PW SKILLS

Lesson Plan

Binary tree-1

PW SKILLS

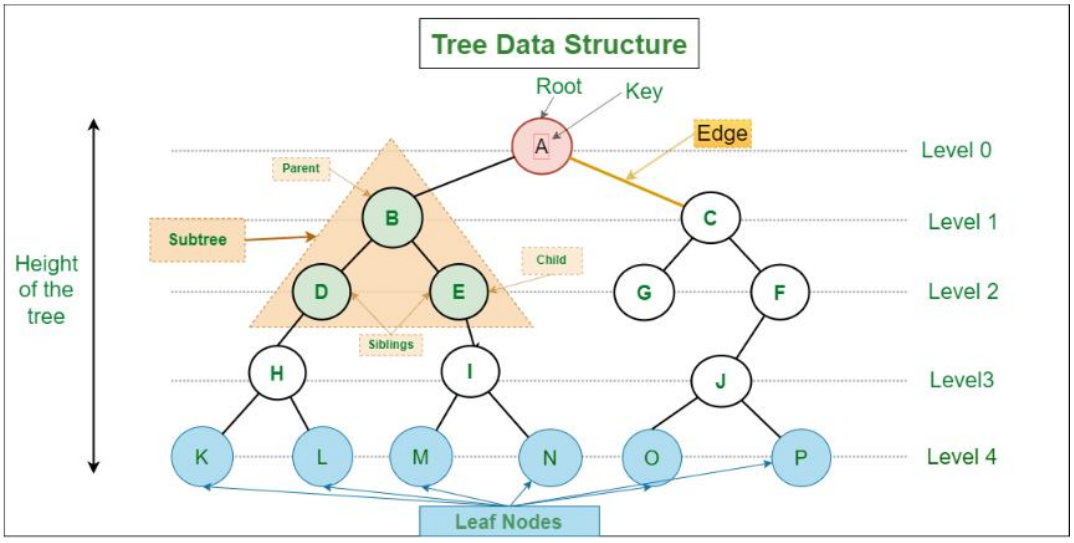
Prerequisites:

- Understanding of basic data structures like arrays and linked lists.
- Understanding of recursion.
- Proficiency in a C++ programming language

Today's Checklist:

- Introduction to Tree Data Structure
- Coding Implementation
- Display a Tree
- Find the sum of tree nodes
- Find the size of the Tree
- Find node with max value
- Find the number of levels of Binary Tree
- Types of Binary Trees
- Diameter of Binary Tree [Leetcode-543]
- Same Tree [Leetcode-100]
- Invert Binary Tree [Leetcode-226]
- Binary Tree Paths [Leetcode-257]
- Lowest Common Ancestor of a Binary tree [Leetcode-236]

Introduction to Tree Data Structure



A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Code:

```
#include <bits/stdc++.h>
using namespace std;
// Function to add an edge between vertices x and y
void addEdge(int x, int y, vector<vector<int> >& adj)
{
    adj[x].push_back(y);
    adj[y].push_back(x);
}
// Function to print the parent of each node
void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{
    // current node is Root, thus, has no parent
    if (parent == 0)
        cout << node << "→Root" << endl;
    else
        cout << node << "→" << parent << endl;
    // Using DFS
    for (auto cur : adj[node])
        if (cur ≠ parent)
            printParents(cur, adj, node);
}
// Function to print the children of each node
void printChildren(int Root, vector<vector<int> >& adj)
{
    // Queue for the BFS
    queue<int> q;
    // pushing the root
    q.push(Root);
    // visit array to keep track of nodes that have been
    // visited
    int vis[adj.size()] = { 0 };
// BFS
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        vis[node] = 1;
        cout << node << "→ ";
        for (auto cur : adj[node])
            if (vis[cur] == 0) {
                cout << cur << " ";
                q.push(cur);
            }
        cout << endl;
    }
}
// Function to print the leaf nodes
void printLeafNodes(int Root, vector<vector<int> >& adj)
{
    // Leaf nodes have only one edge and are not the root
    for (int i = 1; i < adj.size(); i++)
        if (adj[i].size() == 1 && i ≠ Root)
            cout << i << " ";
    cout << endl;
}
// Function to print the degrees of each node
void printDegrees(int Root, vector<vector<int> >& adj)
{
```

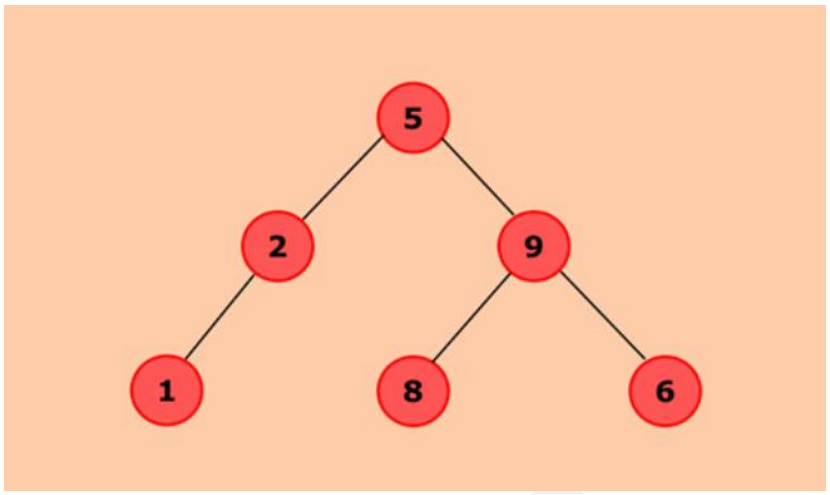
```
for (int i = 1; i < adj.size(); i++) {
        cout << i << ": ";
        // Root has no parent, thus, its degree is equal to
        // the edges it is connected to
        if (i == Root)
            cout << adj[i].size() << endl;
        else
            cout << adj[i].size() - 1 << endl;
    }
}
// Driver code
int main()
{
    // Number of nodes
    int N = 7, Root = 1;
    // Adjacency list to store the tree

vector<vector<int> > adj(N + 1, vector<int>());
    // Creating the tree
    addEdge(1, 2, adj);
    addEdge(1, 3, adj);
    addEdge(1, 4, adj);
    addEdge(2, 5, adj);
    addEdge(2, 6, adj);
    addEdge(4, 7, adj);
    // Printing the parents of each node
    cout << "The parents of each node are:" << endl;
    printParents(Root, adj, 0);

    // Printing the children of each node
    cout << "The children of each node are:" << endl;
    printChildren(Root, adj);

    // Printing the leaf nodes in the tree
    cout << "The leaf nodes of the tree are:" << endl;
    printLeafNodes(Root, adj);

    // Printing the degrees of each node
    cout << "The degrees of each node are:" << endl;
    printDegrees(Root, adj);

    return 0;
}
```

Explanation:

- The code defines functions to work with a tree data structure represented using an adjacency list.
- The addEdge function adds an undirected edge between two vertices in the tree.
- The printParents function prints the parent of each node using depth-first search (DFS).
- The printChildren function prints the children of each node using breadth-first search (BFS).
- The printLeafNodes function prints the leaf nodes of the tree (nodes with only one edge and not the root).
- The printDegrees function prints the degrees of each node (number of edges connected to the node).
- In the main function, a tree is created with 7 nodes, and various properties are printed using the defined functions.
- The tree structure is as follows: 1 is connected to 2, 3, and 4; 2 is connected to 5 and 6; 4 is connected to 7.
- The output includes the parents, children, leaf nodes, and degrees of each node in the tree.

Find sum of tree nodes

Given a Binary Tree of size N, your task is to complete the function sumBt(), that should return the sum of all the nodes of the given binary tree.



For the given tree, sum of nodes of the binary tree will be 1 + 2 + 5 + 8 + 6 + 9 = 31.

Code:

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

struct Node {
    int key;
    struct Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/*Function to find sum of all elements*/
int sumBT(Node* root)
{
      //sum variable to track the sum of
      //all variables.
    int sum = 0;

    queue<Node*> q;

      //Pushing the first level.
    q.push(root);

      //Pushing elements at each level from
      //the tree.
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

          //After popping each element from queue
          //add its data to the sum variable.
        sum += temp->key;

        if (temp->left) {
            q.push(temp->left);
        }
        if (temp->right) {
            q.push(temp->right);
        }
    }
    return sum;
}
```

Explanation:

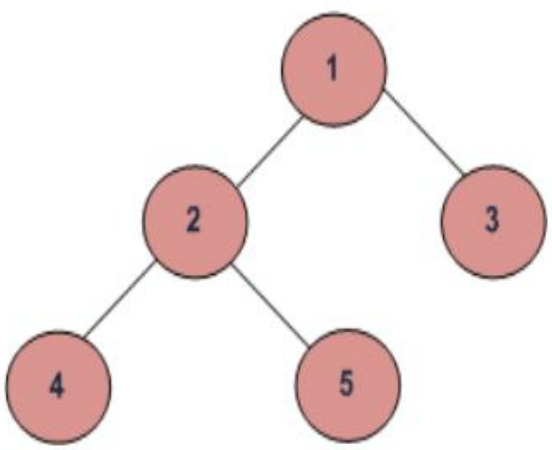
- The code defines a structure Node representing a binary tree node with an integer key, a pointer to the left child, and a pointer to the right child.
- A utility function newNode is defined to create a new node with a given key, initializing its left and right pointers to NULL.
- The main function sumBT takes a binary tree root node as an argument and calculates the sum of all elements in the tree using level order traversal (Breadth-First Search).
- The function initializes a variable sum to 0 to track the sum of all elements.
- It uses a queue to perform level order traversal starting from the root.
- At each level, it dequeues a node, adds its key to the sum, and enqueues its left and right children if they exist.
- The process continues until all nodes are processed, and the final sum is returned.
- The main function does not exist in the provided code; the code focuses on the sumBT function and related utility functions.

Time Complexity: O(n) - Linear time, where n is the number of nodes in the binary tree.

Space Complexity:
In the worst-case balanced scenario: O(2^(height - 1)), where height is the height of the binary tree.
In the worst-case skewed scenario: O(n), where n is the number of nodes in the tree.

Find the size of the Tree

Given a binary tree of size N, you have to count number of nodes in it.



Size of a tree is the number of elements present in the tree. Size of the below tree is 5.

Code:

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

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class node
{
```

```
  public:
    int data;
    node* left;
    node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* Node = new node();
    Node→data = data;
    Node→left = NULL;
    Node→right = NULL;

    return(Node);
}

/* Computes the number of nodes in a tree. */
int size(node* node)
{
    if (node == NULL)
        return 0;
    else
        return(size(node→left) + 1 + size(node→right));
}
```

Explanation:

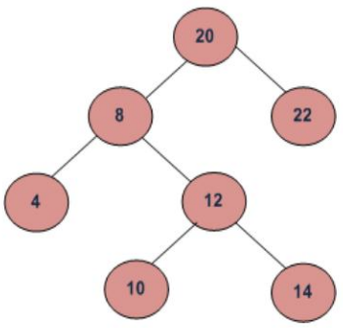
- Defines a class node for binary tree nodes.
- Has data, left child, and right child members.
- newNode creates a new node with given data.
- Initializes left and right pointers to NULL.
- size computes the number of nodes in a tree.
- Recursive approach: size(node->left) + 1 + size(node->right).
- size computes the number of nodes in a tree.
- Recursive approach: size(node->left) + 1 + size(node->right).
- Create instances of node, build a binary tree, and call size.

Time Complexity:
O(n) where n is the number of nodes, due to the recursive traversal of the entire binary tree.

Space Complexity:
O(log(n)) for a balanced tree, O(n) for an unbalanced tree, considering the maximum depth of the recursive call stack.
Additional O(n) space for node allocations.

Find node with max value

Given a binary tree of size N, you have to count number of nodes in it.



Maximum value : 22

Code:

```
#include <iostream>
#include <climits>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

int findMaxValue(TreeNode* root) {
    if (root == nullptr) {
        return INT_MIN;
    }
    int leftMax = findMaxValue(root->left);
    int rightMax = findMaxValue(root->right);
    return std::max(root->val, std::max(leftMax, rightMax));
}
```

Explanation:

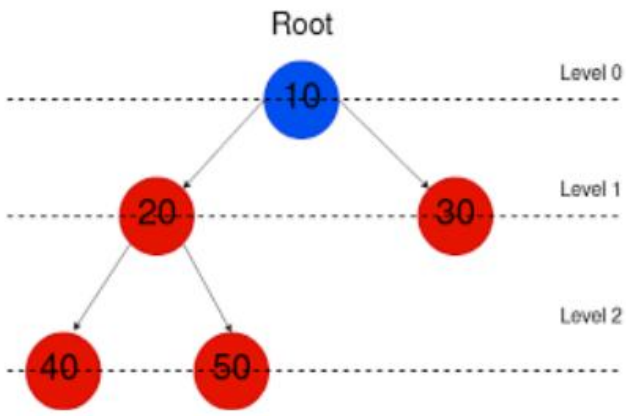
- Defines a binary tree node using a TreeNode struct.
- Each node has an integer value (val) and pointers to left and right children.
- Takes the root of a binary tree.
- Recursively finds the maximum value.
- Returns INT_MIN for an empty tree.
- Calls findMaxValue on left and right subtrees.
- Finds maximum values in the subtrees.
- TreeNode constructor sets initial values for a node.

Time Complexity: O(n) where n is the number of nodes.Visits each node once, linear time.

Space Complexity:
O(h) where h is the height of the tree.
Balanced tree: O(log n).
Skewed tree: O(n) in the worst case due to recursion depth.

Find the number of levels of Binary Tree

Given a binary tree of size N, you have to count number of nodes in it.



Code:

```
#include <iostream>
#include <queue>

using namespace std;

// Define the structure of a binary tree node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr)
{}
};

// Function to find the number of levels in a binary tree
int findNumberOfLevels(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    // Use a queue for level order traversal
    queue<TreeNode*> q;
    q.push(root);

    int levels = 0;

    while (!q.empty()) {
        int size = q.size();  // Number of nodes at the current
level

        // Process all nodes at the current level
        while (size--) {
            TreeNode* current = q.front();
            q.pop();

            // Enqueue left and right children, if any
            if (current->left) {
                q.push(current->left);
            }
            if (current->right) {
                q.push(current->right);

         }
        }

        levels++;
    }

    return levels;
}
```

Explanation:

- Represents a binary tree node.
- Contains an integer value (data) and pointers to left and right children.
- Takes the root of a binary tree as an argument.
- Returns the number of levels in the tree.
- If the tree is empty (root == nullptr), returns 0.
- Uses a queue for level-order traversal.
- Initializes the queue with the root node and sets the initial level count to 0.
- Performs level-order traversal using a while loop.
- Enqueues left and right children of each node.
- Increments the level count after processing all nodes at the current level.

Time Complexity:
The time complexity is O(N), where N is the number of nodes in the binary tree.
Each node is processed once, and in the worst case, all nodes are visited during the level-order traversal.
The while loop iterates through each level, and the inner loop processes all nodes at that level.

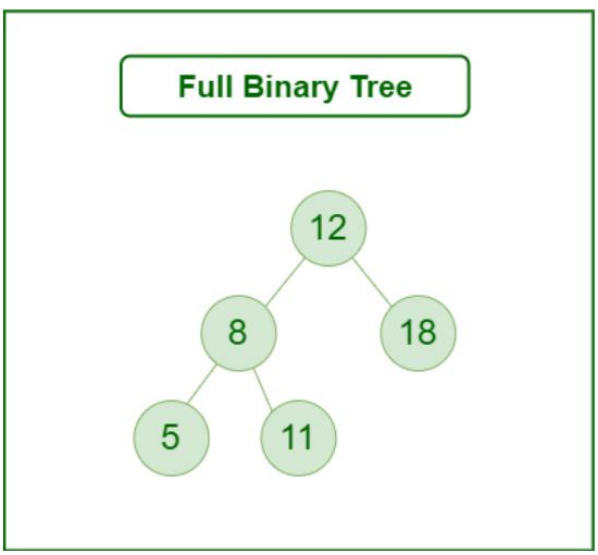
Space Complexity:
The space complexity is O(W), where W is the maximum width (number of nodes at the widest level) of the binary tree.

In the worst case, the queue can store all nodes at the widest level.
The space complexity is also influenced by the maximum number of nodes at a single level, which can be the entire last level.

Types of Binary Trees

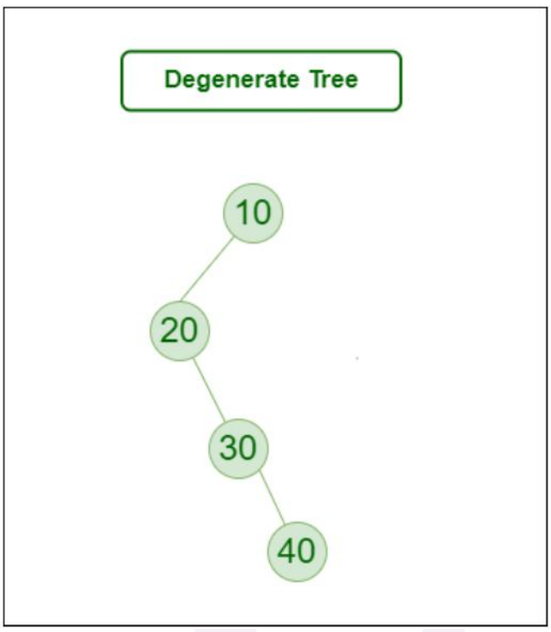
Types of Binary Tree based on the number of children:

Full Binary Tree



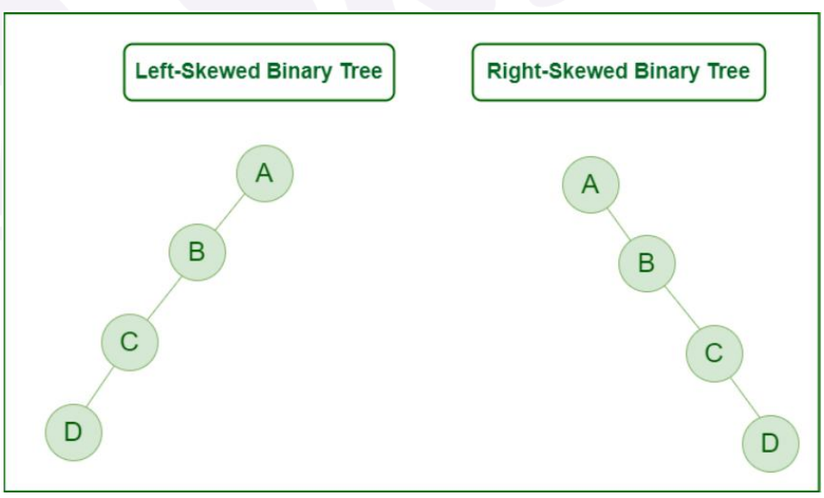
A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children. A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.

Degenerate (or pathological) tree



A Tree where every internal node has one child. Such trees are performance-wise same as the linked list. A degenerate or pathological tree is a tree having a single child either left or right.

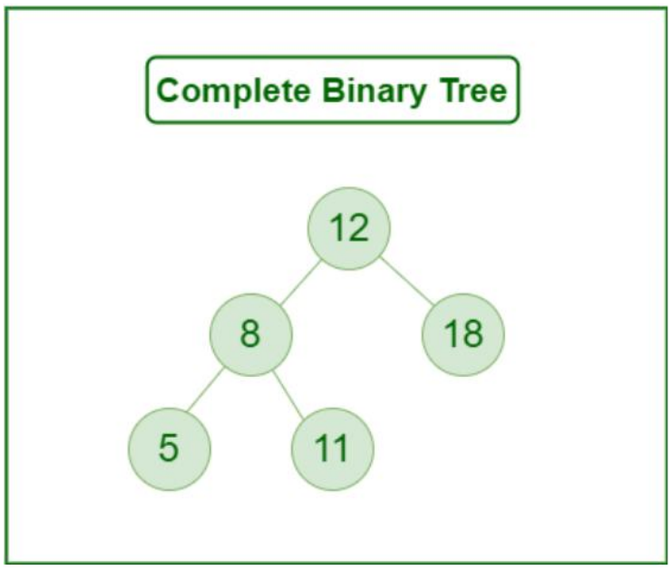
Skewed Binary Tree



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.

Types of Binary Tree On the basis of the completion of levels:

Complete Binary Tree



A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

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

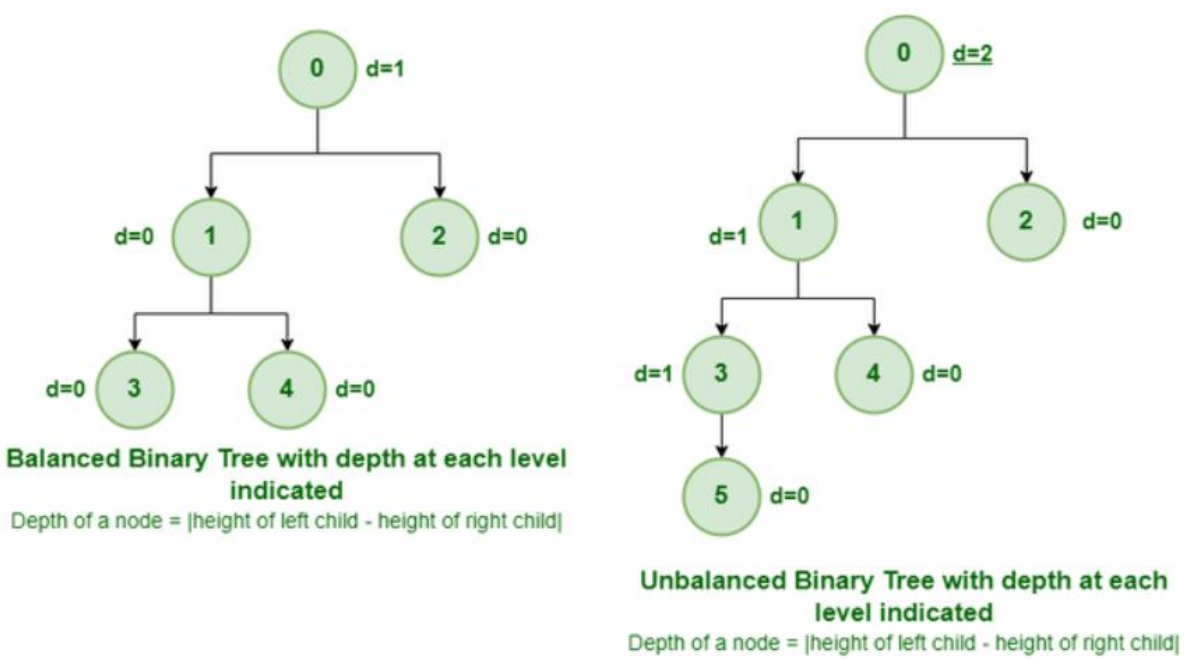
- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are examples of Perfect Binary Trees.
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.

Balanced Binary Tree



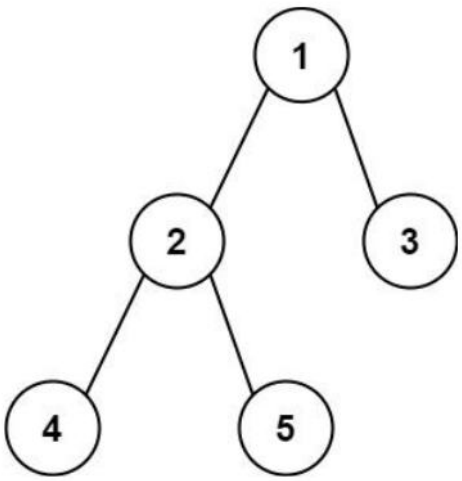
Balanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|

Unbalanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|

Balanced binary trees, like AVL and Red-Black trees, maintain a height of O(log n), ensuring efficient search, insert, and delete operations with O(log n) time complexity.

Diameter of Binary Tree [Leetcode-543]

Given the root of a binary tree, return the length of the diameter of the tree.
The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.
The length of a path between two nodes is represented by the number of edges between them.



Input: root = [1,2,3,4,5]
Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Code:

```
class Solution {
public:
    int d=0;
    int heightTree(TreeNode* root){
        if(!root)
            return 0;
        int lheight=heightTree(root→left);
        int rheight=heightTree(root→right);
        d=max(d,lheight+rheight);//compare the sum with sum of
both heights
        return 1+max(lheight,rheight);
    }
    int diameterOfBinaryTree(TreeNode* root) {
        heightTree(root);
        return d;
    }
};
```

Explanation:

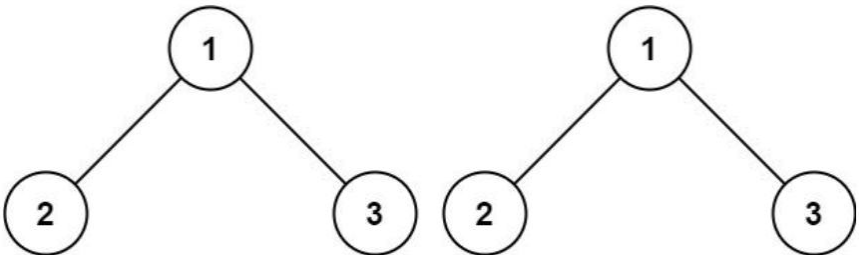
- The code defines a class Solution for finding the diameter of a binary tree.
- It uses a member variable d to store the diameter.
- The heightTree function recursively calculates the height of each subtree.
- Base case: Return 0 if the current node is null.
- Calculate left and right subtree heights (lheight and rheight).
- Update the diameter (d) by comparing it with the sum of left and right subtree heights.
- Return 1 plus the maximum height of left and right subtrees.
- The diameterOfBinaryTree function calls heightTree to calculate the diameter and returns the result.

Time Complexity: O(n) - linear, where n is the number of nodes in the binary tree.
Each node is visited once during the recursive traversal.

Space Complexity: O(h) - linear, where h is the height of the binary tree.
The space is primarily used for the recursive call stack.
In the worst case (skewed tree), space complexity is O(n).
In a balanced tree, space complexity is O(log(n)).

Same Tree [Leetcode-100]

Given the roots of two binary trees p and q, write a function to check if they are the same or not.
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.



Input: p = [1,2,3], q = [1,2,3]
Output: true

Code:

```
class Solution {
public:

  bool isSameTree(TreeNode* p, TreeNode* q) {

        if (p == NULL && q == NULL) {
            return true;
        }
        if (p == NULL || q == NULL) {
            return false;
        }

        if (p->val == q->val) {
            return isSameTree(p->left, q->left) && isSameTree(p
->right, q->right);
        }

        return false;
    }
};
```

Explanation:

- Function isSameTree checks if two binary trees are identical.
- If both trees are NULL, return true (they are identical).
- If only one tree is NULL, return false (they are not identical).
- Compare values of current nodes (p->val and q->val).
- Recursively check left and right subtrees.
- Return logical AND of recursive calls.
- If current node values are not equal, return false.
- If all checks pass, return true; otherwise, return false.

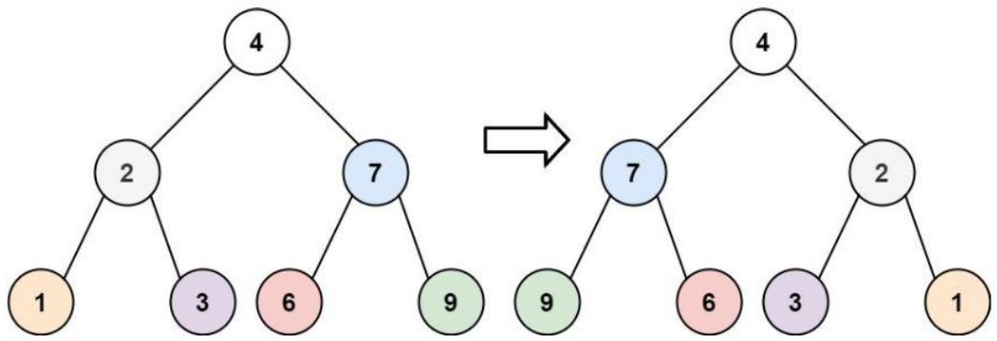
Time Complexity:
The time complexity is O(N), where N is the number of nodes in the larger of the two trees.
In the worst case, the algorithm needs to visit every node once to check for equality.

Space Complexity:
The space complexity is O(H), where H is the height of the larger of the two trees.
This is due to the recursive calls, and in the worst case, the maximum depth of the recursion is the height of the tree.

The recursive call stack contributes to the space complexity.

Invert Binary Tree [Leetcode-226]

Given the root of a binary tree, invert the tree, and return its root.



Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]

Code:

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        // Base Case
        if(root==NULL)
            return NULL;
        invertTree(root->left); //Call the left substree
        invertTree(root->right); //Call the right substree
        // Swap the nodes
        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;
        return root; // Return the root
    }
};
```

Explanation:

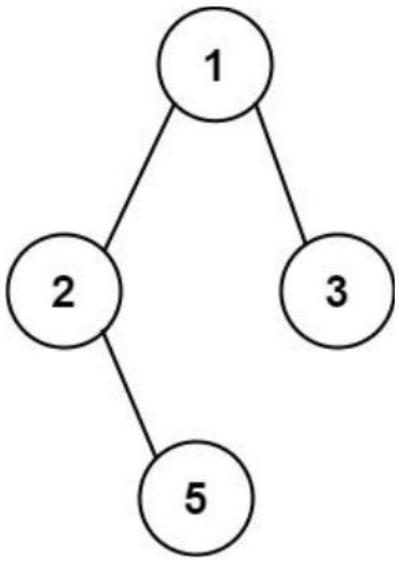
- Inverts a binary tree by swapping left and right subtrees.
- If the current node (root) is NULL, returns NULL.
- Recursively calls itself on the left and right subtrees.
- Swaps the left and right pointers of the current node.
- Returns the modified root after inversion.
- Recursive inversion from root to leaf nodes.
- Assumes a TreeNode structure with left and right pointers.

Time Complexity: O(n) - linear, where n is the number of nodes.
Each node is visited once, and the operations are constant time.

Space Complexity: O(h) - height of the tree.
Recursive calls contribute to the call stack.
In the worst case (unbalanced tree), space complexity is O(n).
In the best case (balanced tree), space complexity is O(log(n)).

Binary Tree Paths [Leetcode-257]

Given the root of a binary tree, return all root-to-leaf paths in any order.
A leaf is a node with no children.



Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]

Code:

```
class Solution {

vector<string> res;
public:

    void helper(TreeNode* root,string s){
        if(!root){return;}
        if(s.empty()){s+=(to_string(root->val));}
        else{s=s+"->"+(to_string(root->val));}
        if(!root->left && !root->right){
            res.push_back(s);
        }
        helper(root->left,s);
        helper(root->right,s);
        // s.erase(s.end() - 3);
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        helper(root,"");
        return res;
    }
};
```

Explanation:

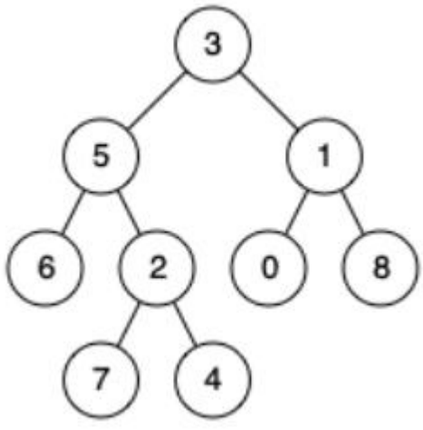
- Solution class with a member variable vector<string> res.
- vector<string> binaryTreePaths(TreeNode* root): Entry point for users to obtain binary tree paths.
- void helper(TreeNode* root, string s): Recursive function to traverse the binary tree and generate paths.
- Parameters: TreeNode* root, string s.
- Base case: Return if the current node is null.
- Update path string s with the current node's value.
- If leaf node, add the path to the result vector.
- Recursively call helper for left and right children.

Time Complexity: O(N) - Linear time, where N is the number of nodes in the binary tree.

Space Complexity: O(N) - Linear space, due to the recursive call stack and the storage of paths in the result vector.

Binary Tree Paths [Leetcode-257]

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Code:

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
        if(!root){
            return NULL;
        }
        if(root==p || root==q)return root;
        TreeNode* l,*r;
        l = lowestCommonAncestor(root->left,p,q);
        r= lowestCommonAncestor(root->right,p,q);
        if(l && r)return root;
        if(l)return l;
        if(r) return r;
        return NULL;
    }
};
```

Explanation:

- Function Purpose: Finds the lowest common ancestor of two nodes (p and q) in a binary tree.
- Base Case: If the current node is NULL, returns NULL.
- Node Match: If the current node is equal to either p or q, returns the current node.
- Recursive Calls: Recursively calls the function on the left and right subtrees and stores the results in l and r.
- Common Ancestor Check: If both l and r are non-NULL, returns the current node as the lowest common ancestor.
- Propagation: If only one of l or r is non-NULL, propagates that non-NULL result up.
- No Match: If both l and r are NULL, returns NULL, indicating no match in the current subtree.

Time Complexity: O(n) - linear time, as each node is visited once.

Space Complexity:

Worst Case: O(n) - the recursion stack can go as deep as the height of the tree, which is n in the worst case.
Average Case (for balanced tree): O(log n) - as the height is log(n).