Push Element At Bottom of Stack CodeStudio

This program illustrates two approaches for adding an element at the bottom of a stack: a recursive approach and an iterative approach. It includes a main function that demonstrates these approaches on a stack containing elements.

Example: Consider the initial stack: 6 5 4 3 2 1

Output: 6 5 4 3 2 1 0

Approach 1: Function to push the given value at the bottom of the stack using recursion

In the recursive approach to push an element at the bottom of a stack, the **pushAtBottomRecusively** function is employed. This function uses recursion to rearrange the stack elements by popping elements and pushing them back onto the stack after the target element has been placed at the bottom. This process effectively places the element at the desired position within the stack.

Time Complexity:

- The function makes a recursive call for each element in the stack, involving a constant amount of work (popping and pushing).
- As a result, the time complexity of this approach is linear, directly proportional to the number of elements in the stack.
- **Time Complexity**: **O(n), where n is the number of elements in the stack.**

Space Complexity:

- The space complexity is influenced by the maximum depth of the call stack due to recursive function calls.
- In the worst case, each recursive call is stored in the call stack, leading to a depth of **n** (number of elements in the stack).
- **Space Complexity**: **O(n), due to the call stack space required for recursion.**

Approach 2: Function to push the given value at the bottom of the stack using iteration

The iterative approach involves using an auxiliary stack (**tempStack**) to temporarily store elements while reordering the original stack. This allows the addition of an element at the bottom by carefully transferring elements back and forth between the original stack and the temporary stack.

Time Complexity:

- The approach requires two iterations over the elements in the stack: one for transferring elements to the temporary stack and another for transferring elements back to the original stack.
- Both iterations are linear in nature, directly proportional to the number of elements in the stack.
- **Time Complexity**: **O(n), where n is the number of elements in the stack.**

Space Complexity:

- The space complexity is influenced by the space required to hold the temporary stack (**tempStack**).
- In the worst case, when all elements are present in the initial stack, the temporary stack holds the same number of elements.
- **Space Complexity**: **O(n), due to the temporary stack's space requirement.**