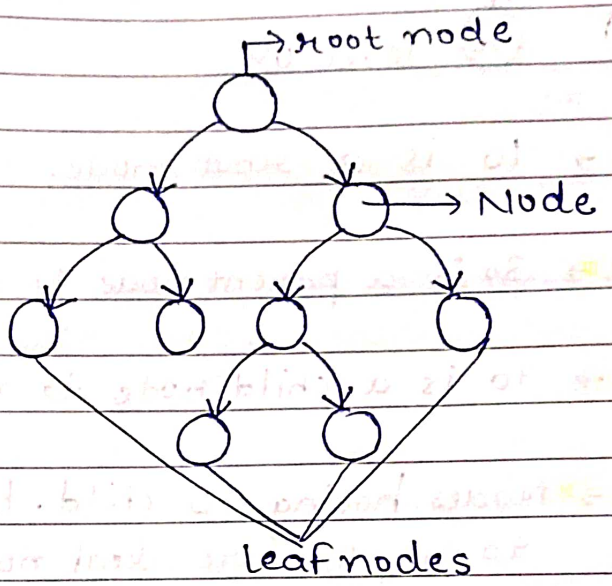
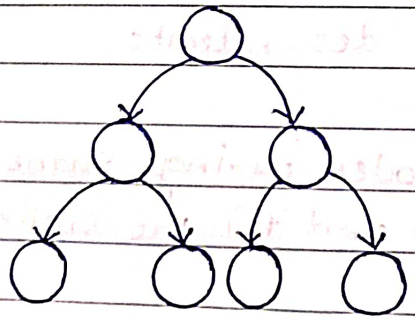
30/11/2023
Thursday

TREES -

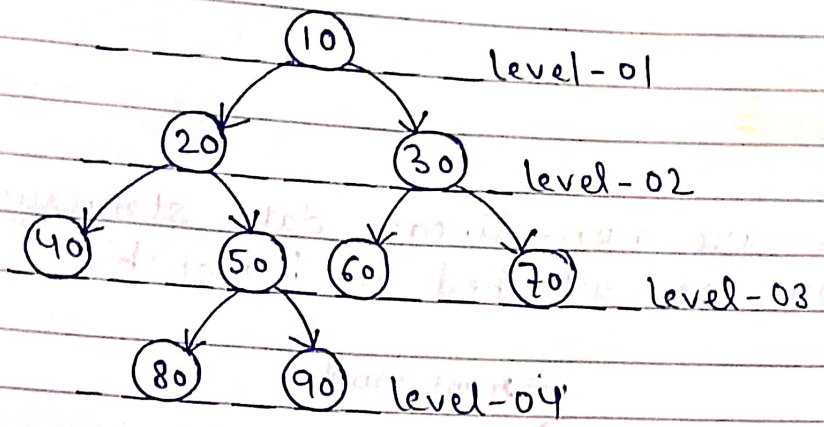
Trees are non-linear data structure in which nodes are attached in hierarchical order.



Binary Tree - Tree having atmost two child nodes



```
class treeNode {
    int data;
    Node* left;
    Node* right;
}
```



Root node → 10 is a root node.

Parent node → 30 is a parent node to node 70.

Child node → 70 is a child node to node 30.

leaf nodes → Nodes having 0 child. Here, 40, 80, 90, 60, 70 are leaf nodes.

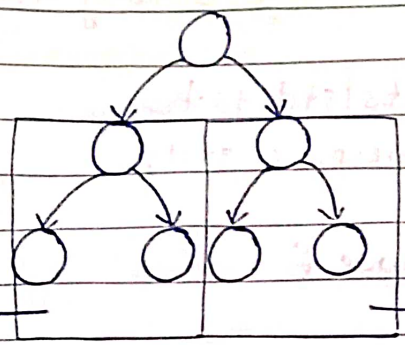
Ancestor nodes → For node 90, (50, 20, 10) are ancestors.

Descendent nodes → for node 20, (50, 90) are descendents.

Sibling nodes → Nodes having same parent. (60 and 70) are siblings.

Neighbour nodes → Nodes having different parents. (50 and 60) are neighbour nodes. They are at same level.

Subtree –



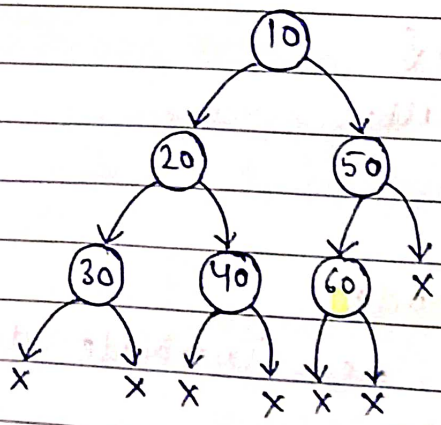
* **Implement binary tree –**

1 node create kr do, rest nodes recursion create kr dega.

Step 1: Create root node.
Step 2: Make a recursive call for left subtree.
Step 3: Make a recursive call for right subtree.

→ If koi valid data milega, then create node.
If koi invalid data milega, then return NULL.

I/P → 10 20 30 -1 -1 40 -1 -1 50 60 -1 -1 -1



Code for creating binary tree -

```
#include <bits/stdc++.h>
using namespace std;

class TreeNode {
 public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode (int val) {
      this->data = val;
      this->left = NULL;
      this->right = NULL;
    }
};

TreeNode * createTree () {

    int data;
    cin >> data;

    if (data == -1) {
        return NULL;
    }

    // create root node
    TreeNode* root = new TreeNode (data);

    // create left node
    root->left = createTree ( );
```

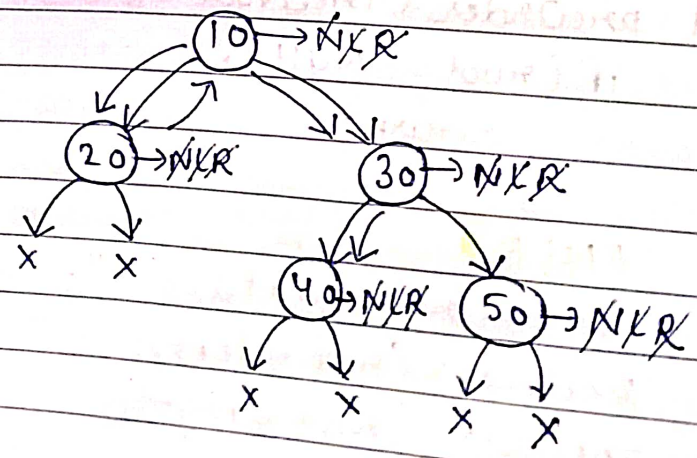
```
//create right node
root->right = createTree();

  return root;
}

int main(){
    TreeNode* root = createTree();
}
```

* 3 types of traversals -

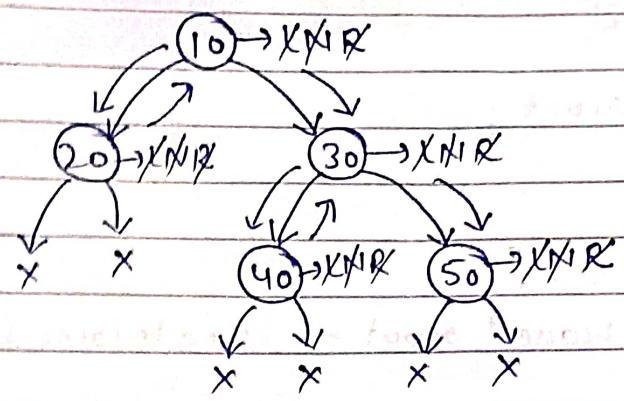
PreOrder = NLR
InOrder = LNR
PostOrder = LRN

N → stands for current node.
L → stands for left part.
R → stands for right part.

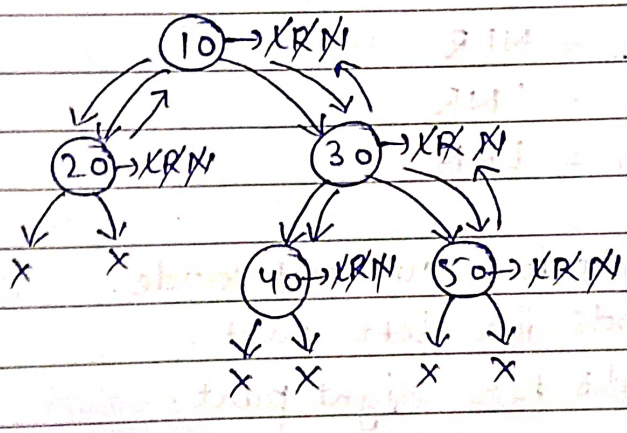
Pre-Order → 10 20 30 40 50



In-Order → 20 10 40 30 50



Post-order → 20 40 50 30 10



Code - PreOrder →

```
void preOrder (TreeNode * root) {
    if (root == NULL) {
        return;
    }
    //NLR
    cout << root->data << " ";
    preOrder (root->left);
    preOrder (root->right);
}
```

InOrder -

Code -

```
void inOrder (TreeNode * root){
    if (root == NULL){
        return;
    }
    // LNR
    inOrder (root -> left);
    cout << root -> data << " ";
    inOrder (root -> right);
}
```

PostOrder -

Code -

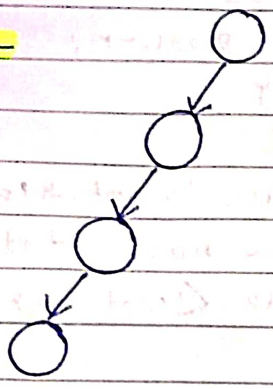
```
void postOrder (TreeNode *root){
    if(root == NULL){
        return;
    }
    // LRN
    postOrder (root -> left);
    postOrder (root -> right);
    cout << root -> data << " ";
}
```

→ **Time complexity** of preOrder, InOrder and post order is O(n) i.e. tree ki har node ko ek baar traverse krna padega.

→ **Space complexity** - In general, S.C. of these traversals is O(h) i.e. height of binary tree. In worst case scenerio, the tree might be skewed either to the left or right. So, in this case, S.C. will be O(n) where n is the nodes present in skewed tree.

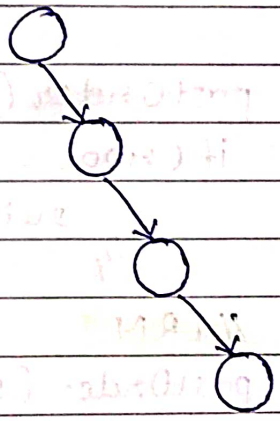
* Skew Tree –

Left skew tree –



All the nodes have left child only.

Right skew tree –



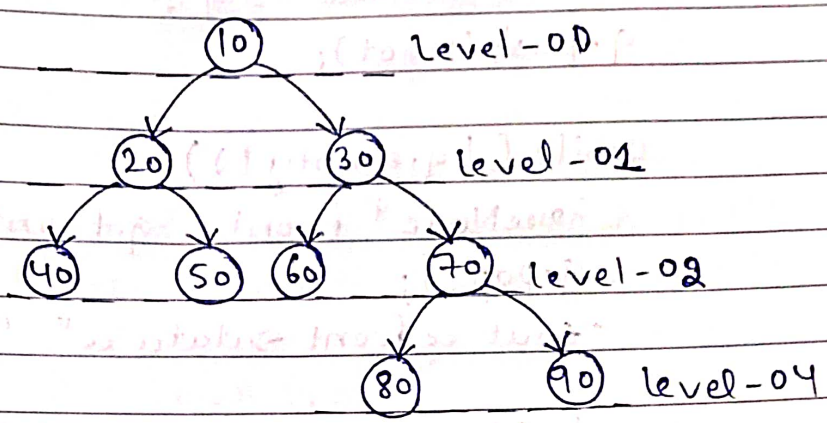
All the nodes have right child only.

Note – Morris traversal takes the constant space. Will deal with it later.

Note – Jb space complexity calculate krni ho to function call stack bna lo and that instant of time Jb maximum space utilise ho rha ho is that space will be considered for S.C.

__/__/__

* **Level Order Traversal -**



10 20 30 40 50 60 70 80 90

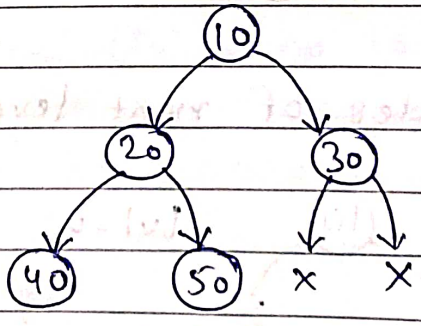
Steps -

Step 1: Initially push root node in queue.

Step 2: Now, traversal starts until queue is not empty.

- ↳ fetch front of queue.
- ↳ push left of front.
- ↳ push right of front.

let for -



10 20 30 40 50

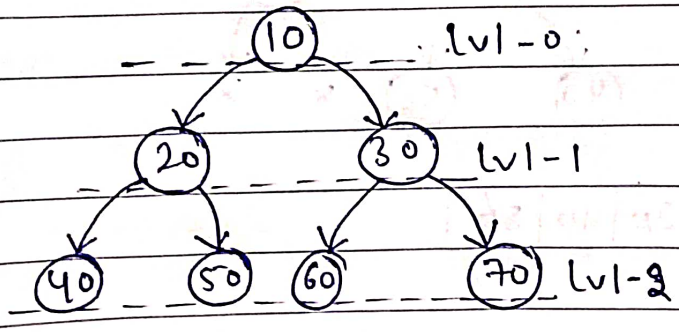
1ϕ	2ϕ	3ϕ	4ϕ	5ϕ

Code -

```
void levelOrderTraversal(TreeNode* root){

    queue<TreeNode*> q;
    q.push(root);

    while(!q.empty()){
        TreeNode* front = q.front();
        q.pop();
        cout << front->data << " ";

        if(front->left != NULL){
            q.push(front->left);
        }
        if(front->right != NULL){
            q.push(front->right);
        }
    }
}
```

* **Level Order Traversal –**

Print the nodes of next level in next line.



o/p – 10
20 30
40 50 60 70

Marker koi bhi exceptional data ho skta hai jo exceptional case handle krega like if data = -1.

Using NULL as a marker in queue jo btayega ab next line me jana hai jb level khtm ho jaye.

Steps -

Step 1: Initially push root node and NULL in queue.

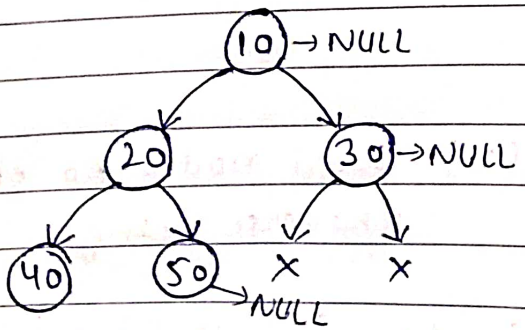
Step 2: Now, traversal starts until queue is not empty.

→ fetch front

→ Front = NULL
- Move to next line.
- If queue is not empty, push NULL in queue.

→ Front is valid node
- Push left of front.
- Push right of front.

Note - Jb ek level finish ho jayega, tb uske next level mei jitni bhi nodes hai vo sb queue mei present hongi always null milne se pehle.



10 20 30
40 50

10	NULL	20	30	NULL	40	50	NULL

Code -

```
void levelTraversal(TreeNode* root){
    queue<TreeNode*> q;
    q.push(root);
    q.push(NULL);

    while(!q.empty()){
        TreeNode* front = q.front();
        q.pop();

        if(front==NULL){
            cout<<endl;
            if(!q.empty()){
                q.push(NULL);
            }
        }
        else{
            cout<<front->data<<" ";
            if(front->left!=NULL){
                q.push(front->left);
            }
            if(front->right!=NULL){
                q.push(front->right);
            }
        }
    }
}
```

Ye wali condition isliye lgayi hai coz jb last level reach kr jayenge or last wala NULL front hoga to fir se NULL push ho jayega and queue kbhi empty nhi hogi or infinite loop mei fss jayegi

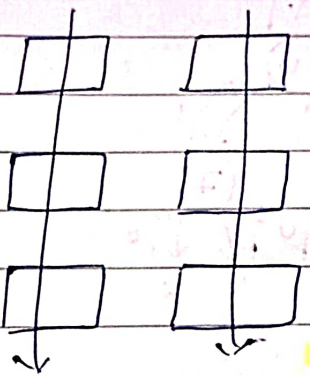
Time complexity - O(n) Saari nodes ko ek baar traverse krenge.

Space complexity - O(b) where b is the maximum no. of nodes in a level i.e. jis level ki maximum breadth hogi.

Breadth-first Search -

Traversing the tree row-wise i.e. left to right.

Depth-first Search -



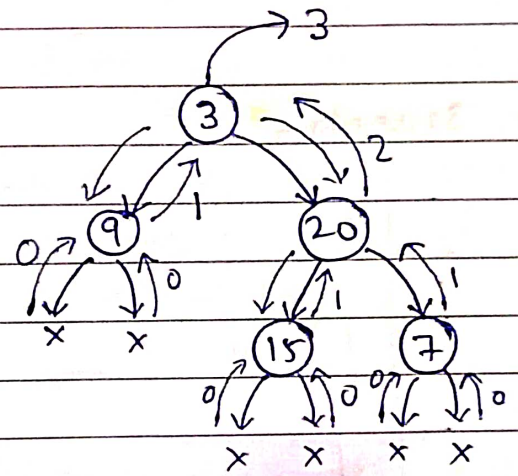
Traversing the tree column-wise i.e. up to down.

* **Height of binary Tree (Leetcode -104)**

A binary tree's maximum depth is the no. of nodes along the longest path from the root node down to farthest leaf node.

I/P - root = [3, 9, 20, null, null, 15, 7]

O/P - 3



Recursive Approach -

Code -

```
class Solution {
  public:
    int maxDepth(TreeNode * root){

        if(root == NULL){
            return 0;
        }
    int leftHeight = maxDepth(root->left);
    int rightHeight = maxDepth(root->right);

    return max(leftHeight, rightHeight)+1;
}
```

Note - Whether it is tree or linked list, always check if it is empty and code karte time ye wala case yaad rakhna hai.

Using level order traversal - Jitne level honge tree ke utni hi height hogi.

Code -

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == NULL) return 0;

        queue<TreeNode*> q;
        q.push(root);
        q.push(NULL);
        int count = 1;

        while (!q.empty()) {
            TreeNode* front = q.front();
            q.pop();

            if (front == NULL) {
                if (!q.empty()) {
                    q.push(NULL);
                    count++;
                }
            }
            else {
                if (front->left != NULL)
                    q.push(front->left);
                if (front->right != NULL)
                    q.push(front->right);
            }
        }
        return count;
    }
};
```

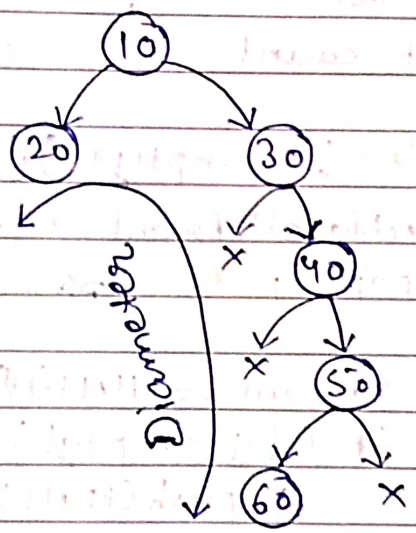
* Diameter of Binary Tree (leetcode-543)

Diameter of binary tree is the length of the longest path between any two nodes in a tree. The path may or may not pass through the root.

The length of path b|w two nodes is represented by number of edges b|w them.

I/P - [10, 20, 30, null, null, 40, null, 50, 60, null]

O/P - 5



3 possibilities -

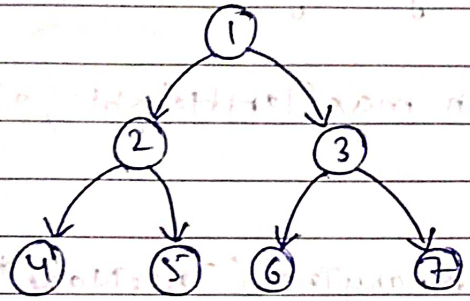
(i) Both nodes are in the left subtree.
(ii) Both nodes are in the right subtree.
(iii) One node is in left subtree and other in right subtree.

Code-

```
class Solution {
public:
    int maxDepth(TreeNode * root) {
        if (root == NULL) return 0;

        int leftHeight = maxDepth(root->left);
        int rightHeight = maxDepth(root->right);

        return max(leftHeight, rightHeight) + 1;
    }

int diameterOfBinaryTree(TreeNode * root) {

        if (root == NULL) return 0;

int option1 = diameterOfBinaryTree(root->left);
int option2 = diameterOfBinaryTree(root->right);
int option3 = maxDepth(root->left) + maxDepth(root->right);

return max(option1, max(option2, option3));
    }
};
```

* **Perfect Binary Tree -**

In which all leaf nodes are at the same level and all non-leaf nodes have 2 children. i.e. all leaf nodes are at maximum depth of tree.



* **Complete Binary Tree -**

In which all the levels are completely filled except the lowest one which is filled from left. i.e. the last leaf element might not have a right sibling.