Question Sheet

1. What is a situation where you had to solve a tough engineering problem?

What was the problem?

What did you do to solve it?

What was the outcome?

1. An enterprise company is interested in building a web service complete with front-end website. You are hired to do the job. What are the considerations and tradeoffs that you consider to do the job?

3.

# Coding

The following 2 questions are expected to be answered without the help of the web.

**Please submit original code.**

Also, for each coding question, please submit test cases to ensure your code is working correctly.

Consideration for coding sections:

1. Functionality – does it work?
2. Algorithmic complexity – does it work efficiently?
3. Testability – how easy is it to test?
4. Test harness – how well do you think through corner cases?
5. Readability – how clean is your code?

## Coding part 1 - 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(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.

**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.

**Implementation:**

**import** java.util.Stack;

**class** MinStack {

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

**public** MinStack() {

}

**public** **void** push(**int** value) {

**if** (minStack.isEmpty()) {

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

**return**;

}

**int** minValue = minStack.peek()[1];

minStack.push(**new** **int**[] { value, Math.*min*(value, minValue) });

}

**public** **void** pop() {

minStack.pop();

}

**public** **int** top() {

**return** minStack.peek()[0];

}

**public** **int** getMin() {

**return** minStack.peek()[1];

}

}

**Junit Test Cases:**

import static org.junit.Assert.\*;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import org.junit.Before;

import org.junit.Test;

public class MinStackTest {

String[] operation;

Integer[] data;

public static final String PUSH = "push";

public static final String POP = "pop";

public static final String GET\_MIN = "getMin";

public static final String TOP = "top";

@Before

public void setup() {

operation = new String[] { "MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin" };

data = new Integer[] { null, -2, 0, -3, null, null, null, null };

}

@Test

public void testMinStack\_returnsMinValue() {

List<Integer> result = new ArrayList<>();

MinStack minStack = new MinStack();

for (int i = 0; i < operation.length; i++) {

if (PUSH.equals(operation[i])) {

minStack.push(data[i]);

} else if (POP.equals(operation[i])) {

minStack.pop();

} else if (GET\_MIN.equals(operation[i])) {

result.add(minStack.getMin());

}

}

assertEquals(Arrays.asList(-3, -2), result);

}

@Test

public void testMinStack\_returnsTopValue() {

List<Integer> result = new ArrayList<>();

MinStack minStack = new MinStack();

for (int i = 0; i < operation.length; i++) {

if (PUSH.equals(operation[i])) {

minStack.push(data[i]);

} else if (POP.equals(operation[i])) {

minStack.pop();

} else if (TOP.equals(operation[i])) {

result.add(minStack.top());

}

}

assertEquals(Arrays.asList(0), result);

}

}

## Coding part 2 - Valid Parentheses

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

**Example 1:**

Input: s = "()"

Output: true

**Example 2:**

Input: s = "()[]{}"

Output: true

**Example 3:**

Input: s = "(]"

Output: false

**Example 4:**

Input: s = "([)]"

Output: false

**Example 5:**

Input: s = "{[]}"

Output: true

**Constraints:**

* 1 <= s.length <= 10^4
* s consists of parentheses only '()[]{}'.

**Implementation:**

**import** java.util.Stack;

**public** **class** ValidParanthesis {

**public** **boolean** validate(String input) {

Stack<Character> stack = **new** Stack<>();

**for** (**int** i = 0; i < input.length(); i++) {

**char** currentCharacter = input.charAt(i);

**if** (currentCharacter == '(' || currentCharacter == '[' || currentCharacter == '{') {

stack.push(currentCharacter);

**continue**;

}

**if** (stack.isEmpty())

**return** **false**;

**char** topCharacter = stack.pop();

**switch** (currentCharacter) {

**case** ')':

**if** (topCharacter != '(')

**return** **false**;

**break**;

**case** '}':

**if** (topCharacter != '{')

**return** **false**;

**break**;

**case** ']':

**if** (topCharacter != '[')

**return** **false**;

**break**;

}

}

**return** (stack.isEmpty());

}

}

**Junit Test Cases:**

**import** **static** org.junit.Assert.*assertFalse*;

**import** **static** org.junit.Assert.*assertTrue*;

**import** org.junit.Before;

**import** org.junit.Test;

**public** **class** ValidParanthesisTest {

ValidParanthesis solution;

@Before

**public** **void** setup() {

solution = **new** ValidParanthesis();

}

@Test

**public** **void** test\_validParanthesis\_case1() {

String input = "()";

*assertTrue*(solution.validate(input));

}

@Test

**public** **void** test\_validParanthesis\_case2() {

String input = "()[]{}";

*assertTrue*(solution.validate(input));

}

@Test

**public** **void** test\_validParanthesis\_case3() {

String input = "(]";

*assertFalse*(solution.validate(input));

}

@Test

**public** **void** test\_validParanthesis\_case4() {

String input = "([)]";

*assertFalse*(solution.validate(input));

}

@Test

**public** **void** test\_validParanthesis\_case5() {

String input = "{[]}";

*assertTrue*(solution.validate(input));

}

}