Stack-Based Problem Solving Templates for LeetCode

1. Valid Parentheses

Problem:

Check if the input string of parentheses is valid. A string is valid if open brackets are closed by the same type of brackets and in the correct order.

Template:

java

Copy code

```
import java.util.Stack;


public class ValidParentheses {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(') {
                stack.push(')');
            } else if (c == '{') {
                stack.push('}');
            } else if (c == '[') {
                stack.push(']');
            } else if (stack.isEmpty() || stack.pop() != c) {
                return false;
            }
        }
        return stack.isEmpty();
    }
```

```
    public static void main(String[] args) {
        ValidParentheses vp = new ValidParentheses();
        System.out.println(vp.isValid("()[]{}")); // true
        System.out.println(vp.isValid("([)]"));   // false
        System.out.println(vp.isValid("{[]}"));   // true
    }
}
```

Explanation:

Use a stack to keep track of opening brackets.

Push the corresponding closing bracket to the stack when an opening bracket is encountered.

If a closing bracket is found, pop from the stack and check if it matches the expected closing bracket.

Ensure the stack is empty at the end for the string to be valid.

2. Daily Temperatures

Problem:

Given a list of daily temperatures, return a list where each element is the number of days you have to wait until a warmer temperature. If there is no future day, put 0.

Template:

java

Copy code

```
import java.util.Stack;

public class DailyTemperatures {
    public int[] dailyTemperatures(int[] temperatures) {
        int[] result = new int[temperatures.length];
        Stack<Integer> stack = new Stack<>();

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

```
            while (!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()]) {
                int index = stack.pop();
                result[index] = i - index;
            }
            stack.push(i);
        }

        return result;
    }

    public static void main(String[] args) {
        DailyTemperatures dt = new DailyTemperatures();
        int[] temperatures = {73, 74, 75, 71, 69, 72, 76, 73};
        int[] result = dt.dailyTemperatures(temperatures);
        for (int r : result) {
            System.out.print(r + " "); // 1 1 4 2 1 1 0 0
        }
    }
}
```

Explanation:

Use a stack to keep track of indices of temperatures.

For each temperature, compare it with the temperature at the index stored at the top of the stack.

If the current temperature is warmer, pop the stack and calculate the number of days waited.

Push the current index onto the stack.

3. Next Greater Element

Problem:

Find the next greater element for each element in the first array from the second array.

Template:

java

Copy code

```
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

public class NextGreaterElement {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Map<Integer, Integer> map = new HashMap<>();
        Stack<Integer> stack = new Stack<>();

        for (int num : nums2) {
            while (!stack.isEmpty() && stack.peek() < num) {
                map.put(stack.pop(), num);
            }
            stack.push(num);
        }

        int[] result = new int[nums1.length];
        for (int i = 0; i < nums1.length; i++) {
            result[i] = map.getOrDefault(nums1[i], -1);
        }

        return result;
    }

    public static void main(String[] args) {
        NextGreaterElement nge = new NextGreaterElement();
```

```
        int[] nums1 = {4, 1, 2};
        int[] nums2 = {1, 3, 4, 2};
        int[] result = nge.nextGreaterElement(nums1, nums2);
        for (int r : result) {
            System.out.print(r + " "); // -1 3 -1
        }
    }
}
```

Explanation:

Use a stack to keep track of elements in the second array.

Use a hashmap to store the next greater element for each number.

For each number in the second array, if it is greater than the number at the top of the stack, pop the stack and update the hashmap.

Finally, use the hashmap to build the result array for the first array.

4. Min Stack

Problem:

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Template:

java

Copy code

```
import java.util.Stack;

class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
```

```
    minStack = new Stack<>();
}

public void push(int x) {
    stack.push(x);
    if (minStack.isEmpty() || x <= minStack.peek()) {
        minStack.push(x);
    }
}

public void pop() {
    if (stack.peek().equals(minStack.peek())) {
        minStack.pop();
    }
    stack.pop();
}

public int top() {
    return stack.peek();
}

public int getMin() {
    return minStack.peek();
}

public static void main(String[] args) {
    MinStack minStack = new MinStack();
    minStack.push(-2);
    minStack.push(0);
```

```
        minStack.push(-3);
        System.out.println(minStack.getMin()); // -3
        minStack.pop();
        System.out.println(minStack.top());    // 0
        System.out.println(minStack.getMin()); // -2
    }
}
```

Explanation:

Use two stacks: one for the stack operations and another to keep track of the minimum values.

When pushing, push to both stacks if the new element is smaller or equal to the current minimum.

When popping, pop from both stacks if the top elements are equal.

The minimum element is always at the top of the min stack.

5. Trapping Rain Water

Problem:

Calculate how much water can be trapped after raining given the heights of bars.

Template:

java

Copy code

```
public class TrappingRainWater {
    public int trap(int[] height) {
        if (height.length == 0) return 0;

        Stack<Integer> stack = new Stack<>();
        int water = 0;

        for (int i = 0; i < height.length; i++) {
            while (!stack.isEmpty() && height[i] > height[stack.peek()]) {
```

```
            int top = stack.pop();
            if (stack.isEmpty()) break;

            int distance = i - stack.peek() - 1;
            int boundedHeight = Math.min(height[i], height[stack.peek()]) - height[top];
            water += distance * boundedHeight;
        }
        stack.push(i);
    }

    return water;
  }

  public static void main(String[] args) {
    TrappingRainWater trw = new TrappingRainWater();
    int[] height = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
    System.out.println(trw.trap(height)); // 6
  }
}
```

Explanation:

Use a stack to keep track of the indices of the bars.

For each bar, calculate the water trapped by using the current bar and the bars in the stack.

Pop from the stack when a bar is encountered that is taller than the bar at the index at the top of the stack.

Calculate the distance and bounded height to find the trapped water.

6. Evaluate Reverse Polish Notation

Problem:

Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN).

Template:

java

Copy code

```
import java.util.Stack;

public class EvaluateRPN {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();

        for (String token : tokens) {
            if (token.equals("+")) {
                stack.push(stack.pop() + stack.pop());
            } else if (token.equals("-")) {
                int b = stack.pop();
                int a = stack.pop();
                stack.push(a - b);
            } else if (token.equals("*")) {
                stack.push(stack.pop() * stack.pop());
            } else if (token.equals("/")) {
                int b = stack.pop();
                int a = stack.pop();
                stack.push(a / b);
            } else {
                stack.push(Integer.parseInt(token));
            }
        }

        return stack.pop();
    }
```

```
    public static void main(String[] args) {
        EvaluateRPN rpn = new EvaluateRPN();
        String[] tokens = {"2", "1", "+", "3", "*"};
        System.out.println(rpn.evalRPN(tokens)); // 9
    }
}
```

Explanation:

Use a stack to store numbers.

When encountering an operator, pop the required number of operands from the stack, apply the operation, and push the result back to the stack.

Continue until all tokens are processed.

The final result is the remaining value in the stack.

7. Basic Calculator

Problem:

Implement a basic calculator to evaluate a simple expression string.

Template:

java

Copy code

```
import java.util.Stack;

public class BasicCalculator {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int num = 0;
        int result = 0;
        int sign = 1;
```

```
    for (char c : s.toCharArray()) {
        if (Character.isDigit(c)) {
            num = num * 10 + (c - '0');
        } else if (c == '+') {
            result += sign * num;
            num = 0;
            sign = 1;
        } else if (c == '-') {
            result += sign * num;
            num = 0;
            sign = -1;
        } else if (c == '(') {
            stack.push(result);
            stack.push(sign);
            result = 0;
            sign = 1;
        } else if (c == ')') {
            result += sign * num;
            result *= stack.pop();
            result += stack.pop();
            num = 0;
        }
    }

    return result + (sign * num);
}

public static void main(String[] args) {
    BasicCalculator calculator = new BasicCalculator();
```

```
        System.out.println(calculator.calculate("1 + 1")); // 2
        System.out.println(calculator.calculate(" 2-1 + 2 ")); // 3
        System.out.println(calculator.calculate("(1+(4+5+2)-3)+(6+8)")); // 23
    }
}
```

Explanation:

Use a stack to handle parentheses and sign changes.

Iterate through the string, updating the result based on current number and sign.

Push the result and sign onto the stack when encountering an opening parenthesis.

Pop from the stack and update the result when encountering a closing parenthesis.

Add the last processed number to the result.

8. Largest Rectangle in Histogram

Problem:

Find the area of the largest rectangle in a histogram given its bar heights.

Template:

java

Copy code

```
import java.util.Stack;

public class LargestRectangleHistogram {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        int index = 0;

        while (index < heights.length) {
            if (stack.isEmpty() || heights[index] >= heights[stack.peek()]) {
```

```
            stack.push(index++);
        } else {
            int height = heights[stack.pop()];
            int width = stack.isEmpty() ? index : index - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
        }
    }

    while (!stack.isEmpty()) {
        int height = heights[stack.pop()];
        int width = stack.isEmpty() ? index : index - stack.peek() - 1;
        maxArea = Math.max(maxArea, height * width);
    }

    return maxArea;
  }

  public static void main(String[] args) {
    LargestRectangleHistogram lra = new LargestRectangleHistogram();
    int[] heights = {2, 1, 5, 6, 2, 3};
    System.out.println(lra.largestRectangleArea(heights)); // 10
  }
}
```

Explanation:

Use a stack to keep track of the indices of the bars.

For each bar, calculate the maximum area by considering it as the smallest bar.

Calculate the width based on the indices in the stack.

Update the maximum area as needed.

9. Simplify Path

Problem:

Given an absolute path for a file (Unix-style), simplify it.

Template:

java

Copy code

```
import java.util.Stack;

public class SimplifyPath {
    public String simplifyPath(String path) {
        Stack<String> stack = new Stack<>();
        String[] components = path.split("/");

        for (String component : components) {
            if (component.equals(".") || component.isEmpty()) {
                continue;
            } else if (component.equals("..")) {
                if (!stack.isEmpty()) {
                    stack.pop();
                }
            } else {
                stack.push(component);
            }
        }

        StringBuilder result = new StringBuilder();
        for (String dir : stack) {
```

```
            result.append("/").append(dir);
        }

        return result.length() > 0 ? result.toString() : "/";
    }

    public static void main(String[] args) {
        SimplifyPath sp = new SimplifyPath();
        System.out.println(sp.simplifyPath("/home/")); // "/home"
        System.out.println(sp.simplifyPath("/../")); // "/"
        System.out.println(sp.simplifyPath("/home//foo/")); // "/home/foo"
    }
}
```

Explanation:

Use a stack to store the valid path components.

Split the path by "/" and iterate through each component.

Ignore "." and empty components, pop the stack for "..", and push valid directory names.

Build the simplified path from the stack.

Feel free to add any additional notes or modify the explanations to suit your needs before converting this document to a PDF.

Template for Implement Queue using Stacks

```
import java.util.Stack;

class MyQueue {
    private Stack<Integer> inputStack;
    private Stack<Integer> outputStack;
```

```
/** Initialize your data structure here. */
public MyQueue() {
    inputStack = new Stack<>();
    outputStack = new Stack<>();
}

/** Push element x to the back of queue. */
public void push(int x) {
    inputStack.push(x);
}

/** Removes the element from the front of queue and returns that element. */
public int pop() {
    if (outputStack.isEmpty()) {
        while (!inputStack.isEmpty()) {
            outputStack.push(inputStack.pop());
        }
    }
    return outputStack.pop();
}

/** Get the front element. */
public int peek() {
    if (outputStack.isEmpty()) {
        while (!inputStack.isEmpty()) {
            outputStack.push(inputStack.pop());
        }
    }
    return outputStack.peek();
```

```
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return inputStack.isEmpty() && outputStack.isEmpty();
    }

    public static void main(String[] args) {
        MyQueue queue = new MyQueue();
        queue.push(1);
        queue.push(2);
        System.out.println(queue.peek());  // 1
        System.out.println(queue.pop());   // 1
        System.out.println(queue.empty()); // false
    }
}
```

Explanation:

Implements a queue using two stacks (inputStack and outputStack).

push(x) - Push elements onto inputStack.

pop() - If outputStack is empty, transfer all elements from inputStack to outputStack. Then pop from outputStack.

peek() - If outputStack is empty, transfer all elements from inputStack to outputStack. Then peek from outputStack.

empty() - Check if both stacks are empty to determine if the queue is empty.

Template for Simplifying Decisions Based on Current Element

java

Copy code

```
import java.util.Stack;
```

```
public class SimplifyDecisions {
    public int simplifyDecisions(int[] nums) {
        Stack<Integer> stack = new Stack<>();
        int sum = 0;

        for (int num : nums) {
            while (!stack.isEmpty() && stack.peek() < num) {
                sum += stack.pop();
            }
            stack.push(num);
        }

        while (!stack.isEmpty()) {
            sum += stack.pop();
        }

        return sum;
    }

    public static void main(String[] args) {
        SimplifyDecisions sd = new SimplifyDecisions();
        int[] nums = {3, 1, 5, 4, 2};
        System.out.println(sd.simplifyDecisions(nums)); // 20
    }
}
```

Explanation:

Iterates through an array (nums) and makes decisions based on the current element compared to the stack's top element.

Adjusts the stack by pushing or popping elements to simplify the total sum based on current conditions.