PW SKILLS

Lesson Plan

Binary Trees - 3
Java

PW SKILLS

Today's Checklist

- Preorder, Inorder, Postorder Traversal - Iterative
- Construct Binary Tree from Preorder & Inorder Traversal
- Boundary Traversal of Tree
- Binary Tree Right Side View
- Top View of Binary Tree
- Path Sum II
- Path Sum III

Preorder Traversal (Iterative)

Code:

```
public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            result.add(node.val);

            if (node.right != null) {
                stack.push(node.right);
            }

            if (node.left != null) {
                stack.push(node.left);
            }
        }

        return result;
    }
```

Explanation:
The preorderTraversal function performs preorder traversal iteratively using a stack.
It initializes an empty result vector to store the traversal.
It starts with the root node and pushes it onto the stack.
While the stack is not empty, it pops nodes and adds their values to the result.
For each node, it first pushes the right child (if present) onto the stack, and then the left child (if present). This order ensures the correct preorder traversal.
Finally, the main function demonstrates the use of preorderTraversal and prints the result.

Time complexity O(n): where 'n' is the number of nodes in the tree, as it visits each node once.
space complexity O(h): where 'h' is the height of the tree, due to the stack space used in the worst-case scenario, which is the height of the tree for a skewed tree.

Inorder Traversal (Iterative)

Code:

```
public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;

        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }

            curr = stack.pop();
            result.add(curr.val);

            curr = curr.right;
        }

        return result;
    }
```

Explanation:
The inorderTraversal function performs inorder traversal iteratively using a stack.
It initializes an empty result vector to store the traversal.
It starts with the root node and traverses down the left side, pushing nodes onto the stack.
It then starts popping nodes from the stack, adding their values to the result, and moves to the right child if present.
This sequence of operations simulates the inorder traversal behavior.
The main function demonstrates the use of inorderTraversal and prints the resulting inorder traversal.

Time complexity is O(n): where 'n' is the number of nodes in the tree, visiting each node exactly once.
Space complexity is O(h): where 'h' is the height of the tree, due to the stack space used, which depends on the height of the tree in the worst case for a skewed tree.

Postorder Traversal (Iterative)

Code:

```
public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Stack<TreeNode> s1 = new Stack<>();
        Stack<TreeNode> s2 = new Stack<>();
        s1.push(root);
```

```
        while (!s1.isEmpty()) {
            TreeNode node = s1.pop();
            s2.push(node);

            if (node.left ≠ null) {
                s1.push(node.left);
            }
            if (node.right ≠ null) {
                s1.push(node.right);
            }
        }

        while (!s2.isEmpty()) {
            result.add(s2.pop().val);
        }

        return result;
    }
```

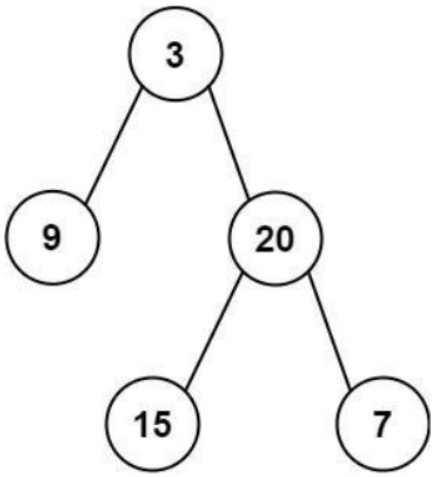
Explanation:
The inorderTraversal function performs inorder traversal iteratively using a stack.
It initializes an empty result vector to store the traversal.
It starts with the root node and traverses down the left side, pushing nodes onto the stack.
It then starts popping nodes from the stack, adding their values to the result, and moves to the right child if present.
This sequence of operations simulates the inorder traversal behavior.
The main function demonstrates the use of inorderTraversal and prints the resulting inorder traversal.

Time complexity is O(n): where 'n' is the number of nodes in the tree, visiting each node exactly once.
Space complexity is O(n): due to the two stacks used, which can hold all nodes in the worst-case scenario.

Q. Construct Binary Tree from Preorder & Inorder Traversal [LeetCode 105]

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Example 1:



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

Example 2:

Input: preorder = [-1], inorder = [-1]
Output: [-1]

Code:

```
public class Solution {
    int preorderIndex;
    Map<Integer, Integer> inorderIndexMap;

    public TreeNode buildTreeHelper(int[] preorder, int left, int
right) {
        if (left > right)
            return null;

        int rootValue = preorder[preorderIndex++];
        TreeNode root = new TreeNode(rootValue);
        int inorderPivotIndex = inorderIndexMap.get(rootValue);

        root.left = buildTreeHelper(preorder, left,
inorderPivotIndex - 1);
        root.right = buildTreeHelper(preorder, inorderPivotIndex
+ 1, right);
        return root;
    }

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        preorderIndex = 0;
        inorderIndexMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderIndexMap.put(inorder[i], i);
        }
        return buildTreeHelper(preorder, 0, preorder.length - 1);
    }
  }
```

Explanation:
PreOrder Traversal- Root->left->right
Inorder Traversal- Left->Root->right

As we can clearly see, in preorder the first node is always the root, and in inorder the middle node is root.

Now, given preorder, we will take the first number and make it our root, now find the same number in inorder traversal, this number's left in inorder traversal will form left subtree and right would form right subtree.

Now recursively solve for left and right subtrees.

Time Complexity: O(n): 'n' being the total number of nodes in the binary tree. The algorithm processes each node once as it constructs the tree from preorder and inorder traversals.

Space Complexity: O(n): The space complexity is also O(n) due to the usage of the hash map (`inorderIndexUmp`) to store the indices of elements from the inorder traversal. The recursion also utilizes stack space proportional to the height of the tree during the tree construction.

Boundary Traversal of Tree

Input: The input is a binary tree represented by its root node. The nodes of the binary tree are represented using a Node class which has a data field to store the value of the node and left and right fields to point to its left and right child nodes respectively.

Output: The output is the boundary traversal of the binary tree. The boundary of a binary tree consists of its left boundary -> leaf nodes -> right boundary.

Code:

```
import java.util.ArrayList;
import java.util.List;

// Node class to represent a node in the binary tree
class Node {
    int data;
    Node left;
    Node right;

    Node(int data) {
        this.data = data;
        left = right = null;
    }
}

public class BoundaryTraversal {
    // Function to print the left boundary of a binary tree
    static void printLeftBoundary(Node root, List<Integer>
boundary) {
        if (root == null)
            return;
        if (root.left != null) {
            boundary.add(root.data);
            printLeftBoundary(root.left, boundary);
        } else if (root.right != null) {
            boundary.add(root.data);
            printLeftBoundary(root.right, boundary);
        }
    }

    // Function to print the right boundary of a binary tree
    static void printRightBoundary(Node root, List<Integer>
boundary) {
```

```
        if (root == null)
            return;
        if (root.right != null) {
            printRightBoundary(root.right, boundary);
            boundary.add(root.data);
        } else if (root.left != null) {
            printRightBoundary(root.left, boundary);
            boundary.add(root.data);
        }
    }

    // Function to print the leaf nodes of a binary tree
    static void printLeaves(Node root, List<Integer> boundary) {
        if (root == null)
            return;
        printLeaves(root.left, boundary);
        if (root.left == null && root.right == null)
            boundary.add(root.data);
        printLeaves(root.right, boundary);
    }

    // Function to print the boundary traversal of a binary tree
    static List<Integer> printBoundary(Node root) {
        List<Integer> boundary = new ArrayList<>();
        if (root == null)
            return boundary;
        boundary.add(root.data);
        printLeftBoundary(root.left, boundary);
        printLeaves(root.left, boundary);
        printLeaves(root.right, boundary);
        printRightBoundary(root.right, boundary);
        return boundary;
    }

    // Example usage:
    public static void main(String[] args) {
        Node root = new Node(20);
        root.left = new Node(8);
        root.left.left = new Node(4);
        root.left.right = new Node(12);
        root.left.right.left = new Node(10);
        root.left.right.right = new Node(14);
        root.right = new Node(22);
        root.right.right = new Node(25);

        List<Integer> boundaryTraversal = printBoundary(root);
        System.out.println("Boundary Traversal: " +
boundaryTraversal);
    }
}
```

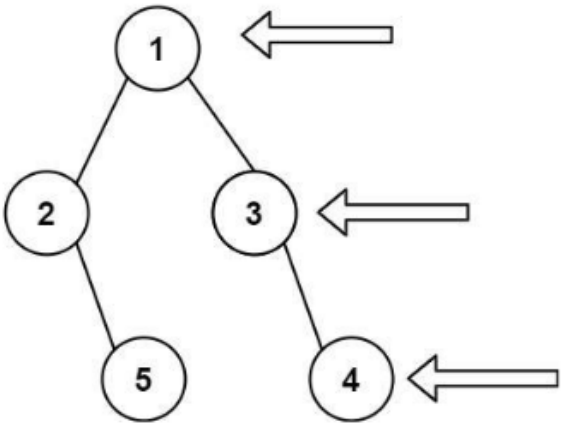
Explanation:
The program first defines a Node class to represent a node in the binary tree.
It then defines three helper functions to print the left boundary, the leaf nodes of the binary tree and the right boundary.
Traverse the left boundary of the tree from the root to the leftmost leaf, adding each node to the result list.
Traverse all the leaf nodes from left to right, adding each node to the result list.
Traverse the right boundary of the tree from the rightmost leaf to the root, adding each node to the result list.
Return the result list containing the boundary traversal of the tree, excluding the duplicates from the left and right boundaries.

Time complexity: The time complexity of the program is O(n), where n is the number of nodes in the binary tree. This is because each node in the binary tree is visited exactly once during the traversal.

Space complexity: The space complexity of the program is O(n) due to the recursive approach used to traverse the binary tree, where the maximum depth of the call stack is equal to the height of the tree. The space complexity can be O(log(n)) if the binary tree is balanced.

Q. Binary Tree Right Side View [LeetCode 199]

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.



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

Code:

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null)
            return result;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        TreeNode node;
```

```
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                node = queue.poll();
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
                if (i == size - 1) {
                    result.add(node.val);
                }
            }
        }
        return result;
    }
}
```

Explanation:
Traverse the tree in the level order fashion.
Push the nodes into the queue level wise.
Check if left and right child exists for each node in the queue one by one.
Push the value of last node of level in the tree into the vector.
Return the vector.

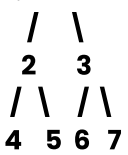
Time Complexity: O(n) - 'n' being the number of nodes in the binary tree. The algorithm traverses each node once in a level-order traversal using a queue.
Space Complexity: O(w) - 'w' being the maximum width of the tree (width refers to the maximum number of nodes at any level). In the worst case, when the tree is completely unbalanced (like a linked list), the queue will hold all nodes at the last level, leading to O(w) space usage.

Q. Top View of Binary Tree

The top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. A node x is there in the output if x is the topmost node at its horizontal distance. The horizontal distance of the left child of a node x is equal to a horizontal distance of x minus 1, and that of a right child is the horizontal distance of x plus 1.

Examples:

Input:



Output: Top view of the above binary tree is: 4 2 1 3 7

Code:

```
public class Solution {
    public void topView(Node root) {
        if (root == null)
            return;

        Queue<Node> queue = new LinkedList<>();
        Map<Integer, Integer> map = new TreeMap<>();
        int hd = 0;
        root.hd = hd;

        // Push node and horizontal distance to queue
        queue.add(root);

        System.out.println("The top view of the tree is: ");

        while (!queue.isEmpty()) {
            hd = root.hd;

            // If the map doesn't contain the horizontal
distance, add it to the map
            if (!map.containsKey(hd)) {
                map.put(hd, root.data);
            }

            if (root.left != null) {
                root.left.hd = hd - 1;
                queue.add(root.left);
            }
            if (root.right != null) {
                root.right.hd = hd + 1;
                queue.add(root.right);
            }
            queue.poll();
            root = queue.peek();
        }

        for (int value : map.values()) {
            System.out.print(value + " ");
        }
    }
}
```

Explanation:
The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to put nodes of the same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of the same horizontal distance below it. Hashing is used to check if a node at a given horizontal distance is seen or not.

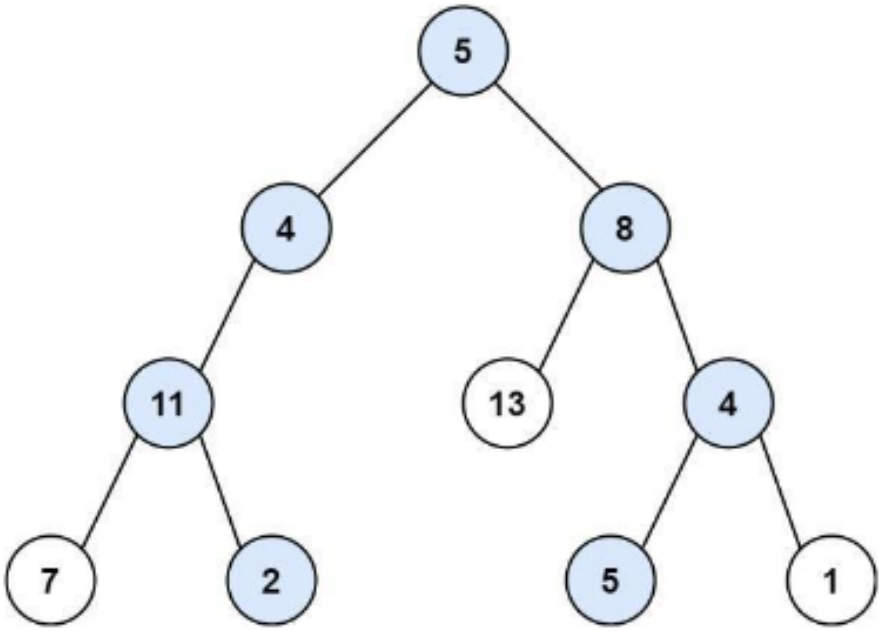
Time complexity: O(N * log(N)), where N is the number of nodes in the given tree.
Auxiliary Space: O(N), As we store nodes in the map and queue.

Q. Path Sum II [LeetCode 113]

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]

Explanation: There are two paths whose sum equals targetSum:
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22

Code:

```
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int
targetSum) {
        List<List<Integer>> resPaths = new ArrayList<>();
        List<Integer> addPath = new ArrayList<>();
        findPaths(root, targetSum, addPath, resPaths);
        return resPaths;
    }
```

```
    private void findPaths(TreeNode root, int sum, List<Integer>
addPath, List<List<Integer>> resPaths) {
        if (root == null) return;

        addPath.add(root.val);
        if (root.val == sum && root.left == null && root.right ==
null) {
            resPaths.add(new ArrayList<>(addPath));
            addPath.remove(addPath.size() - 1); // backtrack by
removing the last element
            return;
        }

        sum -= root.val;
        findPaths(root.left, sum, addPath, resPaths);
        findPaths(root.right, sum, addPath, resPaths);

        addPath.remove(addPath.size() - 1); // backtrack by
removing the last element
    }
}
```

Explanation: This code aims to find all root-to-leaf paths in a binary tree where the sum of node values along the path equals the given target sum. It uses a recursive depth-first search (DFS) approach to traverse the tree, maintaining the current path's sum and nodes.

The `findPaths` function recursively explores the tree, tracking the current path sum and updating the path taken. When a leaf node is reached with a sum equal to the target sum, the path is stored in the result vector.

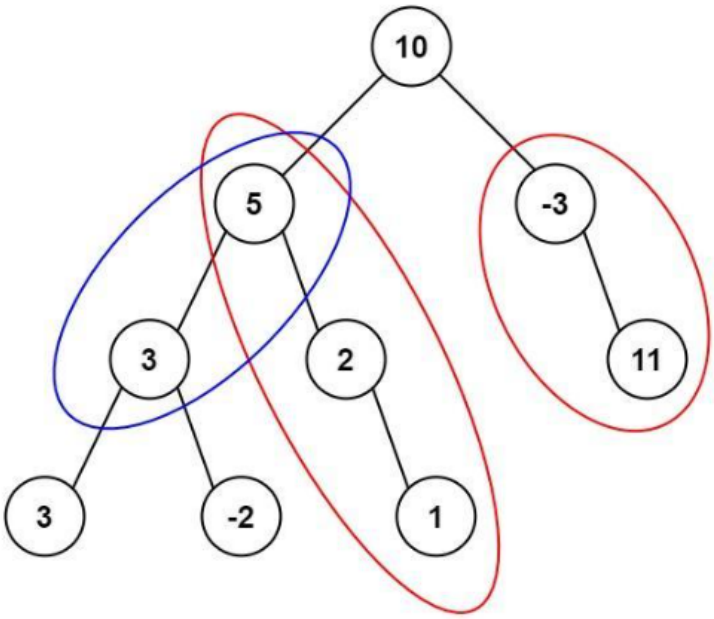
Time Complexity: O(n) - 'n' being the number of nodes in the tree. The algorithm explores each node exactly once, checking the path sum conditions.

Space Complexity: O(h * m) - 'h' is the height of the tree, 'm' is the average number of root-to-leaf paths that sum up to the target. The space includes recursive call stack space and the storage space for the paths found.

Q. Path Sum III [LeetCode 437]

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3

Explanation: The paths that sum to 8 are shown.

Code:

```
public class Solution {
    private int ans = 0;

    private void solve(TreeNode root, int target) {
        if (root == null) return;
        if (target == root.val) ans++;
        solve(root.left, target - root.val);
        solve(root.right, target - root.val);
    }

    public int pathSum(TreeNode root, int targetSum) {
        if (root == null) return 0;
        solve(root, targetSum);
        int leftPathSum = pathSum(root.left, targetSum);
        int rightPathSum = pathSum(root.right, targetSum);
        return ans + leftPathSum + rightPathSum;
    }
}
```

Explanation: Explanation: Recursively take each node as root and check if a path sum equal to target exists or not. If you have any further query, please let me know in the comment section.

Time Complexity: O(n^2), In the worst case, for each node in the tree (`n` nodes), the `solve` function traverses the entire tree again. As a result, for each node, it essentially performs a traversal over the entire tree. Therefore, it results in a nested traversal, making it quadratic.

Space Complexity: O(h), The space complexity here is mainly due to the recursive calls' stack space, which is proportional to the height of the tree (`h`). The recursive function `solve` keeps track of the calls in the stack.

PW SKILLS

THANK YOU !

PW SKILLS