Medium Stack Design Leetcode Link

The problem requires designing a stack data structure with not only the standard operations - push, pop, and top - but also the ability

Problem Description

155. Min Stack

to retrieve the minimum element present in the stack at any time, all in constant time O(1). This is quite challenging because the standard stack data structure does not keep track of the minimum element, and searching for the minimum element typically requires O(n) time where n is the number of elements in the stack. To summarize, MinStack class should support these operations:

push(int val) that adds an element to the stack.

pop() that removes the element at the top of the stack.

MinStack() constructor that initializes the stack.

- top() that retrieves the element at the top of the stack without removing it.
- getMin() that retrieves the minimum element present in the stack.
- Each of these operations must be done in O(1) time complexity. Intuition

To maintain the minimum element information in constant time, we need a strategy that keeps track of the minimum at each state of

the stack. The solution uses two stacks:

 The first stack stk1 behaves like a normal stack, holding all the elements. The second stack stk2 keeps track of minimum elements. Each element in stk2 corresponds to the minimum element of stk1 up to that point.

With each push, we insert the value into stk1 and also update the stk2 with the minimum of the new value and the current minimum,

which is the top element of stk2.

how the solution is implemented for each function of the MinStack class:

Append the value val to the primary stack self.stkl.

Upon pop, we simply remove the top elements from both stk1 and stk2. Given stk2 is always in sync with stk1 but only storing minimums, it ensures that stk2's top always represents the current minimum of stk1.

This design allows for the stack to operate as normal while having a supporting structure, stk2, that efficiently tracks the minimum

element. Thus, all the required operations run in constant time, fulfilling the problem constraints.

Solution Approach The solution involves using two stacks to keep track of the minimum element in conjunction with the main stack operations. Here is

Append inf (infinity) to self.stk2 to handle the edge case when the first element is pushed onto the stack.

top is straightforward as we return the top element of stk1 whereas getMin returns the top element of stk2.

Initialize two empty lists, self.stk1 and self.stk2. The first list self.stk1 is used as the primary stack to store the values. The second list self.stk2 serves as an auxiliary stack to store the minimum values seen so far.

init (self)

push(self, val: int)

• To maintain the minimum in self.stk2, append the minimum of the new value val and the last element in self.stk2 (which is the

current minimum) to self.stk2.

pop(self)

top(self)

 Pop the top element from self.stk1, which is the standard stack operation. Similarly, pop the top element from self.stk2. This ensures that after the pop operation, the minimum value is updated correspondingly in self.stk2.

• Return the top element of self.stk1 by accessing the last element in the list self.stk1[-1] without removing it. This is the

• Return the top element of self.stk2 by accessing the last element in the list self.stk2[-1]. Since self.stk2 is maintained in a

standard 'peek' operation in stack. getMin(self)

way that its top always contains the minimum element of the stack, this gives us the current minimum in constant time.

In conclusion, the algorithm leverages an auxiliary stack self.stk2 that mirrors the push and pop operations of the primary stack

self.stk1. It cleverly keeps track of the minimum element at each level of the stack as elements are added and removed, which

and ensures that all operations - push, pop, top, and getMin — remain constant time 0(1) operations.

allows for constant time retrieval of the minimum element. This elegant solution satisfies the constraints of the problem statement

1 stk1: []

[inf, 5]

1 push(3)

1 push(7)

3 is smaller).

1 getMin()

1 getMin()

2 stk2: [inf]

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. 1. Construct a MinStack. Initially, both stk1 and stk2 are empty:

inf in stk2 is used as a placeholder to simplify the push operation even when the stack is empty. 2. Push the value 5 onto the stack. 1 push(5) stk1 now holds: [5] stk2 updates its minimum value by comparing the current minimum (inf) and 5, choosing the smaller one:

stk1 now holds: [5, 3, 7] stk2 updates by comparing 3 (current minimum in stk2) and 7, which results in: [inf, 5, 3, 3] (since

After popping, stk1 holds: [5, 3] And stk2 synchronously pops as well to maintain the minimum: [inf, 5, 3].

By following the above steps for each operation, it is demonstrated that the auxiliary stack stk2 accurately keeps track of the

Add the minimum value to the min_stack which is the minimum of the new value and the current minimum

minimum elements for the primary stack stk1 at each point of operation, ensuring that getMin() can always retrieve the current

stk1 now holds: [5, 3] stk2 updates: [inf, 5, 3] as 3 is the new minimum. 4. Push the value 7 onto the stack.

3. Push the value 3 onto the stack.

5. Retrieve the current minimum.

6. Pop the top element off the stack. 1 pop()

With the last 7 removed, the new top of stk2 and thus the current minimum is now: 3.

The top of stk2 gives us the minimum: 3.

1 top() The top of stk1 gives us the last pushed value which is not removed yet: 3.

minimum in constant time, 0(1).

Python Solution

13

14

15

16

17

18

19

20

21

22

23

24

25

26

10

11

12

13

14

15

16

19

20

21

22

23

25

26

27

28

29

30

32

33

34

35

36

37

8. Retrieve the top element.

Retrieve the new current minimum.

class MinStack: def __init__(self): # Initialize two stacks; one to hold the actual stack values, # and the other to keep track of the minimum value at any given point. self.stack = [] self.min_stack = [float('inf')] # Initialize with infinity to handle edge case for the first element pushed

def pop(self) -> None:

def top(self) -> int:

28 # min_stack = MinStack()

29 # min_stack.push(-2)

30 # min_stack.push(0)

class MinStack {

public MinStack() {

public void push(int val) {

stack.push(val);

public void pop() {

stack.pop();

public int top() {

public int getMin() {

minValuesStack.pop();

return stack.peek();

self.stack.pop()

self.min_stack.pop()

return self.stack[-1]

27 # Example of how the MinStack class can be used:

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

self.stack.append(val)

Add the value to the main stack

Return the top value of the main stack

// stkl keeps track of all the elements in the stack.

private Deque<Integer> minValuesStack = new ArrayDeque<>();

minValuesStack.push(Math.min(val, minValuesStack.peek()));

// Method to retrieve the top element of the stack without removing it.

private Deque<Integer> stack = new ArrayDeque<>();

minValuesStack.push(Integer.MAX_VALUE);

// Remove the top element of the stack.

// Peek the top element of the stack.

// Method to get the current minimum value in the stack.

// Push the value onto the stack.

self.min_stack.append(min(val, self.min_stack[-1]))

Remove the top value from both main stack and min_stack

def get_min(self) -> int: # Return the current minimum value which is the top value of the min_stack return self.min_stack[-1]

// minValuesStack keeps track of the minimum values at each state of the stack.

// Method to push an element onto the stack. Updates the minimum value as well.

// Method to remove the top element from the stack. Also updates the minimum value.

// Peek the top element of the minValuesStack, which is the current minimum.

// Constructor initializes the minValuesStack with the maximum value an integer can hold.

// Push the smaller between the current value and the current minimum onto the minValuesStack.

// Remove the top element of the minValuesStack, which corresponds to the minimum at that state.

```
31 # min_stack.push(-3)
32 # print(min_stack.get_min()) # returns -3
33 # min_stack.pop()
34 # print(min_stack.top())
                            # returns 0
35 # print(min_stack.get_min()) # returns -2
36
Java Solution
```

return minValuesStack.peek(); 38 39 40 } 41

```
C++ Solution
 1 #include <stack>
   #include <climits> // needed for INT_MAX
   using namespace std;
   class MinStack {
   public:
       // Constructor initializes the auxiliary stack with the maximum integer value.
       MinStack() {
           minValuesStack.push(INT_MAX);
10
11
12
       // Pushes a value onto the stack and updates the minimum value stack.
13
       void push(int val) {
           mainStack.push(val);
14
15
           // Push the current minimum value onto the auxiliary stack.
           // Compares the new value with the current top of the minValuesStack.
16
           minValuesStack.push(std::min(val, minValuesStack.top()));
18
19
       // Pops the top element from both the main stack and the minimum value stack.
20
       void pop() {
21
22
           mainStack.pop();
           minValuesStack.pop();
24
25
26
       // Retrieves the top element of the main stack.
27
       int top() {
28
           return mainStack.top();
29
30
31
       // Retrieves the current minimum value from the auxiliary stack.
32
       int getMin() {
33
           return minValuesStack.top();
34
35
   private:
37
       // Main stack to store the elements.
       stack<int> mainStack;
38
       // Auxiliary stack to store the minimum values.
39
       stack<int> minValuesStack;
40
41 };
42
43 /**
   * The following comments illustrate how a MinStack object is used:
   * MinStack* obj = new MinStack();
    * obj->push(val);
    * obj->pop();
```

minStack.pop(); 18 19 } 20 21 /** * Retrieves the element on top of the stack.

function top(): number {

function getMin(): number {

constant time complexity.

function pop(): void {

stack.pop();

* int param_3 = obj->top();

Typescript Solution

stack.push(val);

50

51

/**

*/

9

11 }

10

12

24

26

28

29

34

36

35 }

27 }

*/

* int param_4 = obj->getMin();

function push(val: number): void {

* @return {number} The top element of the stack.

* Retrieves the minimum element from the stack.

return minStack[minStack.length - 1];

Time and Space Complexity

* @return {number} The current minimum value in the stack.

operations, can retrieve the smallest element in the stack in constant time.

return stack[stack.length - 1];

1 let stack: number[] = []; // This is the primary stack for storing numbers.

* Pushes a number onto the stack and updates the minStack appropriately.

* Removes the element on top of the stack and also updates the minStack.

* @param {number} val - The value to be pushed onto the stack.

minStack.push(Math.min(val, minStack[minStack.length - 1]));

let minStack: number[] = [Infinity]; // This stack keeps track of the minimum values.

```
Time Complexity:
  • <u>init</u>: Initializing the MinStack involves setting up two lists (stacks), which is an O(1) operation.
  • push: This method appends a new element to the stk1 and computes the minimum with the last element of stk2 to append to
    stk2. Both operations are 0(1) since appending to a list in Python and retrieving the last element have constant time complexity.
```

The provided Python code defines a class MinStack which implements a stack that, in addition to the typical push and pop

• pop: Here, we pop from both stk1 and stk2, which are 0(1) operations since popping from the end of a list in Python has

Overall, all of the operations of the MinStack class run in constant time, so we have 0(1) time complexity for push, pop, top, and getMin methods.

stk1 grows linearly with the number of elements pushed, so it has a space complexity of O(n).

getMin: Returning the last element of stk2, which is the current minimum, is also an O(1) operation.

top: This method simply returns the last element of stk1, which is an 0(1) operation.

Space Complexity: • The space complexity of the MinStack class is mainly due to the space used by the two stacks stk1 and stk2. Assuming n is the number of elements pushed into the stack:

stk2 also grows with each push operation. Even though it's used to keep the minimum value at each state, it does in pair

with the stk1, so its size also corresponds to the number of elements pushed, which makes it 0(n) as well. Therefore, the overall space complexity of the MinStack class is O(n) where n is the number of elements in the stack.