654. Maximum Binary Tree

The **key observation** to develop the intuition for this algorithm is understanding the relationship between **monotonic properties** and the **maximum binary tree construction**. Here's the breakdown:

Key Observations:

1. **Parent-Child Relationship Based on Maximum Value**:
    - In the maximum binary tree, the largest number in a subarray becomes the **parent** of all other numbers in that subarray.
    - All smaller numbers to the left of the maximum value form the **left subtree**, and all smaller numbers to the right form the **right subtree**.
    - Thus, a number is only a **direct child** of the closest larger number.

---

1. **Direct Neighboring Relationships**:
    - For a given number, its **parent** will always be the **nearest larger number** (if it exists).
    - The **left child** of a number is the largest number seen **before** it (to the left in the stack).
    - The **right child** of a number is the smallest number **yet to be processed** (stack ensures this order).

---

1. **Stack Properties to Maintain Monotonicity**:
    - A monotonic decreasing stack can track the largest number seen so far:
        - While processing the array from left to right, maintain a stack where the elements are in decreasing order.
        - This ensures that when a new number is encountered:
            - All smaller numbers in the stack will become the **left child** of the new number.

        - The top of the stack after popping (if it exists) will become the **right child** of the new number.

---

1. **Efficiency of the Monotonic Stack**:
    - A monotonic stack avoids repeatedly scanning the array for maximum values:
        - Each number is pushed to and popped from the stack exactly once.
        - This avoids the need for recursive or brute-force operations.

Summary of the Insight:

The core idea is:

- The **parent of a node** is the **nearest larger number** encountered while processing the array.
- The **left child** and **right child** relationships can be determined dynamically by maintaining a **monotonic decreasing stack**.

This observation minimizes the need for recursion or explicit subarray splitting and enables an efficient O(n)O(n)O(n) algorithm.

```
# Tc and Sc: O(n).
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> Opt

        stack_of_nodes = []
```

```
        for num in nums:
            node = TreeNode(num)

            # Set the smaller nodes as the left child
            while stack_of_nodes and stack_of_nodes[-1].val < nu
                node.left = stack_of_nodes.pop()

            # Set the current node as the right child
            if stack_of_nodes:
                # stack_of_nodes[-1] will be the max number
                stack_of_nodes[-1].right = node

            stack_of_nodes.append(node)

        # The root of the tree is the first node added to the st
        return stack_of_nodes[0]
```