Design a Special Stack to Get Minimum value in O(1) Time Complexity CodeStudio

You are given the task of implementing a stack data structure that supports the following operations efficiently: push, pop, getTop, and getMin. The objective is to design each approach in a way that these operations can be performed with O(1) time complexity while maintaining efficient space usage.

Approach 1: MinStack class using two stacks to track minimum value

In this approach, two separate stacks are used to maintain the elements of the main stack and the corresponding minimum values at each step. When an element is pushed onto the stack, the minimum value is updated in the auxiliary minimum stack.

Functions:

- **Push Operation:**
    - Description: Pushes the given value onto the main stack and updates the minimum value stack.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **Pop Operation:**
    - Description: Pops the top element from both the main stack and the minimum value stack.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **GetTop Operation:**
    - Description: Returns the top element of the main stack without removing it.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
- **GetMin Operation:**
    - Description: Returns the minimum value present in the minimum value stack.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**

Approach 2: MinStackDifference class using a single stack and difference to track minimum value

In this approach, a single stack is used to store the elements, along with a variable to track the current minimum value. When pushing an element, the difference between the element and the current minimum is stored if the element is smaller than the current minimum.

Functions:

- **Push Operation:**
    - Description: Pushes the given value onto the stack while updating the current minimum value.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **Pop Operation:**
    - Description: Pops the top element from the stack and updates the current minimum value if needed.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **GetTop Operation:**
    - Description: Returns the top element of the stack without removing it.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
- **GetMin Operation:**
    - Description: Returns the current minimum value stored in the variable.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**

Approach 3: SpecialStackPair class using pairs to track both the element and the minimum value

This approach uses a single stack to store pairs of elements and their corresponding minimum values. When pushing an element, the minimum value is updated and stored along with the element as a pair.

Functions:

- **Push Operation:**

    - Description: Pushes the given value onto the stack along with the corresponding minimum value.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **Pop Operation:**
    - Description: Pops the top element from the stack and updates the current minimum value accordingly.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(N)**
- **GetTop Operation:**
    - Description: Returns the top element of the stack without removing it.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
- **GetMin Operation:**
    - Description: Returns the minimum value stored in the top pair of the stack.
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**

The best Approach:

- If memory usage is not a concern and you want a straightforward implementation, the **MinStack using Two Stacks** might be a good choice.
- If you are looking for a balance between space complexity and ease of implementation, the **MinStackDifference using a Single Stack** could be suitable.
- If you want to avoid storing extra values (differences or separate stacks) and still have a manageable space complexity, the **SpecialStackPair using Pairs** approach might be appealing.