

1 | 定个大目标：（春招）吊打（HR）

2020年11月13日晚

栈和队列刷题

232、Implement Queue using Stacks

1、一入一出：双栈法

①、代码

```
1  class MyQueue {
2      //一、算法思想
3      //通过：双栈实现队列， 一个栈（inStack）， 另一个栈
      (outStack)
4
5      //二、实现细节
6      //1、创建两个栈：用LinkedList， 构造方法中进行初始化
7      //2、写一个outStack为空，则inStack全出栈，入outStack
      的方法： inPutOut
8      //3、入栈时：直接入inStack
9      //4、出栈时：调用 inPutOut 方法
10     //5、判断是否为空：instack && outStack 都为空是才为空
11     //6、获取栈顶元素：调用inPutOut方法， 然后outstak.peek
      即可
12
13     //三、测试用例
14     //1、[1, 2, 3]  --->>> push, push, pop, peek
      [null, null, 2 , 1]
15
16     //四、复杂度
17     //1、Time Complexity: O(n)
18     //2、Space Complexity: O(n)
19
20     private Deque<Integer> inStack;
21     private Deque<Integer> outStack;
22
23     public MyQueue() {
24         inStack = new LinkedList<>();
```

```

25         outStack = new LinkedList<>();
26     }
27
28     /** Push element x to the back of queue. */
29     public void push(int x) {
30         inStack.push(x);
31     }
32
33     /** Removes the element from in front of queue and
34     returns that element. */
35     public int pop() {
36         inPutOut(inStack, outStack);
37
38         return outStack.pop();
39     }
40
41     /** Get the front element. */
42     public int peek() {
43         inPutOut(inStack, outStack);
44         return outStack.peek();
45     }
46
47     /** Returns whether the queue is empty. */
48     public boolean empty() {
49         return inStack.isEmpty() && outStack.isEmpty();
50     }
51
52     /**inStack入栈 outStack的方法*/
53     public void inPutOut(Deque inStack, Deque outStack){
54         if(outStack.isEmpty()){
55             while(!inStack.isEmpty()){
56                 outStack.push(inStack.pop());
57             }
58         }
59     }

```

②、代码图

```

1  class MyQueue {
2      //一、算法思想
3      //通过：双栈实现队列， 一个栈 (inStack)， 另一个栈(outStack)
4
5      //二、实现细节
6      //1、创建两个栈：用LinkedList， 构造方法中进行初始化
7      //2、写一个outStack为空，则inStack全出栈，入outStack的方法： inPutOut
8      //3、入栈时：直接入inStack
9      //4、出栈时：调用 inPutOut 方法
10     //5、判断是否为空： instack && outStack 都为空是才为空
11     //6、获取栈顶元素：调用inPutOut方法， 然后outstak.peek即可
12
13     //三、测试用例
14     //1、[1, 2, 3]  --->>> push, push, pop, peek [null, null, 2, 1]
15
16     //四、复杂度
17     //1、Time Complexity: O(n)
18     //2、Space Complexity: O(n)
19
20     /** Initialize your data structure here. */
21
22     private Deque<Integer> inStack;
23     private Deque<Integer> outStack;
24
25     public MyQueue() {
26         inStack = new LinkedList<>();
27         outStack = new LinkedList<>();
28     }
29
30     /** Push element x to the back of queue. */
31     public void push(int x) {
32         inStack.push(x);
33     }
34
35     /** Removes the element from in front of queue and returns that element. */
36     public int pop() {
37         inPutOut(inStack, outStack);
38
39         return outStack.pop();
40     }
41
42     /** Get the front element. */
43     public int peek() {
44         inPutOut(inStack, outStack);
45         return outStack.peek();
46     }
47
48     /** Returns whether the queue is empty. */
49     public boolean empty() {
50         return inStack.isEmpty() && outStack.isEmpty();
51     }
52
53     /**inStack入栈 outStack的方法*/
54     public void inPutOut(Deque inStack, Deque outStack){
55         if(outStack.isEmpty()){
56             while(!inStack.isEmpty()){
57                 outStack.push(inStack.pop());
58             }
59         }  核心所在：当outStack为空时， inStack全部入栈outStack
60     }
61 }
62
63 /**
64  * Your MyQueue object will be instantiated and called as such:

```

③、复杂度

Success Details >

Runtime: 0 ms, faster than 100.00% of Java online submissions for Implement Queue using Stacks.

Memory Usage: 37 MB, less than 8.75% of Java online submissions for Implement Queue using Stacks.

Next challenges:

Remove Duplicate Letters

Design Log Storage System

Design a Stack With Increment Operation

739、Daily Temperatures (Medium)

1、暴力法：

①、（强烈不推荐）代码如下

```
1 class Solution {
2     //一、算法思路：
3     //通过暴力法：两个for循环
4
5     //二、实现细节
6     //1、a[i] , j = i+1, a[i]与a[j]依次比较
7     //2、如果a[j] >= a[i] 则直接相减：存入a[i]
8     //3、break跳出内层循环
9
10    //三、测试用例：
11    //1、[70, 92, 66]
12    //2、[73, 74, 75, 71]
13
14    //四、Complexity
15    //1、Time Complexity: O(n^2)
16    //2、Space Complexity: O(1)
17    public int[] dailyTemperatures(int[] T) {
18        for(int i = 0; i < T.length; i++){
19            boolean flag = true; //true为
20            T[j] < T[i], 作为标志
21            for(int j = i+1; j < T.length; j++){
22                if(T[j] > T[i]){
23                    T[i] = j-i;
24                    flag = false; //此时
25                    T[j] > T[i]
26                    break;
27                }
28            }
29        }
30    }
31 }
```

```

27         }
28
29         if(flag){ //此时 T[i], 后面没有比它: 气温高的日子
30             T[i] = 0;
31         }
32     }
33
34     return T;
35 }
36 }

```

②、代码图（便于分析）

```

1 * class Solution {
2     //一、思路:
3     //通过暴力法: 两个for循环
4
5     //二、实现细节
6     //1、a[i] , j = i+1, a[i]与a[j]依次比较
7     //2、如果a[j] >= a[i] 则直接相减: 存入a[i]
8     //3、break跳出内层循环
9
10    //三、测试用例:
11    //1、[70, 92, 66]
12    //2、[73, 74, 75, 71]
13
14
15    //四、Complexity
16    //1、Time Complexity: O(n^2)
17    //2、Space Complexity: O(1)
18 * public int[] dailyTemperatures(int[] T) {
19     for(int i = 0; i < T.length; i++){
20         boolean flag = true; //true为T[j] < T[i]
21         for(int j = i+1; j < T.length; j++){
22             if(T[j] > T[i]){
23                 T[i] = j-i; //切记: 下标相减, 才是相差天数
24                 flag = false; //此时 T[j] > T[i]
25                 break;
26             }
27         }
28     }
29
30     if(flag){ //此时 T[i], 后面没有比它: 气温高的日子
31         T[i] = 0;
32     }
33 }
34
35     return T;
36 }
37 }

```

③、复杂度 (LeetCode)

Success Details >

Runtime: 922 ms, faster than 11.40% of Java online submissions for Daily Temperatures.

Memory Usage: 47.4 MB, less than 5.12% of Java online submissions for Daily Temperatures.

Next challenges:

Next Greater Element I

Show off your acceptance:



2、栈 + 数组

①、（推荐）代码如下

```
1  class Solution {
2      //一、算法思路
3      //通过利用：栈（存放遍历温度下标） + 数组（存放升温距离天
   数）
4
5      //二、实现细节
6      //1、栈采用：Deque<Integer> stack = new
   LinkedList<>() --->>> 双端链表模拟，速度优于Stack
7      //2、数组采用： int[] distance;
8
9      //三、测试用例
10     //1、[73, 56, 88]
11     //2、[74, 75, 71, 69, 72, 76, 73]
12
13     //四、complexity
14     //1、Time Complexity: O(n)    进行了for循环 n次 +
   栈中元素出栈次数
15     //2、Space Complexity: O(n)    栈n + 数组n
16
17     public int[] dailyTemperatures(int[] T) {
18         //1、创建一个栈
19         Deque<Integer> stack = new LinkedList<>();
20
21         //2、创建一个数组
```

```

22         int length = T.length; //数组
    长度
23         int[] distance = new int[length];
24
25
26         //3、进行遍历
27         for(int curIndex = 0; curIndex < length;
curIndex++){
28             //distance[curIndex] = 0; //此
    处不用进行初始化
29                                     //因
    为java中：数组没有赋初值时：为0
30
31             while(!stack.isEmpty() && T[curIndex] >
T[stack.peek()]){
32                 int preIndex = stack.pop(); //栈顶
    元素的下标
33                 distance[preIndex] = curIndex - preIndex;
34             }
35
36             stack.push(curIndex);
37         }
38
39         return distance;
40     }
41 }

```

②、代码图（便于分析）

```

1  class Solution {
2      //一、思路
3      //通过利用：栈（存放遍历温度下标） + 数组（存放升温距离天数）
4
5      //二、实现细节
6      //1、栈采用：Deque<Integer> stack = new LinkedList<>() ----> 双端链表模拟，速度优于Stack
7      //2、数组采用： int[] distance;
8
9      //三、测试用例
10     //1、[73, 56, 88]
11     //2、[74, 75, 71, 69, 72, 76, 73]
12
13     //四、complexity
14     //1、Time Complexity: O(n) 进行了for循环 n次 + 栈中元素出栈次数
15     //2、Space Complexity: O(n) 栈 + 数组n
16
17     public int[] dailyTemperatures(int[] T) {
18         //1、创建一个栈
19         Deque<Integer> stack = new LinkedList<>();
20
21         //2、创建一个数组
22         int length = T.length; // 数组长度
23         int[] distance = new int[length];
24
25
26         //3、进行遍历
27         for(int curIndex = 0; curIndex < length; curIndex++){
28             //distance[curIndex] = 0; // 此处不用进行初始化，因为java中：数组没有赋初值，为0
29
30             while(!stack.isEmpty() && T[curIndex] > T[stack.peek()]){
31                 int preIndex = stack.pop(); // 栈顶元素的下标
32                 distance[preIndex] = curIndex - preIndex;
33             }
34
35             stack.push(curIndex);
36         }
37
38         return distance;
39     }
40 }

```

③、复杂度

Success Details >

Runtime: **10 ms**, faster than 94.59% of Java online submissions for Daily Temperatures.

Memory Usage: **47.8 MB**, less than 5.12% of Java online submissions for Daily Temperatures.

Next challenges:

Next Greater Element I

Show off your acceptance:

Time Submitted	Status	Runtime	Memory	Language
<div> <div>Problems</div> <div>Pick One</div> <div> <div>< Prev</div> <div>739/1645</div> <div>Next ></div> </div> </div>				

```

20
21 //2、创建一个数组
22 int length = T.length; // 数组长度
23 int[] distance = new int[length];
24
25
26 //3、进行遍历
27 for(int curIndex = 0; curIndex < length; curIndex++){

```

Testcase Run Code Result Debuquer

Accepted Runtime: 0 ms

Your input [73,74,75,71,69,72,76,73]

Output [1,1,4,2,1,1,0,0]

Expected [1,1,4,2,1,1,0,0]

Console How to create a testcase

503、Next Greater Element II

一、栈 + 数组

①、代码图


```

1  class Solution {
2      //一、算法思想
3      //1、利用栈 + 返回数组：初始化-1
4
5      //二、算法细节
6      //1、将数组：通过取余 2 * length 可以，模拟循环数组
7      //2、遍历需要两遍， 才能最终确认下来， 每一个元素的下一个：最大值
8      //3、提前将：返回数组赋值为 -1， 避免了找不到：赋值-1操作
9      //4、核心：if(!stack.isEmpty() && num > stack.peek()) 则 next[stack.pop()] = num
10     //5、核心：i < length 则 stack.push(i)
11
12     //三、测试用例
13     //1、[1, 2, 1]
14
15     //四、复杂度
16     //1、Time Complexity: O(N)    --> 因为一个 for (2*length次) + 小于等于n次出栈 + i < length入栈
17     //2、Space Complexity: O(N)    --> 一个栈 (入栈n次) + 一个数组长n
18
19     public int[] nextGreaterElements(int[] nums) {
20         //1、获取nums的长度
21         int length = nums.length;
22
23         //2、前置预判断：length == 0 则直接返回
24         if(length == 0){
25             return nums;
26         }
27
28         //3、创建一个栈
29         Deque<Integer> stack = new LinkedList<>();
30
31         //4、创建一个：存放nums[]数组的（下一个大元素的数组）， 并赋值为：-1
32         int maxNext[] = new int[length];
33         Arrays.fill(maxNext, -1);  --> 未雨绸缪
34
35         //5、模拟循环数组：遍历
36         for(int i = 0; i < 2*length; i++){
37             //5.1、记录当前数组元素的：value
38             int curValue = nums[i%length]; 模拟循环数组：用得巧
39
40             //5.2、进行判断
41             while(!stack.isEmpty() && curValue > nums[stack.peek()]){
42                 maxNext[stack.pop()] = curValue;  --> 与栈顶比较：确定下一个大值，用得妙
43             }
44
45             if(i < length){
46                 stack.push(i);  --> 原始长度：都入栈一遍，写得妙
47             }
48         }
49
50         return maxNext;
51     }
52 }

```

②、源代码

```

1  class Solution {
2      //一、算法思想
3      //1、利用栈 + 返回数组：初始化-1
4
5      //二、算法细节
6      //1、将数组：通过取余 2 * length 可以，模拟循环数组
7      //2、遍历需要两遍， 才能最终确认下来， 每一个元素的下一个：最大值
8      //3、提前将：返回数组赋值为 -1， 避免了找不到：赋值-1操作
9      //4、核心：if(!stack.isEmpty() && num > stack.peek()) 则 next[stack.pop()] = num
10     //5、核心：i < length 则 stack.push(i)
11
12     //三、测试用例
13     //1、[1, 2, 1]
14
15     //四、复杂度

```

```

16      //1、Time Complexity: O(N)      -->>因为一个
for (2*length次) + 小于等于n次出栈 + i < length入栈
17      //2、Space Complexity: O(N)      -->>一个栈（入栈n次）
+ 一个数组长n
18
19      public int[] nextGreaterElements(int[] nums) {
20          //1、获取nums的长度
21          int length = nums.length;
22
23          //2、前置预判断: length == 0 则直接返回
24          if(length == 0){
25              return nums;
26          }
27
28          //3、创建一个栈
29          Deque<Integer> stack = new LinkedList<>();
30
31          //4、创建一个: 存放nums[]数组的（下一个大元素的数组），
并赋值为: -1
32          int maxNext[] = new int[length];
33          Arrays.fill(maxNext, -1);
34
35          //5、模拟循环数组: 遍历
36          for(int i = 0; i < 2*length; i++){
37              //5.1、记录当前数组元素的: value
38              int curValue = nums[i%length];
39
40              //5.2、进行判断
41              while(!stack.isEmpty() && curValue >
nums[stack.peek()]){
42                  maxNext[stack.pop()] = curValue;
43              }
44
45              if(i < length){
46                  stack.push(i);
47              }
48          }
49
50          return maxNext;
51
52      }
53  }

```

③、复杂度

Success Details >

Runtime: 6 ms, faster than 94.75% of Java online submissions for Next Greater Element II.

Memory Usage: 41 MB, less than 6.70% of Java online submissions for Next Greater Element II.

Next challenges:

Next Greater Element I

Next Greater Element III

Show off your acceptance:



225、Implement Stack using Queues

一、双端队列法

①、代码图

```

1 class MyStack {
2
3     //一、算法思想
4     //采用：双端队列Deque
5
6     //二、算法细节
7     //1、add时：判断
8     //第一步：（元素）直接入队，求出队列长度 Length
9     //第二步：通过while(length-- > 1)的元素全部出队，再入队，实现逆转， 相当于将（新元素）前的所有元素（都重新入队）
10
11     //2、remove时：直接出栈， 因为（All the calls to pop and top are valid.）
12
13     //3、peek时：直接查看
14
15     //4、empty：直接判断
16
17     //注意：切记poll()和remove()效果是一样的，别乱用
18
19     //三、测试用例
20     //push(1) , push(2) --->> pop() --->> 2
21
22     //四、复杂度
23     //1、Time Complexity: O(n) --->> 主要在 push()处累加和用到了n，其余操作为O(1)
24     //2、Space Complexity: O(n) --->> 双端队列
25
26 /** Initialize your data structure here. */
27
28 private Deque<Integer> queue;
29
30 public MyStack() {
31     queue = new LinkedList<>();
32 }
33
34 /** Push element x onto stack. */
35 public void push(int x) {
36     //1、直接入队
37     queue.add(x);
38
39     //2、求出：队列长度
40     int length = queue.size();
41
42     //3、将入队（新元素）前的（所有元素）重新入队，实现逆转，变成栈的效果
43     while(length-- > 1){
44         queue.add(queue.remove());
45     }
46 }
47
48 /** Removes the element on top of the stack and returns that element. */
49 public int pop() {
50     return queue.remove();
51 }
52
53 /** Get the top element. */
54 public int top() {
55     return queue.peek();
56 }
57
58 /** Returns whether the stack is empty. */
59 public boolean empty() {
60     return queue.isEmpty();
61 }
62 }
63
64 /**
65  * Your MyStack object will be instantiated and called as such:
66  * MyStack obj = new MyStack();
67  * obj.push(x);
68  * int param_2 = obj.pop();
69  * int param_3 = obj.top();
70  * boolean param_4 = obj.empty();

```

②、源代码

```

1 class MyStack {
2
3     //一、算法思想
4     //采用：双端队列Deque
5
6     //二、算法细节
7     //1、add时：判断
8     //第一步：（元素）直接入队，求出队列长度 length
9     //第二步：通过while(length-- > 1)的元素全部出队，
10    再入队，实现逆转， 相当于将（新元素）前的所有元素（都重新入队）
11
12     //2、remove时：直接出栈， 因为（All the calls to pop
13    and top are valid.）

```

```

12
13         //3、peek时：直接查看
14
15         //4、empty：直接判断
16
17         //注意：切记poll()和remove()效果是一样的，别乱用
18
19         //三、测试用例
20         //push(1) ,    push(2)    --->>> pop() --->>> 2
21
22         //四、复杂度
23         //1、Time Complexity: O(n) --->>> 主要在 push()
        处累加和用到了n，其余操作为O(1)
24         //2、Space Complexity: O(n) --->>> 双端队列
25
26         /** Initialize your data structure here. */
27
28         private Deque<Integer> queue;
29
30         public MyStack() {
31             queue = new LinkedList<>();
32         }
33
34         /** Push element x onto stack. */
35         public void push(int x) {
36             //1、直接入队
37             queue.add(x);
38
39             //2、求出：队列长度
40             int length = queue.size();
41
42             //3、将入队（新元素）前的（所有元素）重新入队， 实现逆
        转，变成栈的效果
43             while(length-- > 1){
44                 queue.add(queue.remove());
45             }
46         }
47
48         /** Removes the element on top of the stack and
        returns that element. */
49         public int pop() {
50             return queue.remove();
51         }

```

```
52
53     /** Get the top element. */
54     public int top() {
55         return queue.peek();
56     }
57
58     /** Returns whether the stack is empty. */
59     public boolean empty() {
60         return queue.isEmpty();
61     }
62 }
63
```

③、复杂度

Success Details >

Runtime: 0 ms, faster than 100.00% of Java online submissions for Implement Stack using Queues.

Memory Usage: 36.8 MB, less than 8.40% of Java online submissions for Implement Stack using Queues.

Next challenges:

Trapping Rain Water

Basic Calculator

Zigzag Iterator

Show off your acceptance:



155、Min Stack (Easy)

一、栈 + Node 结点法

①、代码图

```

1 class MinStack {
2     //一、算法思想
3     //利用：内部结点类Node + 模拟栈Deque 实现， 始终保持栈顶元素(结点.minValue属性)存放着（栈最小值）
4
5     //二、算法细节
6     //1、创建一个内部类：Node
7     //成员变量(value):      存放当前：入栈值x
8     //成员变量(minValue) :   存放栈中：最小值
9
10    //2、创建一个LinkedList模拟栈
11
12    //3、在push：进行判断
13    //1、当stack为空时： 直接Node入栈，
14    //2、当stack不为空时： 将 (Node.minValue值) 与 (栈顶元素.minValue) 进行比较
15    //若Node.minValue < 栈顶元素.minValue 则直接入栈
16    //否则：将Node.minValue = 栈顶元素.minValue 再入栈
17    //3、stack.getMin() ---->> 直接获取(栈顶.minValue)即可
18    //4、其余：pop, peek, isEmpty 不用更改
19
20    //三、测试用例
21    //push 1 ---->> push -2 ---->> push 0 ---->>getMin = -2, pop = 0, getMin = -2,
22
23    //四、复杂度
24    //Time Complexity: O(1) ---->> push、pop、top、getMin： 这几个方法（都是常数阶），所以为O(1)
25    //Space Complexity: O(n) ---->> 用到了：栈 + Node， 当最坏情况是，全部入栈，所以为 O(n)
26
27
28    //1、创建一个栈
29    private Deque<Node> stack;
30
31    /** initialize your data structure here. */
32    public MinStack() {
33        stack = new LinkedList<>();
34    }
35
36    public void push(int x) {
37        if(!stack.isEmpty() && x > stack.peek().minValue){ //此时栈不为空， （当前栈顶）存放的最小值：依然是最小值
38            int minValue = stack.peek().minValue;
39            stack.push(new Node(x, minValue));
40        }else{
41            stack.push(new Node(x, x));
42        }
43    }
44
45    public void pop() {
46        stack.pop();
47    }
48
49    public int top() {
50        return stack.peek().value;
51    }
52
53    public int getMin() {
54        return stack.peek().minValue;
55    }
56
57    private class Node{
58        private int value;      //存放当前：入栈值x
59        private int minValue;    //存放栈中：最小值
60
61        public Node(int value, int minValue){
62            this.value = value;
63            this.minValue = minValue;
64        }
65    }
66
67

```

此处为精髓：

- 1、如果：栈不为空 && x > 栈顶.minValue 也就是最小值时，最小值如旧。
- 2、否则：x就是最小值， 直接入栈，成为新的（最小值）

②、源代码

```

1 class MinStack {
2     //一、算法思想
3     //利用：内部结点类Node + 模拟栈Deque 实现， 始终保持栈
4     顶元素(结点.minValue属性)存放着（栈最小值）
5
6     //二、算法细节
7     //1、创建一个内部类：Node
8     //成员变量(value):      存放当前：入栈值x
9     //成员变量(minValue) :   存放栈中：最小值
10
11    //2、创建一个LinkedList模拟栈
12
13    //3、在push：进行判断

```

```

13         //1、当stack为空时： 直接Node入栈，
14         //2、当stack不为空时：将（Node.minValue值） 与
        （栈顶元素.minValue） 进行比较
15         //若Node.minValue < 栈顶元素.minValue 则直
        接入栈
16         //否则：将Node.minValue = 栈顶元素.minValue
        再入栈
17         //3、stack.getMin() --->>> 直接获取(栈
        顶.minValue)即可
18         //4、其余：pop, peek, isEmpty 不用更改
19
20     //三、测试用例
21     //push 1 -->>> push -2 -->>> push 0      ----
    >>getMin = -2,  pop = 0,  getMin = -2,
22
23     //四、复杂度
24     //Time Complexity: O(1)    --->>> push、
    pop、top、getMin: 这几个方法（都是常数阶），所以为O(1)
25     //Space Complexity: O(n)    --->>> 用到了：栈
    + Node, 当最坏情况是，全部入栈， 每次x都比栈顶的（minValue）要
    小,O(n)跑不掉了
26
27
28     //1、创建一个栈
29     private Deque<Node> stack;
30
31     /** initialize your data structure here. */
32     public MinStack() {
33         stack = new LinkedList<>();
34     }
35
36     public void push(int x) {
37         if(!stack.isEmpty() && x > stack.peek().minValue){
        //此时栈不为空， （当前栈顶）存放的最小值：依然是最小值
38             int minValue = stack.peek().minValue;
39             stack.push(new Node(x, minValue));
40         }else{
41             stack.push(new Node(x, x));
42         }
43     }
44 }
45
46     public void pop() {

```



```

47         stack.pop();
48     }
49
50     public int top() {
51         return stack.peek().value;
52     }
53
54     public int getMin() {
55         return stack.peek().minValue;
56     }
57
58     private class Node{
59         private int value;           //存放当前：入栈值x
60         private int minValue;       //存放栈中：最小值
61
62         public Node(int value, int minValue){
63             this.value = value;
64             this.minValue = minValue;
65         }
66     }
67 }
68
69 /**
70  * Your MinStack object will be instantiated and called as
71  * such:
72  * MinStack obj = new MinStack();
73  * obj.push(x);
74  * obj.pop();
75  * int param_3 = obj.top();
76  * int param_4 = obj.getMin();
77  */

```

③、复杂度

Success Details >

Runtime: **4 ms**, faster than 92.81% of Java online submissions for Min Stack.

Memory Usage: **40.9 MB**, less than 50.53% of Java online submissions for Min Stack.

Next challenges:

Sliding Window Maximum

Max Stack

Show off your acceptance:



二、数据栈 + 辅助（最小值）栈

①、代码图

```
1 * class MinStack {
2     //一、算法思想
3     //运用：数据栈 + 辅助（最小值）栈 = 双栈模式
4
5     //二、算法细节
6     //1、创建两个栈：
7         //①、Deque<Integer> dataStack = new LinkedList<>();
8         //②、Deque<Integer> minStack = new LinkedList<>();
9     //2、push时：
10        //①、直接入栈：dataStack
11        //②、判断minStack是否为空
12        //如果为空：直接入栈
13        //否则：判断入栈值（与）minStack.peek()大小
14        //如果：入栈值（小于等于），则入栈，因为 = 不入，会导致出栈后，拿peek时空指针
15        //否则：不入栈
16    //3、pop时：将dataStack直接入栈，并记录值 value，并用value 和 minStack.peek()比较大小
17    //如果：相等时，minStack.pop()也出栈
18
19
20    //三、测试用例
21    //push 1, push -2, push 0 --->>> getMin() = -2, pop() = 0, getMin() = -2
22
23    //四、复杂度
24    //Time Complexity : O(1) --->>> 所有方法操作都是：O(1)
25    //Space Complexity: O(n) --->>> 双栈
26
27    //1、创建：数据栈 + 辅助（最小值）栈
28    private Deque<Integer> dataStack;
29    private Deque<Integer> minStack;
30
31    /** initialize your data structure here. */
32    public MinStack() {
33        dataStack = new LinkedList<>();
34        minStack = new LinkedList<>();
35    }
36
37    public void push(int x) {
38        dataStack.push(x);
39        if(minStack.isEmpty() || x <= minStack.peek()){ // 切记：x == 最小值也要入栈
40            minStack.push(x); // 因为出栈时：最小栈可能也会出栈，不入栈导致：最小栈.peek时空指针
41        }
42    }
43    // 切记：此处(等于 =)也要入栈，否则待pop()出栈后，去getMin()会出现：空指针
44    public void pop() {
45        int value = dataStack.pop();
46        if(value == minStack.peek()){
47            minStack.pop();
48        }
49    }
50
51    public int top() {
52        return dataStack.peek();
53    }
54
55    public int getMin() {
56        return minStack.peek();
57    }
58 }
```

②、源代码

```
1 class MinStack {
2     //一、算法思想
3     //运用：数据栈 + 辅助（最小值）栈 = 双栈模式
4
5     //二、算法细节
6     //1、创建两个栈：
7         //①、Deque<Integer> dataStack = new
LinkedList<>();
8         //②、Deque<Integer> minStack = new
LinkedList<>();
9     //2、push时：
10        //①、直接入栈：dataStack
11        //②、判断minStack是否为空
12        //如果为空：直接入栈
```

```

13 //否则：判断入栈值 （与）
minStack.peek()大小
14 //如果：入栈值（小于等于），
    则入栈， 因为 = 不入，会导致出栈后，拿peek时空指针
15 //否则：不入栈
16 //3、pop时：将dataStack直接入栈， 并记录值 value， 并
    用value 和 minStack.peek()比较大小
17 //如果：相等时，minStack.pop()也出栈
18
19
20 //三、测试用例
21 //push 1, push -2, push 0    --->>> getMin() =
    -2, pop() = 0, getMin() = -2
22
23 //四、复杂度
24 //Time Complexity : O(1)    -->>>所有方法操作都是：
    O(1)
25 //Space Complexity: O(n)    -->>> 双栈
26
27 //1、创建：数据栈 + 辅助（最小值）栈
28 private Deque<Integer> dataStack;
29 private Deque<Integer> minStack;
30
31 /** initialize your data structure here. */
32 public MinStack() {
33     dataStack = new LinkedList<>();
34     minStack = new LinkedList<>();
35 }
36
37 public void push(int x) {
38     dataStack.push(x);
39     if(minStack.isEmpty() || x <= minStack.peek()){ //
        切记：x == 最小值也要入栈
40         minStack.push(x); //
        因为出栈时：最小栈可能也会出栈，不入栈导致：最小栈.peek时空指针
41     }
42 }
43
44 public void pop() {
45     int value = dataStack.pop();
46     if(value == minStack.peek()){
47         minStack.pop();
48     }

```

```
49     }
50
51     public int top() {
52         return dataStack.peek();
53     }
54
55     public int getMin() {
56         return minStack.peek();
57     }
58 }
59
```

③、复杂度

Success Details >

Runtime: 4 ms, faster than 92.81% of Java online submissions for Min Stack.

Memory Usage: 41.2 MB, less than 22.00% of Java online submissions for Min Stack.

Next challenges:

Sliding Window Maximum

Max Stack

Show off your acceptance:



三、自定义：链栈

①、代码图

```

1 class MinStack {
2     //一、算法思路
3     //采用：自己创建（一个链栈） + 头插法
4
5     //二、算法细节
6     //1、创建：链结点LinkStack（数据结构）
7     //int value: 存放当前入栈值
8     //int minValue: 存放栈最小值
9     //StackNode next: 存放下一个结点
10    //2、push时：若链栈 LinkStack == null --->>> 直接入栈，最小值为x
11    //否则：用Math.min(x, LinkStack.minValue)找出最小值，然后入栈
12    //入栈：采用头插法，通过head = LinkStack的构造方法，巧妙插入
13    //3、pop时：直接head = head.next
14    //4、top时：直接head.value
15    //5、getMin时：直接head.getMin
16
17    //三、测试用例
18    //push 1, push -2, push 0 --->>> getMin() = -2, pop() = 0, getMin() = -2
19
20    //四、复杂度
21    //Time Complexity: O(1)
22    //Space Complexity: O(n)
23
24    //1、创建一个：链栈的头指针
25    private LinkedNode head;
26
27    /** initialize your data structure here. */
28    public MinStack() {
29
30    }
31
32    public void push(int x) {
33        if(head == null){
34            head = new LinkedNode(x, x, head);
35        }else{
36            head = new LinkedNode(x, Math.min(x, head.minValue), head);
37        }
38    }
39    //核心在于：传入当前链表，通过构造方法，进行（头插法）操作
40
41    public void pop() {
42        head = head.next;
43    }
44
45    public int top() {
46        return head.value;
47    }
48
49    public int getMin() {
50        return head.minValue;
51    }
52
53    private class LinkedNode{
54        private int value; //存放当前入栈值
55        private int minValue; //存放栈最小值
56        private LinkedNode next; //存放下一个结点
57
58        LinkedNode(int value, int minValue, LinkedNode head){
59            this.value = value;
60            this.minValue = minValue;
61            this.next = head; //精髓
62        }
63    }
64 }

```

②、源代码

```

1 class MinStack {
2     //一、算法思路
3     //采用：自己创建（一个链栈） + 头插法
4
5     //二、算法细节
6     //1、创建：链结点LinkStack（数据结构）
7     //int value: 存放当前入栈值
8     //int minValue: 存放栈最小值
9     //StackNode next: 存放下一个结点
10    //2、push时：若链栈 LinkStack == null --->>> 直接
    入栈，最小值为x
11    //否则：用Math.min(x, LinkStack.minValue)
    找出最小值，然后入栈

```

```
12         //入栈：采用头插法， 通过head = LinkStack的构造方
    法，巧妙插入
13         //3、pop时：直接head = head.next
14         //4、top时：直接head.value
15         //5、getMin时：直接head.getMin
16
17     //三、测试用例
18         //push 1, push -2, push 0    --->>> getMin() = -2,
    pop() = 0, getMin() = -2
19
20     //四、复杂度
21         //Time Complexity:  O(1)
22         //Space Complexity:  O(n)
23
24
25     //1、创建一个：链栈的头指针
26     private ListNode head;
27
28     /** initialize your data structure here. */
29     public MinStack() {
30
31     }
32
33     public void push(int x) {
34         if(head == null){
35             head = new ListNode(x, x, head);
36         }else{
37             head = new ListNode(x, Math.min(x,
    head.minValue), head);
38         }
39     }
40
41     public void pop() {
42         head = head.next;
43     }
44
45     public int top() {
46         return head.value;
47     }
48
49     public int getMin() {
50         return head.minValue;
51     }
```

```

52
53     private class ListNode{
54         private int value;           //存放当前入栈值
55         private int minValue;        //存放栈最小值
56         private ListNode next;       //存放下一个结点
57
58         ListNode(int value, int minValue, ListNode
head){
59             this.value = value;
60             this.minValue = minValue;
61             this.next = head;
62         }
63     }
64 }
65

```

③、复杂度

Success Details >

Runtime: 4 ms, faster than 92.81% of Java online submissions for Min Stack.

Memory Usage: 40.5 MB, less than 85.51% of Java online submissions for Min Stack.

Next challenges:

Sliding Window Maximum

Max Stack

Show off your acceptance:



20、Valid Parentheses (Easy)

一、栈 + 直接遍历

①、代码图

1 | 可以补充：前置判断括号个数， `if(length % 2 == 1) return false;`

```

1  class Solution {
2      //一、算法思想
3      //采用：遍历字符串 + 栈
4
5      //二、算法细节
6      //1、创建一个栈: Deque<Character> stack = new LinkedList<>()
7      //2、遍历字符串，通过if、else if 进行选择
8      //①、遇到( , [, { 左括号则入栈它们的（右括号）
9      //②、遇到), ], } 时
10         //判断：栈是否为空， 如果为空 return false
11         //判断：右括号是否 == 出栈元素， 如果不等于 return false
12         //3、return stack.isEmpty() 作为是否是：有效括号的结果
13
14
15     //三、测试用例
16     //1、{[]} --->>> return false
17     //2、{[]}] --->>> return true
18     //3、[] --->>> return true
19
20     //四、复杂度
21     //1、Time Complexity: O(n)          --> 因为要遍历length长度
22     //2、Space Complexity: O(n)         --> 用到了：栈
23
24     //五、优缺点
25     //1、advantage: 通俗易懂， 代码量少
26     //2、shortcoming: 拓展性不好， 一旦匹配变多， if将变多， 而且修改起来困难
27
28     public boolean isValid(String s) {
29         //1、创建一个栈
30         Deque<Character> stack = new LinkedList<>();
31
32         //2、获取字符串长度
33         int length = s.length();          //因为题目给定: 1 <= s.length <= 10^4 所以不用前置判断了
34
35         //3、遍历字符串
36         for(int i = 0; i < length; i++){
37             Character c = s.charAt(i);
38
39             if(c == '(')
40                 stack.push(')');
41             else if(c == '[')
42                 stack.push(']');
43             else if(c == '{')
44                 stack.push('}');
45             else if(stack.isEmpty() || (stack.pop() != c))
46                 return false;
47         }
48
49         return stack.isEmpty();
50     }
51 }

```

②、源代码

```

1  class Solution {
2      //一、算法思想
3      //采用：遍历字符串 + 栈
4
5      //二、算法细节
6      //1、创建一个栈: Deque<Character> stack = new
LinkedList<>()
7      //2、遍历字符串，通过if、else if 进行选择
8      //①、遇到( , [, { 左括号则入栈它们的（右括号）
9      //②、遇到), ], } 时
10         //判断：栈是否为空， 如果为空 return false
11         //判断：右括号是否 == 出栈元素， 如果不等于
return false
12         //3、return stack.isEmpty() 作为是否是：有效括号的结
果

```



```

13
14
15 //三、测试用例
16 //1、([]) --->>> return false
17 //2、{[]} --->>> return true
18 //3、[] --->>> return true
19
20 //四、复杂度
21 //1、Time Complexity: O(n) -->> 因为要遍历
length长度
22 //2、Space Complexity: O(n) -->> 用到了：栈
23
24 //五、优缺点
25 //1、advantage: 通俗易懂， 代码量少
26 //2、shortcoming: 拓展性不好， 一旦匹配变多， if将变
多， 而且修改起来困难
27
28 public boolean isValid(String s) {
29 //1、创建一个栈
30 Deque<Character> stack = new LinkedList<>();
31
32 //2、获取字符串长度
33 int length = s.length(); //因为题目给定：1
<= s.length <= 10^4 所以不用前置判断了
34
35 //3、遍历字符串
36 for(int i = 0; i < length; i++){
37 Character c = s.charAt(i);
38
39 if(c == '(')
40 stack.push(')');
41 else if(c == '[')
42 stack.push(']');
43 else if(c == '{')
44 stack.push('}');
45 else if(stack.isEmpty() || (stack.pop() != c))
46 return false;
47 }
48
49 return stack.isEmpty();
50 }
51 }

```

③、复杂度

Success [Details >](#)

Runtime: 1 ms, faster than 99.17% of Java online submissions for Valid Parentheses.

Memory Usage: 37.3 MB, less than 40.64% of Java online submissions for Valid Parentheses.

Next challenges:

[Generate Parentheses](#)

[Longest Valid Parentheses](#)

[Remove Invalid Parentheses](#)

[Check If Word Is Valid After Substitutions](#)

Show off your acceptance:



二、遍历字符串 + 栈 + Map

①、代码图

```

1  class Solution {
2      //一、算法思想
3      //采用：遍历字符串 + 栈 + Map
4
5      //二、算法细节
6      //1、创建一个栈：Deque<Character> stack = new LinkedList<>()
7      //2、创建一个：Map<Character, Character> = new HashMap<>()
8      //3、将()、[], {}, 放入map中， 左括号为key， 右括号为value
9      //4、进行遍历：当前字符到 (Map 查找)
10     //若找到：则将Map的value入栈
11     //若找不到：判断当前栈是否为空 || 当前遍历字符 != 出栈元素
12     //栈若为空：说明（右括号）找不到匹配的（左括号） --> 直接return false
13     //当前遍历字符 != 出栈元素：说明匹配不上 --> 直接return false
14     //5、最后用：stack.isEmpty()的原因：因为若能匹配，则都出栈了，不能匹配则（栈不为空）
15
16     //三、测试用例
17     //1、([]) --->> return false
18     //2、{[]} --->> return true
19     //3、[] --->> return true
20
21     //四、复杂度
22     //1、Time Complexity: O(n) --> 因为要遍历length长度
23     //2、Space Complexity: O(n) --> 最坏超过 n，因为用到了Map
24
25     //五、优缺点
26     //1、advantage： 添加了前置判断、拓展性强（增加匹配元素时）易于添加
27     //2、shortcoming： 耗费了空间
28
29     public boolean isValid(String s) {
30         //1、获取字符串长度
31         int length = s.length(); //因为题目给定：1 <= s.length <= 10^4 所以对此做判断
32
33         if(length % 2 == 1){ 此处用得巧
34             return false; //奇数个括号，不可能匹配
35         }
36
37         //2、创建一个栈
38         Deque<Character> stack = new LinkedList<>();
39
40
41         //3、创建一个Map，并将成对括号：放入
42         Map<Character, Character> map = new HashMap<>(); //直接用map.put()而不用：匿名内部类，统一初
43         始化
44         map.put('(', ')'); //是因为：map.put()的效率更高
45         map.put('[', ']');
46         map.put('{', '}');
47
48
49         //4、遍历字符串
50         for(int i = 0; i < length; i++){
51             Character c = s.charAt(i);
52             if(map.containsKey(c)) containsKey()中是有s的，注意了
53                 stack.push(map.get(c));
54             else if(stack.isEmpty() || c != stack.pop())
55                 return false;
56         }
57
58         return stack.isEmpty();
59     }
60 }

```

②、源代码

```

1  class Solution {
2      //一、算法思想
3      //采用：遍历字符串 + 栈 + Map
4
5      //二、算法细节
6      //1、创建一个栈：Deque<Character> stack = new
7      LinkedList<>()
8      //2、创建一个：Map<Character, Character> = new
9      HashMap<>()
10     //3、将()、[], {}, 放入map中， 左括号为key， 右括号为
11     value

```

```

9      //4、进行遍历：当前字符到（Map查找）
10      //若找到：则将Map的value入栈
11      //若找不到：判断当前栈是否为空 || 当前遍历字符
    != 出栈元素
12      //栈若为空：说明（右括号）找不到匹配的（左
    括号） -->> 直接return false
13      //当前遍历字符 != 出栈元素： 说明匹配不上
    -->> 直接return false
14      //5、最后用：stack.isEmpty()的原因：因为若能匹配，则都
    出栈了，不能匹配则（栈不为空）
15
16      //三、测试用例
17      //1、([]]    --->>> return false
18      //2、{[]}    --->>> return true
19      //3、[]      --->>> return true
20
21      //四、复杂度
22      //1、Time Complexity: O(n)          -->> 因为要遍历
    length长度
23      //2、Space Complexity: O(n)        -->> 最坏超过 n
    , 因为用到了Map
24
25      //五、优缺点
26      //1、advantage:  添加了前置判断、拓展性强（增加匹配元
    素时）易于添加
27      //2、shortcoming:  耗费了空间
28
29      public boolean isValid(String s) {
30          //1、获取字符串长度
31          int length = s.length();          //因为题目给定：1
    <= s.length <= 10^4 所以对此做：判断
32
33          if(length % 2 == 1){
34              return false;                //奇数个括号， 不可
    能匹配
35          }
36
37          //2、创建一个栈
38          Deque<Character> stack = new LinkedList<>();
39
40
41          //3、创建一个Map，并将成对括号：放入

```

```

42      Map<Character, Character> map = new HashMap<>();
      //直接用map.put()而不用：匿名内部类，统一初始化
43      map.put('(', ')');
      //是因为：map.put()的效率更高
44      map.put('[', ']');
45      map.put('{', '}');
46
47
48
49      //4、遍历字符串
50      for(int i = 0; i < length; i++){
51          Character c = s.charAt(i);
52          if(map.containsKey(c))
53              stack.push(map.get(c));
54          else if(stack.isEmpty() || c != stack.pop())
55              return false;
56      }
57
58      return stack.isEmpty();
59  }
60 }

```

③、复杂度

Success Details >

Runtime: 1 ms, faster than 99.17% of Java online submissions for Valid Parentheses.

Memory Usage: 37.5 MB, less than 32.48% of Java online submissions for Valid Parentheses.

Next challenges:

Generate Parentheses

Longest Valid Parentheses

Remove Invalid Parentheses

Check If Word Is Valid After Substitutions

Show off your acceptance:



三、哨兵 + stack

①、代码图



②、源代码

```

1  class Solution {
2      //1、创建Map
3      private static final Map<Character, Character> map =
new HashMap<>(){
4          {
5              put('(', ')');
6              put('[', ']');
7              put('{', '}');
8              put('?', '?');          //作为哨兵
9          }
10     };
11
12     public boolean isValid(String s) {
13         //1、前置欲判断
14         if(s.length() % 2 == 1 ||
!map.containsKey(s.charAt(0)))
15             return false;
16
17         //2、创建一个栈
18         Deque<Character> stack = new LinkedList<>();
19         stack.push('?');          // (?, ?)作为哨兵
20
21         //3、遍历：判断
22         for(char c : s.toCharArray()){
23             if(map.containsKey(c)){
24                 stack.push(c);
25             }else if(map.get(stack.pop()) != c){
26                 return false;
27             }
28         }
29
30         return stack.size() == 1;
31     }
32 }

```

③、复杂度

Success [Details >](#)

Runtime: 1 ms, faster than 99.17% of Java online submissions for Valid Parentheses.

Memory Usage: 37.3 MB, less than 51.26% of Java online submissions for Valid Parentheses.

Next challenges:

[Generate Parentheses](#)

[Longest Valid Parentheses](#)

[Remove Invalid Parentheses](#)

[Check If Word Is Valid After Substitutions](#)

Show off your acceptance:

