COL106 : Data Structures and Algorithms, Semester II 2024-25
Practice Programming Questions Trees and Binary tree

January 2025

Instructions

- Please use the following questions as practice questions for learning about List datastructure.
- The questions with * next to them should be attempted during the lab sessions, and your solutions must be uploaded on moodlenew. Note that your submissions will not be evaluated, but will be used as a mark of your attendance. We will filter out all the submissions that are not from the lab workstations. So do not use your laptops for submitting the programs.

Questions

A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and right child. It is widely used in applications such as searching, sorting, and hierarchical data representation. Special types of binary trees include Binary Search Trees (BSTs), where the left child contains values smaller than the parent and the right child contains values greater than the parent, and Balanced Trees, which maintain a height balance to ensure efficient operations. Here's a simple implementation of a binary tree Node in **Java**:

```java
class Node {
    int value;
    Node left, right;

    public Node(int value) {
        this.value = value;
        left = right = null;
    }
}
```

Listing 1: Binary Tree Implementation

1. * **Construct BT using traversal**

   Given two integer arrays `preorder` and `inorder`, where:

   - `preorder` is the preorder traversal of a binary tree (BT).
   - `inorder` is the inorder traversal of the same BT.

   Construct and return the `root` of the BT.

   Example

```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output:
    3
   / \
  9   20
     /  \
    15   7

The preorder traversal is [3, 9, 20, 15, 7].
The inorder traversal is [9, 3, 15, 20, 7].
```

2. ***ZigZag Level order traversal**

   Given the `root` of a binary tree, return the **zigzag level order traversal** of its nodes' values. This traversal alternates between left-to-right and right-to-left for each level.

   Input Format The input is given as a reference to the `root` of a binary tree. The tree is represented by nodes, where each node contains the following:

   - `val` (an integer): The value of the node.
   - `left` (a reference): The left child of the node.
   - `right` (a reference): The right child of the node.

Examples

Example 1:

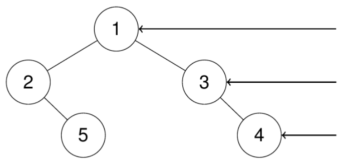
```
Input: root = [3,9,20,null,null,15,7]
Output: [3,20,9,15,7]

Explanation:
    3
   / \
  9   20
     /  \
    15   7

The zigzag level order traversal is:
Level 0: [3] (left-to-right)
Level 1: [20, 9] (right-to-left)
Level 2: [15, 7] (left-to-right)
```

3. *** Binary Tree Right Side View**

   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.

   Example

   Input:



Output:

$$[1, 3, 4]$$

From the right side, the visible nodes are $[1, 3, 4]$.

4. ***Validate Binary Tree** Given the `root` of a binary tree, determine if it is a valid binary search tree (BST). A valid binary search tree is defined as follows:

   - The left subtree of a node contains only nodes with keys **less than** the node's key.
   - The right subtree of a node contains only nodes with keys **greater than** the node's key.
   - Both the left and right subtrees must also be binary search trees.

   Input Format The input is given as a reference to the root node of a binary tree. **Example**

```
Input: root = [2, 1, 3]
Output: true

Explanation:
    2
   / \
  1   3

The given tree is a valid binary search tree.
```

   Example 2:

```
Input: root = [5, 1, 4, null, null, 3, 6]
Output: false

Explanation:
    5
   / \
  1   4
     / \
    3   6

The root node's right child (4) contains a value less than the root (5),
violating the BST property.
```

5. **Merge BSTs** You are given n Binary Search Tree (BST) root nodes for n separate BSTs stored in an array `trees` (0-indexed). Each BST in `trees` has at most 3 nodes, and no two roots have the same value.

   In one operation, you can:

   - Select two distinct indices i and j such that the value stored at one of the leaf nodes of `trees[i]` is equal to the root value of `trees[j]`.

- Replace the leaf node in `trees[i]` with `trees[j]`.
- Remove `trees[j]` from `trees`.

Return the root of the resulting BST if it is possible to form a valid BST after performing $n-1$ operations. If it is impossible to create a valid BST, return `null`.

Input Format

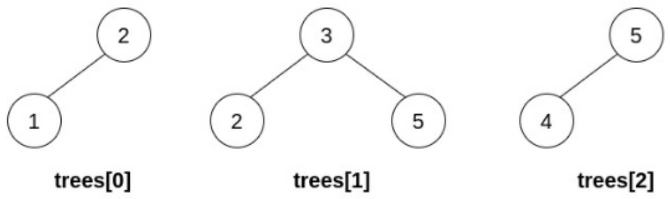
- `trees`: An array of `TreeNode` objects where each element represents the root of a BST.
  - Each BST has at most 3 nodes.
  - No two roots have the same value.

Output Format

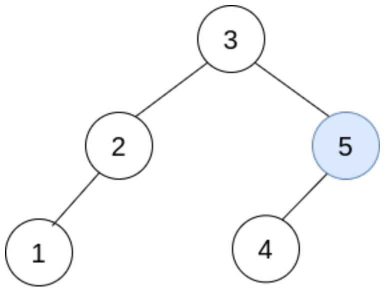
Return the root node of the resulting BST if it is possible to combine all the given BSTs into a single valid BST. Otherwise, return `null`.

Example

- **input**



- **output**



6. **k^{th} smallest element of a binary search tree:**

   Given the root of a binary search tree and an integer k, return the k^{th} smallest value among all the node values in the tree.

   Input Format: The input is given as a reference to the `root` of a binary tree and an integer k. The tree is represented by nodes, where each node contains the following:

   - `val` (an integer): The value of the node.
   - `left` (a reference): The left child of the node.
   - `right` (a reference): The right child of the node.

Output Format: Your function should return an integers, the k^{th} smallest element of the BST. It is guaranteed that a suitable k^{th} smallest element will always exist.

Example 1:

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3

Explanation:
        5
       / \
      3   6
     / \
    2   4
   /
  1
```

The 3rd smallest element is 3.

Example 2:

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

Explanation:
        3
       / \
      1   4
       \
        2
```

The 1st smallest element is 1.

7. **Maximum Path Sum:**

   In a tree, a path is a sequence of nodes where each consecutive pair is directly connected by an edge, and no node appears more than once in the sequence. It is not necessary that the path includes the root. The sum of a path is the total of all node values in that path.

   Given the root of a binary tree, find and return the highest possible sum of any non-empty path.

   Input Format: The input is given as a reference to the `root` of a binary tree. The tree is represented by nodes, where each node contains the following:

   - `val` (an integer): The value of the node.
   - `left` (a reference): The left child of the node.
   - `right` (a reference): The right child of the node.

   Output Format: Your function should return a single integer which is the maximum sum of any non-empty path. Return 0 if no non-empty path exists.

   Example 1:

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

```
Explanation:
    1
   / \
  2   3

The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:

```
Input: root = [-5, 1, 4, null, null, 3, 6]
Output: 13

Explanation:
    -5
   / \
  1   4
     / \
    3   6

The optimal path is 3 -> 4 -> 6 with a path sum of 3 + 4 + 6 = 13.
```

Follow Up Question: The diameter/width of a tree is defined as the number of edges on the longest path between any two nodes.

```
In Example1 above, it is 2 for the longest path 2 -> 1 -> 3.
For Example2, it is 3 for the longest paths 1 -> -5 -> 4 -> 3 or 1 -> -5 -> 4 -> 6.
```

Can you use the solution for the maximum path sum problem (with slight modification) to find the length of diameter of any n-ary tree?

8. **k^{th} ancestor of a tree node:**

You are given a tree with n nodes numbered from 0 to $n-1$ in the form of a parent array `parent`, where `parent[i]` is the parent of the i^{th} node. The root of the tree is node 0. Find the k^{th} ancestor of a given node.

The k^{th} ancestor of a tree node is the k^{th} node in the path from that node to the root node.

Input Format:

- `parent[]` (an array of integers): `parent[i]` is the parent of the i^{th} node. The root of the tree is node 0
- `findAncestorArray[][]` (a 2-d array of integers): Array of ($\texttt{node}, k$) values, where the k^{th} ancestor of the `node` needs to be found. Return -1 if k^{th} ancestor doesn't exist.

Output Format: Your function should return an array of integers, where the i^{th} element represents the answer to the i^{th} query of `findAncestorArray`.

Example 1:

```
Input: parents = [-1, 0, 0, 1, 1, 2, 2], findAncestorArray = [[3, 1], [5, 2], [6, 3]]
Output: [1,0,-1]

Explanation:
    0
    / \
   1   2
  / \ / \
 3  4 5  6
```

The first ancestor of 3 is 1, second ancestor of 5 is 0 and third ancestor of 6 doesn't exist.

9. **Maximum distance for each node:**

   You are given a tree consisting of n nodes. Your task is to determine for each node the maximum distance to any another node.

   Input Format:

   - `parent[]` (an array of integers): `parent[i]` is the parent of the i^{th} node. The root of the tree is node 0

   Output Format: Your function should return an array of integers, where the i^{th} element represents the maximum distance from i^{th} node to any other node.

   Example 1:

```
Input: parent = [-1, 0, 0, 2, 2], Output: [2,3,2,3,3]

Explanation:
    0
   / \
  1   2
     / \
    3   4
```

   Farthest from 0 are 3 or 4, from 1 are 3 or 4, from 2 is 1, and for 3 and 4 is 1. Therefore, we get farthest distances as 2,3,2,3 and 3 respectively.