Binary Trees- 2

Construct a Binary Tree from Preorder and Inorder Traversal

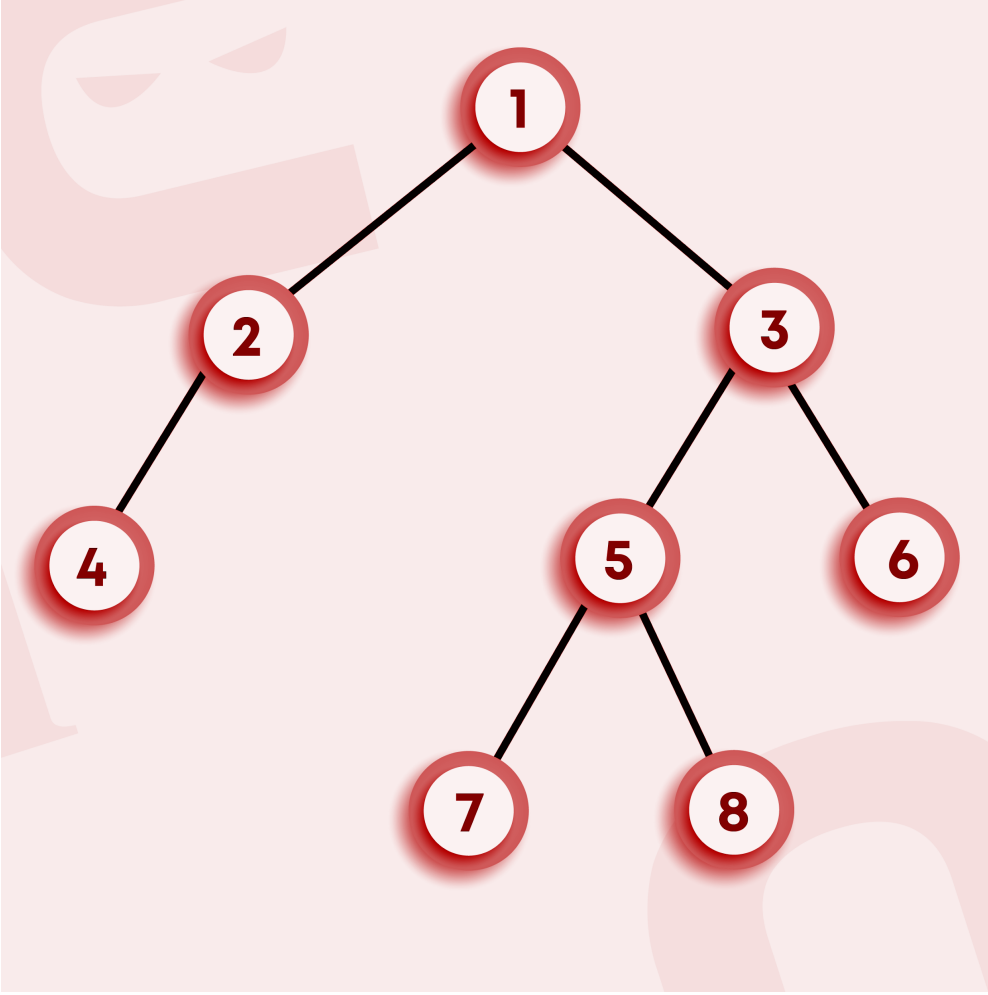
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



- The idea is to start with the root node, which would be the first item in the preorder sequence, and find the boundary of its left and right subtree in the inorder array.
- Now, all keys before the root node in the inorder array become part of the left subtree, and all the indices after the root node become part of the right subtree.
- We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-
Inorder: {4, 2, 1, 7, 5, 8, 3, 6}
Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

- The root will be the first element in the preorder sequence, i.e. 1.
- Next, we locate the index of the root node in the inorder sequence.
- Since 1 is the root node, all nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}.
- Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Thus we get:

Left subtree:	**Right subtree:**
Inorder : {4, 2}	**Inorder :** {7, 5, 8, 3, 6}
Preorder : {2, 4}	**Preorder :** {3, 5, 7, 8, 6}

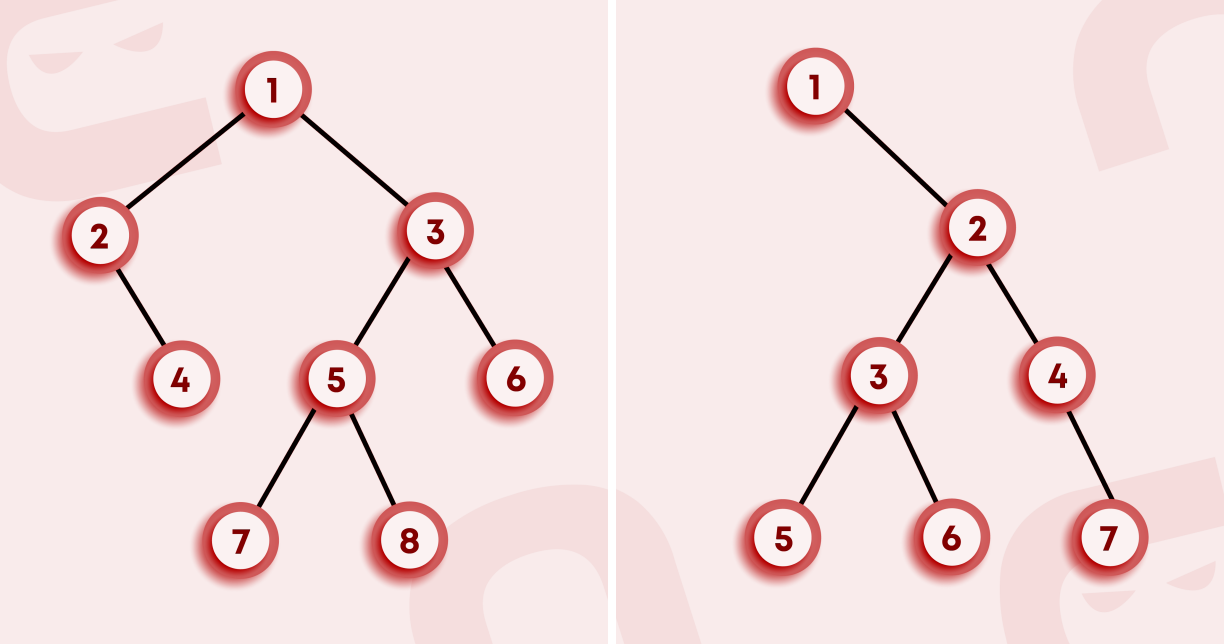
Follow the above approach until the complete tree is constructed. Now let us look at the code for this problem.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

The Diameter of a Binary tree

- The **diameter** of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes.
- The diameter of the binary tree may pass through the root (not necessary).

For example, the figure below shows two binary trees having diameters 6 and 5, respectively. The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```
def height(node):

    # Base Case : Tree is empty
    if node is None:
        return 0

    # If tree is not empty then height = 1 + max of left
    # height and right heights
    return 1 + max(height(node.left), height(node.right))

def diameter(root):

    # Base Case when tree is empty
    if root is None:
        return 0

    # Get the height of left and right subtrees
    leftH= height(root.left)
    rightH = height(root.right)

    # Get the diameter of left and right subtrees
    leftD = diameter(root.left)
    rightD = diameter(root.right)

    # Return max of the three
    return max(leftH + rightH + 1, max(leftD, rightD))
```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be **O(n).**
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to **O(n*h)**. (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, **h** is the height of the tree, which could be **O(n^2)**.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra **n** traversals for each node.

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find the height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a NULL tree, height and diameter both are equal to 0. Hence, the pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = `max(leftHeight, rightHeight)`

Diameter = `max(leftHeight + rightHeight, leftDiameter, rightDiameter)`

To create a pair class, follow the syntax below:

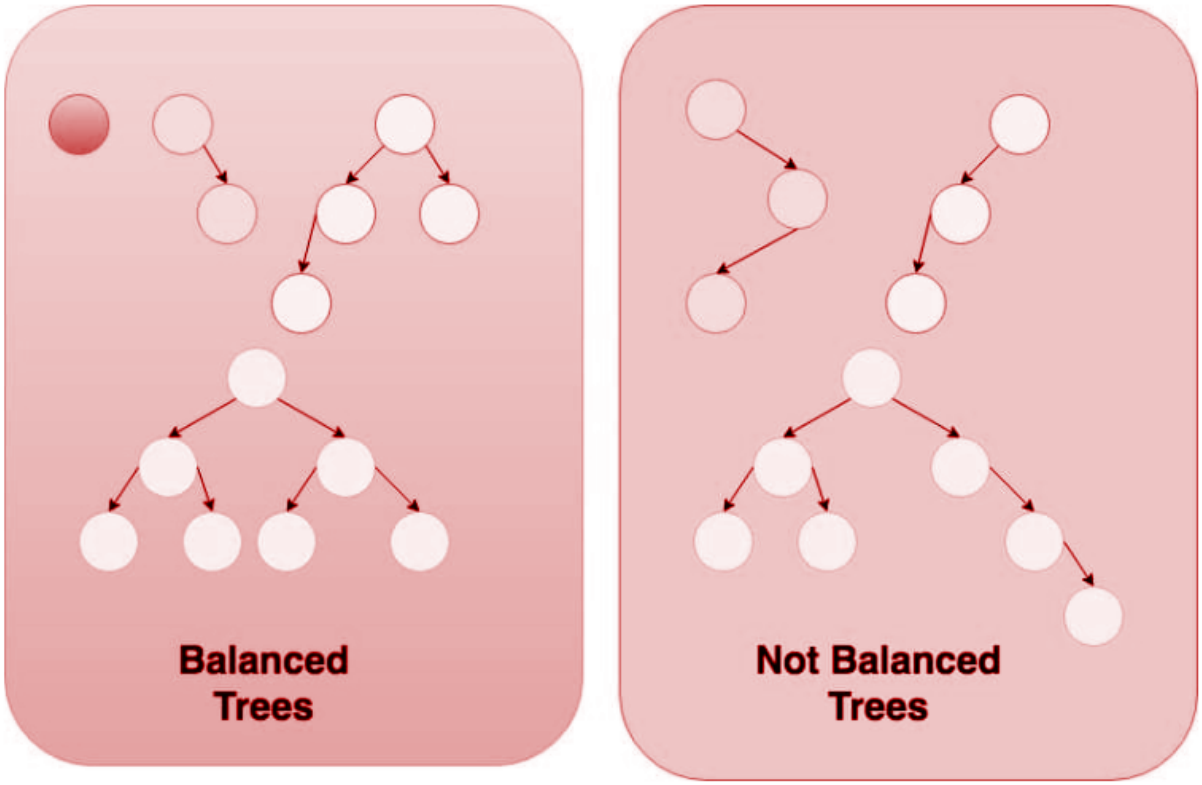
```
class Pair:
    def __init__(self, data): # Constructor to create a new Pair
        self.data = data
        self.left = self.right = None
```

It can be observed that we are just traversing each node once while making recursive calls and the rest of all other operations are performed in constant time, hence the time complexity of this program is **O(n)**, where **n** is the number of nodes.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Balanced Binary Tree

A binary tree is balanced if, for each node in the tree, the difference between the height of the right subtree and the left subtree is at most one.



Check if a Binary Tree is Balanced

From the definition of a balanced tree, we can conclude that a binary tree is balanced if at every node in the tree:

- The right subtree is balanced.
- The left subtree is balanced.
- The difference between the height of the left subtree and the right subtree is at most 1

Let's define a recursive function **is_balanced()** that takes a root node as an argument and returns a boolean value that represents whether the tree is balanced or not.

Let's also define a helper function **get_height()** that returns the height of a tree. Notice that **get_height()** is also implemented recursively

```
def get_height(root):#Returns height of the tree
    if root is None:
        return 0
    return 1 + max(get_height(root.left , get_height(root.right))

def is_balanced(root):
    # a None tree is balanced
    if root is None:
        return True
    return is_balanced(root.right) and is_balanced(root.left) and
abs(get_height(root.left) - get_height(root.right)) <= 1
```

The **is_balanced()** function returns true if the right subtree and the left subtree are balanced, and if the difference between their height does not exceed 1.

Check if a Binary Tree is Balanced- Improved Solution

Here, we are using two recursive functions: one that checks if a tree is balanced, and another one that returns the height of a tree. Can we achieve the same goal by using only one recursive function?

Let us define our recursive function **is_balanced_helper()** to be a function that takes one argument, the tree root and returns an integer such that:

- If the tree is balanced, return the height of the tree
- If the tree is not balanced, return -1

Notice that this new **is_balanced_helper()** can be easily implemented recursively as well by following these rules:

- Apply is_balanced_helper on both the right and left subtrees
- If either the right or left subtrees returns -1, then we should return -1 (because our tree is not balanced if either subtree is not balanced)
- If both subtrees return an integer value (indicating the heights of the subtrees), then we check the difference between these heights.
- If the difference doesn't exceed 1, then we return the height of this tree. Otherwise, we return -1

Let us now look at its implementation:

```
def is_balanced_helper(root):
    if root is None: # a None tree is balanced
        return 0

 # if the left subtree is not balanced, then tree is not balanced
    left_height = is_balanced_helper(root.left)
    if left_height == -1:
        return -1

    # if the right subtree is not balanced,then tree is not balanced
    right_height = is_balanced_helper(root.right)
    if right_height == -1:
        return -1

    # If the difference in heights is greater than 1
    if abs(left_height - right_height) > 1:
        return -1

    # this tree is balanced, return its height
    return max(left_height, right_height) + 1
```

Finally we know that, if **is_balanced_helper()** returns a number that is greater than -1, the tree is balanced, otherwise, it is not.

```
def is_balanced(root):
    return is_balanced_helper(root) > -1
```

Practice problems:

- https://www.hackerrank.com/challenges/tree-top-view/problem
- https://www.codechef.com/problems/BTREEKK
- https://www.spoj.com/problems/TREEVERSE/
- https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/approximate/largest-cycle-in-a-tree-9113b3ab/