

```

1  ### ☑ Easy Stack & Queue Problems
2
3  ##### 1. **Parenthesis Checker**
4
5  public boolean isValid(String s) {
6      Stack<Character> stack = new Stack<>();
7      for (char c : s.toCharArray()) {
8          if (c == '(' || c == '{' || c == '[') stack.push(c);
9          else if (stack.isEmpty() || 
10                  (c == ')' && stack.pop() != '(') ||
11                  (c == '}' && stack.pop() != '{') ||
12                  (c == ']' && stack.pop() != '[')) return false;
13     }
14     Empty();
15 }
16
17
18 ##### 2. **Reverse a String using Stack**
19
20 public String reverse(String s) {
21     Stack<Character> stack = new Stack<>();
22     for (char c : s.toCharArray()) stack.push(c);
23     StringBuilder sb = new StringBuilder();
24     while (!stack.isEmpty()) sb.append(stack.pop());
25     return sb.toString();
26 }
27
28
29 ##### 3. **Two Stacks in an Array**
30
31 class TwoStacks {
32     int[] arr;
33     int top1, top2;
34     public TwoStacks(int n) {
35         ;
36         top2 = n;
37     }
38     void push1(int x) {
39         if (top1 + 1 < top2) arr[++top1] = x;
40     }
41     void push2(int x) {
42         if (top1 + 1 < top2) arr[--top2] = x;
43     }
44     int pop1() {
45         return top1 >= 0 ? arr[top1--] : -1;
46     }
47     int pop2() {
48         return top2 < arr.length ? arr[top2++] : -1;
49     }
50 }
51
52
53 ##### ☑ Medium Stack & Queue Problems
54
55 ##### 4. **Queue Using Stacks**
56
57 class MyQueue {
58     Stack<Integer> in = new Stack<>();
59     Stack<Integer> out = new Stack<>();
60
61     public void push(int x) {
62         in.push(x);
63     }
64
65     public int pop() {
66         peek();
67         return out.pop();
68     }
69 }
```

```

70     public int peek() {
71         if (out.isEmpty())
72             while (!in.isEmpty()) out.push(in.pop());
73         return out.peek();
74     }
75
76     public boolean empty() {
77         return in.isEmpty() && out.isEmpty();
78     }
79 }
80
81
82 ##### 5. **Stack Using Queues**
83
84 class MyStack {
85     Queue<Integer> q = new LinkedList<>();
86
87     public void push(int x) {
88         q.add(x);
89         for (int i = 0; i < q.size() - 1; i++)
90             q.add(q.remove());
91     }
92
93     public int pop() {
94         return q.remove();
95     }
96
97     public int top() {
98         return q.peek();
99     }
100
101    public boolean empty() {
102        return q.isEmpty();
103    }
104 }
105
106
107 ##### 6. **Next Greater Element**
108
109 public int[] nextGreater(int[] nums) {
110     Stack<Integer> stack = new Stack<>();
111     int[] res = new int[nums.length];
112     for (int i = nums.length - 1; i >= 0; i--) {
113         (!stack.isEmpty() && stack.peek() <= nums[i]) stack.pop();
114         res[i] = stack.isEmpty() ? -1 : stack.peek();
115         stack.push(nums[i]);
116     }
117     return res;
118 }
119
120
121 ##### Hard Stack Problems
122
123 ##### 7. **Stack with getMin() in O(1)**
124
125 class MinStack {
126     Stack<Integer> stack = new Stack<>();
127     Stack<Integer> minStack = new Stack<>();
128
129     public void push(int x) {
130         stack.push(x);
131         if (minStack.isEmpty() || x <= minStack.peek()) minStack.push(x);
132     }
133
134     public void pop() {
135         if (stack.pop().equals(minStack.peek())) minStack.pop();
136     }
137
138     public int top() {

```

```

139         return stack.peek();
140     }
141
142     public int getMin() {
143         return minStack.peek();
144     }
145 }
146
147
148 #### 8. **Largest Rectangular Area in Histogram**
149
150 public int largestRectangleArea(int[] heights) {
151     Stack<Integer> stack = new Stack<>();
152     int maxArea = 0, i = 0;
153     while (i <= heights.length) {
154         int h = (i == heights.length) ? 0 : heights[i];
155         && h < heights[stack.peek()] {
156             int height = heights[stack.pop()];
157             int width = stack.isEmpty() ? i : i - stack.peek() - 1;
158             maxArea = Math.max(maxArea, height * width);
159         }
160         stack.push(i++);
161     }
162     return maxArea;
163 }
164
165
166 #### 9. **Evaluate RPN Expressions**
167
168 public int evalRPN(String[] tokens) {
169     Stack<Integer> stack = new Stack<>();
170     for (String token : tokens) {
171         if ("+-*/".contains(token)) {
172             int b = stack.pop(), a = stack.pop();
173             switch (token) {
174                 case "+": stack.push(a + b); break;
175                 case "-": stack.push(a - b); break;
176                 case "*": stack.push(a * b); break;
177                 case "/": stack.push(a / b); break;
178             }
179         } else {
180             stack.push(Integer.parseInt(token));
181         }
182     }
183     return stack.pop();
184 }
185
186
187 #### 10. **Normalize Path Names**
188
189 public String simplifyPath(String path) {
190     part : path.split("/") {
191         if (part.equals(".")) {
192             if (!stack.isEmpty()) stack.pop();
193         } else if (!part.isEmpty() && !part.equals(".")) {
194             stack.push(part);
195         }
196     }
197     return "/" + String.join("/", stack);
198 }
199
200
201 #### 11. **Compute Buildings with a Sunset View**
202
203 public List<Integer> sunsetView(int[] heights) {
204     Stack<Integer> stack = new Stack<>();
205     for (int i = heights.length - 1; i >= 0; i--) {
206         if (stack.isEmpty() || heights[i] > heights[stack.peek()])
207             stack.push(i);

```

```

208     }
209     Collections.reverse(stack);
210     return stack;
211 }
212
213
214 ##### 12. **Compute Binary Tree Nodes in Order of Increasing Depth**
215
216 public List<List<Integer>> levelOrder(TreeNode root) {
217     List<List<Integer>> result = new ArrayList<>();
218     if<Integer> level = new ArrayList<>();
219     for (int i = 0; i < size; i++) {
220         TreeNode node = queue.poll();
221         level.add(node.val);
222         if (node.left != null) queue.offer(node.left);
223         if (node.right != null) queue.offer(node.right);
224     }
225     result.add(level);
226 }
227 return result;
228 }
229
230
231 ##### 13. **Implement a Circular Queue**
232
233 class MyCircularQueue {
234     int[] data;
235     int front, rear, size;
236
237     public MyCircularQueue(int k) {
238         data = new int[k];
239         0; rear = -1; size = 0;
240     }
241
242     public boolean enqueue(int value) {
243         if (isFull()) return false;
244         rear = (rear + 1) % data.length;
245         data[rear] = value;
246         size++;
247         return true;
248     }
249
250     public boolean dequeue() {
251         if (isEmpty()) return false;
252         front = (front + 1) % data.length;
253         size--;
254         return true;
255     }
256
257     public int Front() {
258         return isEmpty() ? -1 : data[front];
259     }
260     public int Rear() {
261         return isEmpty() ? -1 : data[rear];
262     }
263
264     public boolean isEmpty() {
265         return size == 0;
266     }
267
268     public boolean isFull() {
269         return size == data.length;
270     }
271
272
273 ##### 14. **Implement a Queue with Max API**
274
275 class MaxQueue {
276     Queue<Integer> queue = new LinkedList<>();

```

```

277     Deque<Integer> maxDeque = new LinkedList<>();
278
279     public void enqueue(int x) {
280         queue.offer(x);
281         while (!maxDeque.isEmpty() && maxDeque.getLast() < x)
282             maxDeque.removeLast();
283         maxDeque.addLast(x);
284     }
285
286     public int dequeue() {
287         int val = queue.poll();
288         if (val == maxDeque.peek()) maxDeque.poll();
289         return val;
290     }
291
292     public int getMax() {
293         return maxDeque.peek();
294     }
295 }
296
297 ### ☑ Advanced Stack Problems
298
299 ##### 15. **The Celebrity Problem**
300
301 public int findCelebrity(int[][] knows, int n) {
302     int celeb = 0;
303     for (int i = 1; i < n; i++)
304         if (knows[celeb][i] == 1) celeb = i;
305
306     for (int i = 0; i < n; i++) {
307         if (i != celeb && (knows[celeb][i] == 1 || knows[i][celeb] == 0))
308             return -1;
309     }
310     return celeb;
311 }
312
313
314 ##### 16. **Valid Stack Permutation**
315
316 public boolean isValidPermutation(int[] input, int[] output) {
317     Stack<Integer> stack = new Stack<>();
318     int j = 0;
319     for (int val : input) {
320         stack.push(val);
321         while (!stack.isEmpty() && stack.peek() == output[j]) {
322             stack.pop();
323             j++;
324         }
325     }
326     return stack.isEmpty();
327 }
328
329
330 ##### 17. **Stack with getRandom() in O(1)**
331
332 class RandomStack {
333     List<Integer> list = new ArrayList<>();
334     Map<Integer, Integer> map = new HashMap<>();
335
336     public void push(int x) {
337         map.put(x, list.size());
338         list.add(x);
339     }
340
341     public void pop() {
342         int last = list.get(list.size() - 1);
343         map.remove(last);
344         list.remove(list.size() - 1);
345     }

```

```

346
347     public int getRandom() {
348         return list.get(new Random().nextInt(list.size()));
349     }
350 }
351
352
353 ##### 18. **k Stacks in a Single Array**
354
355 class KStacks {
356     int[] arr, top, next;
357     int free;
358
359     public KStacks(int k, int n) {
360         arr = new int[n];
361         top = new int[k];
362         next = new int[n];
363         Arrays.fill(top, -1);
364         for (int i = 0; i < n - 1; i++) next[i] = i + 1;
365         next[n - 1] = -1;
366         free = 0;
367     }
368
369     public void push(int x, int sn) {
370         int i = free;
371         free = next[i];
372         next[i] = top[sn];
373         top[sn] = i;
374         arr[i] = x;
375     }
376
377     public int pop(int sn) {
378         int i = top[sn];
379         top[sn] = next[i];
380         next[i] = free;
381         free = i;
382         return arr[i];
383     }
384 }
385
386
387 ##### 19. **Largest Rectangular Area in Histogram**
388 *(Already provided earlier)*
389
390 ---
391
392 ##### 20. **Clone a Stack Without Extra Space**
393
394 public Stack<Integer> cloneStack(Stack<Integer> original) {
395     reverse(original);
396     Stack<Integer> clone = new Stack<>();
397     reverse(original);
398     for (int val : original) clone.push(val);
399     return clone;
400 }
401
402 private void reverse(Stack<Integer> stack) {
403     if (stack.isEmpty()) return;
404     int temp = stack.pop();
405     reverse(stack);
406     insertAtBottom(stack, temp);
407 }
408
409 private void insertAtBottom(Stack<Integer> stack, int val) {
410     if (stack.isEmpty()) {
411         stack.push(val);
412         return;
413     }
414     int temp = stack.pop();

```

```

415     insertAtBottom(stack, val);
416     stack.push(temp);
417 }
418
419
420 ##### 21. **Custom Browser History**
421
422 class BrowserHistory {
423     Stack<String> back = new Stack<>();
424     Stack<String> forward = new Stack<>();
425     String current;
426
427     public BrowserHistory(String homepage) {
428         current = homepage;
429     }
430
431     public void visit(String url) {
432         back.push(current);
433         current = url;
434         forward.clear();
435     }
436
437     public String back() {
438         if (!back.isEmpty()) {
439             forward.push(current);
440             current = back.pop();
441         }
442         return current;
443     }
444
445     public String forward() {
446         if (!forward.isEmpty()) {
447             back.push(current);
448             current = forward.pop();
449         }
450         return current;
451     }
452 }
453
454
455 ##### 22. **Maximum Rectangle with All 1s**
456
457 public int maximalRectangle(char[][] matrix) {
458     if (matrix.length == 0) return 0;
459     int[] heights = new int[matrix[0].length];
460     int maxArea = 0;
461     for (char[] row : matrix) {
462         for (int i = 0; i < row.length; i++)
463             heights[i] = row[i] == '1' ? heights.max(maxArea, largestRectangleArea(
464                 heights));
465     }
466     return maxArea;
467 }
468
469
470 ##### 23. **Sort a Stack Using Recursion**
471
472 public void sortStack(Stack<Integer> stack) {
473     if (!stack.isEmpty()) {
474         int temp = stack.pop();
475         sortStack(stack);
476         insertSorted(stack, temp);
477     }
478 }
479
480 private void insertSorted(Stack<Integer> stack, int val) {
481     if (stack.isEmpty() || val > stack.peek()) {
482         stack.push(val);
483         return;
484     }

```

```

483     }
484     int temp = stack.pop();
485     insertSorted(stack, val);
486     stack.push(temp);
487 }
488
489
490 ##### 24. **Stack with findMiddle() and deleteMiddle()**
491
492 class DLLNode {
493     int val;
494     DLLNode prev, next;
495     DLLNode(int val) { this.val = val; }
496 }
497
498 class StackWithMiddle {
499     DLLNode head, mid;
500     int count = 0;
501
502     public void push(int val) {
503         DLLNode node = new DLLNode(val);
504         node.next = head;
505         if (head != null) head.prev = node;
506         head = node;
507         count++;
508         if (count == 1) mid = node;
509         else if (count % 2 == 0) mid = mid.prev;
510     }
511
512     public int pop() {
513         if (head == null) return -1;
514         int val = head.val;
515         head = head.next;
516         if (head != null) head.prev = null;
517         count--;
518         if (count % 2 == 1) mid = mid.next;
519         return val;
520     }
521
522     public int findMiddle() {
523         return mid != null ? mid.val : -1;
524     }
525
526     public void deleteMiddle() {
527         if (mid == null) return;
528         if (mid.prev != null) mid.prev.next = mid.next;
529         if (mid.next != null) mid.next.prev = mid.prev;
530         if (count % 2 == 0) mid = mid.next;
531         else mid = mid.prev;
532         count--;
533     }
534 }
535

```