Problem 155: Min Stack

Problem Information

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

Problem Description

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the

MinStack

class:

MinStack()

initializes the stack object.

void push(int val)

pushes the element

val

onto the stack.

void pop()

removes the element on the top of the stack.

int top()

gets the top element of the stack.

int getMin()

retrieves the minimum element in the stack.

You must implement a solution with

O(1)

time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"] [[],[-2],[0],[-3],[],[],[],[]]

Output

[null,null,null,null,-3,null,0,-2]

Explanation

MinStack minStack = new MinStack(); minStack.push(-2); minStack.push(0); minStack.push(-3); minStack.getMin(); // return -3 minStack.pop(); minStack.top(); // return 0 minStack.getMin(); // return -2

Constraints:

-2

31

<= val <= 2

31

- 1

Methods

pop

,

top

and

getMin

operations will always be called on

non-empty

stacks.

At most

3 * 10

4

calls will be made to

push

,

pop

,

top

, and

getMin

.

Code Snippets

C++:

```
class MinStack {
public:
MinStack() {

}

void push(int val) {

}

void pop() {

}

int top() {

}

int getMin() {

}
};

/**
* Your MinStack object will be instantiated and called as such:
* MinStack* obj = new MinStack();
* obj->push(val);
* obj->pop();
* int param_3 = obj->top();
* int param_4 = obj->getMin();
*/
```

Java:

```
class MinStack {

public MinStack() {

}

public void push(int val) {

}

public void pop() {

}

public int top() {

}

public int getMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = new MinStack();
* obj.push(val);
* obj.pop();
* int param_3 = obj.top();
* int param_4 = obj.getMin();
*/
```

Python3:

```
class MinStack:

def __init__(self):



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



def pop(self) -> None:
```

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


def getMin(self) -> int:



# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

Python:

```
class MinStack(object):

def __init__(self):


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


def pop(self):
"""
:rtype: None
"""


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

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



# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

JavaScript:

```
var MinStack = function() {

};

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

};

/**
* @return {void}
*/
MinStack.prototype.pop = function() {

};

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

};
```

```
/**
* @return {number}
*/
MinStack.prototype.getMin = function() {

};

/**
* Your MinStack object will be instantiated and called as such:
* var obj = new MinStack()
* obj.push(val)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

TypeScript:

```
class MinStack {
constructor() {

}

push(val: number): void {

}

pop(): void {

}

top(): number {

}

getMin(): number {

}
}

/**
```

```
* Your MinStack object will be instantiated and called as such:
* var obj = new MinStack()
* obj.push(val)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

C#:

```
public class MinStack {

public MinStack() {

}

public void Push(int val) {

}

public void Pop() {

}

public int Top() {

}

public int GetMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = new MinStack();
* obj.Push(val);
* obj.Pop();
* int param_3 = obj.Top();
* int param_4 = obj.GetMin();
*/
```

C:

```
typedef struct {

} MinStack;


MinStack* minStackCreate() {

}

void minStackPush(MinStack* obj, int val) {

}

void minStackPop(MinStack* obj) {

}

int minStackTop(MinStack* obj) {

}

int minStackGetMin(MinStack* obj) {

}

void minStackFree(MinStack* obj) {

}

/**
* Your MinStack struct will be instantiated and called as such:
* MinStack* obj = minStackCreate();
* minStackPush(obj, val);

* minStackPop(obj);

* int param_3 = minStackTop(obj);
* int param_4 = minStackGetMin(obj);

* minStackFree(obj);
*/
```

Go:

```
type MinStack struct {

}


func Constructor() MinStack {

}


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

}


func (this *MinStack) Pop() {

}


func (this *MinStack) Top() int {

}


func (this *MinStack) GetMin() int {

}


/**
* Your MinStack object will be instantiated and called as such:
* obj := Constructor();
* obj.Push(val);
* obj.Pop();
* param_3 := obj.Top();
* param_4 := obj.GetMin();
*/
```

Kotlin:

```
class MinStack() {

fun push(`val`: Int) {

}

fun pop() {

}

fun top(): Int {

}

fun getMin(): Int {

}

}

/**
* Your MinStack object will be instantiated and called as such:
* var obj = MinStack()
* obj.push(`val`)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

Swift:

```
class MinStack {

init() {
}

func push(_ val: Int) {

}

func pop() {

}

func top() -> Int {

}

func getMin() -> Int {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* let obj = MinStack()
* obj.push(val)
* obj.pop()
* let ret_3: Int = obj.top()
* let ret_4: Int = obj.getMin()
*/
```

Rust:

```
struct MinStack {

}



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

```
fn new() -> Self {

}

fn push(&self, val: i32) {

}

fn pop(&self) {

}

fn top(&self) -> i32 {

}

fn get_min(&self) -> i32 {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* let obj = MinStack::new();
* obj.push(val);
* obj.pop();
* let ret_3: i32 = obj.top();
* let ret_4: i32 = obj.get_min();
*/
```

Ruby:

```
class MinStack
def initialize()

end



=begin
:type val: Integer
:rtype: Void
=end
```

```
def push(val)

end


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

end


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

end


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

end


end

# Your MinStack object will be instantiated and called as such:
# obj = MinStack.new()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.get_min()
```

PHP:

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

}

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

}

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

}

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

}

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

}
}

/**
* Your MinStack object will be instantiated and called as such:
* $obj = MinStack();
* $obj->push($val);
* $obj->pop();
* $ret_3 = $obj->top();
* $ret_4 = $obj->getMin();
```

```
*/
```

Dart:

```
class MinStack {

MinStack() {

}

void push(int val) {

}

void pop() {

}

int top() {

}

int getMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = MinStack();
* obj.push(val);
* obj.pop();
* int param3 = obj.top();
* int param4 = obj.getMin();
*/
```

Scala:

```
class MinStack() {

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

def pop(): Unit = {

}

def top(): Int = {

}

def getMin(): Int = {

}

}

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

Elixir:

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

end

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

end

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

end
```

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

end

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

end
end

# Your functions will be called as such:
# MinStack.init_()
# MinStack.push(val)
# MinStack.pop()
# param_3 = MinStack.top()
# param_4 = MinStack.get_min()

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

Erlang:

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

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

-spec min_stack_pop() -> any().
min_stack_pop() ->
.

-spec min_stack_top() -> integer().
min_stack_top() ->
.

-spec min_stack_get_min() -> integer().
min_stack_get_min() ->
```

```
.


%% Your functions will be called as such:
%% min_stack_init_(),
%% min_stack_push(Val),
%% min_stack_pop(),
%% Param_3 = min_stack_top(),
%% Param_4 = min_stack_get_min(),

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

Racket:

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

(init-field)

; push : exact-integer? -> void?
(define/public (push val)
)
; pop : -> void?
(define/public (pop)
)
; top : -> exact-integer?
(define/public (top)
)
; get-min : -> exact-integer?
(define/public (get-min)
)))

;; Your min-stack% object will be instantiated and called as such:
;; (define obj (new min-stack%))
;; (send obj push val)
;; (send obj pop)
;; (define param_3 (send obj top))
;; (define param_4 (send obj get-min))
```

Solutions

C++ Solution:

```
/*
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 MinStack {
public:
MinStack() {

}

void push(int val) {

}

void pop() {

}

int top() {

}

int getMin() {

}
};

/**
* Your MinStack object will be instantiated and called as such:
* MinStack* obj = new MinStack();
* obj->push(val);
* obj->pop();
* int param_3 = obj->top();
* int param_4 = obj->getMin();
*/
```

Java Solution:

```
/**
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 MinStack {

public MinStack() {

}

public void push(int val) {

}

public void pop() {

}

public int top() {

}

public int getMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = new MinStack();
* obj.push(val);
* obj.pop();
* int param_3 = obj.top();
* int param_4 = obj.getMin();
*/
```

Python3 Solution:

```
"""
Problem: Min Stack
Difficulty: Medium
Tags: stack

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 MinStack:

def __init__(self):



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

Python Solution:

```
class MinStack(object):

def __init__(self):



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

```
def pop(self):
"""
:rtype: None
"""


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


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



# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

JavaScript Solution:

```
/**
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 MinStack = function() {
};

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

};

/**
* @return {void}
*/
MinStack.prototype.pop = function() {

};

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

};

/**
* @return {number}
*/
MinStack.prototype.getMin = function() {

};

/**
* Your MinStack object will be instantiated and called as such:
* var obj = new MinStack()
* obj.push(val)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

TypeScript Solution:

```
/**
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 MinStack {
constructor() {

}

push(val: number): void {

}

pop(): void {

}

top(): number {

}

getMin(): number {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* var obj = new MinStack()
* obj.push(val)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

C# Solution:

```
/*
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 MinStack {

public MinStack() {

}

public void Push(int val) {

}

public void Pop() {

}

public int Top() {

}

public int GetMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = new MinStack();
* obj.Push(val);
* obj.Pop();
* int param_3 = obj.Top();
* int param_4 = obj.GetMin();
*/
```

C Solution:

```
/*
* Problem: Min Stack
* Difficulty: Medium
* Tags: stack
*
* 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 {

} MinStack;


MinStack* minStackCreate() {

}

void minStackPush(MinStack* obj, int val) {

}

void minStackPop(MinStack* obj) {

}

int minStackTop(MinStack* obj) {

}

int minStackGetMin(MinStack* obj) {

}

void minStackFree(MinStack* obj) {

}
```

```
/**
* Your MinStack struct will be instantiated and called as such:
* MinStack* obj = minStackCreate();
* minStackPush(obj, val);

* minStackPop(obj);

* int param_3 = minStackTop(obj);

* int param_4 = minStackGetMin(obj);

* minStackFree(obj);
*/
```

Go Solution:

```
// Problem: Min Stack
// Difficulty: Medium
// Tags: stack
//
// 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 MinStack struct {

}


func Constructor() MinStack {

}


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

}


func (this *MinStack) Pop() {
}


func (this *MinStack) Top() int {

}


func (this *MinStack) GetMin() int {

}


/**
* Your MinStack object will be instantiated and called as such:
* obj := Constructor();
* obj.Push(val);
* obj.Pop();
* param_3 := obj.Top();
* param_4 := obj.GetMin();
*/
```

Kotlin Solution:

```
class MinStack() {

fun push(`val`: Int) {

}

fun pop() {

}

fun top(): Int {

}

fun getMin(): Int {

}

}

/**
* Your MinStack object will be instantiated and called as such:
* var obj = MinStack()
* obj.push(`val`)
* obj.pop()
* var param_3 = obj.top()
* var param_4 = obj.getMin()
*/
```

Swift Solution:

```

class MinStack {

init() {

}

func push(_ val: Int) {

}

func pop() {

}

func top() -> Int {

}

func getMin() -> Int {

}
}

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

```
* let obj = MinStack()
* obj.push(val)
* obj.pop()
* let ret_3: Int = obj.top()
* let ret_4: Int = obj.getMin()
*/
```

Rust Solution:

```
// Problem: Min Stack
// Difficulty: Medium
// Tags: stack
//
// 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 MinStack {

}


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

fn new() -> Self {

}

fn push(&self, val: i32) {

}

fn pop(&self) {

}

fn top(&self) -> i32 {
}

fn get_min(&self) -> i32 {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* let obj = MinStack::new();
* obj.push(val);
* obj.pop();
* let ret_3: i32 = obj.top();
* let ret_4: i32 = obj.get_min();
*/
```

Ruby Solution:

```
class MinStack
def initialize()

end



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

end



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

end
```

```


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

end



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

end



end

# Your MinStack object will be instantiated and called as such:
# obj = MinStack.new()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.get_min()
```

PHP Solution:

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

}

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

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

}

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

}

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

}
}

/**
* Your MinStack object will be instantiated and called as such:
* $obj = MinStack();
* $obj->push($val);
* $obj->pop();
* $ret_3 = $obj->top();
* $ret_4 = $obj->getMin();
*/
```

Dart Solution:

```
class MinStack {

MinStack() {

}
void push(int val) {

}

void pop() {

}

int top() {

}

int getMin() {

}
}

/**
* Your MinStack object will be instantiated and called as such:
* MinStack obj = MinStack();
* obj.push(val);
* obj.pop();
* int param3 = obj.top();
* int param4 = obj.getMin();
*/
```

Scala Solution:

```
class MinStack() {

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

}

def pop(): Unit = {

}

def top(): Int = {

}
def getMin(): Int = {

}

}

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

Elixir Solution:

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

end

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

end

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

end

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

end

@spec get_min() :: integer
def get_min() do
```

```
end
end

# Your functions will be called as such:
# MinStack.init_()
# MinStack.push(val)
# MinStack.pop()
# param_3 = MinStack.top()
# param_4 = MinStack.get_min()

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

Erlang Solution:

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

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

-spec min_stack_pop() -> any().
min_stack_pop() ->
.

-spec min_stack_top() -> integer().
min_stack_top() ->
.

-spec min_stack_get_min() -> integer().
min_stack_get_min() ->
.



%% Your functions will be called as such:
%% min_stack_init_(),
%% min_stack_push(Val),
%% min_stack_pop(),
%% Param_3 = min_stack_top(),
%% Param_4 = min_stack_get_min(),

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

Racket Solution:

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

(init-field)

; push : exact-integer? -> void?
(define/public (push val)
)
; pop : -> void?
(define/public (pop)
)
; top : -> exact-integer?
(define/public (top)
)
; get-min : -> exact-integer?
(define/public (get-min)
)))

;; Your min-stack% object will be instantiated and called as such:
;; (define obj (new min-stack%))
;; (send obj push val)
;; (send obj pop)
;; (define param_3 (send obj top))
;; (define param_4 (send obj get-min))
```