Leet Code : 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.

Here are two different functional ways to implement a MinStack in Scala:

1. Using two stacks:

This implementation uses two stacks as:

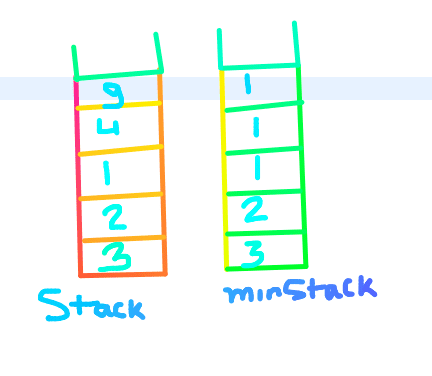
Stack 1: To store the actual elements.

Stack 2 : To store the minimum values encountered .

*Whenever a new element is pushed onto the stack, if it is smaller than or equal to the current minimum, it is also pushed onto the minStack.*

*When an element is popped from the stack, if it is the current minimum, it is also popped from the minStack.*

*The top of the minStack always stores the current minimum value.*



Using 2 stacks Implementation

class MinStack() {  
 private val stack = scala.collection.mutable.Stack[Int]()  
 private val minStack = scala.collection.mutable.Stack[Int]()  
  
 def push(x: Int): Unit = {  
 stack.push(x)  
 if (minStack.isEmpty || x <= minStack.top) minStack.push(x)  
 }  
  
 def pop(): Unit = {  
 if (stack.top == minStack.top) minStack.pop()  
 stack.pop()  
 }  
  
 def top(): Int = stack.top  
  
 def getMin(): Int = minStack.top  
}

2. Using a single stack with tuple

This implementation uses a single stack to store tuples of the form (element, minimum encountered).

*Whenever a new element is pushed onto the stack, its value and the current minimum are stored in a tuple and pushed onto the stack. The minimum value is updated whenever a new minimum is encountered.*

*When an element is popped from the stack, the corresponding tuple is also popped.*

*The top of the stack always stores the current element and the current minimum value.*

class MinStack() {  
 private val stack = scala.collection.mutable.Stack[(Int, Int)]()  
 private var min = Int.MaxValue  
  
 def push(x: Int): Unit = {  
 min = math.min(x, min)  
 stack.push((x, min))  
 }  
  
 def pop(): Unit = stack.pop()  
  
 def top(): Int = stack.top.\_1  
  
 def getMin(): Int = stack.top.\_2  
}

3. Using a single stack and a minimum variable

*We define a variable minEle that stores the current minimum element in the stack.*

*Now the interesting part is, how to handle the case when the minimum element is removed.?*

*We push “2x — minEle” into the stack instead of x so that the previous minimum element can be retrieved using the current minEle and its value stored in the stack.*

Follow the given steps to implement the stack operations:

Push(x): Insert x at the top of the stack

* If the stack is empty, insert x into the stack and make minEle equal to x.
* If the stack is not empty, compare x with minEle. Two cases arise:
* If x is greater than or equal to minEle, simply insert x.
* If x is less than minEle, insert (2\*x — minEle) into the stack and make minEle equal to x.  
  For example, let the previous minEle be 3. Now we want to insert 2. We update minEle as 2 and insert 2\*2–3 = 1 into the stack

Pop(): Removes an element from the top of the stack

* Remove the element from the top. Let the removed element be y. Two cases arise:
* If y is greater than or equal to minEle, the minimum element in the stack is still minEle.
* If y is less than minEle, the minimum element now becomes (2\*minEle — y), so update (minEle = 2\*minEle — y). This is where we retrieve the previous minimum from the current minimum and its value in the stack.  
  For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as 2\*2–1 = 3

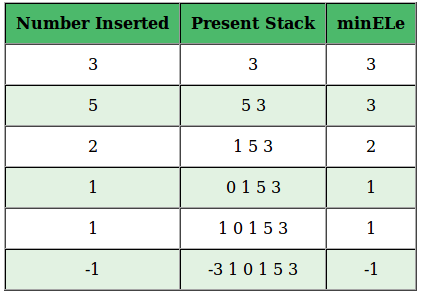
Important Points:

*Stack doesn’t hold the actual value of an element if it is minimum so far.*

*The actual minimum element is always stored in the minEle variable.*

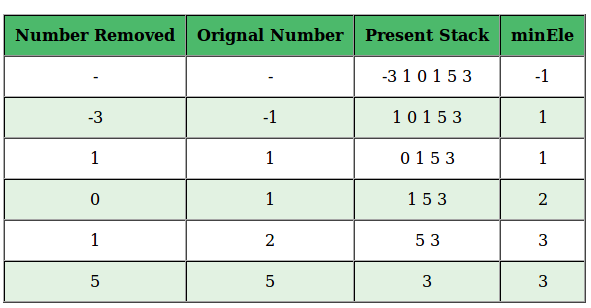
import java.util.Stack  
  
class MinStack {  
 private val s = new Stack[Node]  
  
 private class Node(val value: Int, val min: Int)  
  
 def push(x: Int): Unit = {  
 if (s.isEmpty()) {  
 s.push(new Node(x, x))  
 } else {  
 val min = Math.min(s.peek().min, x)  
 s.push(new Node(x, min))  
 }  
 }  
  
 def pop(): Int = {  
 s.pop().value  
 }  
  
 def top(): Int = {  
 s.peek().value  
 }  
  
 def getMin(): Int = {  
 s.peek().min  
 }  
}  
  
object MinStackDemo {  
 def main(args: Array[String]): Unit = {  
 val s = new MinStack()  
 // Function calls  
 s.push(-1)  
 s.push(10)  
 s.push(-4)  
 s.push(0)  
 println(s.getMin())  
 println(s.pop())  
 println(s.pop())  
 println(s.getMin())  
 }  
}

Push()



* *Number to be Inserted: 3, Stack is empty, so insert 3 into stack and minEle = 3.*
* *Number to be Inserted: 5, Stack is not empty, 5> minEle, insert 5 into stack and minEle = 3.*
* *Number to be Inserted: 2, Stack is not empty, 2< minEle, insert (2\*2–3 = 1) into stack and minEle = 2.*
* *Number to be Inserted: 1, Stack is not empty, 1< minEle, insert (2\*1–2 = 0) into stack and minEle = 1.*
* *Number to be Inserted: 1, Stack is not empty, 1 = minEle, insert 1 into stack and minEle = 1.*
* *Number to be Inserted: -1, Stack is not empty, -1 < minEle, insert (2\*-1–1 = -3) into stack and minEle = -1.*

Pop()



* *initially the minimum element minEle in the stack is -1.*
* *Number removed: -3, Since -3 is less than the minimum element the original number being removed is minEle which is -1, and the new minEle = 2\*-1 — (-3) = 1*
* *Number removed: 1, 1 == minEle, so number removed is 1 and minEle is still equal to 1.*
* *Number removed: 0, 0< minEle, original number is minEle which is 1 and new minEle = 2\*1–0 = 2.*
* *Number removed: 1, 1< minEle, original number is minEle which is 2 and new minEle = 2\*2–1 = 3.*
* *Number removed: 5, 5> minEle, original number is 5 and minEle is still 3*