Problem 232: Implement Queue using Stacks

Problem Information

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

Problem Description

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (

push

,

peek

,

pop

, and

empty

).

Implement the

MyQueue

class:

void push(int x)

Pushes element x to the back of the queue.

int pop()

Removes the element from the front of the queue and returns it.

int peek()

Returns the element at the front of the queue.

boolean empty()

Returns

true

if the queue is empty,

false

otherwise.

Notes:

You must use

only

standard operations of a stack, which means only

push to top

,

peek/pop from top

,

size

, and

is empty

operations are valid.

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

Example 1:

Input

["MyQueue", "push", "push", "peek", "pop", "empty"] [[], [1], [2], [], [], []]

Output

[null, null, null, 1, 1, false]

Explanation

MyQueue myQueue = new MyQueue(); myQueue.push(1); // queue is: [1] myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue) myQueue.peek(); // return 1 myQueue.pop(); // return 1, queue is [2] myQueue.empty(); // return false

Constraints:

1 <= x <= 9

At most

100

calls will be made to

push

,

pop

,

peek

, and

empty

.

All the calls to

pop

and

peek

are valid.

Follow-up:

Can you implement the queue such that each operation is

amortized

O(1)

time complexity? In other words, performing

n

operations will take overall

O(n)

time even if one of those operations may take longer.

Code Snippets

C++:

```
class MyQueue {
public:
MyQueue() {

}

void push(int x) {

}

int pop() {

}

int peek() {

}

bool empty() {

}
};

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

Java:

```
class MyQueue {

public MyQueue() {

}

public void push(int x) {

}

public int pop() {

}

public int peek() {

}

public boolean empty() {

}
}

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

Python3:

```
class MyQueue:

def __init__(self):



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



def pop(self) -> int:
```

```
def peek(self) -> int:


def empty(self) -> bool:



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

Python:

```
class MyQueue(object):

def __init__(self):


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


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


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

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



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

JavaScript:

```
var MyQueue = function() {

};

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

};

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

};

/**
* @return {number}
*/
MyQueue.prototype.peek = function() {

};
```

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

};

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

TypeScript:

```
class MyQueue {
constructor() {

}

push(x: number): void {

}

pop(): number {

}

peek(): number {

}

empty(): boolean {

}
}

/**
```

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

C#:

```
public class MyQueue {

public MyQueue() {

}

public void Push(int x) {

}

public int Pop() {

}

public int Peek() {

}

public bool Empty() {

}
}

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

C:

```
typedef struct {

} MyQueue;


MyQueue* myQueueCreate() {

}

void myQueuePush(MyQueue* obj, int x) {

}

int myQueuePop(MyQueue* obj) {

}

int myQueuePeek(MyQueue* obj) {

}

bool myQueueEmpty(MyQueue* obj) {

}

void myQueueFree(MyQueue* obj) {

}

/**
* Your MyQueue struct will be instantiated and called as such:
* MyQueue* obj = myQueueCreate();
* myQueuePush(obj, x);

* int param_2 = myQueuePop(obj);

* int param_3 = myQueuePeek(obj);
* bool param_4 = myQueueEmpty(obj);

* myQueueFree(obj);
*/
```

Go:

```
type MyQueue struct {

}


func Constructor() MyQueue {

}


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

}


func (this *MyQueue) Pop() int {

}


func (this *MyQueue) Peek() int {

}


func (this *MyQueue) Empty() bool {

}


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

Kotlin:

```
class MyQueue() {

fun push(x: Int) {

}

fun pop(): Int {

}

fun peek(): Int {

}

fun empty(): Boolean {

}

}

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

Swift:

```
class MyQueue {

init() {
}

func push(_ x: Int) {

}

func pop() -> Int {

}

func peek() -> Int {

}

func empty() -> Bool {

}
}

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

Rust:

```
struct MyQueue {

}


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

```
fn new() -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

fn peek(&self) -> i32 {

}

fn empty(&self) -> bool {

}
}

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

Ruby:

```
class MyQueue
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 peek()

end



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

end



end

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

PHP:

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

}

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

}

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

}

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

}

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

}
}

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

```
*/
```

Dart:

```
class MyQueue {

MyQueue() {

}

void push(int x) {

}

int pop() {

}

int peek() {

}

bool empty() {

}
}

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

Scala:

```
class MyQueue() {

def push(x: Int): Unit = {
}

def pop(): Int = {

}

def peek(): Int = {

}

def empty(): Boolean = {

}

}

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

Elixir:

```
defmodule MyQueue 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 peek() :: integer
def peek() do

end

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

end
end

# Your functions will be called as such:
# MyQueue.init_()
# MyQueue.push(x)
# param_2 = MyQueue.pop()
# param_3 = MyQueue.peek()
# param_4 = MyQueue.empty()

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

Erlang:

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

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

-spec my_queue_pop() -> integer().
my_queue_pop() ->
.

-spec my_queue_peek() -> integer().
my_queue_peek() ->
.

-spec my_queue_empty() -> boolean().
my_queue_empty() ->
```

```
.



%% Your functions will be called as such:
%% my_queue_init_(),
%% my_queue_push(X),
%% Param_2 = my_queue_pop(),
%% Param_3 = my_queue_peek(),
%% Param_4 = my_queue_empty(),

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

Racket:

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

(init-field)

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

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

Solutions

C++ Solution:

```
/*
* Problem: Implement Queue using Stacks
* 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 MyQueue {
public:
MyQueue() {

}

void push(int x) {

}

int pop() {

}

int peek() {

}

bool empty() {

}
};

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

Java Solution:

```
/**
* Problem: Implement Queue using Stacks
* 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 MyQueue {

public MyQueue() {

}

public void push(int x) {

}

public int pop() {

}

public int peek() {

}

public boolean empty() {

}
}

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

Python3 Solution:

```
"""
Problem: Implement Queue using Stacks
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 MyQueue:

def __init__(self):



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

Python Solution:

```
class MyQueue(object):

def __init__(self):



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

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



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



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




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

JavaScript Solution:

```
/**
* Problem: Implement Queue using Stacks
* 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 MyQueue = function() {
};

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

};

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

};

/**
* @return {number}
*/
MyQueue.prototype.peek = function() {

};

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

};

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

TypeScript Solution:

```
/**
* Problem: Implement Queue using Stacks
* 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 MyQueue {
constructor() {

}

push(x: number): void {

}

pop(): number {

}

peek(): number {

}

empty(): boolean {

}
}

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

C# Solution:

```
/*
* Problem: Implement Queue using Stacks
* 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 MyQueue {

public MyQueue() {

}

public void Push(int x) {

}

public int Pop() {

}

public int Peek() {

}

public bool Empty() {

}
}

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

C Solution:

```
/*
* Problem: Implement Queue using Stacks
* 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 {

} MyQueue;


MyQueue* myQueueCreate() {

}

void myQueuePush(MyQueue* obj, int x) {

}

int myQueuePop(MyQueue* obj) {

}

int myQueuePeek(MyQueue* obj) {

}

bool myQueueEmpty(MyQueue* obj) {

}

void myQueueFree(MyQueue* obj) {

}
```

```
/**
* Your MyQueue struct will be instantiated and called as such:
* MyQueue* obj = myQueueCreate();
* myQueuePush(obj, x);

* int param_2 = myQueuePop(obj);

* int param_3 = myQueuePeek(obj);

* bool param_4 = myQueueEmpty(obj);

* myQueueFree(obj);
*/
```

Go Solution:

```
// Problem: Implement Queue using Stacks
// 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 MyQueue struct {

}


func Constructor() MyQueue {

}


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

}


func (this *MyQueue) Pop() int {
}


func (this *MyQueue) Peek() int {

}


func (this *MyQueue) Empty() bool {

}


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

Kotlin Solution:

```
class MyQueue() {

fun push(x: Int) {

}

fun pop(): Int {

}

fun peek(): Int {

}

fun empty(): Boolean {

}

}

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

Swift Solution:

```

class MyQueue {

init() {

}

func push(_ x: Int) {

}

func pop() -> Int {

}

func peek() -> Int {

}

func empty() -> Bool {

}
}

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

```
* let obj = MyQueue()
* obj.push(x)
* let ret_2: Int = obj.pop()
* let ret_3: Int = obj.peek()
* let ret_4: Bool = obj.empty()
*/
```

Rust Solution:

```
// Problem: Implement Queue using Stacks
// 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 MyQueue {

}



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

fn new() -> Self {

}

fn push(&self, x: i32) {

}

fn pop(&self) -> i32 {

}

fn peek(&self) -> i32 {
}

fn empty(&self) -> bool {

}
}

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

Ruby Solution:

```
class MyQueue
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 peek()

end


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

end


end

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

PHP Solution:

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

}

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

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

}

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

}

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

}
}

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

Dart Solution:

```
class MyQueue {

MyQueue() {

}
void push(int x) {

}

int pop() {

}

int peek() {

}

bool empty() {

}
}

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

Scala Solution:

```
class MyQueue() {

def push(x: Int): Unit = {

}

def pop(): Int = {

}

def peek(): Int = {

}
def empty(): Boolean = {

}

}

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

Elixir Solution:

```
defmodule MyQueue 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 peek() :: integer
def peek() do

end

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

```
end
end

# Your functions will be called as such:
# MyQueue.init_()
# MyQueue.push(x)
# param_2 = MyQueue.pop()
# param_3 = MyQueue.peek()
# param_4 = MyQueue.empty()

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

Erlang Solution:

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

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

-spec my_queue_pop() -> integer().
my_queue_pop() ->
.

-spec my_queue_peek() -> integer().
my_queue_peek() ->
.

-spec my_queue_empty() -> boolean().
my_queue_empty() ->
.



%% Your functions will be called as such:
%% my_queue_init_(),
%% my_queue_push(X),
%% Param_2 = my_queue_pop(),
%% Param_3 = my_queue_peek(),
%% Param_4 = my_queue_empty(),

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

Racket Solution:

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

(init-field)

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

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