PW SKILLS

Lesson Plan

Binary tree-2

PW SKILLS

Prerequisites:

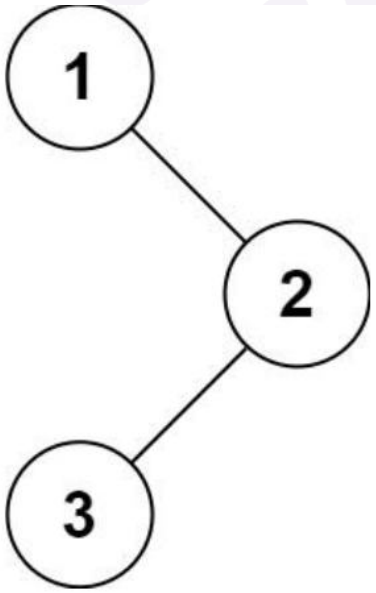
- Understanding of basic data structures like arrays and linked lists.
- Understanding of recursion.
- Proficiency in a C++ programming language

Today's Checklist:

- Traversals in Tree (Preorder, Inorder, Postorder) [Leetcode-144, 94, 145]
- How recursion works on tree?
- Print elements of nth level
- Level Order Traversal [Leetcode-102]
- Level Order Traversal (Using Queue)
- Construct Tree from Level Order Traversal

Preorder Traversal In Tree [Leetcode-144]

Given the root of a binary tree, return the preorder traversal of its nodes' values.



Input: root = [1,null,2,3]
Output: [1,2,3]

Example 2:
Input: root = []
Output: []

Code:

```
class Solution {
public:
void preorder(TreeNode* root,vector<int> &ans){
         if(root==NULL)return;
         ans.push_back(root->val);
         preorder(root->left,ans);
         preorder(root->right,ans);
}
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        preorder(root,ans);
        return ans;
    }
};
```

Explanation:

- Recursive function for preorder traversal.
- Takes a binary tree node and a vector as parameters.
- Appends the current node's value to the vector.
- Recursively calls itself for the left and right children.
- Initiates an empty vector.
- Calls the preorder function to perform preorder traversal starting from the root.
- Returns the vector containing the preorder traversal result.

Time Complexity: O(N) - Linear time complexity since each node is visited once.

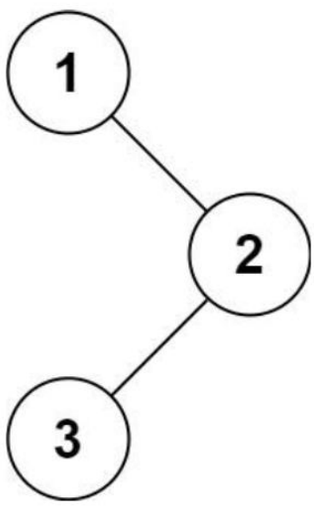
Space Complexity:

Worst Case: O(N) - Due to the call stack in the case of a skewed tree.
Average Case (Balanced Tree): O(log N) - Height of the call stack in a balanced tree.

Inorder Traversal In Tree [Leetcode-94]

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:



Input: root = [1,null,2,3]
Output: [1,3,2]

Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [1]
Output: [1]

Code:

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        helper(root, result);
        return result;
    }

    void helper(TreeNode* root, vector<int>& result) {

if (root != nullptr) {
            helper(root->left, result);
            result.push_back(root->val);
            helper(root->right, result);
        }
    }
};
```

Explanation:

- Takes a TreeNode* as input (root of the binary tree).
- Returns a vector of integers representing the inorder traversal of the tree.
- Calls the helper method to perform the actual traversal.
- Recursive function for inorder traversal.
- Takes a TreeNode* (current node) and a reference to a vector of integers as parameters.
- Calls helper on the left subtree.
- Appends the value of the current node to the result vector.
- Calls helper on the right subtree.

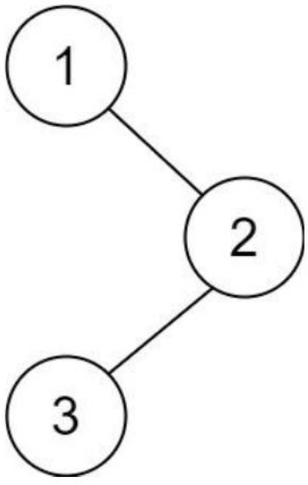
Time Complexity: O(N) - Linear time, where N is the number of nodes in the binary tree. Each node is visited once during the inorder traversal.

Space Complexity: O(N) - Linear space, where N is the number of nodes in the binary tree. The space is used for the call stack during recursion and the result vector storing node values.

Postorder Traversal In Tree [Leetcode-145]

Given the root of a binary tree, return the postorder traversal of its nodes' values.

Example 1:



Input: root = [1,null,2,3]
Output: [3,2,1]

Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [1]
Output: [1]

Code:

```
class Solution {
public:
    void solver(TreeNode* root,vector<int>&ans){
        if(root==NULL){
            return ;
        }
        if(root->left){
            solver(root->left,ans);
        }
        if(root->right){
            solver(root->right,ans);
        }

  ans.push_back(root->val);
    }
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        solver(root,ans);
        return ans;
    }
};
```

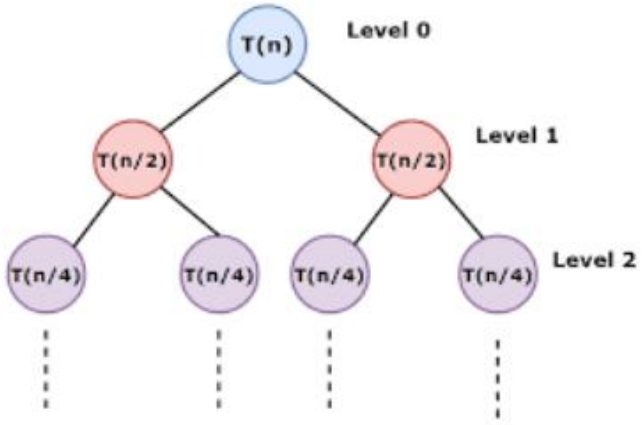
Explanation:

- Private function solver recursively traverses the tree in postorder.
- Base case: Returns if the current root is NULL.
- Traverses left and right subtrees recursively.
- Pushes the value of the current root onto a vector (ans).
- Public function postorderTraversal initializes an empty vector and calls solver.
- The final result, representing the postorder traversal, is returned as a vector.

Time Complexity: O(N) - linear time complexity, as each node is visited once during the postorder traversal.

Space Complexity: O(N) - linear space complexity due to the recursive call stack and the vector (ans) storing node values.

How recursion works on tree?



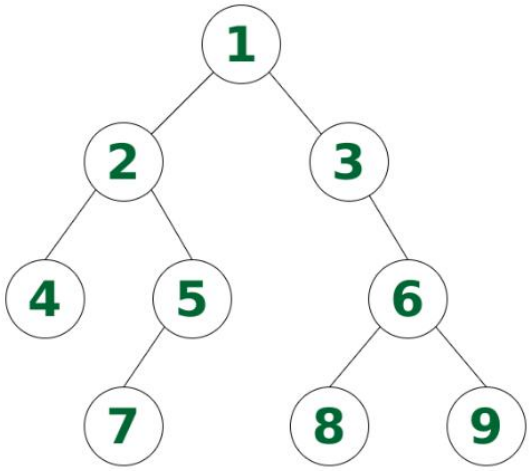
Explanation:

- Base Case:Define a simple scenario where the function returns a result directly, often involving leaf nodes or empty subtrees.
- Recursive Step:Break down the problem into smaller sub-problems by applying the function recursively to each subtree.
- Combining Results:Combine results from recursive calls to get the final result for the current node or level of the tree.
- This approach is efficient and often leads to concise code, as each node is treated as the root of its own subtree.

Print elements of nth level

Given a binary tree and an integer K, the task is to print all the integers at the Kth level in the tree from left to right.

Input: Tree in the image below, K = 3



Output: 4 5 6
Explanation: All the nodes present in level 3 of above binary tree from left to right are 4, 5, and 6.

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
    int data;
    struct Node *left, *right;
};

void printNodes(Node* root, int level, int K)
{
    // Base Case
    if (root == NULL) {
        return;
    }

    printNodes(root->left, level + 1, K);

    printNodes(root->right, level + 1, K);

    if (K == level) {
        cout << root->data << " ";
    }
}
 Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
```

Explanation:

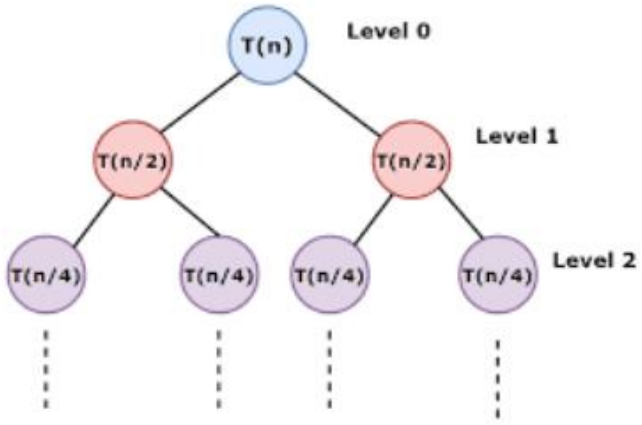
- The code defines a Node structure for a binary tree with data, left, and right pointers.
- printNodes is a recursive function to print nodes at a specific level (K) in a binary tree using DFS.
- Base case: If the current node is NULL, return.
- Recursively calls printNodes for the left and right subtrees, incrementing the level.
- Prints the data of the current node if its level matches the target level (K).
- newNode creates a new tree node with the given data.
- Example usage involves creating a binary tree and calling printNodes with the root and desired level (K).
- Assumes a 0-indexed tree where the root is at level 0.
- The code focuses on depth-first traversal and level-based printing of nodes in a binary tree.

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

Space Complexity: O(H) - linear space, where H is the height of the binary tree (max depth of recursive call stack).

Worst case: O(N) for a skewed tree.
Best case: O(log N) for a balanced tree.
Recursive approach utilizes the call stack.

How recursion works on tree?



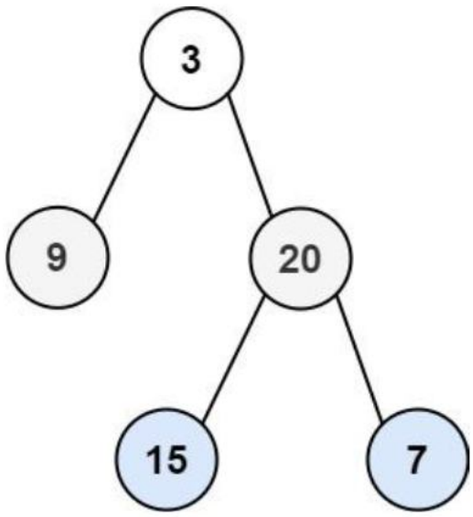
Explanation:

- Base Case:Define a simple scenario where the function returns a result directly, often involving leaf nodes or empty subtrees.
- Recursive Step:Break down the problem into smaller sub-problems by applying the function recursively to each subtree.
- Combining Results:Combine results from recursive calls to get the final result for the current node or level of the tree.
- This approach is efficient and often leads to concise code, as each node is treated as the root of its own subtree.

Binary Tree Level Order Traversal [Leetcode-102]

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example 1:



Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]

Example 2:

Input: root = [1]
Output: [[1]]

Example 3:

Input: root = []
Output: []

Code:

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        int count = 0;
        level(root , count , result);
        return result;

    }
private:
    void level(TreeNode* root , int count , vector<vector<int>>
&result){
        if(root == NULL){
            return ;
        }

        if(result.size() <= count){
            result.push_back(vector<int>());
        }
        result[count].push_back(root->val);
        count++;
        level(root->left , count , result);
        level(root->right , count , result);

    }
};
```

Explanation:

- Conduct level-order traversal on a binary tree.
- Recursive approach.
- Maintains a 2D vector (result) for storing levels.
- Takes root node as input.
- Initializes result and calls private function.
- Recursive traversal.
- Base case: Null node.
- Updates result vector for each level.
- Returns a 2D vector (result).
- Each inner vector represents node values at a specific level.

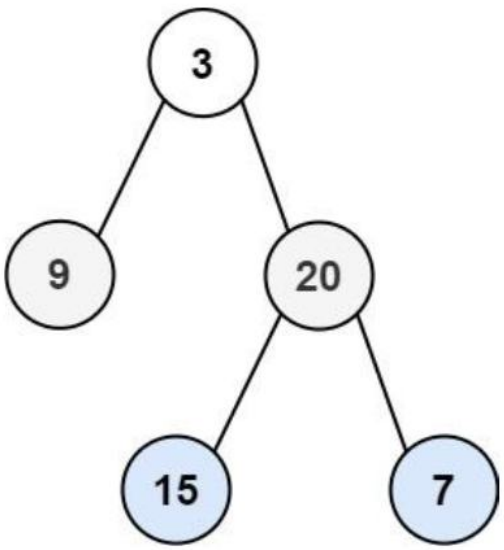
Time Complexity: O(N)
The code visits each node once in a depth-first manner.

Space Complexity: O(H)
The space complexity depends on the maximum height of the call stack during recursion.
In the worst case (skewed tree), the height is N, resulting in O(N) space complexity.
In the average case (balanced tree), the height is log(N), resulting in O(log(N)) space complexity.

Binary Tree Level Order Traversal [Leetcode-102] (Using Queue)

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example 1:



Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]

Example 2:

Input: root = [1]
Output: [[1]]

Example 3:

Input: root = []
Output: []

Code:

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if(root==NULL)return ans;
        queue<TreeNode*>q;
        q.push(root);
        while(!q.empty()){
            int s=q.size();
            vector<int>v;
            for(int i=0;i<s;i++){
                TreeNode *node=q.front();
                q.pop();
                if(node->left!=NULL)q.push(node->left);
                if(node->right!=NULL)q.push(node->right);
                v.push_back(node->val);
            }
            ans.push_back(v);
        }
        return ans;
    }
};
```

Explanation:

- Objective: Perform level-order traversal of a binary tree and return the result.
- Data Structures: ans (2D vector for traversal outcome), q (Queue for level-order traversal).
- Traversal Process: Enqueue root into q; while q not empty, process each level, enqueue children, and append values to ans.
- Return: Final 2D vector ans representing the level-order traversal.

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node is visited once.

Space Complexity: O(M) where M is the maximum number of nodes at any level in the binary tree. In the worst case, the queue (q) will store all nodes at a single level.

Construct Tree from Level Order Traversal

Given an array of elements, the task is to insert these elements in level order and construct a tree.

Code:

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

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

struct Node* newNode(int value)
{
    Node* n = new Node;
    n->key = value;
    n->left = NULL;
    n->right = NULL;
    return n;
}

struct Node* insertValue(struct Node* root, int value,
                         queue<Node *>& q)
{
    Node* node = newNode(value);
    if (root == NULL)
        root = node;

    else if (q.front()->left == NULL)
        q.front()->left = node;

    else {
        q.front()->right = node;
        q.pop();
    }

    q.push(node);
    return root;
}

Node* createTree(int arr[], int n)
```

```
{
    Node* root = NULL;
    queue<Node*> q;
    for (int i = 0; i < n; i++)
      root = insertValue(root, arr[i], q);
    return root;
}

// This is used to verify the logic.
void levelOrder(struct Node* root)
{
    if (root == NULL)
        return;
    queue<Node*> n;
    n.push(root);
    while (!n.empty()) {
        cout << n.front()->key << " ";
        if (n.front()->left != NULL)
            n.push(n.front()->left);
        if (n.front()->right != NULL)
            n.push(n.front()->right);
        n.pop();
    }
}
```

Explanation:

- The code defines a binary tree structure with a level-order insertion method.
- It uses a queue to construct the binary tree in a level-order manner.
- The createTree function builds a binary tree from an array of integers.
- The levelOrder function prints the nodes of the tree in level-order traversal.
- The code is a concise illustration of binary tree construction and level-order traversal.

Time Complexity:
The time complexity for inserting each element into the binary tree is O(1) on average because each node is inserted once.

The createTree function iterates through the array of size n, resulting in a time complexity of O(n).
The levelOrder function performs a level-order traversal, visiting each node once, leading to a time complexity of O(n).

Space Complexity:
The space complexity is O(n) as the queue can store up to n nodes during the construction of the binary tree. The queue is the primary contributor to space usage, and its size is proportional to the number of nodes in the binary tree.

The space complexity is determined by the maximum number of nodes at any level in the binary tree during construction.