1381. Design a Stack With Increment Operation

Medium Stack Design Array Leetcode Link

Problem Description

This problem requires designing a stack that not only has the usual push and pop functionalities but also a special feature that allows incrementing the bottom-most `k` elements by a given value. The class `CustomStack` should be initialized with a `maxSize`, which is the maximum number of elements that the stack can hold. The stack supports three operations:

1. `push(int x)`: This method should push the value `x` onto the stack only if the number of elements in the stack is less than the `maxSize`.
2. `pop()`: This method should remove the top element of the stack and return its value. If the stack is empty, it should return `-1`.
3. `increment(int k, int val)`: This method should increment the bottom `k` elements of the stack by the value `val`. If there are fewer than `k` elements in the stack, it increments all of them.

Intuition

To solve this problem efficiently, one could keep a secondary array, called `add`, alongside the primary stack data structure, which is an array called `stk`. The array `add` should have the same length as `stk`, and it's used to store the incremental values.

The key idea behind the solution is to use lazy incrementation. When the `increment` method is called, instead of incrementing every single one of the bottom `k` elements, the value is added to the `add` array at the index that corresponds to the k-th element (or the last element if `k` is greater than the number of elements in the stack). When an element is popped, the increment value at the current index is added to the popped element to reflect all the increments up to that point. If there are elements below it, the increment is passed down to the next element in the stack, ensuring all the increments are applied lazily during the pop operations.

This approach is efficient because it does not require incrementing each element individually, which could be costly for a large number of `inc` operations or a large `k`. Instead, the increment is deferred until an element is popped, making both push and increment operations O(1) time complexity, with the pop operation being O(1) as well.

Solution Approach

The implementation of `CustomStack` relies on two arrays, `stk` to actually hold the stack elements, and `add` to keep track of the increment values that need to be applied to each element. The helper variable `i` acts as a pointer to the current top of the stack, with `i = 0` indicating that the stack is empty.

Here's how each method of the `CustomStack` works according to the solution approach:

`__init__(self, maxSize: int)`

The constructor initializes the two arrays, `stk` and `add`, with a length of `maxSize` and fills them with zeros. It also initializes `i` to 0, which represents the initial position of the stack pointer.

`push(self, x: int) -> None`

The `push` method checks whether the current stack size (`i`) is less than the maximum permissible size before proceeding to add an element. If there's space, the method places the value `x` at the current top position (given by `i`) in `stk` and increments `i` by 1 to reflect the new top of the stack.

`pop(self) -> int`

The `pop` method first checks if the stack is empty by checking if `i` is 0 or less. If the stack is empty, it returns `-1`. If not, it decrements `i` by 1 to point to the current top of the stack, retrieves the original value plus any increment value stored in `add` for that position, and returns it. If there is another element below the popped element (`i > 0`), the increment value for the current position is passed down to the next element by adding it to the `add` value of the next (now the top) element. Finally, the increment value of the popped position is reset to 0.

`increment(self, k: int, val: int) -> None`

The `increment` operation is done lazily. Instead of iterating through the bottom `k` elements, it finds the position `i` in the `add` array which corresponds to the k-th element or the last element if the stack's current size is less than `k`. It then adds the `val` to this `add[i]`. When a `pop` operation is executed later, this value will be added to the returned element and carried over to the next element if necessary.

The data structures used, namely the two arrays `stk` and `add`, are efficient for the operations that the `CustomStack` class is expected to perform. The lazy increment pattern avoids unnecessary work during `increment` operations, ensuring that the `pop` and `push` operations remain efficient even with a large number of increments.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we initialize a `CustomStack` with `maxSize = 3`.

1. **Initialization**

```
CustomStack(maxSize = 3)
stk = [0, 0, 0]
add = [0, 0, 0]
i   = 0 (stack is empty at this stage)
```

2. **Push Operations**

```
push(5) -> stk = [5, 0, 0], add = [0, 0, 0], i = 1
push(7) -> stk = [5, 7, 0], add = [0, 0, 0], i = 2
push(3) -> stk = [5, 7, 3], add = [0, 0, 0], i = 3
```

3. **Increment Operation**

```
increment(2, 100) -> stk = [5, 7, 3], add = [100, 100, 0], i = 3
(We added 100 to the first two elements, however, we store this increment in the `add` array rather than change `stk` directly.)
```

4. **Pop Operation**

```
pop() -> stk = [5, 7, 0], add = [100, 100, 0], i = 2
(We pop the top element, which would be 3 + 100 from `add[2]` = 103, and since there's no element below it, we don't need to tra

pop() -> stk = [5, 0, 0], add = [100, 0, 0], i = 1
(We pop off 7 + 100 from `add[1]` = 107, and we pass the increment value 100 to the next element by setting `add[0]` to 200.)

pop() -> stk = [0, 0, 0], add = [0, 0, 0], i = 0
(The final pop is 5 + 200 from `add[0]` = 205, and we set `add[0]` back to 0 as it's no longer needed.)
```

Based on this example, we can observe that the `increment` operation updates the `add` array rather than every element, and the incremented amounts are only applied when `pop` operations occur. This makes accurate use of the lazy incrementation strategy where the `increment` operation is O(1) and doesn't depend on the number of elements it affects. The `pop` operation remains O(1) because it only involves a constant number of actions regardless of the size of the stack or the increment values.

Python Solution

```python
class CustomStack:
    def __init__(self, max_size: int):
        # Initialize the stack with the given max size.
        # The actual stack is maintained in a list initialized with zeros.
        self.stack = [0] * max_size
        # The add list is used to store the additive increments for each position.
        self.add = [0] * max_size
        # The index `current_size` tracks the number of elements in the stack.
        self.current_size = 0

    def push(self, x: int) -> None:
        # Push an element onto the stack if there is space available.
        if self.current_size < len(self.stack):
            self.stack[self.current_size] = x
            self.current_size += 1

    def pop(self) -> int:
        # Pop the top element from the stack and apply any increments to it.
        if self.current_size <= 0:
            return -1  # Return -1 if the stack is empty.
        self.current_size -= 1
        result = self.stack[self.current_size] + self.add[self.current_size]
        # Transfer the increment to the next element to be popped, if applicable.
        if self.current_size > 0:
            self.add[self.current_size - 1] += self.add[self.current_size]
        # Reset the increment for the current position.
        self.add[self.current_size] = 0
        return result  # Return the final value after applying the increment.

    def increment(self, k: int, val: int) -> None:
        # Increment the bottom `k` elements of the stack by `val`.
        limit = min(k, self.current_size) - 1  # Determine the actual limit to increment.
        if limit >= 0:
            self.add[limit] += val  # Apply the increment to the `limit` position.

# Example usage:
# max_size = 3
# custom_stack = CustomStack(max_size)
# custom_stack.push(1)
# custom_stack.push(2)
# print(custom_stack.pop())  # Outputs: 2
# custom_stack.push(2)
# custom_stack.increment(2, 1)  # Stack becomes [2,3]
# print(custom_stack.pop())  # Outputs: 3
# print(custom_stack.pop())  # Outputs: 2
```

Java Solution

```java
class CustomStack {
    private int[] stack;  // Array to store stack elements
    private int[] increments;  // Array to store increment operations
    private int topIndex;  // Points to the next free spot in the 'stack' array (and the current size of the stack)

    // Constructor to initialize the stack and increments array with the given maxSize
    public CustomStack(int maxSize) {
        stack = new int[maxSize];
        increments = new int[maxSize];
        topIndex = 0;
    }

    // Method to push an element onto the top of the stack if there is space available
    public void push(int x) {
        if (topIndex < stack.length) {
            stack[topIndex++] = x;
        }
    }

    // Method to remove and return the top element of the stack; if the stack is empty, returns -1
    public int pop() {
        if (topIndex <= 0) { // Check if stack is empty
            return -1;
        }
        int result = stack[--topIndex] + increments[topIndex]; // Get the top element with the increment
        if (topIndex > 0) { // If there's still an element below the top, transfer the increment
            increments[topIndex - 1] += increments[topIndex];
        }
        increments[topIndex] = 0; // Reset the increment for the current index since it's been applied
        return result;
    }

    // Method to increment the bottom 'k' elements by 'val'
    public void increment(int k, int val) {
        if (topIndex > 0) { // Check if stack is not empty
            int index = Math.min(topIndex, k) - 1; // Determine the index until which the increments should be applied
            increments[index] += val; // Apply the increment to the kth element or the current top, whichever is smaller
        }
    }
}

/**
 * The following operations can be performed on an instance of CustomStack:
 * - Create a new stack with a maximum size `maxSize`.
 * - Push an integer `x` onto the stack if there is space.
 * - Pop and return the top element of the stack; if the stack is empty, return -1.
 * - Increment the bottom `k` elements of the stack by `val`.
 *
 * Usage:
 * CustomStack customStack = new CustomStack(maxSize);
 * customStack.push(x);
 * int topElement = customStack.pop();
 * customStack.increment(k, val);
 */
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For min function

class CustomStack {
public:
    // Constructor to initialize the stack with a maximum size
    CustomStack(int maxSize) {
        stack_.resize(maxSize);
        additional_.resize(maxSize);
        size_ = 0; // Stack starts empty
    }

    // Method to push an element to the top of the stack if there's space
    void push(int x) {
        if (size_ < stack_.size()) {
            stack_[size_] = x; // Insert element at the top of the stack
            size_++; // Increase the stack size
        }
    }

    // Method to pop the top element from the stack
    int pop() {
        // Check if the stack is empty
        if (size_ <= 0) {
            return -1; // If empty, return -1 as per the problem statement
        }
        // Decrement size and get the top value with additional increment applied
        size_--;
        int value = stack_[size_] + additional_[size_];
        if (size_ > 0) {
            // Propagate the increment to the below element (if any)
            additional_[size_ - 1] += additional_[size_];
        }
        // Reset the additional increment for the popped element
        additional_[size_] = 0;
        return value; // Return the popped value
    }

    // Method to increment the bottom k elements of the stack by val
    void increment(int k, int val) {
        if (size_ > 0) {
            // Only apply to the bottom k elements or all elements if size is less than k
            int limit = std::min(k, size_);
            additional_[limit - 1] += val; // Apply increment to the k-th element from the bottom
        }
    }

private:
    std::vector<int> stack_;     // Vector to store the stack elements
    std::vector<int> additional_; // Vector to store additional increments to apply on pop
    int size_;                   // Current size of the stack (number of elements)
};

// Below this line is where the CustomStack class could be used:
/*
int main() {
    CustomStack* obj = new CustomStack(maxSize);
    obj->push(x);
    int param_2 = obj->pop();
    obj->increment(k,val);

    // Remember to delete the object created with 'new' to prevent memory leaks
    delete obj;
    return 0;
}
*/
```

Typescript Solution

```typescript
// Global variables to act as the stack, the "add" track array, and the index pointer.
let stack: number[];
let addTrack: number[];
let currentIndex: number;

// Function to initialize the stack with a maximum size.
function initializeCustomStack(maxSize: number): void {
    stack = Array(maxSize).fill(0);
    addTrack = Array(maxSize).fill(0);
    currentIndex = 0;
}

// Function to push a new element onto the stack if there's space.
function pushToCustomStack(value: number): void {
    if (currentIndex < stack.length) {
        stack[currentIndex++] = value;
    }
}

// Function to pop the top element from the stack, with additional logic for increment overflow.
function popFromCustomStack(): number {
    if (currentIndex <= 0) {
        return -1;
    }
    const topValue = stack[--currentIndex] + addTrack[currentIndex];
    if (currentIndex > 0) {
        addTrack[currentIndex - 1] += addTrack[currentIndex];
    }
    addTrack[currentIndex] = 0;
    return topValue;
}

// Function to increment the bottom k elements of the stack by a specified value.
function incrementBottomElements(k: number, value: number): void {
    let incrementIndex = Math.min(currentIndex, k) - 1;
    if (incrementIndex >= 0) {
        addTrack[incrementIndex] += value;
    }
}

// Please note that these global functions assume that the initializeCustomStack function is called first to set up the stack.
```

Time and Space Complexity

Time Complexity:

- **init**: `O(n)` where `n` is `maxSize`. This is because we're initializing two arrays `stk` and `add`, each of `maxSize` length, with zeroes.
- **push**: `O(1)` for each operation. Inserting an element into the stack is done by directly indexing into the array, which is a constant-time operation.
- **pop**: `O(1)` for each operation. Removing an element from the stack is also done by directly indexing into the array. The additional operation of transferring the increment value to the next element is constant-time as well, as it involves only a direct index and an arithmetic operation.
- **increment**: `O(1)` for each operation. Even though the increment operation affects `k` elements, we're lazily applying this operation by only updating one item in the `add` array, so we're not iterating through the entire `k` elements at the time of calling this method.

Space Complexity:

- Overall space complexity is `O(n)` where `n` is `maxSize`. This is due to maintaining two additional arrays (`stk` and `add`) that store the elements of the stack and the increment operations to be applied.