

Problem 1381: Design a Stack With Increment Operation

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

Design a stack that supports increment operations on its elements.

Implement the

CustomStack

class:

CustomStack(int maxSize)

Initializes the object with

maxSize

which is the maximum number of elements in the stack.

void push(int x)

Adds

x

to the top of the stack if the stack has not reached the

maxSize

int pop()

Pops and returns the top of the stack or

-1

if the stack is empty.

void inc(int k, int val)

Increments the bottom

k

elements of the stack by

val

. If there are less than

k

elements in the stack, increment all the elements in the stack.

Example 1:

Input

```
["CustomStack","push","push","pop","push","push","push","increment","increment","pop","pop"
,"pop","pop"] [[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
```

Output

```
[null,null,null,2,null,null,null,null,103,202,201,-1]
```

Explanation

```
CustomStack stk = new CustomStack(3); // Stack is Empty []
stk.push(1); // stack becomes [1]
stk.push(2); // stack becomes [1, 2]
stk.pop(); // return 2 --> Return top of the stack 2, stack
becomes [1]
stk.push(2); // stack becomes [1, 2]
stk.push(3); // stack becomes [1, 2, 3]
stk.push(4); // stack still [1, 2, 3], Do not add another elements as size is 4
stk.increment(5,
100); // stack becomes [101, 102, 103]
stk.increment(2, 100); // stack becomes [201, 202,
103]
stk.pop(); // return 103 --> Return top of the stack 103, stack becomes [201, 202]
stk.pop(); // return 202 --> Return top of the stack 202, stack becomes [201]
stk.pop(); // return 201 --> Return top of the stack 201, stack becomes []
stk.pop(); // return -1 --> Stack is empty
return -1.
```

Constraints:

$1 \leq \text{maxSize}, x, k \leq 1000$

$0 \leq \text{val} \leq 100$

At most

1000

calls will be made to each method of

increment

,

push

and

pop

each separately.

Code Snippets

C++:

```

class CustomStack {
public:
CustomStack(int maxSize) {

}

void push(int x) {

}

int pop() {

}

void increment(int k, int val) {

}

};

/**
* Your CustomStack object will be instantiated and called as such:
* CustomStack* obj = new CustomStack(maxSize);
* obj->push(x);
* int param_2 = obj->pop();
* obj->increment(k,val);
*/

```

Java:

```

class CustomStack {

public CustomStack(int maxSize) {

}

public void push(int x) {

}

public int pop() {

}

```

```

public void increment(int k, int val) {

}

}

/***
* Your CustomStack object will be instantiated and called as such:
* CustomStack obj = new CustomStack(maxSize);
* obj.push(x);
* int param_2 = obj.pop();
* obj.increment(k,val);
*/

```

Python3:

```

class CustomStack:

def __init__(self, maxSize: int):

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

def pop(self) -> int:

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

# Your CustomStack object will be instantiated and called as such:
# obj = CustomStack(maxSize)
# obj.push(x)
# param_2 = obj.pop()
# obj.increment(k,val)

```

Python:

```

class CustomStack(object):

def __init__(self, maxSize):
    """

```

```

:type maxSize: int
"""

def push(self, x):
    """
:type x: int
:rtype: None
"""

def pop(self):
    """
:rtype: int
"""

def increment(self, k, val):
    """
:type k: int
:type val: int
:rtype: None
"""

# Your CustomStack object will be instantiated and called as such:
# obj = CustomStack(maxSize)
# obj.push(x)
# param_2 = obj.pop()
# obj.increment(k,val)

```

JavaScript:

```

/**
 * @param {number} maxSize
 */
var CustomStack = function(maxSize) {

};

/**

```

```

* @param {number} x
* @return {void}
*/
CustomStack.prototype.push = function(x) {

};

/***
* @return {number}
*/
CustomStack.prototype.pop = function() {

};

/***
* @param {number} k
* @param {number} val
* @return {void}
*/
CustomStack.prototype.increment = function(k, val) {

};

/***
* Your CustomStack object will be instantiated and called as such:
* var obj = new CustomStack(maxSize)
* obj.push(x)
* var param_2 = obj.pop()
* obj.increment(k, val)
*/

```

TypeScript:

```

class CustomStack {
constructor(maxSize: number) {

}

push(x: number): void {
}

```

```
pop(): number {  
  
}  
  
increment(k: number, val: number): void {  
  
}  
  
}  
  
}  
  
/**  
 * Your CustomStack object will be instantiated and called as such:  
 * var obj = new CustomStack(maxSize)  
 * obj.push(x)  
 * var param_2 = obj.pop()  
 * obj.increment(k,val)  
 */
```

C#:

```
public class CustomStack {  
  
    public CustomStack(int maxSize) {  
  
    }  
  
    public void Push(int x) {  
  
    }  
  
    public int Pop() {  
  
    }  
  
    public void Increment(int k, int val) {  
  
    }  
  
    }  
  
    /**  
     * Your CustomStack object will be instantiated and called as such:  
     * CustomStack obj = new CustomStack(maxSize);  
     * obj.Push(x);
```

```
* int param_2 = obj.Pop();
* obj.Increment(k, val);
*/
```

C:

```
typedef struct {

} CustomStack;

CustomStack* customStackCreate(int maxSize) {

}

void customStackPush(CustomStack* obj, int x) {

}

int customStackPop(CustomStack* obj) {

}

void customStackIncrement(CustomStack* obj, int k, int val) {

}

void customStackFree(CustomStack* obj) {

}

/**
 * Your CustomStack struct will be instantiated and called as such:
 * CustomStack* obj = customStackCreate(maxSize);
 * customStackPush(obj, x);

 * int param_2 = customStackPop(obj);

 * customStackIncrement(obj, k, val);
 */
```

```
* customStackFree(obj);  
*/
```

Go:

```
type CustomStack struct {  
  
}  
  
func Constructor(maxSize int) CustomStack {  
  
}  
  
func (this *CustomStack) Push(x int) {  
  
}  
  
func (this *CustomStack) Pop() int {  
  
}  
  
func (this *CustomStack) Increment(k int, val int) {  
  
}  
  
/**  
* Your CustomStack object will be instantiated and called as such:  
* obj := Constructor(maxSize);  
* obj.Push(x);  
* param_2 := obj.Pop();  
* obj.Increment(k,val);  
*/
```

Kotlin:

```

class CustomStack(maxSize: Int) {

    fun push(x: Int) {

    }

    fun pop(): Int {

    }

    fun increment(k: Int, `val`: Int) {

    }

    /**
     * Your CustomStack object will be instantiated and called as such:
     * var obj = CustomStack(maxSize)
     * obj.push(x)
     * var param_2 = obj.pop()
     * obj.increment(k,`val`)
     */
}

```

Swift:

```

class CustomStack {

    init(_ maxSize: Int) {

    }

    func push(_ x: Int) {

    }

    func pop() -> Int {

    }

    func increment(_ k: Int, _ val: Int) {
}

```

```
}

}

/***
* Your CustomStack object will be instantiated and called as such:
* let obj = CustomStack(maxSize)
* obj.push(x)
* let ret_2: Int = obj.pop()
* obj.increment(k, val)
*/

```

Rust:

```
struct CustomStack {

}

/***
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl CustomStack {

fn new(maxSize: i32) -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

fn increment(&self, k: i32, val: i32) {

}
}

/**

```

```
* Your CustomStack object will be instantiated and called as such:  
* let obj = CustomStack::new(maxSize);  
* obj.push(x);  
* let ret_2: i32 = obj.pop();  
* obj.increment(k, val);  
*/
```

Ruby:

```
class CustomStack  
  
=begin  
:type max_size: Integer  
=end  
def initialize(max_size)  
  
end
```

```
=begin  
:type x: Integer  
:rtype: Void  
=end  
def push(x)  
  
end
```

```
=begin  
:rtype: Integer  
=end  
def pop( )  
  
end
```

```
=begin  
:type k: Integer  
:type val: Integer  
:rtype: Void  
=end  
def increment(k, val)
```

```
end

end

# Your CustomStack object will be instantiated and called as such:
# obj = CustomStack.new(max_size)
# obj.push(x)
# param_2 = obj.pop()
# obj.increment(k, val)
```

PHP:

```
class CustomStack {

    /**
     * @param Integer $maxSize
     */
    function __construct($maxSize) {

    }

    /**
     * @param Integer $x
     * @return NULL
     */
    function push($x) {

    }

    /**
     * @return Integer
     */
    function pop() {

    }

    /**
     * @param Integer $k
     * @param Integer $val
     * @return NULL
     */
}
```

```
function increment($k, $val) {  
}  
}  
  
}  
  
/**  
 * Your CustomStack object will be instantiated and called as such:  
 * $obj = CustomStack($maxSize);  
 * $obj->push($x);  
 * $ret_2 = $obj->pop();  
 * $obj->increment($k, $val);  
 */
```

Dart:

```
class CustomStack {  
  
CustomStack(int maxSize) {  
  
}  
  
void push(int x) {  
  
}  
  
int pop() {  
  
}  
  
void increment(int k, int val) {  
  
}  
  
}  
  
}  
  
/**  
 * Your CustomStack object will be instantiated and called as such:  
 * CustomStack obj = CustomStack(maxSize);  
 * obj.push(x);  
 * int param2 = obj.pop();  
 * obj.increment(k, val);  
 */
```

Scala:

```
class CustomStack(_maxSize: Int) {  
  
  def push(x: Int): Unit = {  
  
  }  
  
  def pop(): Int = {  
  
  }  
  
  def increment(k: Int, `val`: Int): Unit = {  
  
  }  
  
  /**  
   * Your CustomStack object will be instantiated and called as such:  
   * val obj = new CustomStack(maxSize)  
   * obj.push(x)  
   * val param_2 = obj.pop()  
   * obj.increment(k,`val`)  
   */
```

Elixir:

```
defmodule CustomStack do  
  @spec init_(max_size :: integer) :: any  
  def init_(max_size) do  
  
  end  
  
  @spec push(x :: integer) :: any  
  def push(x) do  
  
  end  
  
  @spec pop() :: integer  
  def pop() do
```

```

end

@spec increment(k :: integer, val :: integer) :: any
def increment(k, val) do

end
end

# Your functions will be called as such:
# CustomStack.init_(max_size)
# CustomStack.push(x)
# param_2 = CustomStack.pop()
# CustomStack.increment(k, val)

# CustomStack.init_ will be called before every test case, in which you can
do some necessary initializations.

```

Erlang:

```

-spec custom_stack_init_(MaxSize :: integer()) -> any().
custom_stack_init_(MaxSize) ->
.

-spec custom_stack_push(X :: integer()) -> any().
custom_stack_push(X) ->
.

-spec custom_stack_pop() -> integer().
custom_stack_pop() ->
.

-spec custom_stack_increment(K :: integer(), Val :: integer()) -> any().
custom_stack_increment(K, Val) ->
.

%% Your functions will be called as such:
%% custom_stack_init_(MaxSize),
%% custom_stack_push(X),
%% Param_2 = custom_stack_pop(),
%% custom_stack_increment(K, Val),

```

```
%% custom_stack_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket:

```
(define custom-stack%
  (class object%
    (super-new)

    ; max-size : exact-integer?
    (init-field
      max-size)

    ; push : exact-integer? -> void?
    (define/public (push x)
      )

    ; pop : -> exact-integer?
    (define/public (pop)
      )

    ; increment : exact-integer? exact-integer? -> void?
    (define/public (increment k val)
      )))

;; Your custom-stack% object will be instantiated and called as such:
;; (define obj (new custom-stack% [max-size max-size]))
;; (send obj push x)
;; (define param_2 (send obj pop))
;; (send obj increment k val)
```

Solutions

C++ Solution:

```
/*
 * Problem: Design a Stack With Increment Operation
 * Difficulty: Medium
 * Tags: array, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
```

```

* Space Complexity: O(1) to O(n) depending on approach
*/



class CustomStack {
public:
CustomStack(int maxSize) {

}

void push(int x) {

}

int pop() {

}

void increment(int k, int val) {

};

}

/**
* Your CustomStack object will be instantiated and called as such:
* CustomStack* obj = new CustomStack(maxSize);
* obj->push(x);
* int param_2 = obj->pop();
* obj->increment(k,val);
*/

```

Java Solution:

```

/***
* Problem: Design a Stack With Increment Operation
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

```

```

class CustomStack {

    public CustomStack(int maxSize) {

    }

    public void push(int x) {

    }

    public int pop() {

    }

    public void increment(int k, int val) {

    }

}

/**
 * Your CustomStack object will be instantiated and called as such:
 * CustomStack obj = new CustomStack(maxSize);
 * obj.push(x);
 * int param_2 = obj.pop();
 * obj.increment(k,val);
 */

```

Python3 Solution:

```

"""
Problem: Design a Stack With Increment Operation
Difficulty: Medium
Tags: array, stack

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class CustomStack:

```

```
def __init__(self, maxSize: int):

    def push(self, x: int) -> None:
        # TODO: Implement optimized solution
        pass
```

Python Solution:

```
class CustomStack(object):

    def __init__(self, maxSize):
        """
        :type maxSize: int
        """

    def push(self, x):
        """
        :type x: int
        :rtype: None
        """

    def pop(self):
        """
        :rtype: int
        """

    def increment(self, k, val):
        """
        :type k: int
        :type val: int
        :rtype: None
        """

# Your CustomStack object will be instantiated and called as such:
```

```
# obj = CustomStack(maxSize)
# obj.push(x)
# param_2 = obj.pop()
# obj.increment(k,val)
```

JavaScript Solution:

```
/**
 * Problem: Design a Stack With Increment Operation
 * Difficulty: Medium
 * Tags: array, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

/**
 * @param {number} maxSize
 */
var CustomStack = function(maxSize) {

};

/**
 * @param {number} x
 * @return {void}
 */
CustomStack.prototype.push = function(x) {

};

/**
 * @return {number}
 */
CustomStack.prototype.pop = function() {

};

/**
 * @param {number} k
```

```

* @param {number} val
* @return {void}
*/
CustomStack.prototype.increment = function(k, val) {

};

/** 
* Your CustomStack object will be instantiated and called as such:
* var obj = new CustomStack(maxSize)
* obj.push(x)
* var param_2 = obj.pop()
* obj.increment(k,val)
*/

```

TypeScript Solution:

```

/**
* Problem: Design a Stack With Increment Operation
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class CustomStack {
constructor(maxSize: number) {

}

push(x: number): void {

}

pop(): number {

}

increment(k: number, val: number): void {

```

```

}

}

/** 
 * Your CustomStack object will be instantiated and called as such:
 * var obj = new CustomStack(maxSize)
 * obj.push(x)
 * var param_2 = obj.pop()
 * obj.increment(k,val)
 */

```

C# Solution:

```

/*
 * Problem: Design a Stack With Increment Operation
 * Difficulty: Medium
 * Tags: array, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

public class CustomStack {

    public CustomStack(int maxSize) {

    }

    public void Push(int x) {

    }

    public int Pop() {

    }

    public void Increment(int k, int val) {

```

```

}

/**
 * Your CustomStack object will be instantiated and called as such:
 * CustomStack obj = new CustomStack(maxSize);
 * obj.Push(x);
 * int param_2 = obj.Pop();
 * obj.Increment(k,val);
 */

```

C Solution:

```

/*
 * Problem: Design a Stack With Increment Operation
 * Difficulty: Medium
 * Tags: array, stack
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

```

```

typedef struct {

} CustomStack;

CustomStack* customStackCreate(int maxSize) {

}

void customStackPush(CustomStack* obj, int x) {

}

int customStackPop(CustomStack* obj) {

}

```

```

void customStackIncrement(CustomStack* obj, int k, int val) {

}

void customStackFree(CustomStack* obj) {

}

/**
 * Your CustomStack struct will be instantiated and called as such:
 * CustomStack* obj = customStackCreate(maxSize);
 * customStackPush(obj, x);

 * int param_2 = customStackPop(obj);

 * customStackIncrement(obj, k, val);

 * customStackFree(obj);
 */

```

Go Solution:

```

// Problem: Design a Stack With Increment Operation
// Difficulty: Medium
// Tags: array, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type CustomStack struct {

}

func Constructor(maxSize int) CustomStack {
}
```

```

func (this *CustomStack) Push(x int) {

}

func (this *CustomStack) Pop() int {

}

func (this *CustomStack) Increment(k int, val int) {

}

/**
 * Your CustomStack object will be instantiated and called as such:
 * obj := Constructor(maxSize);
 * obj.Push(x);
 * param_2 := obj.Pop();
 * obj.Increment(k,val);
 */

```

Kotlin Solution:

```

class CustomStack(maxSize: Int) {

    fun push(x: Int) {

    }

    fun pop(): Int {

    }

    fun increment(k: Int, `val`: Int) {

    }
}

```

```
/**  
 * Your CustomStack object will be instantiated and called as such:  
 * var obj = CustomStack(maxSize)  
 * obj.push(x)  
 * var param_2 = obj.pop()  
 * obj.increment(k,`val`)  
 */
```

Swift Solution:

```
class CustomStack {  
  
    init(_ maxSize: Int) {  
  
    }  
  
    func push(_ x: Int) {  
  
    }  
  
    func pop() -> Int {  
  
    }  
  
    func increment(_ k: Int, _ val: Int) {  
  
    }  
}  
  
/**  
 * Your CustomStack object will be instantiated and called as such:  
 * let obj = CustomStack(maxSize)  
 * obj.push(x)  
 * let ret_2: Int = obj.pop()  
 * obj.increment(k, val)  
 */
```

Rust Solution:

```
// Problem: Design a Stack With Increment Operation
// Difficulty: Medium
// Tags: array, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct CustomStack {

}

/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl CustomStack {

fn new(maxSize: i32) -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

fn increment(&self, k: i32, val: i32) {

}

}

/** 
* Your CustomStack object will be instantiated and called as such:
* let obj = CustomStack::new(maxSize);
* obj.push(x);
* let ret_2: i32 = obj.pop();
* obj.increment(k, val);
*/
}
```

Ruby Solution:

```
class CustomStack

=begin
:type max_size: Integer
=end
def initialize(max_size)

end

=begin
:type x: Integer
:rtype: Void
=end
def push(x)

end

=begin
:rtype: Integer
=end
def pop( )

end

=begin
:type k: Integer
:type val: Integer
:rtype: Void
=end
def increment(k, val)

end

end

# Your CustomStack object will be instantiated and called as such:
```

```
# obj = CustomStack.new(max_size)
# obj.push(x)
# param_2 = obj.pop()
# obj.increment(k, val)
```

PHP Solution:

```
class CustomStack {
    /**
     * @param Integer $maxSize
     */
    function __construct($maxSize) {

    }

    /**
     * @param Integer $x
     * @return NULL
     */
    function push($x) {

    }

    /**
     * @return Integer
     */
    function pop() {

    }

    /**
     * @param Integer $k
     * @param Integer $val
     * @return NULL
     */
    function increment($k, $val) {

    }
}
```

```
* Your CustomStack object will be instantiated and called as such:  
* $obj = CustomStack($maxSize);  
* $obj->push($x);  
* $ret_2 = $obj->pop();  
* $obj->increment($k, $val);  
*/
```

Dart Solution:

```
class CustomStack {  
  
CustomStack(int maxSize) {  
  
}  
  
void push(int x) {  
  
}  
  
int pop() {  
  
}  
  
void increment(int k, int val) {  
  
}  
  
}  
  
/**  
* Your CustomStack object will be instantiated and called as such:  
* CustomStack obj = CustomStack(maxSize);  
* obj.push(x);  
* int param2 = obj.pop();  
* obj.increment(k,val);  
*/
```

Scala Solution:

```
class CustomStack(_maxSize: Int) {  
  
def push(x: Int): Unit = {
```

```

}

def pop(): Int = {

}

def increment(k: Int, `val`: Int): Unit = {

}

/**
* Your CustomStack object will be instantiated and called as such:
* val obj = new CustomStack(maxSize)
* obj.push(x)
* val param_2 = obj.pop()
* obj.increment(k,`val`)
*/

```

Elixir Solution:

```

defmodule CustomStack do
  @spec init_(max_size :: integer) :: any
  def init_(max_size) do

    end

    @spec push(x :: integer) :: any
    def push(x) do

      end

      @spec pop() :: integer
      def pop() do

        end

        @spec increment(k :: integer, val :: integer) :: any
        def increment(k, val) do

```

```

end
end

# Your functions will be called as such:
# CustomStack.init_(max_size)
# CustomStack.push(x)
# param_2 = CustomStack.pop()
# CustomStack.increment(k, val)

# CustomStack.init_ will be called before every test case, in which you can
do some necessary initializations.

```

Erlang Solution:

```

-spec custom_stack_init_(MaxSize :: integer()) -> any().
custom_stack_init_(MaxSize) ->
.

-spec custom_stack_push(X :: integer()) -> any().
custom_stack_push(X) ->
.

-spec custom_stack_pop() -> integer().
custom_stack_pop() ->
.

-spec custom_stack_increment(K :: integer(), Val :: integer()) -> any().
custom_stack_increment(K, Val) ->
.

%% Your functions will be called as such:
%% custom_stack_init_(MaxSize),
%% custom_stack_push(X),
%% Param_2 = custom_stack_pop(),
%% custom_stack_increment(K, Val),

%% custom_stack_init_ will be called before every test case, in which you can
do some necessary initializations.

```

Racket Solution:

```
(define custom-stack%
  (class object%
    (super-new)

    ; max-size : exact-integer?
    (init-field
      max-size)

    ; push : exact-integer? -> void?
    (define/public (push x)
      )
    ; pop : -> exact-integer?
    (define/public (pop)
      )
    ; increment : exact-integer? exact-integer? -> void?
    (define/public (increment k val)
      )))

;; Your custom-stack% object will be instantiated and called as such:
;; (define obj (new custom-stack% [max-size max-size]))
;; (send obj push x)
;; (define param_2 (send obj pop))
;; (send obj increment k val)
```