Question Sheet

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

What was the problem?

Ans:

Design an email notification system for delay in file reception and dealay in file report generation.

What did you do to solve it?

Ans:

1. Built a scheduler-based service that fetches config dynamically form mongo.

2. Depending on scheduler, check file reception and report generation from logs

3. Trigger delay on confirmation from logs.

What was the outcome?

Ans:

Efficient, dynamic system to trigger and monitor file reception and report generation

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 doing the job?

Ans:

1. Security
2. Performance and reliability
3. Deployment and run time
4. API Management

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.

## **Ans**:

**import** java.util.Stack;

**class** MinStack {

Stack<Integer> mainStack;

Stack<Integer> minStack;

**public** MinStack() {

mainStack = **new** Stack<>();

minStack = **new** Stack<>();

}

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

mainStack.add(x);

**if** (minStack.empty() || (!minStack.empty() && x <= minStack.peek())) {

minStack.add(x);

}

}

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

**if** (mainStack.empty()) {

**return**;

}

**int** x = mainStack.pop();

**if** (x == minStack.peek()) {

minStack.pop();

}

}

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

**return** mainStack.peek();

}

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

**if** (minStack.empty()) {

**return** -1;

}

**return** minStack.peek();

}

// Test code

**public** **static** **void** main(String[] args) {

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

Integer[] values = { **null**, -2, 0, -3, **null**, **null**, **null**, **null** };

MinStack minStack = **new** MinStack();

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

**if** ("MinStack".equals(input[i])) {

System.***out***.print("null ");

} **else** **if** ("push".equals(input[i])) {

minStack.push(values[i]);

System.***out***.print("null ");

} **else** **if** ("pop".equals(input[i])) {

minStack.pop();

System.***out***.print("null ");

} **else** **if** ("getMin".equals(input[i])) {

System.***out***.print(minStack.getMin() + " ");

} **else** **if** ("top".equals(input[i])) {

System.***out***.print(minStack.top() + " ");

}

}

}

}

**Junit Test class**

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

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** MinStackTest {

@Test

**public** **void** testMinValue() {

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

Integer[] values = { **null**, -2, 0, -3, **null**, **null**, **null**, **null** };

List<Integer> expected = Arrays.*asList*(-3, -2);

List<Integer> output = **new** ArrayList<>();

MinStack minStack = **new** MinStack();

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

**if** ("MinStack".equals(input[i])) {

} **else** **if** ("push".equals(input[i])) {

minStack.push(values[i]);

} **else** **if** ("pop".equals(input[i])) {

minStack.pop();

} **else** **if** ("getMin".equals(input[i])) {

output.add(minStack.getMin());

} **else** **if** ("top".equals(input[i])) {

}

}

*assertEquals*(expected, output);

}

@Test

**public** **void** testTop() {

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

Integer[] values = { **null**, -2, 0, -3, **null**, **null**, **null**, **null** };

List<Integer> expected = Arrays.*asList*(0);

List<Integer> output = **new** ArrayList<>();

MinStack minStack = **new** MinStack();

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

**if** ("MinStack".equals(input[i])) {

} **else** **if** ("push".equals(input[i])) {

minStack.push(values[i]);

} **else** **if** ("pop".equals(input[i])) {

minStack.pop();

} **else** **if** ("getMin".equals(input[i])) {

} **else** **if** ("top".equals(input[i])) {

output.add(minStack.top());

}

}

*assertEquals*(expected, output);

}

}

## 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 '()[]{}'.

Ans:

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Stack;

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

**private** **static** Map<Character, Character> *pairsMap*;

**static** {

*pairsMap* = **new** HashMap<Character, Character>();

*pairsMap*.put(')', '(');

*pairsMap*.put('}', '{');

*pairsMap*.put(']', '[');

}

**public** **static** **boolean** isValidParanthesis(String s) {

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

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

**char** c = s.charAt(i);

**if** (*pairsMap*.containsKey(c)) {

**char** top = stack.empty() ? '$' : stack.pop();

**if** (top != *pairsMap*.get(c)) {

**return** **false**;

}

} **else** {

stack.push(c);

}

}

**return** stack.isEmpty();

}

// Test code

**public** **static** **void** main(String[] args) {

String s = "()";

System.***out***.println(*isValidParanthesis*(s));

s = "()[]{}";

System.***out***.println(*isValidParanthesis*(s));

s = "(]";

System.***out***.println(*isValidParanthesis*(s));

s = "([)]";

System.***out***.println(*isValidParanthesis*(s));

s = "{[]}";

System.***out***.println(*isValidParanthesis*(s));

}

}

**Junit Test class :**

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

**import** org.junit.Test;

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

@Test

**public** **void** test\_paranthesis1() {

ValidParanthesis impl = **new** ValidParanthesis();

String s = "()";

*assertTrue*(impl.*isValidParanthesis*(s));

}

@Test

**public** **void** test\_paranthesis2() {

ValidParanthesis impl = **new** ValidParanthesis();

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

*assertTrue*(impl.*isValidParanthesis*(s));

}

@Test

**public** **void** test\_paranthesis3() {

ValidParanthesis impl = **new** ValidParanthesis();

String s = "(]";

*assertFalse*(impl.*isValidParanthesis*(s));

}

@Test

**public** **void** test\_paranthesis4() {

ValidParanthesis impl = **new** ValidParanthesis();

String s = "([)]";

*assertFalse*(impl.*isValidParanthesis*(s));

}

@Test

**public** **void** test\_paranthesis5() {

ValidParanthesis impl = **new** ValidParanthesis();

String s = "{[]}";

*assertTrue*(impl.*isValidParanthesis*(s));

}

}