Stack Using Dynamic Array CodeStudio

This C++ program demonstrates the implementation of a stack data structure using a dynamic array. The Stack class contains methods to perform basic stack operations such as push, pop, isEmpty, isFull, getTop, and getSize. The stack starts with a default capacity of 10 and dynamically resizes itself using a doubling strategy whenever it becomes full.

The **Stack** class is defined with private member variables: **arr** (pointer to the dynamic array), **capacity** (maximum capacity of the stack), and **top** (index of the top element).

1. **Constructor (Stack()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: The constructor initializes the stack by allocating memory for the dynamic array (**arr**) with a fixed initial capacity. It sets the **top** to -1. This operation takes constant time and space.
2. **isEmpty (bool isEmpty()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: This function checks whether the stack is empty by examining the value of **top**. It involves a simple comparison and returns the result. It operates in constant time and uses constant space.
3. **isFull (bool isFull()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: Similar to **isEmpty**, this function checks if the stack is full by comparing the value of **top** to the capacity. It's a straightforward comparison and operates in constant time with constant space.
4. **push (void push(int value)):**
    - **Time Complexity: O(1) average (amortized), O(n) worst case (when resizing is needed)**
    - **Space Complexity: O(n) in the worst case (when resizing)**
    - Explanation: Pushing an element onto the stack involves a few steps:

- Checking if the stack is full (O(1)).
- If full, resizing the dynamic array (O(n)) and copying the existing elements (O(n)).
- Updating the **top** index and adding the new element (O(1)).

- The amortized time complexity of **push** is still O(1) because resizing doesn't happen every time and the resizing cost gets distributed over multiple pushes.

5. **pop (void pop()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: Popping the top element involves simply decrementing the **top** index by one. This is a constant time and space operation.
6. **getTop (int getTop()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: Returning the top element involves accessing the element at the index **top** of the dynamic array, which is a constant time operation. The space used is also constant.
7. **getSize (int getSize()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: The size of the stack is represented by **top + 1**. Calculating this and returning it takes constant time and uses constant space.
8. **Destructor (~Stack()):**
    - **Time Complexity: O(1)**
    - **Space Complexity: O(1)**
    - Explanation: The destructor deallocates the memory used by the dynamic array, which is a constant time and space operation.