Lesson:

Interview Problems on Binary Trees

PW SKILLS

Pre-Requisites

- Arrays
- Linked List
- Binary trees

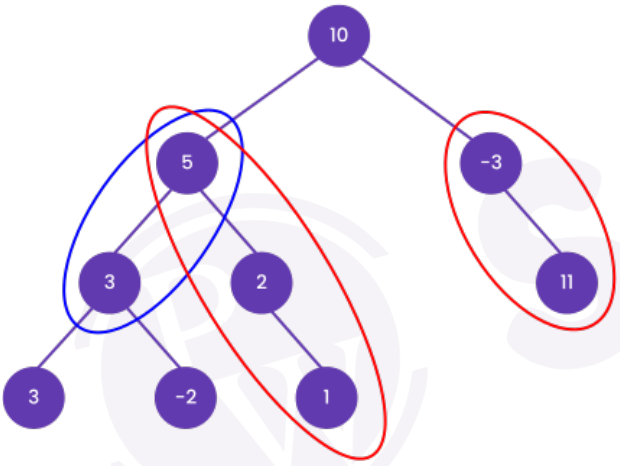
List of Concepts Involved

- Problems based on binary trees.

Problem-1: Leetcode 437. Path Sum III

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



Code Link:https://pastebin.com/wixZ6p9t

Explanation:

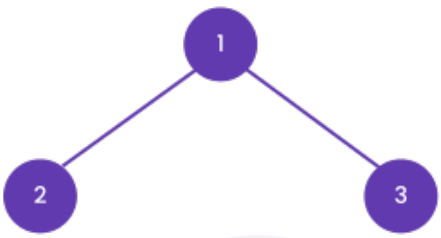
- We maintain a running sum and an unordered map called pathCount. The pathCount map stores the count of each running sum encountered while traversing the tree.
- The idea is to traverse the binary tree in a depth-first manner and at each node, calculate the running sum by adding the node's value to the previous running sum. We then check if there exists a previous running sum (runningSum - target) in the pathCount map. If it exists, it means there is a subpath in the tree whose sum is equal to the target. We increment the count by the value stored in pathCount[runningSum - target].
- After checking for the current node, we update the pathCount map by incrementing the count for the current running sum. This is because we want to keep track of all possible subpaths that can extend from the current node.
- We then recursively process the left and right subtrees, passing the updated running sum and pathCount map. We add the counts returned by the recursive calls to the current count.
- Finally, before backtracking to the parent node, we decrement the count for the current running sum in the pathCount map. This is done to ensure that we don't count paths that go beyond the current node.

Output:

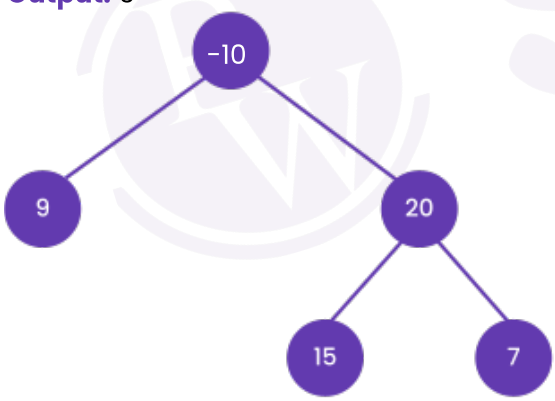
```
Number of paths with sum 8: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Problem-2: Leetcode 124. Binary Tree Maximum Path Sum

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.The path sum of a path is the sum of the node's values in the path.
Given the root of a binary tree, return the maximum path sum of any non-empty path.



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



Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

Code link: https://pastebin.com/h1hW7xn7

Explanation:
To find the maximum path sum, we need to consider two scenarios for each node:

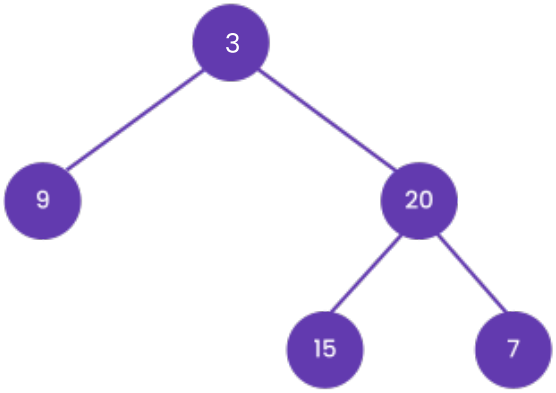
1. The maximum path sum that includes the current node: In this case, we consider the node's value along with the maximum path sum from its left and right subtrees. The maximum path sum can be calculated as the maximum value among the following:
   - Node's value alone
   - Node's value + maximum path sum from the left subtree
   - Node's value + maximum path sum from the right subtree
2. The maximum path sum that goes through the current node: In this case, we consider the node's value along with the maximum path sum from both its left and right subtrees. The maximum path sum can be calculated as the maximum value among the following:
   - Node's value alone
   - Node's value + maximum path sum from the left subtree + maximum path sum from the right subtree

- We update a variable maxSum to keep track of the maximum path sum encountered so far during the traversal.
- The recursive function maxPathSumHelper implements this logic. It takes a node as input and returns the maximum path sum that includes the current node. It also updates maxSum with the maximum path sum seen so far.
- During the recursive traversal, we calculate the maximum path sums for the left and right subtrees by recursively calling maxPathSumHelper. Then, we calculate the maximum path sum for the current node by considering the node's value along with the maximum path sums from its left and right subtrees.
- At each step, we update maxSum with the maximum path sum seen so far. Eventually, when the recursion reaches the leaf nodes, the maximum path sum for the entire tree is stored in maxSum.
- Finally, the maxPathSum function initializes maxSum with the minimum possible value and calls the maxPathSumHelper function on the root node to find the maximum path sum in the tree.

Problem 3: Leetcode 105. Construct Binary Tree from Preorder and Inorder Traversal

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]

Solution:
https://pastebin.com/CgZEJ1MZ

Explanation:

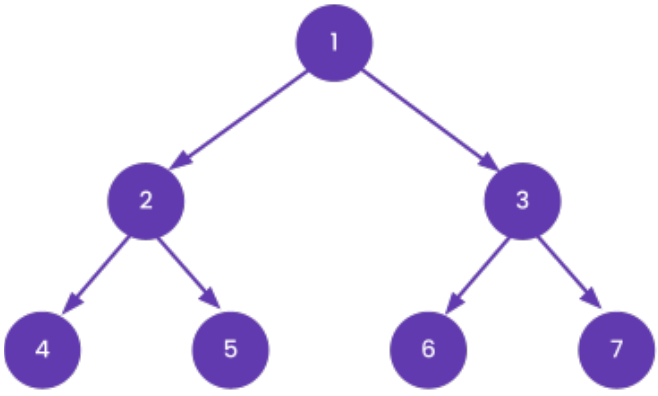
1. **BuildTreeHelper Function:**
   - This recursive helper function constructs the binary tree using preorder and inorder traversals.
   - It takes the following parameters:
     - **preorder:** The vector containing the preorder traversal.
     - **preStart and preEnd:** The starting and ending indices of the current subtree in the preorder traversal.
     - **inorder:** The vector containing the inorder traversal.
     - **inStart and inEnd:** The starting and ending indices of the current subtree in the inorder traversal.
     - **inorderMap:** An unordered map to store the indices of elements in the inorder traversal.
   - **Time Complexity:** The time complexity of this function is O(n), where n is the number of nodes in the binary tree. This is because the function processes each node exactly once.
   - **Space Complexity:** The space complexity is O(n) as well. This is due to the space used by the recursive function stack during the construction of the tree.
2. **buildTree Function:**
   - This function initializes the inorderMap and calls the buildTreeHelper function.
   - It takes the preorder and inorder vectors as input.
   - **Time Complexity:** The time complexity of this function is also O(n), where n is the number of nodes in the binary tree. This is because it builds the inorderMap in O(n) time and then calls the buildTreeHelper function.
   - **Space Complexity:** The space complexity is O(n) as it uses the inorderMap to store the indices of elements in the inorder traversal.

Overall, the code has an efficient time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the binary tree.

Problem 4: Leetcode 889. Construct Binary Tree from Preorder and Postorder Traversal

Given two integer arrays, preorder and postorder where preorder is the preorder traversal of a binary tree of distinct values and postorder is the postorder traversal of the same tree, reconstruct and return the binary tree. If there exist multiple answers, you can return any of them.

Example 1:



Input: preorder = [1,2,4,5,3,6,7], postorder = [4,5,2,6,7,3,1]
Output: [1,2,3,4,5,6,7]

Example 2:
Input: preorder = [1], postorder = [1]
Output: [1]

Code link: https://pastebin.com/kjr1TGTK

Explanation: The constructFromPrePost function takes the preorder and postorder vectors as input and constructs a binary tree using these traversals.

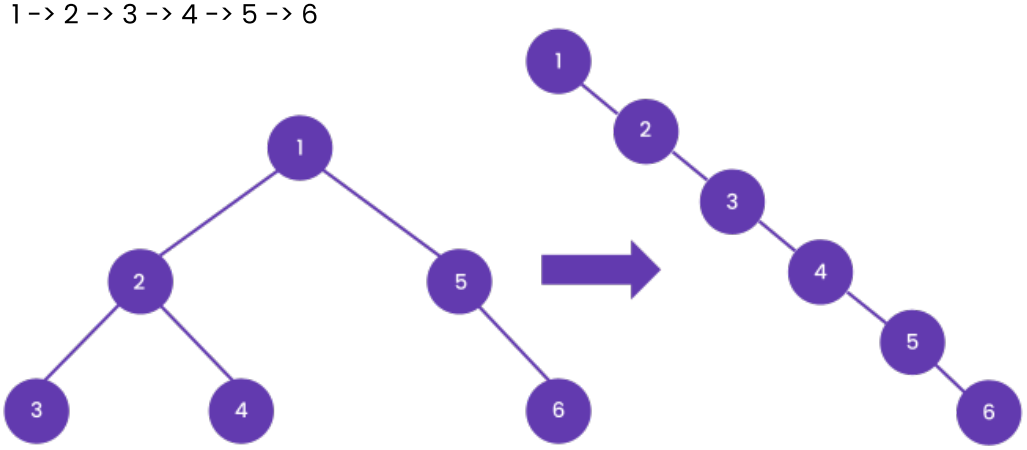
- It initializes a postIndexMap unordered_mapx to store the indices of elements in the postorder traversal for efficient lookup.
- The function then calls the buildTreeHelper function, which is a private helper function that recursively constructs the binary tree.
- The buildTreeHelper function selects the root node based on the first element of the preorder traversal and creates the root node.
- It then recursively constructs the left and right subtrees by determining the sizes of the subtrees using the postorder traversal and the postIndexMap.
- **Time Complexity:** Both the constructFromPrePost and buildTreeHelper functions have a time complexity of O(n), where n is the number of nodes in the binary tree. This is because each node is processed once.
- **Space Complexity:** The space complexity is O(n) as well. This is due to the space used by the recursive function stack during the construction of the tree and the postIndexMap that stores the indices of elements in the postorder traversal.

Problem-5: Leetcode 114. Flatten Binary Tree to Linked List

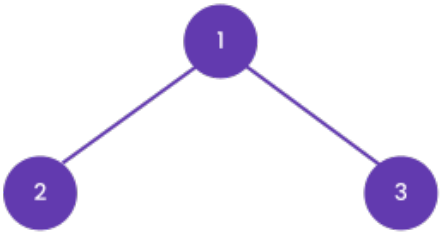
Given the root of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
- The "linked list" should be in the same order as a **pre-order traversal** of the binary tree.

Input1: root = [1,2,5,3,4,null,6]
Output1:
1 -> 2 -> 3 -> 4 -> 5 -> 6



Input2:



Output2:
1 -> 2 -> 3

Code link: https://pastebin.com/Ky7g3E2G

Explanation:

- The idea behind flattening a binary tree to a linked list is to perform a pre-order traversal of the tree and modify the tree in-place to form a linked list.
- Specifically, for each node node in the tree, we recursively flatten the left and right subtrees of node.
- Set the left child of node to null.
- Set the right child of node to the flattened left subtree.
- Append the flattened right subtree to the end of the linked list formed by the flattened left subtree.
- We can perform the above steps recursively until we reach the leaf nodes of the tree.
- In our code, the flatten() function takes the root of the binary tree as input and modifies the tree in-place to form a linked list.
- The function recursively flattens the left and right subtrees of the root, sets the left child of the root to null, sets the right child of the root to the flattened left subtree, and appends the flattened right subtree to the end of the linked list formed by the flattened left subtree.
- The function terminates when it reaches the leaf nodes of the tree.

Problem-6: Leetcode 2385 Amount of time for binary tree to be infected

You are given the root of a binary tree with unique values, and an integer start. At minute 0, an infection starts from the node with value start.
Each minute, a node becomes infected if:

- The node is currently uninfected.
- The node is adjacent to an infected node.

Return the number of minutes needed for the entire tree to be infected.
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]

Code link: https://pastebin.com/urgVGxaF

Explanation:

- To solve the problem, we can use a breadth-first search (BFS) approach.
- Initialize a queue and a set to store infected nodes.
- Enqueue the starting node (with the given start value) into the queue and mark it as infected in the set.
- Initialize a variable minutes to 0, which represents the number of minutes passed.

- While the queue is not empty: Increment minutes by 1 to simulate the passage of time. Process all nodes in the current level of the tree. For each node in the current level. Check if any of its adjacent nodes are uninfected. If an uninfected adjacent node is found. Enqueue the uninfected node into the queue. Mark the uninfected node as infected in the set.
- After the BFS traversal, check if all nodes in the tree are infected.
- If all nodes are infected, return the value of minutes (the number of minutes passed).
- If not all nodes are infected, it means the infection could not reach all nodes, so return -1 to indicate that it is not possible to infect the entire tree.
- The main idea is to simulate the spread of infection starting from the given starting node, using a BFS traversal.
- Each level of the BFS represents one minute passing, and we check for uninfected adjacent nodes to continue the spread.
- By using the queue to process nodes level by level and the set to keep track of infected nodes, we can determine the minimum number of minutes required for the entire tree to be infected.

Upcoming Class Teaser:

- Binary Search Trees