☑ Easy Stack & Queue Problems

1. **Parenthesis Checker**

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

2. **Reverse a String using Stack**

```
public String reverse(String s) {
    Stack<Character> stack = new Stack<>();
    for (char c : s.toCharArray()) stack.push(c);
    StringBuilder sb = new StringBuilder();
    while (!stack.isEmpty()) sb.append(stack.pop());
    return sb.toString();
}
```

3. **Two Stacks in an Array**

```
class TwoStacks {
    int[] arr;
    int top1, top2;
    public TwoStacks(int n) {
       ;
        top2 = n;
    }
    void push1(int x) {
        if (top1 + 1 < top2) arr[++top1] = x;
    }
    void push2(int x) {
        if (top1 + 1 < top2) arr[--top2] = x;
    }
    int pop1() {
        return top1 >= 0 ? arr[top1--] : -1;
    }
    int pop2() {
        return top2 < arr.length ? arr[top2++] : -1;
    }
}
```

☑ Medium Stack & Queue Problems

4. **Queue Using Stacks**

```
class MyQueue {
    Stack<Integer> in = new Stack<>();
    Stack<Integer> out = new Stack<>();

    public void push(int x) {
        in.push(x);
    }

    public int pop() {
        peek();
        return out.pop();
    }

```

```
    public int peek() {
        if (out.isEmpty())
            while (!in.isEmpty()) out.push(in.pop());
        return out.peek();
    }

    public boolean empty() {
        return in.isEmpty() && out.isEmpty();
    }
}


#### 5. **Stack Using Queues**

class MyStack {
    Queue<Integer> q = new LinkedList<>();

    public void push(int x) {
        q.add(x);
        for (int i = 0; i < q.size() - 1; i++)
            q.add(q.remove());
    }

    public int pop() {
        return q.remove();
    }

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

    public boolean empty() {
        return q.isEmpty();
    }
}


#### 6. **Next Greater Element**

public int[] nextGreater(int[] nums) {
    Stack<Integer> stack = new Stack<>();
    int[] res = new int[nums.length];
    for (int i = nums.length - 1; i >= 0; i--) {
        (!stack.isEmpty() && stack.peek() <= nums[i]) stack.pop();
        res[i] = stack.isEmpty() ? -1 : stack.peek();
        stack.push(nums[i]);
    }
    return res;
}


### ☑ Hard Stack Problems

#### 7. **Stack with getMin() in O(1)**

class MinStack {
    Stack<Integer> stack = new Stack<>();
    Stack<Integer> 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.pop().equals(minStack.peek())) minStack.pop();
    }

    public int top() {
```

```
        return stack.peek();
    }

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


#### 8. **Largest Rectangular Area in Histogram**

public int largestRectangleArea(int[] heights) {
    Stack<Integer> stack = new Stack<>();
    int maxArea = 0, i = 0;
    while (i <= heights.length) {
        int h = (i == heights.length) ? 0 : heights[i];
 && h < heights[stack.peek()]) {
            int height = heights[stack.pop()];
            int width = stack.isEmpty() ? i : i - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
        }
        stack.push(i++);
    }
    return maxArea;
}


#### 9. **Evaluate RPN Expressions**

public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();
    for (String token : tokens) {
        if ("+-*/".contains(token)) {
            int b = stack.pop(), a = stack.pop();
            switch (token) {
                case "+": stack.push(a + b); break;
                case "-": stack.push(a - b); break;
                case "*": stack.push(a * b); break;
                case "/": stack.push(a / b); break;
            }
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();
}


#### 10. **Normalize Path Names**

public String simplifyPath(String path) {
    part : path.split("/")) {
        if (part.equals("..")) {
            if (!stack.isEmpty()) stack.pop();
        } else if (!part.isEmpty() && !part.equals(".")) {
            stack.push(part);
        }
    }
    return "/" + String.join("/", stack);
}


#### 11. **Compute Buildings with a Sunset View**

public List<Integer> sunsetView(int[] heights) {
    Stack<Integer> stack = new Stack<>();
    for (int i = heights.length - 1; i >= 0; i--) {
        if (stack.isEmpty() || heights[i] > heights[stack.peek()])
            stack.push(i);
```

```
    }
    Collections.reverse(stack);
    return stack;
}


#### 12. **Compute Binary Tree Nodes in Order of Increasing Depth**

public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        result.add(level);
    }
    return result;
}


#### 13. **Implement a Circular Queue**

class MyCircularQueue {
    int[] data;
    int front, rear, size;

    public MyCircularQueue(int k) {
        data = new int[k];
        0; rear = -1; size = 0;
    }

    public boolean enQueue(int value) {
        if (isFull()) return false;
        rear = (rear + 1) % data.length;
        data[rear] = value;
        size++;
        return true;
    }

    public boolean deQueue() {
        if (isEmpty()) return false;
        front = (front + 1) % data.length;
        size--;
        return true;
    }

    public int Front() {
        return isEmpty() ? -1 : data[front];
       public int Rear() {
        return isEmpty() ? -1 : data[rear];
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == data.length;
    }
}


#### 14. **Implement a Queue with Max API**

class MaxQueue {
    Queue<Integer> queue = new LinkedList<>();
```

```
    Deque<Integer> maxDeque = new LinkedList<>();

    public void enqueue(int x) {
        queue.offer(x);
        while (!maxDeque.isEmpty() && maxDeque.getLast() < x)
            maxDeque.removeLast();
        maxDeque.addLast(x);
    }

    public int dequeue() {
        int val = queue.poll();
        if (val == maxDeque.peek()) maxDeque.poll();
        return val;
    }

    public int getMax() {
        return maxDeque.peek();
    }
}
```

☑ Advanced Stack Problems

15. **The Celebrity Problem**

```
public int findCelebrity(int[][] knows, int n) {
    int celeb = 0;
    for (int i = 1; i < n; i++)
        if (knows[celeb][i] == 1) celeb = i;

    for (int i = 0; i < n; i++) {
        if (i != celeb && (knows[celeb][i] == 1 || knows[i][celeb] == 0))
            return -1;
    }
    return celeb;
}
```

16. **Valid Stack Permutation**

```
public boolean isValidPermutation(int[] input, int[] output) {
    Stack<Integer> stack = new Stack<>();
    int j = 0;
    for (int val : input) {
        stack.push(val);
        while (!stack.isEmpty() && stack.peek() == output[j]) {
            stack.pop();
            j++;
        }
    }
    return stack.isEmpty();
}
```

17. **Stack with getRandom() in O(1)**

```
class RandomStack {
    List<Integer> list = new ArrayList<>();
    Map<Integer, Integer> map = new HashMap<>();

    public void push(int x) {
        map.put(x, list.size());
        list.add(x);
    }

    public void pop() {
        int last = list.get(list.size() - 1);
        map.remove(last);
        list.remove(list.size() - 1);
    }
```

```
    public int getRandom() {
        return list.get(new Random().nextInt(list.size()));
    }
}


#### 18. **k Stacks in a Single Array**

class KStacks {
    int[] arr, top, next;
    int free;

    public KStacks(int k, int n) {
        arr = new int[n];
        top = new int[k];
        next = new int[n];
        Arrays.fill(top, -1);
        for (int i = 0; i < n - 1; i++) next[i] = i + 1;
        next[n - 1] = -1;
        free = 0;
    }

    public void push(int x, int sn) {
        int i = free;
        free = next[i];
        next[i] = top[sn];
        top[sn] = i;
        arr[i] = x;
    }

    public int pop(int sn) {
        int i = top[sn];
        top[sn] = next[i];
        next[i] = free;
        free = i;
        return arr[i];
    }
}


#### 19. **Largest Rectangular Area in Histogram**
*(Already provided earlier)*

---

#### 20. **Clone a Stack Without Extra Space**

public Stack<Integer> cloneStack(Stack<Integer> original) {
    reverse(original);
    Stack<Integer> clone = new Stack<>();
    reverse(original);
    for (int val : original) clone.push(val);
    return clone;
}

private void reverse(Stack<Integer> stack) {
    if (stack.isEmpty()) return;
    int temp = stack.pop();
    reverse(stack);
    insertAtBottom(stack, temp);
}

private void insertAtBottom(Stack<Integer> stack, int val) {
    if (stack.isEmpty()) {
        stack.push(val);
        return;
    }
    int temp = stack.pop();
```

```
    insertAtBottom(stack, val);
    stack.push(temp);
}


#### 21. **Custom Browser History**

class BrowserHistory {
    Stack<String> back = new Stack<>();
    Stack<String> forward = new Stack<>();
    String current;

    public BrowserHistory(String homepage) {
        current = homepage;
    }

    public void visit(String url) {
        back.push(current);
        current = url;
        forward.clear();
    }

    public String back() {
        if (!back.isEmpty()) {
            forward.push(current);
            current = back.pop();
        }
        return current;
    }

    public String forward() {
        if (!forward.isEmpty()) {
            back.push(current);
            current = forward.pop();
        }
        return current;
    }
}


#### 22. **Maximum Rectangle with All 1s**

public int maximalRectangle(char[][] matrix) {
    if (matrix.length == 0) return 0;
    int[] heights = new int[matrix[0].length];
    int maxArea = 0;
    for (char[] row : matrix) {
        for (int i = 0; i < row.length; i++)
            heights[i] = row[i] == '1' ? heights.max(maxArea, largestRectangleArea(
            heights));
    }
    return maxArea;
}


#### 23. **Sort a Stack Using Recursion**

public void sortStack(Stack<Integer> stack) {
    if (!stack.isEmpty()) {
        int temp = stack.pop();
        sortStack(stack);
        insertSorted(stack, temp);
    }
}

private void insertSorted(Stack<Integer> stack, int val) {
    if (stack.isEmpty() || val > stack.peek()) {
        stack.push(val);
        return;
```

```
    }
    int temp = stack.pop();
    insertSorted(stack, val);
    stack.push(temp);
}


#### 24. **Stack with findMiddle() and deleteMiddle()**

class DLLNode {
    int val;
    DLLNode prev, next;
    DLLNode(int val) { this.val = val; }
}

class StackWithMiddle {
    DLLNode head, mid;
    int count = 0;

    public void push(int val) {
        DLLNode node = new DLLNode(val);
        node.next = head;
        if (head != null) head.prev = node;
        head = node;
        count++;
        if (count == 1) mid = node;
        else if (count % 2 == 0) mid = mid.prev;
    }

    public int pop() {
        if (head == null) return -1;
        int val = head.val;
        head = head.next;
        if (head != null) head.prev = null;
        count--;
        if (count % 2 == 1) mid = mid.next;
        return val;
    }

    public int findMiddle() {
        return mid != null ? mid.val : -1;
    }

    public void deleteMiddle() {
        if (mid == null) return;
        if (mid.prev != null) mid.prev.next = mid.next;
        if (mid.next != null) mid.next.prev = mid.prev;
        if (count % 2 == 0) mid = mid.next;
        else mid = mid.prev;
        count--;
    }
}

```