Problem 225: Implement Stack using Queues

Problem Information

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

Problem Description

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (

push

,

top

,

pop

, and

empty

).

Implement the

MyStack

class:

void push(int x)

Pushes element x to the top of the stack.

int pop()

Removes the element on the top of the stack and returns it.

int top()

Returns the element on the top of the stack.

boolean empty()

Returns

true

if the stack is empty,

false

otherwise.

Notes:

You must use

only

standard operations of a queue, which means that only

push to back

,

peek/pop from front

,

size

and

is empty

operations are valid.

Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input

["MyStack", "push", "push", "top", "pop", "empty"] [[], [1], [2], [], [], []]

Output

[null, null, null, 2, 2, false]

Explanation

MyStack myStack = new MyStack(); myStack.push(1); myStack.push(2); myStack.top(); // return 2 myStack.pop(); // return 2 myStack.empty(); // return False

Constraints:

1 <= x <= 9

At most

100

calls will be made to

push

,

pop

,

top

, and

empty

.

All the calls to

pop

and

top

are valid.

Follow-up:

Can you implement the stack using only one queue?

Code Snippets

C++:

```
class MyStack {
public:
MyStack() {

}

void push(int x) {
```

```
}

int pop() {

}

int top() {

}

bool empty() {

}
};

/**
* Your MyStack object will be instantiated and called as such:
* MyStack* obj = new MyStack();
* obj->push(x);
* int param_2 = obj->pop();
* int param_3 = obj->top();
* bool param_4 = obj->empty();
*/
```

Java:

```
class MyStack {

public MyStack() {

}

public void push(int x) {

}

public int pop() {

}

public int top() {
```

```
}

public boolean empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = new MyStack();
* obj.push(x);
* int param_2 = obj.pop();
* int param_3 = obj.top();
* boolean param_4 = obj.empty();
*/
```

Python3:

```
class MyStack:

def __init__(self):


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


def pop(self) -> int:


def top(self) -> int:


def empty(self) -> bool:



# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
```

```
# param_4 = obj.empty()
```

Python:

```
class MyStack(object):

def __init__(self):


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


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


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


def empty(self):
"""
:rtype: bool
"""




# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

JavaScript:

```
var MyStack = function() {

};

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

};

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

};

/**
* @return {number}
*/
MyStack.prototype.top = function() {

};

/**
* @return {boolean}
*/
MyStack.prototype.empty = function() {

};

/**
* Your MyStack object will be instantiated and called as such:
* var obj = new MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
```

```
* var param_4 = obj.empty()
*/
```

TypeScript:

```
class MyStack {
constructor() {

}

push(x: number): void {

}

pop(): number {

}

top(): number {

}

empty(): boolean {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* var obj = new MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.empty()
*/
```

C#:

```
public class MyStack {

public MyStack() {
```

```
}

public void Push(int x) {

}

public int Pop() {

}

public int Top() {

}

public bool Empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = new MyStack();
* obj.Push(x);
* int param_2 = obj.Pop();
* int param_3 = obj.Top();
* bool param_4 = obj.Empty();
*/
```

C:

```
typedef struct {

} MyStack;


MyStack* myStackCreate() {

}
```

```
void myStackPush(MyStack* obj, int x) {

}

int myStackPop(MyStack* obj) {

}

int myStackTop(MyStack* obj) {

}

bool myStackEmpty(MyStack* obj) {

}

void myStackFree(MyStack* obj) {

}

/**
* Your MyStack struct will be instantiated and called as such:
* MyStack* obj = myStackCreate();
* myStackPush(obj, x);

* int param_2 = myStackPop(obj);

* int param_3 = myStackTop(obj);

* bool param_4 = myStackEmpty(obj);

* myStackFree(obj);
*/
```

Go:

```
type MyStack struct {

}


func Constructor() MyStack {
```

```
}


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

}


func (this *MyStack) Pop() int {

}


func (this *MyStack) Top() int {

}


func (this *MyStack) Empty() bool {

}


/**
* Your MyStack object will be instantiated and called as such:
* obj := Constructor();
* obj.Push(x);
* param_2 := obj.Pop();
* param_3 := obj.Top();
* param_4 := obj.Empty();
*/
```

Kotlin:

```
class MyStack() {

fun push(x: Int) {

}

fun pop(): Int {
```

```
}

fun top(): Int {

}

fun empty(): Boolean {

}

}

/**
* Your MyStack object will be instantiated and called as such:
* var obj = MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.empty()
*/
```

Swift:

```
class MyStack {

init() {

}

func push(_ x: Int) {

}

func pop() -> Int {

}

func top() -> Int {

}
```

```
func empty() -> Bool {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* let obj = MyStack()
* obj.push(x)
* let ret_2: Int = obj.pop()
* let ret_3: Int = obj.top()
* let ret_4: Bool = obj.empty()
*/
```

Rust:

```
struct MyStack {

}


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

fn new() -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

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

```
}

fn empty(&self) -> bool {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* let obj = MyStack::new();
* obj.push(x);
* let ret_2: i32 = obj.pop();
* let ret_3: i32 = obj.top();
* let ret_4: bool = obj.empty();
*/
```

Ruby:

```
class MyStack
def initialize()

end


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

end


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

end


=begin
```

```
:rtype: Integer
=end
def top()

end


=begin
:rtype: Boolean
=end
def empty()

end



end

# Your MyStack object will be instantiated and called as such:
# obj = MyStack.new()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

PHP:

```
class MyStack {
/**
*/
function __construct() {

}

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

}

/**
```

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

}

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

}

/**
* @return Boolean
*/
function empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* $obj = MyStack();
* $obj->push($x);
* $ret_2 = $obj->pop();
* $ret_3 = $obj->top();
* $ret_4 = $obj->empty();
*/
```

Dart:

```
class MyStack {

MyStack() {

}

void push(int x) {

}
```

```
int pop() {

}

int top() {

}

bool empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = MyStack();
* obj.push(x);
* int param2 = obj.pop();
* int param3 = obj.top();
* bool param4 = obj.empty();
*/
```

Scala:

```
class MyStack() {

def push(x: Int): Unit = {

}

def pop(): Int = {

}

def top(): Int = {

}

def empty(): Boolean = {

}
```

```
}

/**
* Your MyStack object will be instantiated and called as such:
* val obj = new MyStack()
* obj.push(x)
* val param_2 = obj.pop()
* val param_3 = obj.top()
* val param_4 = obj.empty()
*/
```

Elixir:

```
defmodule MyStack do
@spec init_() :: any
def init_() do

end

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

end

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

end

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

end

@spec empty() :: boolean
def empty() do

end
end

# Your functions will be called as such:
# MyStack.init_()
```

```
# MyStack.push(x)
# param_2 = MyStack.pop()
# param_3 = MyStack.top()
# param_4 = MyStack.empty()

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

Erlang:

```
-spec my_stack_init_() -> any().
my_stack_init_() ->
.

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

-spec my_stack_pop() -> integer().
my_stack_pop() ->
.

-spec my_stack_top() -> integer().
my_stack_top() ->
.

-spec my_stack_empty() -> boolean().
my_stack_empty() ->
.


%% Your functions will be called as such:
%% my_stack_init_(),
%% my_stack_push(X),
%% Param_2 = my_stack_pop(),
%% Param_3 = my_stack_top(),
%% Param_4 = my_stack_empty(),

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

Racket:

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

(init-field)

; push : exact-integer? -> void?
(define/public (push x)
)
; pop : -> exact-integer?
(define/public (pop)
)
; top : -> exact-integer?
(define/public (top)
)
; empty : -> boolean?
(define/public (empty)
)))

;; Your my-stack% object will be instantiated and called as such:
;; (define obj (new my-stack%))
;; (send obj push x)
;; (define param_2 (send obj pop))
;; (define param_3 (send obj top))
;; (define param_4 (send obj empty))
```

Solutions

C++ Solution:

```
/*
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class MyStack {
public:
MyStack() {

}

void push(int x) {

}

int pop() {

}

int top() {

}

bool empty() {

}
};

/**
* Your MyStack object will be instantiated and called as such:
* MyStack* obj = new MyStack();
* obj->push(x);
* int param_2 = obj->pop();
* int param_3 = obj->top();
* bool param_4 = obj->empty();
*/
```

Java Solution:

```
/**
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
* Approach: Optimized algorithm based on problem constraints
```

```
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class MyStack {

public MyStack() {

}

public void push(int x) {

}

public int pop() {

}

public int top() {

}

public boolean empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = new MyStack();
* obj.push(x);
* int param_2 = obj.pop();
* int param_3 = obj.top();
* boolean param_4 = obj.empty();
*/
```

Python3 Solution:

```
"""
Problem: Implement Stack using Queues
Difficulty: Easy
```

```
Tags: stack, queue

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

class MyStack:

def __init__(self):


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

Python Solution:

```
class MyStack(object):

def __init__(self):


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


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


def top(self):
"""
:rtype: int
"""
```

```
def empty(self):
"""
:rtype: bool
"""



# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

JavaScript Solution:

```
/**
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/


var MyStack = function() {

};

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

};

/**
```

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

};

/**
* @return {number}
*/
MyStack.prototype.top = function() {

};

/**
* @return {boolean}
*/
MyStack.prototype.empty = function() {

};

/**
* Your MyStack object will be instantiated and called as such:
* var obj = new MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.empty()
*/
```

TypeScript Solution:

```
/**
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class MyStack {
constructor() {

}

push(x: number): void {

}

pop(): number {

}

top(): number {

}

empty(): boolean {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* var obj = new MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.empty()
*/
```

C# Solution:

```
/*
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

public class MyStack {

public MyStack() {

}

public void Push(int x) {

}

public int Pop() {

}

public int Top() {

}

public bool Empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = new MyStack();
* obj.Push(x);
* int param_2 = obj.Pop();
* int param_3 = obj.Top();
* bool param_4 = obj.Empty();
*/
```

C Solution:

```
/*
* Problem: Implement Stack using Queues
* Difficulty: Easy
* Tags: stack, queue
*
```

```
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/



typedef struct {

} MyStack;


MyStack* myStackCreate() {

}

void myStackPush(MyStack* obj, int x) {

}

int myStackPop(MyStack* obj) {

}

int myStackTop(MyStack* obj) {

}

bool myStackEmpty(MyStack* obj) {

}

void myStackFree(MyStack* obj) {

}

/**
* Your MyStack struct will be instantiated and called as such:
* MyStack* obj = myStackCreate();
* myStackPush(obj, x);
```

```
* int param_2 = myStackPop(obj);

* int param_3 = myStackTop(obj);

* bool param_4 = myStackEmpty(obj);

* myStackFree(obj);
*/
```

Go Solution:

```
// Problem: Implement Stack using Queues
// Difficulty: Easy
// Tags: stack, queue
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

type MyStack struct {

}


func Constructor() MyStack {

}


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

}


func (this *MyStack) Pop() int {

}


func (this *MyStack) Top() int {
```

```
}


func (this *MyStack) Empty() bool {

}


/**
* Your MyStack object will be instantiated and called as such:
* obj := Constructor();
* obj.Push(x);
* param_2 := obj.Pop();
* param_3 := obj.Top();
* param_4 := obj.Empty();
*/
```

Kotlin Solution:

```
class MyStack() {

fun push(x: Int) {

}

fun pop(): Int {

}

fun top(): Int {

}

fun empty(): Boolean {

}

}

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

```
* var obj = MyStack()
* obj.push(x)
* var param_2 = obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.empty()
*/
```

Swift Solution:

```
class MyStack {

init() {

}

func push(_ x: Int) {

}

func pop() -> Int {

}

func top() -> Int {

}

func empty() -> Bool {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* let obj = MyStack()
* obj.push(x)
* let ret_2: Int = obj.pop()
* let ret_3: Int = obj.top()
* let ret_4: Bool = obj.empty()
*/
```

Rust Solution:

```
// Problem: Implement Stack using Queues
// Difficulty: Easy
// Tags: stack, queue
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

struct MyStack {

}



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

fn new() -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

fn top(&self) -> i32 {

}

fn empty(&self) -> bool {

}
}
```

```
/**
* Your MyStack object will be instantiated and called as such:
* let obj = MyStack::new();
* obj.push(x);
* let ret_2: i32 = obj.pop();
* let ret_3: i32 = obj.top();
* let ret_4: bool = obj.empty();
*/
```

Ruby Solution:

```
class MyStack
def initialize()

end


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

end


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

end


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

end
```

```
=begin
:rtype: Boolean
=end
def empty()

end


end

# Your MyStack object will be instantiated and called as such:
# obj = MyStack.new()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

PHP Solution:

```
class MyStack {
/**
*/
function __construct() {

}

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

}

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

}
```

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

}

/**
* @return Boolean
*/
function empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* $obj = MyStack();
* $obj->push($x);
* $ret_2 = $obj->pop();
* $ret_3 = $obj->top();
* $ret_4 = $obj->empty();
*/
```

Dart Solution:

```
class MyStack {

MyStack() {

}

void push(int x) {

}

int pop() {

}
```

```
int top() {

}

bool empty() {

}
}

/**
* Your MyStack object will be instantiated and called as such:
* MyStack obj = MyStack();
* obj.push(x);
* int param2 = obj.pop();
* int param3 = obj.top();
* bool param4 = obj.empty();
*/
```

Scala Solution:

```
class MyStack() {

def push(x: Int): Unit = {

}

def pop(): Int = {

}

def top(): Int = {

}

def empty(): Boolean = {

}

}

/**
```

```
* Your MyStack object will be instantiated and called as such:
* val obj = new MyStack()
* obj.push(x)
* val param_2 = obj.pop()
* val param_3 = obj.top()
* val param_4 = obj.empty()
*/
```

Elixir Solution:

```
defmodule MyStack do
@spec init_() :: any
def init_() do

end

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

end

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

end

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

end

@spec empty() :: boolean
def empty() do

end
end

# Your functions will be called as such:
# MyStack.init_()
# MyStack.push(x)
# param_2 = MyStack.pop()
```

```
# param_3 = MyStack.top()
# param_4 = MyStack.empty()

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

Erlang Solution:

```
-spec my_stack_init_() -> any().
my_stack_init_() ->
.

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

-spec my_stack_pop() -> integer().
my_stack_pop() ->
.

-spec my_stack_top() -> integer().
my_stack_top() ->
.

-spec my_stack_empty() -> boolean().
my_stack_empty() ->
.


%% Your functions will be called as such:
%% my_stack_init_(),
%% my_stack_push(X),
%% Param_2 = my_stack_pop(),
%% Param_3 = my_stack_top(),
%% Param_4 = my_stack_empty(),

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

Racket Solution:

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

(init-field)

; push : exact-integer? -> void?
(define/public (push x)
)
; pop : -> exact-integer?
(define/public (pop)
)
; top : -> exact-integer?
(define/public (top)
)
; empty : -> boolean?
(define/public (empty)
)))

;; Your my-stack% object will be instantiated and called as such:
;; (define obj (new my-stack%))
;; (send obj push x)
;; (define param_2 (send obj pop))
;; (define param_3 (send obj top))
;; (define param_4 (send obj empty))
```