Implement Two Stacks using One Array CodeStudio

You are tasked with creating a data structure to implement two stacks, Stack A and Stack B, using a single array. Each stack should have its own set of operations for manipulation.

Stack A:

1. **pushA(num)**: Add the integer **num** to Stack A.
2. **popA()**: Remove and return the top element from Stack A. If the stack is empty, return -1.
3. **getTopA()**: Return the top element of Stack A without removing it. If the stack is empty, return -1.
4. **getSizeA()**: Return the number of elements in Stack A.

Stack B:

1. **pushB(num)**: Add the integer **num** to Stack B.
2. **popB()**: Remove and return the top element from Stack B. If the stack is empty, return -1.
3. **getTopB()**: Return the top element of Stack B without removing it. If the stack is empty, return -1.
4. **getSizeB()**: Return the number of elements in Stack B.

Both stacks share the same underlying array and are implemented using it.

Approach 1: Implement Two stacks in One Array by dividing Array in two halves

In this approach, the array is divided into two halves, and each stack occupies one half. The class **TwoStackDivided** is implemented to maintain two stacks within the same array. The size of each stack is determined by dividing the array size by 2 (**mid**). Two pointers, **top1** and **top2**, are used to keep track of the top elements of the two stacks.

- **push1(int num)**: Pushes an element onto stack 1 if there is space available.
- **push2(int num)**: Pushes an element onto stack 2 if there is space available.
- **pop1()**: Pops an element from stack 1 if it's not empty.
- **pop2()**: Pops an element from stack 2 if it's not empty.
- **getTop1()**: Returns the top element of stack 1.
- **getTop2()**: Returns the top element of stack 2.
- **getSize1()**: Returns the size of stack 1.

- **getSize2()**: Returns the size of stack 2.

Approach 2: Implement Two Stacks Using a Single Array with Shared Space

In this approach, both stacks share the same space within the array. The class **TwoStack** is implemented to maintain two stacks using shared space within the array. Two pointers, **top1** and **top2**, are used to keep track of the top elements of the two stacks.

- **push1(int num)**: Pushes an element onto stack 1 if there is space available.
- **push2(int num)**: Pushes an element onto stack 2 if there is space available.
- **pop1()**: Pops an element from stack 1 if it's not empty.
- **pop2()**: Pops an element from stack 2 if it's not empty.
- **getTop1()**: Returns the top element of stack 1.
- **getTop2()**: Returns the top element of stack 2.
- **getSize1()**: Returns the size of stack 1.
- **getSize2()**: Returns the size of stack 2.

Which approach to use:

The second approach, which involves implementing two stacks using a single array with shared space, is generally more desirable due to its flexibility and better space utilization compared to the first approach of dividing the array into two halves.

Advantages of the Second Approach (Shared Space):

1. **Space Utilization:** In the second approach, the two stacks share the same array space, allowing them to utilize the available memory more efficiently. This is particularly advantageous when the sizes of the two stacks are not balanced. In contrast, the first approach divides the array into equal halves, which can lead to inefficient space utilization if one stack requires more space than the other.
2. **Flexibility:** The second approach allows for dynamic allocation of space between the two stacks. If one stack needs more space than the other, it can utilize the available space, as long as the total space limit is not exceeded. This flexibility can be important in scenarios where the usage patterns of the two stacks are unpredictable or can change over time.

Issues with the First Approach (Dividing Array in Two Halves):

1. **Balancing Stacks:** In the first approach, if one stack requires more space than the other, it can result in inefficient space utilization. If stack 1 requires more space than

its allocated half, it cannot utilize the unused space from stack 2, leading to wasted memory.

2. **Fixed Allocation:** The first approach pre-allocates equal space for both stacks, which can lead to inefficient memory usage if the sizes of the two stacks are significantly different. This can limit the overall capacity of the stacks.
3. **Limitations on Stack Sizes:** If one stack reaches its allocated half of the array, it cannot grow further, even if the other stack is small or empty. This can limit the usability of the stacks in scenarios where one stack's size varies significantly.

In summary, the second approach provides greater flexibility and efficient space utilization, making it a more versatile solution when implementing two stacks within a single array.