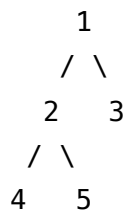
Advanced DSA : Trees 5: Problems on Trees

Problem 1 Invert Binary Tree

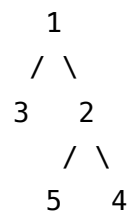
Given the root node of a binary tree, write a function to invert the tree.

Example

Original Binary Tree :

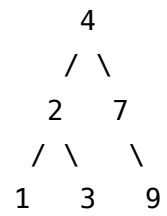


After Inverting the Binary Tree :



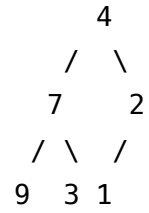
Question

Select the correct inverted binary tree for this given tree:

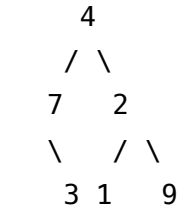


Choices

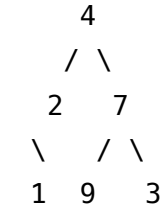
- [ ] **Option 1:**



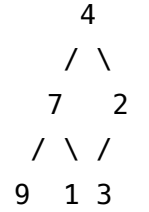
- [ ] **Option 2:**



- [ ] **Option 3:**



- [x] **Option 4:**



:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Solution

To solve this problem,we can recursively invert the binary tree by swapping the left and right subtrees for each node.

```
void invertTree(TreeNode * root) {
    if (root == nullptr) {
        return; // Return if the root is null
    }

    // Use a temporary variable to swap left and right subtrees
    TreeNode * temp = root -> left;
    root -> left = root -> right;
    root -> right = temp;

    // Recursively invert the left and right subtrees
    invertTree(root -> left);
    invertTree(root -> right);
}
```

Complexity

Time Complexity: O(N)

Space Complexity: O(H)

Problem 2 Equal Tree Partition

Given the root of a binary tree, return ***true*** if the tree can be split into two non-empty subtrees with equal sums, or ***false*** otherwise.

Example 1

```
Input:
    5
   / \
  10 10
    /  \
   2   3

Output: True
```

Explanation:

```
    5
   /
  10

Sum: 15


   10
  /  \
 2    3

Sum: 15
```

Example 2

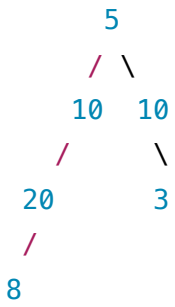
```
Input:
                    1
                   / \
                  2   10
                     /  \
                    2    15
Output: false
```

Explanation:

There is no way to split the tree into two subtrees with equal sums.

Question

Check whether the given tree can be split into two non-empty subtrees with equal sums or not.



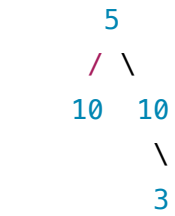
Choices

- [x] Yes, It is possible.
- [ ] It is impossible.

Explanation:

Yes It is possible to split the tree into two non-empty subtrees with sum 28.

Sub-Tree 1:



Sub-Tree 2:

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Solution

1. **Total Sum Check**:
   If the total sum of all nodes in the binary tree is odd, it is impossible to divide the tree into two subtrees with equal sums. This is because the sum of two equal values is always even, and if the total sum is odd, it cannot be divided equally into two parts.
2. **Subtree Sum Check**:
   If we can find a subtree in the binary tree with a sum equal to half of the total sum, we can split the tree into two equal partitions by removing the edge leading to the root of that subtree. This means that we don't necessarily need to compare sums of all possible subtrees, but we can look for a single subtree that meets the subtree sum check condition.

Pseudocode

```
int sum(TreeNode * root) {
    if (!root) {
        return 0;
    }
    return sum(root -> left) + sum(root -> right) + root -> val;
}

bool hasSubtreeWithHalfSum(TreeNode * root, int totalSum) {
    if (!root) {
        return false;
    }

    int leftSum = sum(root -> left);
    int rightSum = sum(root -> right);

    if ((leftSum == totalSum / 2 || rightSum == totalSum / 2) || hasSubtreeWithHalfSum(
        return true;
    }

    return false;
}

bool isEqualTreePartition(TreeNode * root) {
    if (!root) {
        return false; // An empty tree cannot be partitioned
    }

    int totalSum = sum(root);

    if (totalSum % 2 == 1) {
        return false; // If the total sum is odd, partition is not possible
    }

    return hasSubtreeWithHalfSum(root, totalSum);
}
```

Complexity

Time Complexity: O(N)

Space Complexity: O(H)

Problem 3 Next Pointer in Binary Tree

Given a perfect binary tree initially with all next pointers set to nullptr, modify the tree in-place to connect each node's next pointer to the next node in the same level from left to right, following an in-order traversal.

Example

```
Input:
                        1
                      /  \
                     2    3
                    / \  / \
                   4  5 6  7

Output :
                        1 -> nullptr
                      /  \
                     2 -> 3 -> nullptr
                    / \  / \
                   4 -> 5 -> 6 -> 7 -> nullptr
```

:::warning
Please take some time to think about the bruteforce approach on your own before reading further.....
:::

Brute force solution

Level order Traversal :

1. We check if the binary tree is empty; if so, we return the root since there's nothing to connect.
2. A queue is created for level order traversal, initialized with the root node.
3. In the main loop, we process nodes at the current level.
4. At the start of each level, we determine the number of nodes at the current level (levelSize).
5. In the inner loop, we process each node at the current level:
    - We dequeue the current node from the front of the queue.
    - If the current node is not the last node in the level (i.e., i < levelSize - 1), we update its next pointer to point to the front of the queue, which connects nodes from left to right within the same level.

    - We enqueue the left and right children of the current node (if they exist) into the queue for the next level.
6. The loop continues until all levels are processed.
7. Finally, the function returns the modified root of the binary tree, which now has next pointers connecting nodes at the same level, except for the last node in each level, whose next pointer remains nullptr.

Pseudocode :

```
Node * connect(Node * root) {
        // Check if the tree is empty
        if (root is null) {
            return null;
        }

        // Create a queue and enqueue the root
        queue < Node * > q;
        q.push(root);

        // Traverse the tree level by level
        while (!q.empty()) {
            int levelSize = q.size();

            // Process nodes at the current level
            for (int i = 0; i < levelSize; ++i) {
                Node * node = q.front();
                q.pop();

                // Connect the current node to the next node in the same level
                if (i < levelSize - 1) {
                    node -> next = q.front();
                }

                // Enqueue the left and right children (if they exist) for the next lev
                if (node has a left child) {
                    q.push(node 's left child);
                    }
                    if (node has a right child) {
                        q.push(node 's right child);
                        }
                    }
                }

                // Return the modified root
                return root;
            }
```

Complexity

Time Complexity: O(N)

Space Complexity: O(N)

Optimized Solution:

1. We create a dummy node and a temp pointer initially pointing to it.
2. We traverse the tree level by level from left to right.
3. For each node:

- If it has a left child, we connect the temp node's next pointer to the left child and update temp.
- If it has a right child, we connect the temp node's next pointer to the right child and update temp.

4. Level Completion:
5. When the current level is done, we move to the next level by updating root to the dummy node's next. We reset dummy's next and reset temp to the dummy node.
6. We repeat these steps until all levels are traversed.
7. The loop ends when there are no more levels to traverse.

Pseudocode

```
void populateNextPointers(Node * root) {
        if (!root) {
            return;
        }

        Node * dummy = new Node(-1);
        Node * temp = dummy;

        while (root != nullptr) {
            if (root -> left != nullptr) {
                temp -> next = root -> left;
                temp = temp -> next;
            }
            if (root -> right != nullptr) {
                temp -> next = root -> right;
                temp = temp -> next;
            }
            root = root -> next;
            if (root == nullptr) {
                root = dummy -> next;
                dummy -> next = nullptr;
                temp = dummy;
            }
        }
```

Complexity

Time Complexity: O(N)

Space Complexity: O(1)

Problem 4 Check if Root to Leaf Path Sum Equals to K

Given a binary tree and an integer k, determine if there exists a root-to-leaf path in the tree such that adding up all the node values along the path equals k.

Example:

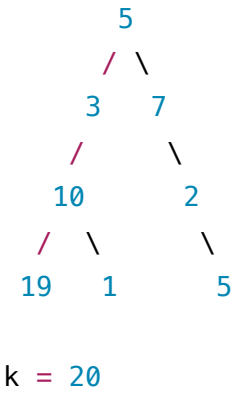
```
Input:
                Binary Tree:
                       5
                      / \
                     4   8
                    /   / \
                   11  13  4
                  /  \      \
                 7    2      1

                k = 22

Output: true
```

Explanation:

In the given binary tree, there exists a root-to-leaf path 5 -> 4 -> 11 -> 2 with a sum of 5 + 4 + 11 + 2 = 22, which equals k. Therefore, the function should return true.

Question

Tell if there exists a root to leaf path with sum value `k = 19`



Choices

- [x] true
- [ ] false

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Solution:

- To solve this problem,we first check if the current node is a leaf node (having no left and right children) and if the current value equals k. If both conditions are met, it returns true, indicating that a valid path is found.
- If not, it recursively checks the left and right subtrees with a reduced sum (k - root->val).
- It returns true if there's a path in either the left or right subtree, indicating that a valid path is found.

Pseudocode

```
bool hasPathSum(TreeNode * root, int k) {
    if (!root) {
        return false; // No path if the tree is empty
    }

    if (!root -> left && !root -> right) {
        return (k == root -> val);
    }

    return hasPathSum(root -> left, k - root -> val) || hasPathSum(root -> right, k - r
}
```

Complexity

Time Complexity: O(N)
Space Complexity: O(H)

Problem 5 Diameter of Binary Tree

Given a binary tree, find the length of the longest path between any two nodes in the tree. This path may or may not pass through the root.

Definition of Diameter: The diameter of a binary tree is defined as the number of nodes along the longest path between any two leaf nodes in the tree. This path may or may not pass through the root.

Question

How would you find the diameter of a binary tree?

Choices

- [ ] Add the height of the left and right subtrees.
- [ ] Count the number of nodes in the tree.
- [x] The maximum of the following three: Diameter of the left subtree, Diameter of the right subtree, Sum of the heights of the left and right subtrees plus one
- [ ] Divide the height of the tree by 2.

Example:

Example that illustrates that the diameter of the tree can pass through a root node.

```
Input:

                    1
                   / \
                  2   3
                 / \
                4   5

Output: 4
```

Explanation:

The diameter of the binary tree shown above is the path 4 -> 2 -> 1 -> 3, which contains four nodes.

Example:

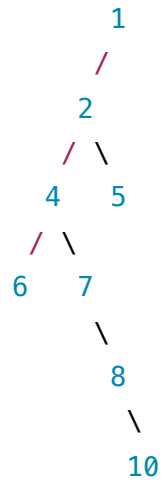
Example that illustrates that the diameter of the tree can pass through a non-root node:

```
Input:
                    1
                   /
                  2
                 / \
                4   5
               / \   \
              6   7   3

Output: 5
```

Explanation:
The diameter of the binary tree shown above is the path 6 - 4 - 2 - 5 - 3, which includes 5 nodes.

Question

What is the diameter of the Given Binary Tree.



Choices

- [x] 6
- [ ] 5
- [ ] 7
- [ ] 4

Explanation:

The path 1 -> 2 -> 4 -> 7 -> 8 -> 10 has 6 nodes, which is the diameter of the given tree.

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Solution:

1. To solve this problem,we initialize a diameter variable to 0 to track the maximum diameter.
2. Define a helper function that recursively computes both the height of the tree and the diameter
3. In the helper function:
    - If the node is null, we return -1 to signify no height.
    - We recursively find left and right subtree heights.
    - We update diameter with the maximum diameter found, including the current node and connecting path (2 units).
    - The height of the current node is the max of left and right subtree heights, plus 1.
    - Call the helper function with the root of the binary tree from the main function or method.
4. Retrieve and use the maximum diameter found during traversal as the result.

Pseudocode:

```
int diameterOfBinaryTree(TreeNode * root) {
    int diameter = 0;

    // Helper function to calculate height and update diameter
    std:: function < int(TreeNode * ) > calculateHeight = [ & ](TreeNode * node) {
        if (!node) {
            return -1; // Height of a null node is -1
        }

        int leftHeight = calculateHeight(node -> left);
        int rightHeight = calculateHeight(node -> right);

        // Update diameter with the current node's diameter
        diameter = std::max(diameter, leftHeight + rightHeight + 2);

        // Return the height of the current node
        return std::max(leftHeight, rightHeight) + 1;
    };

    calculateHeight(root); // Start the recursive calculation

    return diameter;
}
```

Complexity

Time Complexity: O(N)

Space Complexity: O(H)