<https://leetcode.com/problems/min-stack/>

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:**

* -231 <= val <= 231 - 1
* Methods pop, top and getMin operations will always be called on **non-empty** stacks.
* At most 3 \* 104 calls will be made to push, pop, top, and getMin.

**Attempt 1: 2023-04-04**

**Solution 1: Two Stacks (30 min, not suggest because we would better not implement Stack with standard library stack)**

**Style 1: Elements stored on stack and minStack might be different**

class MinStack {

Stack<Integer> stack;

Stack<Integer> minStack;

public MinStack() {

stack = new Stack<Integer>();

minStack = new Stack<Integer>();

}

public void push(int val) {

stack.push(val);

if(minStack.isEmpty() || minStack.peek() >= val) {

minStack.push(val);

}

}

public void pop() {

// Don't use == to compare two Integer, the two -1024 pop

// out from stack and minStack are different objects

/\*\*

The test case used find out '==' is the issue

================================

Input

["MinStack","push","push","push","push","pop","getMin","pop","getMin","pop","getMin"]

[[],[512],[-1024],[-1024],[512],[],[],[],[],[],[]]

Output

[null,null,null,null,null,null,-1024,null,-1024,null,-1024]

Expected

[null,null,null,null,null,null,-1024,null,-1024,null,512]

For the correct 'equals' logic looks like below

1.Bottom -> Top after all pushes

stack: {512,-1024,-1024,512}

minStack: {512,-1024,-1024}

================================

2.pop

stack: {512,-1024,-1024}

minStack: {512,-1024,-1024}

================================

3.getMin

minStack: -1024

================================

4.pop

stack: {512,-1024}

minStack: {512,-1024}

================================

5.getMin

minStack: -1024

================================

6.pop

stack: {512}

minStack: {512}

================================

7.getMin

minStack: 512

\*/

//if(stack.peek() == minStack.peek()) {

if(stack.peek().equals(minStack.peek())) {

minStack.pop();

}

stack.pop();

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**Refer to**

<https://leetcode.com/problems/min-stack/solutions/49016/c-using-two-stacks-quite-short-and-easy-to-understand/>

**This style not push or pop current minimum value to minStack each time, it only push and pop minimum value when necessary, which means the elements stored on stack and minStack might be different.**

class MinStack {

private:

stack<int> s1;

stack<int> s2;

public:

void push(int x) {

s1.push(x);

if (s2.empty() || x <= getMin()) s2.push(x);

}

void pop() {

if (s1.top() == getMin()) s2.pop();

s1.pop();

}

int top() {

return s1.top();

}

int getMin() {

return s2.top();

}

};

**Style 2: Elements stored on stack and minStack exactly same**

class MinStack {

Stack<Integer> stack;

Stack<Integer> minStack;

public MinStack() {

stack = new Stack<Integer>();

minStack = new Stack<Integer>();

}

public void push(int val) {

stack.push(val);

if(minStack.isEmpty() || minStack.peek() >= val) {

minStack.push(val);

} else {

minStack.push(minStack.peek());

}

}

public void pop() {

stack.pop();

minStack.pop();

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**Refer to**

<https://www.jiuzhang.com/solutions/min-stack/>

class MinStack {

Stack<Integer> stack;

Stack<Integer> minStack;

/\*\* initialize your data structure here. \*/

public MinStack() {

stack = new Stack<Integer>();

minStack = new Stack<Integer>();

}

public void push(int x) {

// Style 1:

if(minStack.isEmpty() || minStack.peek() >= x) {

minStack.push(x);

} else {

minStack.push(minStack.peek());

}

// Style 2:

// Refer to

// http://www.jiuzhang.com/solutions/min-stack/

// if (minStack.isEmpty()) {

// minStack.push(number);

// } else {

// minStack.push(Math.min(number, minStack.peek()));

// }

stack.push(x);

}

public void pop() {

minStack.pop();

stack.pop();

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

/\*\*

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

\* MinStack obj = new MinStack();

\* obj.push(x);

\* obj.pop();

\* int param\_3 = obj.top();

\* int param\_4 = obj.getMin();

\*/

**Solution 2: Linked List (30 min, best answer so far, tricky on inserting new node prior than old one)**

class MinStack {

private class Node {

int val;

int min;

Node next;

public Node(int val, int min, Node next) {

this.val = val;

this.min = min;

this.next = next;

}

}

private Node head;

public MinStack() {

}

// newest node -> relative newer node -> ... -> oldest node

public void push(int val) {

if(head == null) {

head = new Node(val, val, null);

} else {

head = new Node(val, Math.min(val, head.min), head);

}

}

// Remove most recent inserted node

public void pop() {

head = head.next;

}

public int top() {

return head.val;

}

public int getMin() {

return head.min;

}

}

**Refer to**

<https://leetcode.com/problems/min-stack/solutions/49010/clean-6ms-java-solution>

This is what exactly the interviewer want, design a stack by yourself.

class MinStack {

private Node head;

public void push(int x) {

if (head == null)

head = new Node(x, x, null);

else

head = new Node(x, Math.min(x, head.min), head);

}

public void pop() {

head = head.next;

}

public int top() {

return head.val;

}

public int getMin() {

return head.min;

}

private class Node {

int val;

int min;

Node next;

private Node(int val, int min, Node next) {

this.val = val;

this.min = min;

this.next = next;

}

}

}

**Solution 3: One Stack and Pair (30 min)**

class MinStack {

Stack<int[]> minStack;

public MinStack() {

minStack = new Stack<int[]>();

}

public void push(int val) {

if(minStack.isEmpty()) {

minStack.push(new int[] {val, val});

} else {

int min = Math.min(val, minStack.peek()[1]);

minStack.push(new int[] {val, min});

}

}

public void pop() {

minStack.pop();

}

public int top() {

return minStack.peek()[0];

}

public int getMin() {

return minStack.peek()[1];

}

}

/\*\*

\* 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();

\*/

**Refer to**

<https://leetcode.com/problems/min-stack/solutions/1209254/c-simple-code-with-one-stack/>

I came up with this simple solution using just a single [stack.Here](http://stack.Here) I am using **Stack of Pair of Int**. The first value of the pair would store the element of the normal stack and the second value would store the minimum up to that point in the [stack.So](http://stack.So) even if the minimum element of the stack is removed from the top, we still have a backup of the next minimum element in the pair. So for every element pushed in the stack, it stores its corresponding minimum value.

For example, let's do a **Dry Run** of an example.

["MinStack","push","push","push","push","push","getMin","pop","pop","top","push","getMin"]

[[],[5],[-2],[3],[-10],[20],[],[],[],[],[30],[]]

* We push 5,-2,3,-10,20 in the stack.
* If the stack is empty we push {val,val} in the stack else we push {val,min([s.top](http://s.top)().second,val)} which is basically minimum upto that point.
* Hence {5,5},{-2,-2},{3,-2},{-10,-10},{20,-10} are pushed in the stack.
* To pop simply do stack.pop()
* To get the top return [stack.top](http://stack.top)().first;
* Now we pop 20 and -10 from the stack
* The elements in the stack would be {5,5},{-2,-2},{3,-2}
* On pushing 30 to the stack
* The elements in the stack would be {5,5},{-2,-2},{3,-2},{30,-2}.

The **Output** of the code would be:

[null,null,null,null,null,null,-10,null,null,3,null,-2]

All the operations are one liners expect the Push operation which is a 2 liner.

class MinStack {

public:

vector< pair<int,int> > s;

MinStack() { }

void push(int val) {

if(s.empty())

s.push\_back({val,val});

else

s.push\_back({val,min(s.back().second,val)});

}

void pop() { s.pop\_back(); }

int top() { return s.back().first; }

int getMin() { return s.back().second; }

};

The Time complexity of each operation is O(1)

The Space complexity is O(N)