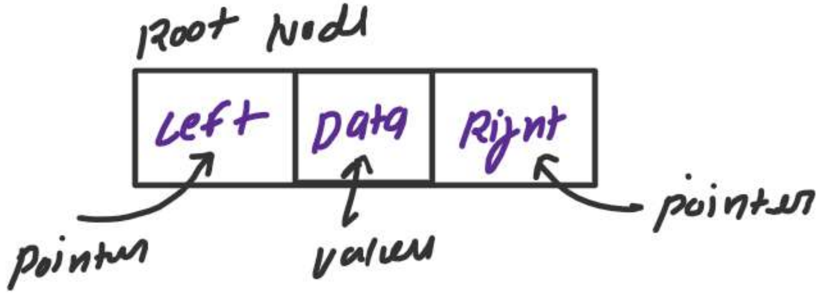
30/11/2023

BINARY TREE CLASS - 1
HOMEWORK

Types of Binary Tree

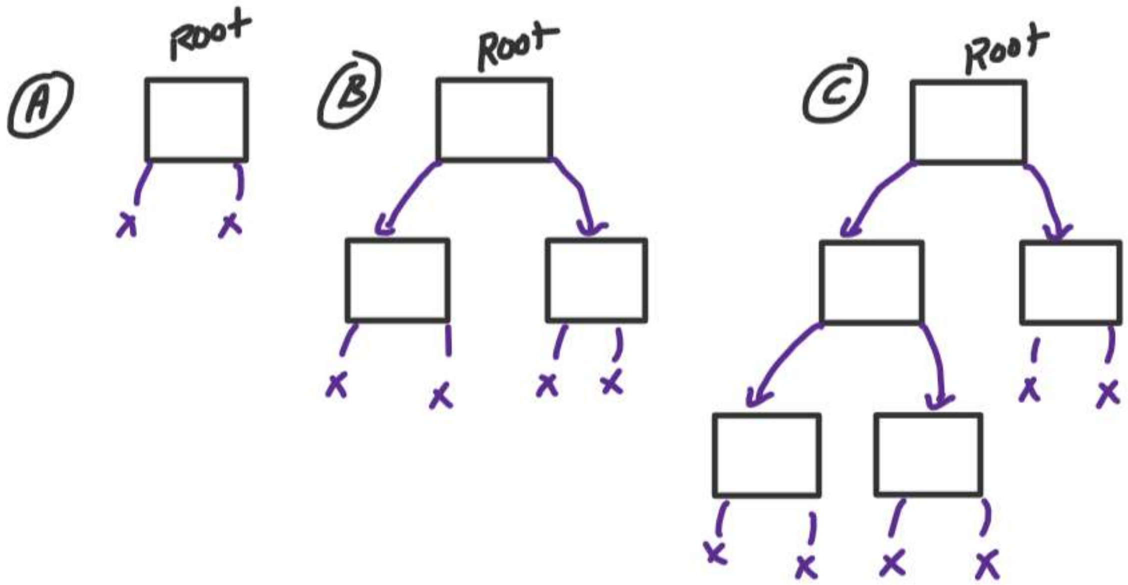
Types of Binary Tree based on the number of children:

1. *Full Binary Tree*
2. *Perfect Binary Tree*
3. *Complete Binary Tree*
4. *Degenerate or Pathological Tree*
5. *Skewed Binary Tree*
6. *Balanced Binary Tree*



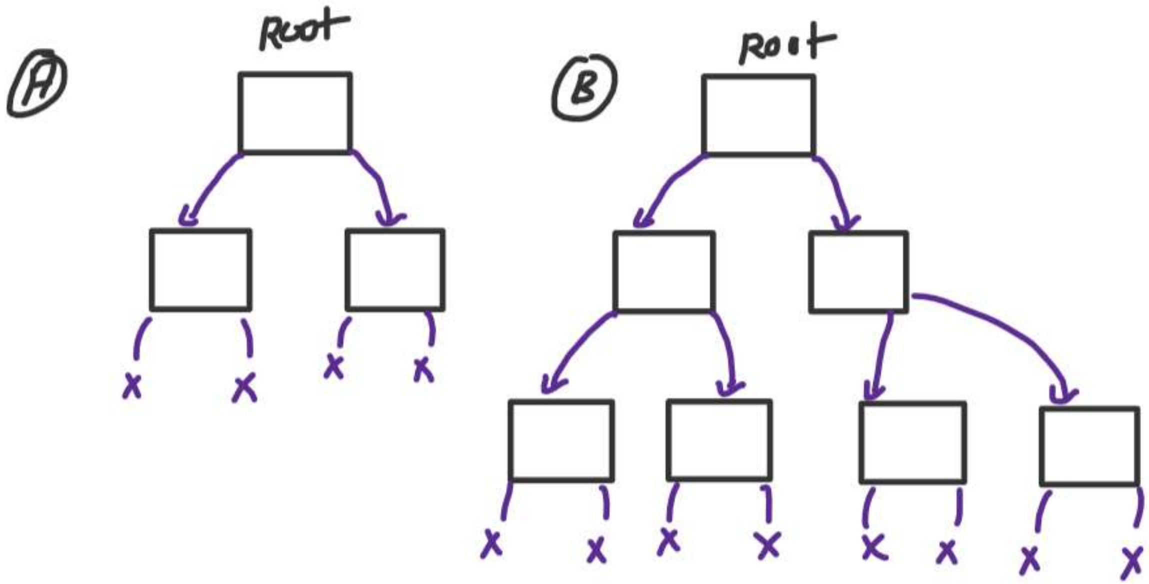
1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



2. ***Perfect Binary Tree***

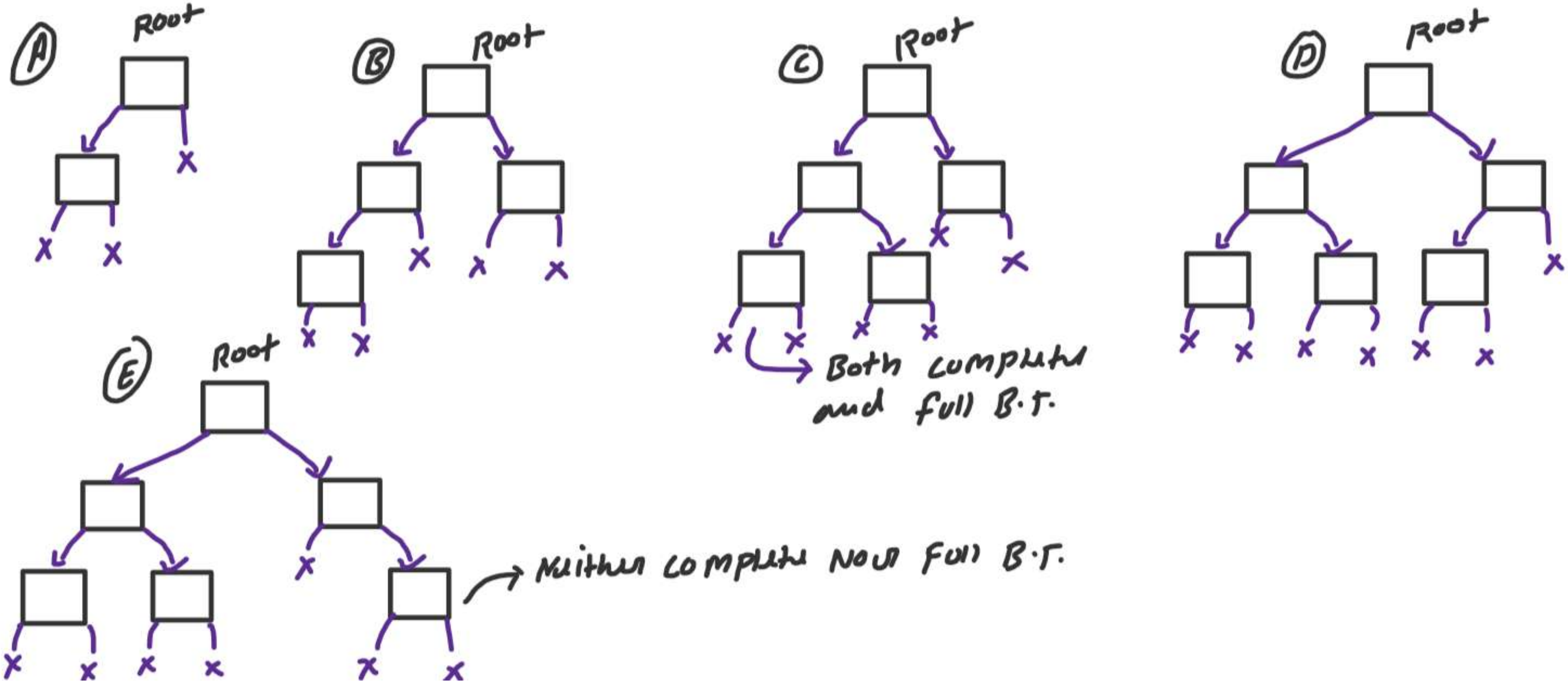
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



3. Complete Binary Tree

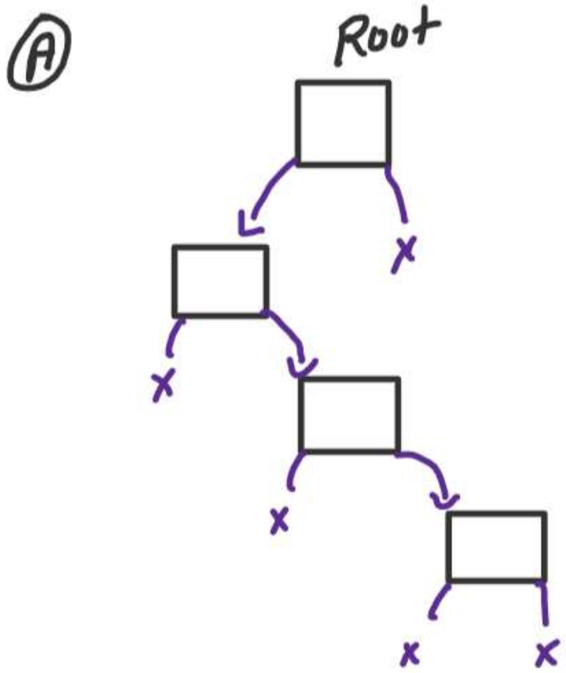
A complete binary tree is just like a full binary tree, but with two major differences

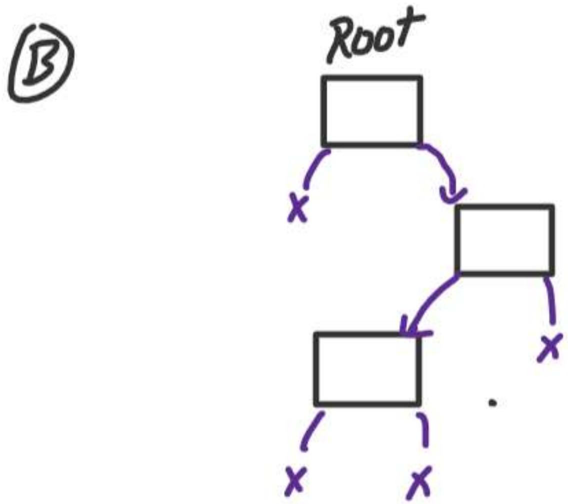
I. *Every level must be completely filled*
II. *All the leaf elements must lean towards the left.*
III. *The last leaf element might not have a right sibling*
i.e. a complete binary tree doesn't have to be a full binary tree.



4. Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a single child either left or right.

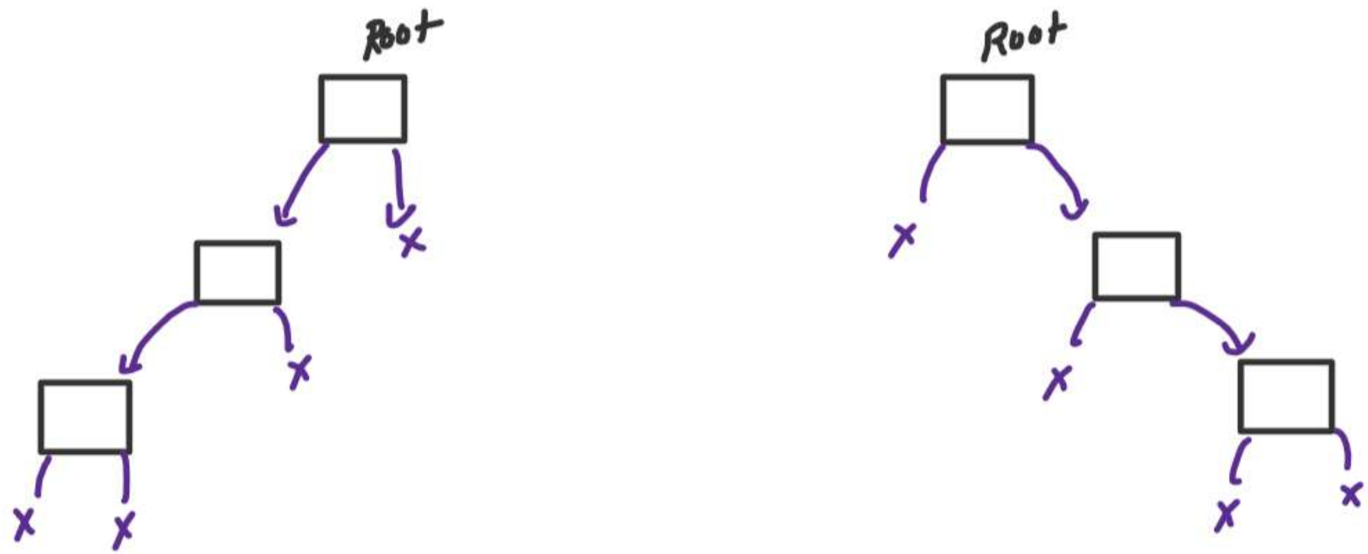




5. Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus,

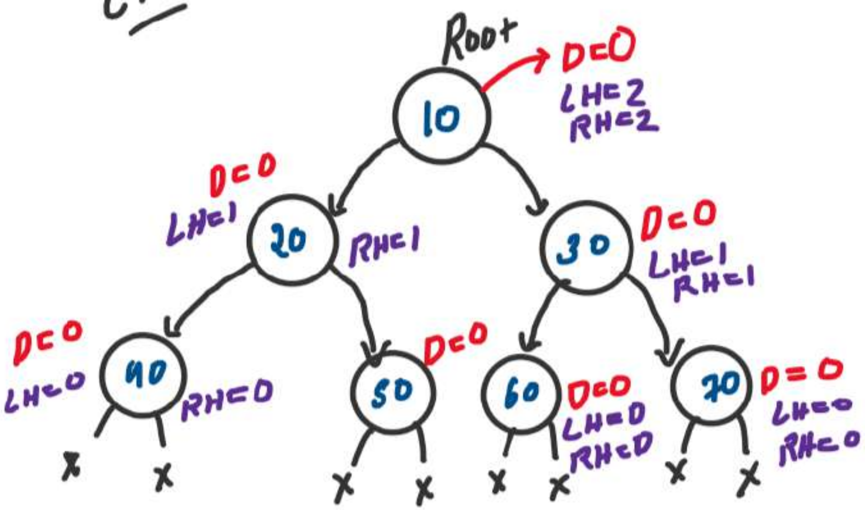
There are two types of skewed binary tree:
left-skewed binary tree *and* ***right-skewed binary tree***



6. Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.

Ex=1

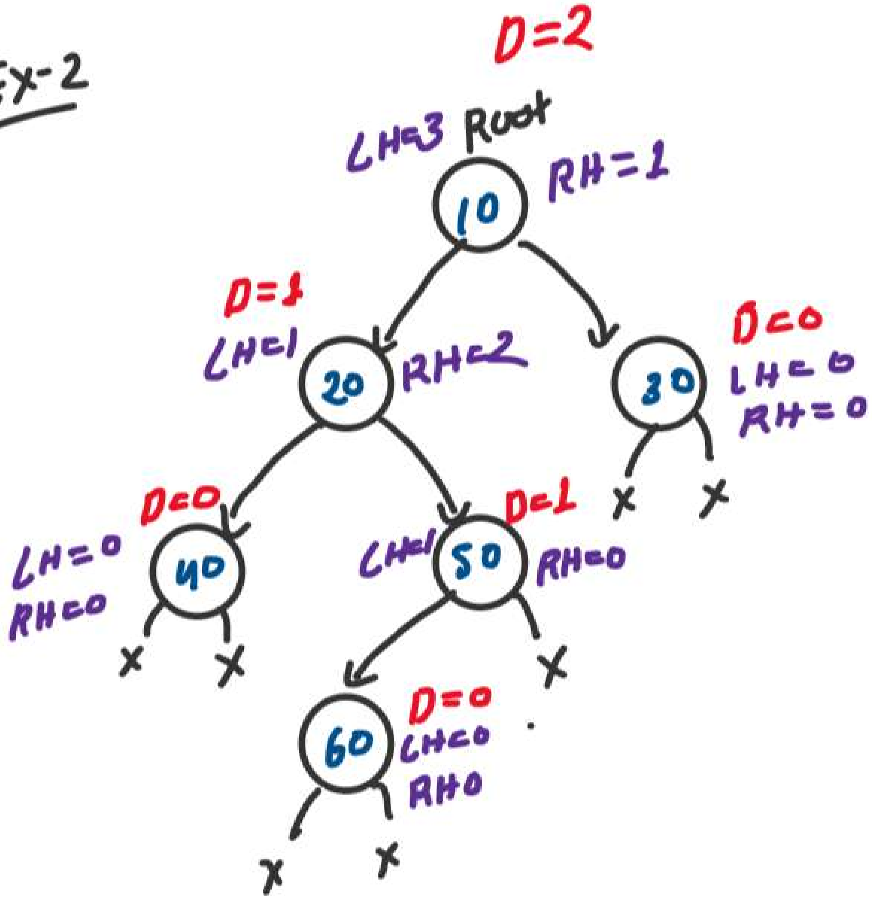


What is Balanced Binary Tree?

Left and Right Sub Tree ki height ka diff.^{n} at most 1 Hona chahiye.

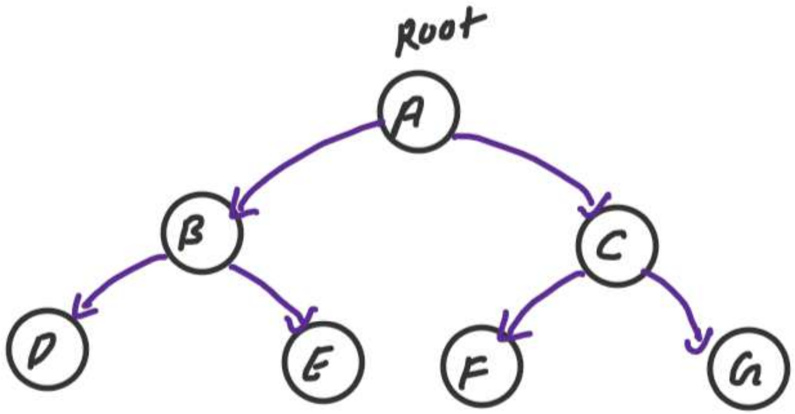
$$\boxed{L_H - R_H \in [0,1]}$$

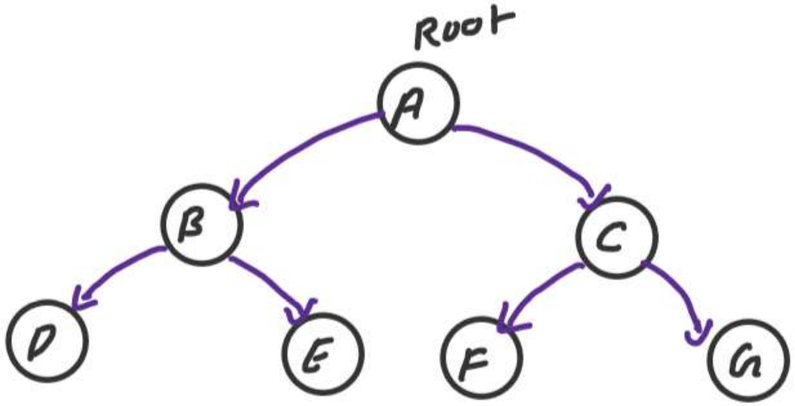
Ex-2



This is not a Balanced Binary TREE

BFS & DFS Algorithms of Binary Tree





BFS - Breath First Search

Level-Order Traversal

A B C D E F G

DFS - Depth First Search

Pre-Order Traversal (NLR)

A B D E C F G

In-Order Traversal (LNR)

D B E A F C G

Post-Order Traversal (LRN)

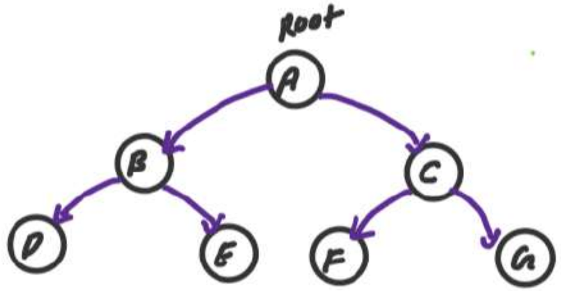
D E B F G C A

DFS Traversals:

1. Pre-order traversal (NLR)
2. In order traversal (LNR)
3. Post order traversal (LRN)

BFS Traversals:

4. Level order traversal



Why do we care?

There are many tree questions that can be solved using any of the above four traversals.

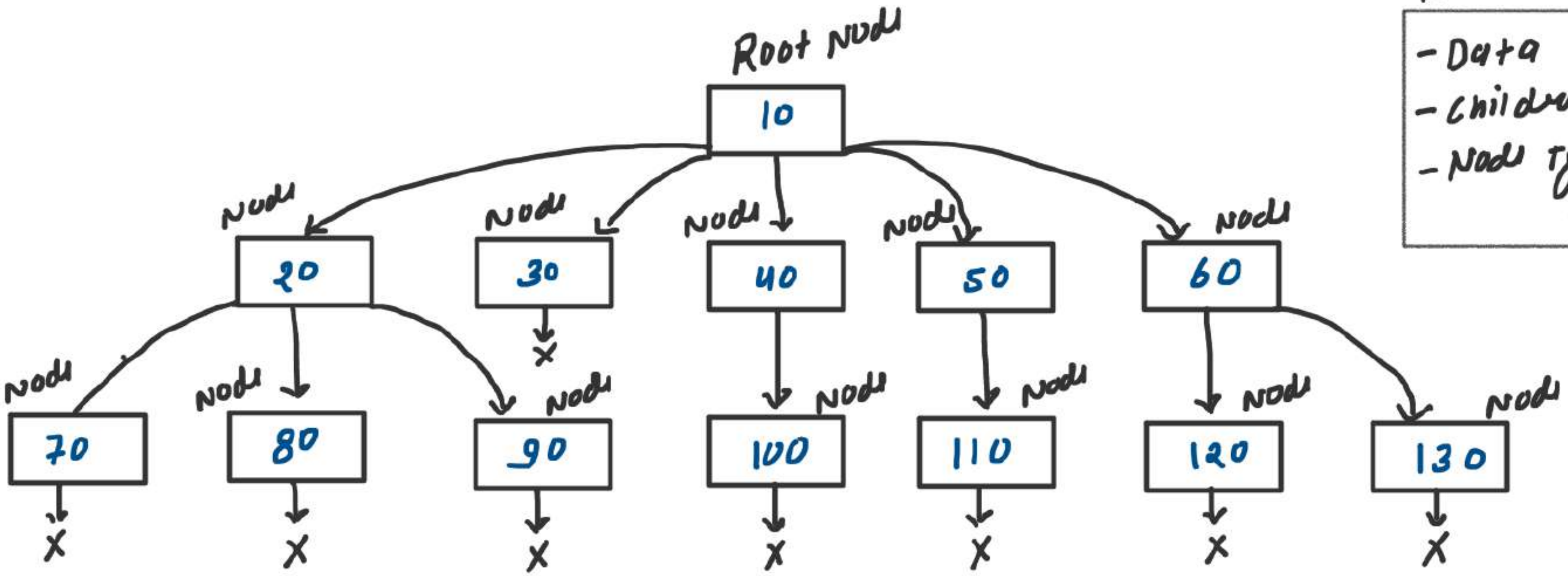
Using DFS Traversal:

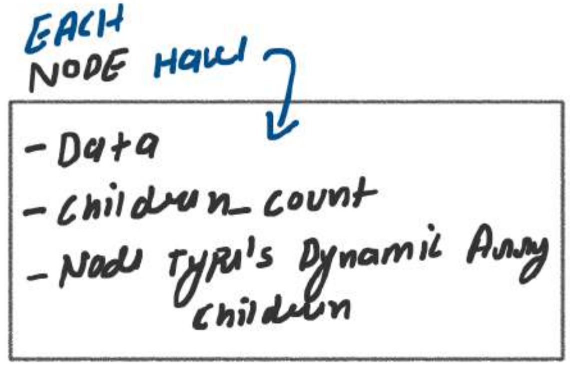
a.) Finding Size of Tree.
b.) Finding Height of Tree.
c.) Finding Max or Min element in a Tree.
d.) Diameter of Binary Tree.
e.) Print nodes at K distance.
f.) Checking if a binary tree is subtree of another binary tree.
g.) Ancestors of a given node.

Using BFS Traversal:

a.) Maximum Width of Binary Tree.
b.) Left View of Tree.
c.) Connect Nodes at same level.

Generic Tree

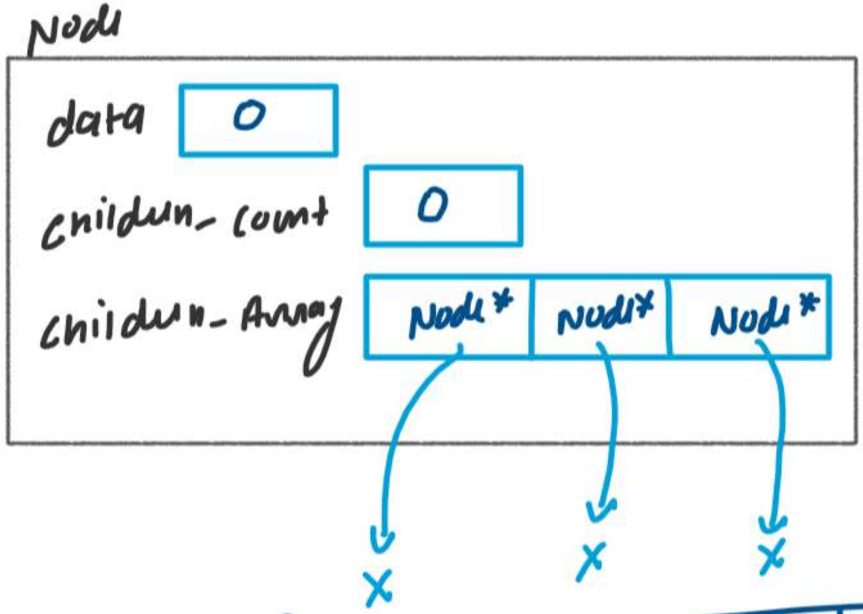




```
class Node {

    int data;

    int children_count
    Node* *children;

}
```

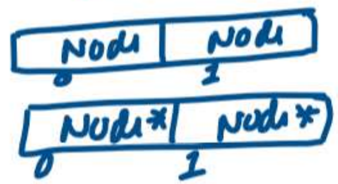
This is a Dynamic Array

Children store Node* Type Data



✗ Node* children = new Node[children_count]; ✗

✓ Node** children = new Node*[children_count]; ✓

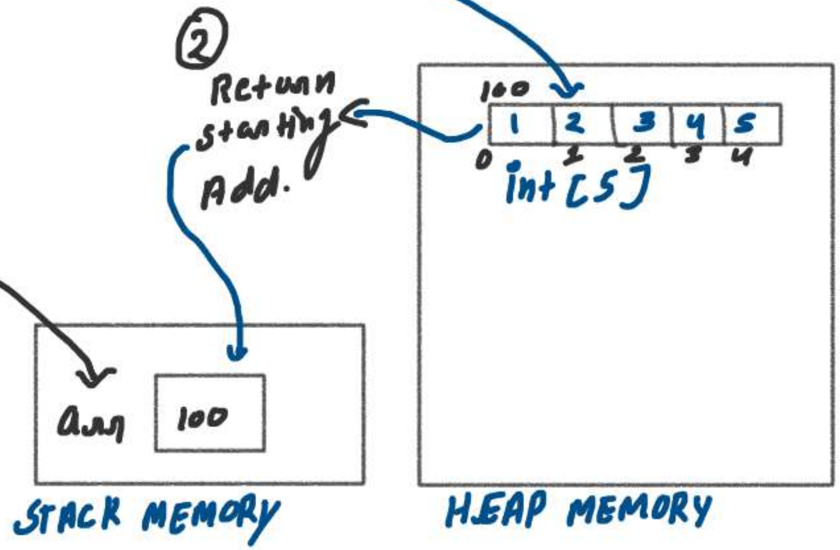


DYNAMIC MEMORY ALLOCATION

int *arr = new int[5];

① Dynamic Allocation

STATIC Allocation



deAllocate karna mat bhoolna

**

delete[] arr;

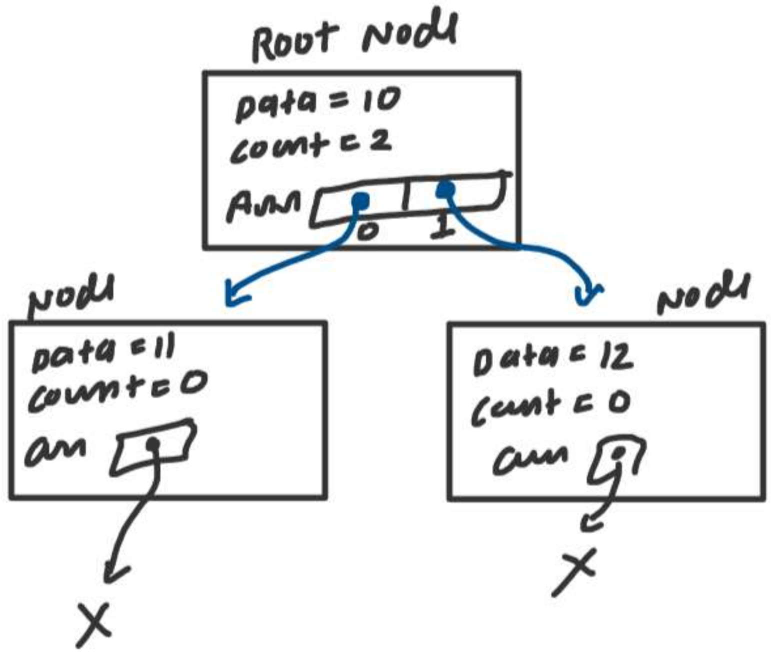
Example 1:

TREE INPUT

```
Enter root data: 10
Enter Children count for 10 node: 2
Enter root data: 11
Enter Children count for 11 node: 0
Enter root data: 12
Enter Children count for 12 node: 0
```

TREE OUTPUT

```
10
11 12
```



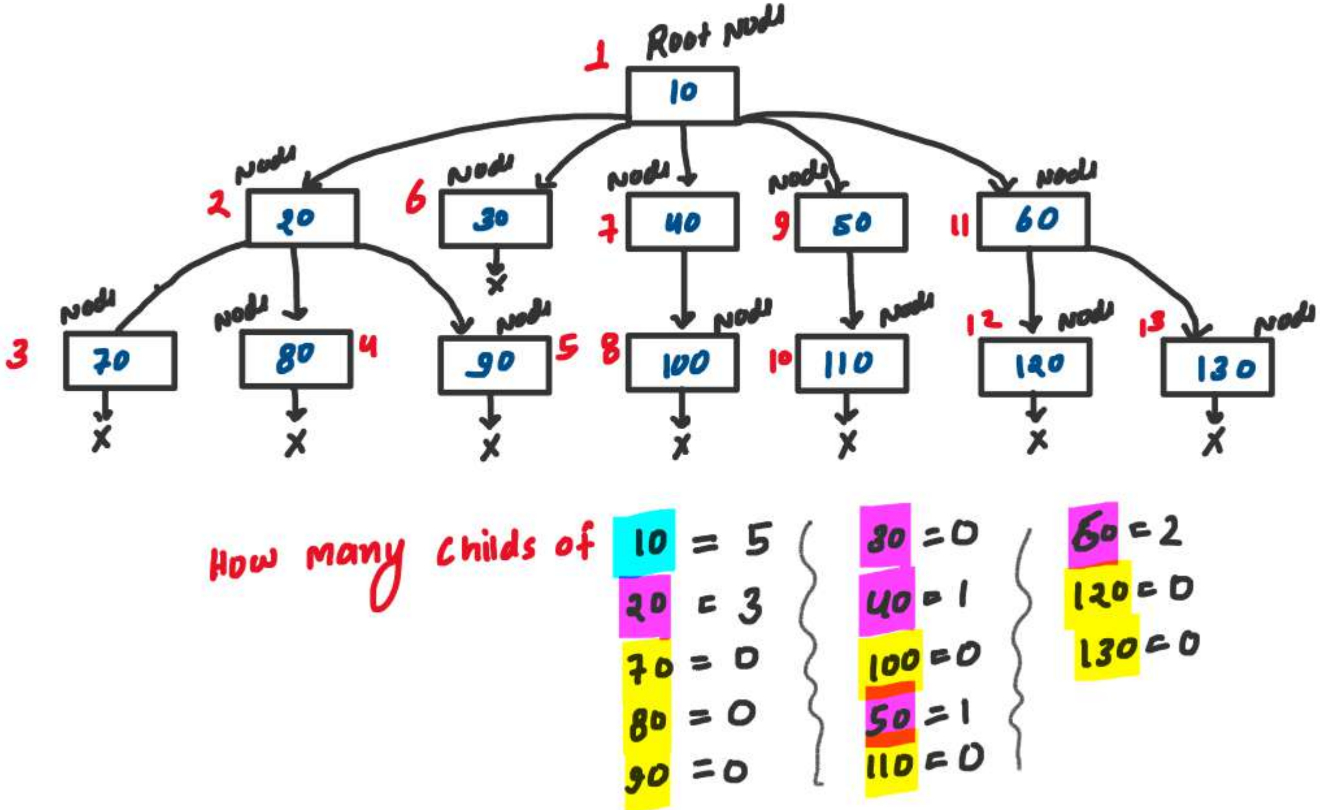
Example 2:

TREE INPUT

```
Enter root data: 10
Enter Children count for 10 node: 5
Enter root data: 20
Enter Children count for 20 node: 3
Enter root data: 70
Enter Children count for 70 node: 0
Enter root data: 80
Enter Children count for 80 node: 0
Enter root data: 90
Enter Children count for 90 node: 0
Enter root data: 30
Enter Children count for 30 node: 0
Enter root data: 40
Enter Children count for 40 node: 1
Enter root data: 100
Enter Children count for 100 node: 0
Enter root data: 50
Enter Children count for 50 node: 1
Enter root data: 110
Enter Children count for 110 node: 0
Enter root data: 60
Enter Children count for 60 node: 2
Enter root data: 120
Enter Children count for 120 node: 0
Enter root data: 130
Enter Children count for 130 node: 0
```

TREE OUTPUT

```
10
20 30 40 50 60
70 80 90 100 110 120 130
```



```cpp
// GENERIC TREE PROBLEM
#include <iostream>
#include<queue>
using namespace std;

class Node{
    public:
    int data;
    int children_count;
    Node** children;

    Node(int value) {
        this->data = value;
        this->children_count = 0;
        this->children = NULL;
    }
};

Node* takeInput(){
    int data, count;
    cout<<"Enter root data: ";
    cin>>data;
    cout<<"Enter Children count for "<<data<<" node: ";
    cin>>count;

    // Create Root Node
    Node* root = new Node(data);

    // Create Child Node of Root Node
    root->children_count = count;
    // Dynamic Array to store links to children
    root->children = new Node[count];
    for(int i=0;i<count;i++){
        root->children[i] = takeInput();
    }
    return root;
}

int main() {
    takeInput();
    return 0;
}
```

```cpp
// LEVEL ORDER TRAVERSAL OF A GENERIC TREE PROBLEM
#include <iostream>
#include<queue>
using namespace std;

class Node{
    ...
};

Node* takeInput(){
    ...
}

void levelOrderPrint(Node* root){
    queue<Node*> q;
    q.push(root);
    q.push(NULL);

    while(!q.empty()){
        auto front = q.front();
        q.pop();
        if(front == NULL){
            cout<<endl;
            if(!q.empty()){
                q.push(NULL);
            }
        }
        else{
            cout<< front->data <<" ";
            for(int i=0;i<front->children_count;i++){
                if(front->children[i]){
                    q.push(front->children[i]);
                }
            }
        }
    }
}

int main() {
    Node* root = takeInput();
    levelOrderPrint(root);
    return 0;
}
```