    {
        // Your code here
        if(T==NULL && S==NULL)
        {
            return true;
        }
        else if(T==NULL)
        {
            return false;
        }
        return same(T,S) || isSubTree(T->left,S) || isSubTree(T->right,S);
    }
```

29.