Problem 1172: Dinner Plate Stacks

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You have an infinite number of stacks arranged in a row and numbered (left to right) from

0

, each of the stacks has the same maximum capacity.

Implement the

DinnerPlates

class:

DinnerPlates(int capacity)

Initializes the object with the maximum capacity of the stacks

capacity

.

void push(int val)

Pushes the given integer

val

into the leftmost stack with a size less than

capacity

.

int pop()

Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns

-1

if all the stacks are empty.

int popAtStack(int index)

Returns the value at the top of the stack with the given index

index

and removes it from that stack or returns

-1

if the stack with that given index is empty.

Example 1:

Input

["DinnerPlates", "push", "push", "push", "push", "push", "popAtStack", "push", "push", "popAtStack", "popAtStack", "pop", "pop", "pop", "pop", "pop"] [[2], [1], [2], [3], [4], [5], [0], [20], [21], [0], [2], [], [], [], [], []]

Output

[null, null, null, null, null, null, 2, null, null, 20, 21, 5, 4, 3, 1, -1]

Explanation:

DinnerPlates D = DinnerPlates(2); // Initialize with capacity = 2 D.push(1); D.push(2); D.push(3); D.push(4); D.push(5); // The stacks are now: 2 4 1 3 5 ■ ■ ■ D.popAtStack(0); // Returns 2. The stacks are now: 4 1 3 5 ■ ■ ■ D.push(20); // The stacks are now: 20 4 1 3 5 ■ ■ ■ D.push(21); // The stacks are now: 20 4 21 1 3 5 ■ ■ ■ D.popAtStack(0); // Returns 20. The stacks are now: 4 21 1 3 5 ■ ■ ■ D.popAtStack(2); // Returns 21. The stacks are now: 4 1 3 5 ■ ■ ■ D.pop() // Returns 5. The stacks are now: 4 1 3 ■ ■ D.pop() // Returns 4. The stacks are now: 1 3 ■ ■ D.pop() // Returns 3. The stacks are now: 1 ■ D.pop() // Returns 1. There are no stacks. D.pop() // Returns -1. There are still no stacks.

Constraints:

1 <= capacity <= 2 * 10

4

1 <= val <= 2 * 10

4

0 <= index <= 10

5

At most

2 * 10

5

calls will be made to

push

,

pop

, and

popAtStack

.

Code Snippets

C++:

```
class DinnerPlates {
public:
DinnerPlates(int capacity) {

}

void push(int val) {

}

int pop() {

}

int popAtStack(int index) {

}
};

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates* obj = new DinnerPlates(capacity);
* obj->push(val);
* int param_2 = obj->pop();
* int param_3 = obj->popAtStack(index);
*/
```

Java:

```
class DinnerPlates {
```

```
public DinnerPlates(int capacity) {

}

public void push(int val) {

}

public int pop() {

}

public int popAtStack(int index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = new DinnerPlates(capacity);
* obj.push(val);
* int param_2 = obj.pop();
* int param_3 = obj.popAtStack(index);
*/
```

Python3:

```
class DinnerPlates:

def __init__(self, capacity: int):


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


def pop(self) -> int:


def popAtStack(self, index: int) -> int:
```

```
# Your DinnerPlates object will be instantiated and called as such:
# obj = DinnerPlates(capacity)
# obj.push(val)
# param_2 = obj.pop()
# param_3 = obj.popAtStack(index)
```

Python:

```
class DinnerPlates(object):

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



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



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



def popAtStack(self, index):
"""
:type index: int
:rtype: int
"""




# Your DinnerPlates object will be instantiated and called as such:
# obj = DinnerPlates(capacity)
# obj.push(val)
# param_2 = obj.pop()
# param_3 = obj.popAtStack(index)
```

JavaScript:

```
/**
* @param {number} capacity
*/
var DinnerPlates = function(capacity) {

};

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

};

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

};

/**
* @param {number} index
* @return {number}
*/
DinnerPlates.prototype.popAtStack = function(index) {

};

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = new DinnerPlates(capacity)
* obj.push(val)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

TypeScript:

```
class DinnerPlates {
constructor(capacity: number) {

}

push(val: number): void {

}

pop(): number {

}

popAtStack(index: number): number {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = new DinnerPlates(capacity)
* obj.push(val)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

C#:

```
public class DinnerPlates {

public DinnerPlates(int capacity) {

}

public void Push(int val) {

}

public int Pop() {

}

public int PopAtStack(int index) {
```

```
}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = new DinnerPlates(capacity);
* obj.Push(val);
* int param_2 = obj.Pop();
* int param_3 = obj.PopAtStack(index);
*/
```

C:

```
typedef struct {

} DinnerPlates;


DinnerPlates* dinnerPlatesCreate(int capacity) {

}

void dinnerPlatesPush(DinnerPlates* obj, int val) {

}

int dinnerPlatesPop(DinnerPlates* obj) {

}

int dinnerPlatesPopAtStack(DinnerPlates* obj, int index) {

}

void dinnerPlatesFree(DinnerPlates* obj) {

}
```

```
/**
* Your DinnerPlates struct will be instantiated and called as such:
* DinnerPlates* obj = dinnerPlatesCreate(capacity);
* dinnerPlatesPush(obj, val);

* int param_2 = dinnerPlatesPop(obj);

* int param_3 = dinnerPlatesPopAtStack(obj, index);

* dinnerPlatesFree(obj);
*/
```

Go:

```
type DinnerPlates struct {

}


func Constructor(capacity int) DinnerPlates {

}


func (this *DinnerPlates) Push(val int) {

}


func (this *DinnerPlates) Pop() int {

}


func (this *DinnerPlates) PopAtStack(index int) int {

}


/**
* Your DinnerPlates object will be instantiated and called as such:
* obj := Constructor(capacity);
```

```
* obj.Push(val);
* param_2 := obj.Pop();
* param_3 := obj.PopAtStack(index);
*/
```

Kotlin:

```
class DinnerPlates(capacity: Int) {

fun push(`val`: Int) {

}

fun pop(): Int {

}

fun popAtStack(index: Int): Int {

}

}

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = DinnerPlates(capacity)
* obj.push(`val`)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

Swift:

```
class DinnerPlates {

init(_ capacity: Int) {

}

func push(_ val: Int) {
```

```
}

func pop() -> Int {

}

func popAtStack(_ index: Int) -> Int {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* let obj = DinnerPlates(capacity)
* obj.push(val)
* let ret_2: Int = obj.pop()
* let ret_3: Int = obj.popAtStack(index)
*/
```

Rust:

```
struct DinnerPlates {

}



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

fn new(capacity: i32) -> Self {

}

fn push(&self, val: i32) {

}

fn pop(&self) -> i32 {
```

```
}

fn pop_at_stack(&self, index: i32) -> i32 {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* let obj = DinnerPlates::new(capacity);
* obj.push(val);
* let ret_2: i32 = obj.pop();
* let ret_3: i32 = obj.pop_at_stack(index);
*/
```

Ruby:

```
class DinnerPlates

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

end


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

end


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

end
```

```
=begin
:type index: Integer
:rtype: Integer
=end
def pop_at_stack(index)

end



end

# Your DinnerPlates object will be instantiated and called as such:
# obj = DinnerPlates.new(capacity)
# obj.push(val)
# param_2 = obj.pop()
# param_3 = obj.pop_at_stack(index)
```

PHP:

```
class DinnerPlates {
/**
* @param Integer $capacity
*/
function __construct($capacity) {

}

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

}

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

```
}

/**
* @param Integer $index
* @return Integer
*/
function popAtStack($index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* $obj = DinnerPlates($capacity);
* $obj->push($val);
* $ret_2 = $obj->pop();
* $ret_3 = $obj->popAtStack($index);
*/
```

Dart:

```
class DinnerPlates {

DinnerPlates(int capacity) {

}

void push(int val) {

}

int pop() {

}

int popAtStack(int index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
```

```
* DinnerPlates obj = DinnerPlates(capacity);
* obj.push(val);
* int param2 = obj.pop();
* int param3 = obj.popAtStack(index);
*/
```

Scala:

```
class DinnerPlates(_capacity: Int) {

def push(`val`: Int): Unit = {

}

def pop(): Int = {

}

def popAtStack(index: Int): Int = {

}

}

/**
* Your DinnerPlates object will be instantiated and called as such:
* val obj = new DinnerPlates(capacity)
* obj.push(`val`)
* val param_2 = obj.pop()
* val param_3 = obj.popAtStack(index)
*/
```

Elixir:

```
defmodule DinnerPlates do
@spec init_(capacity :: integer) :: any
def init_(capacity) do

end

@spec push(val :: integer) :: any
def push(val) do
```

```
end

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

end

@spec pop_at_stack(index :: integer) :: integer
def pop_at_stack(index) do

end
end

# Your functions will be called as such:
# DinnerPlates.init_(capacity)
# DinnerPlates.push(val)
# param_2 = DinnerPlates.pop()
# param_3 = DinnerPlates.pop_at_stack(index)

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

Erlang:

```
-spec dinner_plates_init_(Capacity :: integer()) -> any().
dinner_plates_init_(Capacity) ->
.

-spec dinner_plates_push(Val :: integer()) -> any().
dinner_plates_push(Val) ->
.

-spec dinner_plates_pop() -> integer().
dinner_plates_pop() ->
.

-spec dinner_plates_pop_at_stack(Index :: integer()) -> integer().
dinner_plates_pop_at_stack(Index) ->
.
```

```
%% Your functions will be called as such:
%% dinner_plates_init_(Capacity),
%% dinner_plates_push(Val),
%% Param_2 = dinner_plates_pop(),
%% Param_3 = dinner_plates_pop_at_stack(Index),

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

Racket:

```
(define dinner-plates%
(class object%
(super-new)

; capacity : exact-integer?
(init-field
capacity)

; push : exact-integer? -> void?
(define/public (push val)
)
; pop : -> exact-integer?
(define/public (pop)
)
; pop-at-stack : exact-integer? -> exact-integer?
(define/public (pop-at-stack index)
)))

;; Your dinner-plates% object will be instantiated and called as such:
;; (define obj (new dinner-plates% [capacity capacity]))
;; (send obj push val)
;; (define param_2 (send obj pop))
;; (define param_3 (send obj pop-at-stack index))
```

Solutions

C++ Solution:

```
/*
* Problem: Dinner Plate Stacks
* Difficulty: Hard
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DinnerPlates {
public:
DinnerPlates(int capacity) {

}

void push(int val) {

}

int pop() {

}

int popAtStack(int index) {

}
};

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates* obj = new DinnerPlates(capacity);
* obj->push(val);
* int param_2 = obj->pop();
* int param_3 = obj->popAtStack(index);
*/
```

Java Solution:

```
/**
* Problem: Dinner Plate Stacks
* Difficulty: Hard
```

```
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DinnerPlates {

public DinnerPlates(int capacity) {

}

public void push(int val) {

}

public int pop() {

}

public int popAtStack(int index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = new DinnerPlates(capacity);
* obj.push(val);
* int param_2 = obj.pop();
* int param_3 = obj.popAtStack(index);
*/
```

Python3 Solution:

```
"""
Problem: Dinner Plate Stacks
Difficulty: Hard
Tags: hash, stack, queue, heap
```

```
Approach: Use hash map for O(1) lookups
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(n) for hash map
"""

class DinnerPlates:

def __init__(self, capacity: int):


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

Python Solution:

```
class DinnerPlates(object):

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



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



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



def popAtStack(self, index):
"""
:type index: int
:rtype: int
```

```
"""



# Your DinnerPlates object will be instantiated and called as such:
# obj = DinnerPlates(capacity)
# obj.push(val)
# param_2 = obj.pop()
# param_3 = obj.popAtStack(index)
```

JavaScript Solution:

```
/**
* Problem: Dinner Plate Stacks
* Difficulty: Hard
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} capacity
*/
var DinnerPlates = function(capacity) {

};

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

};

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

```
};

/**
* @param {number} index
* @return {number}
*/
DinnerPlates.prototype.popAtStack = function(index) {

};

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = new DinnerPlates(capacity)
* obj.push(val)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

TypeScript Solution:

```
/**
* Problem: Dinner Plate Stacks
* Difficulty: Hard
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DinnerPlates {
constructor(capacity: number) {

}

push(val: number): void {

}

pop(): number {
```

```
}

popAtStack(index: number): number {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = new DinnerPlates(capacity)
* obj.push(val)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

C# Solution:

```
/*
* Problem: Dinner Plate Stacks
* Difficulty: Hard
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

public class DinnerPlates {

public DinnerPlates(int capacity) {

}

public void Push(int val) {

}

public int Pop() {

}
```

```
public int PopAtStack(int index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = new DinnerPlates(capacity);
* obj.Push(val);
* int param_2 = obj.Pop();
* int param_3 = obj.PopAtStack(index);
*/
```

C Solution:

```
/*
* Problem: Dinner Plate Stacks
* Difficulty: Hard
* Tags: hash, stack, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/




typedef struct {

} DinnerPlates;


DinnerPlates* dinnerPlatesCreate(int capacity) {

}

void dinnerPlatesPush(DinnerPlates* obj, int val) {

}
```

```
int dinnerPlatesPop(DinnerPlates* obj) {

}

int dinnerPlatesPopAtStack(DinnerPlates* obj, int index) {

}

void dinnerPlatesFree(DinnerPlates* obj) {

}

/**
* Your DinnerPlates struct will be instantiated and called as such:
* DinnerPlates* obj = dinnerPlatesCreate(capacity);
* dinnerPlatesPush(obj, val);

* int param_2 = dinnerPlatesPop(obj);

* int param_3 = dinnerPlatesPopAtStack(obj, index);

* dinnerPlatesFree(obj);
*/
```

Go Solution:

```
// Problem: Dinner Plate Stacks
// Difficulty: Hard
// Tags: hash, stack, queue, heap
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

type DinnerPlates struct {

}


func Constructor(capacity int) DinnerPlates {
```

```
}


func (this *DinnerPlates) Push(val int) {

}


func (this *DinnerPlates) Pop() int {

}


func (this *DinnerPlates) PopAtStack(index int) int {

}


/**
* Your DinnerPlates object will be instantiated and called as such:
* obj := Constructor(capacity);
* obj.Push(val);
* param_2 := obj.Pop();
* param_3 := obj.PopAtStack(index);
*/
```

Kotlin Solution:

```
class DinnerPlates(capacity: Int) {

fun push(`val`: Int) {

}

fun pop(): Int {

}

fun popAtStack(index: Int): Int {
```

```
}

}

/**
* Your DinnerPlates object will be instantiated and called as such:
* var obj = DinnerPlates(capacity)
* obj.push(`val`)
* var param_2 = obj.pop()
* var param_3 = obj.popAtStack(index)
*/
```

Swift Solution:

```
class DinnerPlates {

init(_ capacity: Int) {

}

func push(_ val: Int) {

}

func pop() -> Int {

}

func popAtStack(_ index: Int) -> Int {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* let obj = DinnerPlates(capacity)
* obj.push(val)
* let ret_2: Int = obj.pop()
* let ret_3: Int = obj.popAtStack(index)
*/
```

Rust Solution:

```
// Problem: Dinner Plate Stacks
// Difficulty: Hard
// Tags: hash, stack, queue, heap
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

struct DinnerPlates {

}



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

fn new(capacity: i32) -> Self {

}

fn push(&self, val: i32) {

}

fn pop(&self) -> i32 {

}

fn pop_at_stack(&self, index: i32) -> i32 {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* let obj = DinnerPlates::new(capacity);
* obj.push(val);
```

```
* let ret_2: i32 = obj.pop();
* let ret_3: i32 = obj.pop_at_stack(index);
*/
```

Ruby Solution:

```
class DinnerPlates

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

end


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

end


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

end


=begin
:type index: Integer
:rtype: Integer
=end
def pop_at_stack(index)

end
```

```
end

# Your DinnerPlates object will be instantiated and called as such:
# obj = DinnerPlates.new(capacity)
# obj.push(val)
# param_2 = obj.pop()
# param_3 = obj.pop_at_stack(index)
```

PHP Solution:

```
class DinnerPlates {
/**
* @param Integer $capacity
*/
function __construct($capacity) {

}

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

}

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

}

/**
* @param Integer $index
* @return Integer
*/
function popAtStack($index) {

}
```

```
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* $obj = DinnerPlates($capacity);
* $obj->push($val);
* $ret_2 = $obj->pop();
* $ret_3 = $obj->popAtStack($index);
*/
```

Dart Solution:

```
class DinnerPlates {

DinnerPlates(int capacity) {

}

void push(int val) {

}

int pop() {

}

int popAtStack(int index) {

}
}

/**
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = DinnerPlates(capacity);
* obj.push(val);
* int param2 = obj.pop();
* int param3 = obj.popAtStack(index);
*/
```

Scala Solution:

```
class DinnerPlates(_capacity: Int) {

def push(`val`: Int): Unit = {

}

def pop(): Int = {

}

def popAtStack(index: Int): Int = {

}

}

/**
* Your DinnerPlates object will be instantiated and called as such:
* val obj = new DinnerPlates(capacity)
* obj.push(`val`)
* val param_2 = obj.pop()
* val param_3 = obj.popAtStack(index)
*/
```

Elixir Solution:

```
defmodule DinnerPlates do
@spec init_(capacity :: integer) :: any
def init_(capacity) do

end

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

end

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

end
```

```
@spec pop_at_stack(index :: integer) :: integer
def pop_at_stack(index) do

end
end

# Your functions will be called as such:
# DinnerPlates.init_(capacity)
# DinnerPlates.push(val)
# param_2 = DinnerPlates.pop()
# param_3 = DinnerPlates.pop_at_stack(index)

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

Erlang Solution:

```
-spec dinner_plates_init_(Capacity :: integer()) -> any().
dinner_plates_init_(Capacity) ->
.

-spec dinner_plates_push(Val :: integer()) -> any().
dinner_plates_push(Val) ->
.

-spec dinner_plates_pop() -> integer().
dinner_plates_pop() ->
.

-spec dinner_plates_pop_at_stack(Index :: integer()) -> integer().
dinner_plates_pop_at_stack(Index) ->
.


%% Your functions will be called as such:
%% dinner_plates_init_(Capacity),
%% dinner_plates_push(Val),
%% Param_2 = dinner_plates_pop(),
%% Param_3 = dinner_plates_pop_at_stack(Index),

%% dinner_plates_init_ will be called before every test case, in which you
```

```
can do some necessary initializations.
```

Racket Solution:

```
(define dinner-plates%
(class object%
(super-new)

; capacity : exact-integer?
(init-field
capacity)

; push : exact-integer? -> void?
(define/public (push val)
)
; pop : -> exact-integer?
(define/public (pop)
)
; pop-at-stack : exact-integer? -> exact-integer?
(define/public (pop-at-stack index)
)))

;; Your dinner-plates% object will be instantiated and called as such:
;; (define obj (new dinner-plates% [capacity capacity]))
;; (send obj push val)
;; (define param_2 (send obj pop))
;; (define param_3 (send obj pop-at-stack index))
```