```
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

```
class MyQueue {
    //一、算法思想
        //通过：双栈实现队列， 一个栈（inStack）, 另一个栈(outStack)

    //二、实现细节
        //1、创建两个栈：用LinkedList， 构造方法中进行初始化
        //2、写一个outStack为空，则inStack全出栈，入outStack的方法： inPutOut
        //3、入栈时：直接入inStack
        //4、出栈时：调用 inPutOut 方法
        //5、判断是否为空：instack && outStack 都为空是才为空
        //6、获取栈顶元素：调用inPutOut方法， 然后outstak.peek即可

    //三、测试用例
        //1、[1, 2, 3]   --->>> push, push, pop, peek [null, null, 2 , 1]

    //四、复杂度
        //1、Time  Complexity：O(n)
        //2、Space Complexity: O(n)

    private Deque<Integer> inStack;
    private Deque<Integer> outStack;

    public MyQueue() {
        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
    returns that element. */
34      public int pop() {
35          inPutOut(inStack, outStack);
36
37          return outStack.pop();
38      }
39
40      /** Get the front element. */
41      public int peek() {
42          inPutOut(inStack, outStack);
43          return outStack.peek();
44      }
45
46      /** Returns whether the queue is empty. */
47      public boolean empty() {
48          return inStack.isEmpty() && outStack.isEmpty();
49      }
50
51      /**inStack入栈 outStack的方法*/
52      public void inPutOut(Deque inStack, Deque outStack){
53          if(outStack.isEmpty()){
54              while(!inStack.isEmpty()){
55                  outStack.push(inStack.pop());
56              }
57          }
58      }
59  }
```

②、代码图

```java
class MyQueue {
    //一、算法思想
        //通过：双栈实现队列，  一个栈（inStack），另一个栈(outStack)

    //二、实现细节
        //1、创建两个栈：用LinkedList，  构造方法中进行初始化
        //2、写一个outStack为空，则inStack全出栈，入outStack的方法：  inPutOut
        //3、入栈时：直接入inStack
        //4、出栈时：调用 inPutOut 方法
        //5、判断是否为空：instack && outStack 都为空是才为空
        //6、获取栈顶元素：调用inPutOut方法，  然后outstak.peek即可

    //三、测试用例
        //1、[1, 2, 3]   --->>> push, push, pop, peek  [null, null, 2 , 1]

    //四、复杂度
        //1、Time  Complexity：O(n)
        //2、Space Complexity：O(n)

    /** Initialize your data structure here. */

    private Deque<Integer> inStack;
    private Deque<Integer> outStack;

    public MyQueue() {
        inStack = new LinkedList<>();
        outStack = new LinkedList<>();
    }

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

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        inPutOut(inStack, outStack);

        return outStack.pop();
    }

    /** Get the front element. */
    public int peek() {
        inPutOut(inStack, outStack);
        return outStack.peek();
    }

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

    /**inStack入栈 outStack的方法*/
    public void inPutOut(Deque inStack, Deque outStack){
        if(outStack.isEmpty()){
            while(!inStack.isEmpty()){
                outStack.push(inStack.pop());
            }
        }
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
```

核心所在：当outStack为空时， inStack全部入栈outStack

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

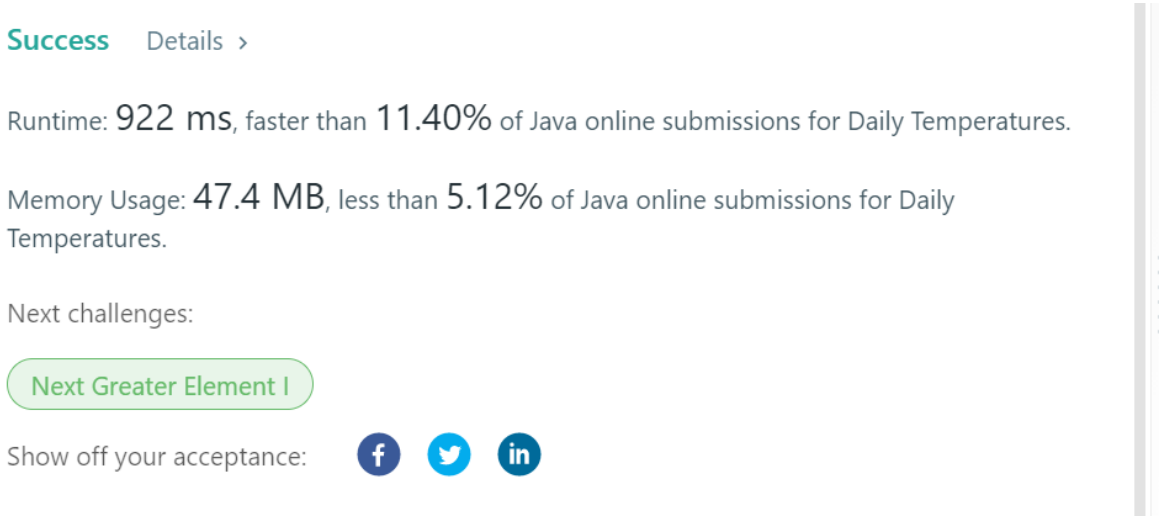
```
class Solution {
    //一、算法思路：
         //通过暴力法：两个for循环

    //二、实现细节
        //1、a[i] , j = i+1, a[i]与a[j]依次比较
        //2、如果a[j] >= a[i] 则直接相减：存入a[i]
        //3、break跳出内层循环

    //三、测试用例：
        //1、[70, 92, 66]
        //2、[73, 74, 75, 71]

    //四、Complexity
        //1、Time  Complexity: O(n^2)
        //2、Space Complexity: O(1)
    public int[] dailyTemperatures(int[] T) {
        for(int i = 0; i < T.length; i++){
            boolean flag = true;                //true为T[j] < T[i]，作为标志
            for(int j = i+1; j < T.length; j++){
                if(T[j] > T[i]){
                    T[i] = j-i;
                    flag = false;               //此时T[j] > T[i]
                    break;
                }

```

```
                }

                if(flag){ //此时 T[i], 后面没有比它：气温高的日子
                    T[i] = 0;
                }
            }

            return T;
        }
}
```

②、代码图（便于分析）


```
class Solution {
    //一、算法思路：
         //通过暴力法：两个for循环

    //二、实现细节
        //1、a[i] , j = i+1, a[i]与a[j]依次比较
        //2、如果a[j] >= a[i] 则直接相减：存入a[i]
        //3、break跳出内层循环

    //三、测试用例：
        //1、[70, 92, 66]
        //2、[73, 74, 75, 71]


    //四、Complexity
       //1、Time  Complexity: O(n^2)
       //2、Space Complexity: O(1)
    public int[] dailyTemperatures(int[] T) {
        for(int i = 0; i < T.length; i++){
            boolean flag = true;                            //true为T[j] < T[i]
            for(int j = i+1; j < T.length; j++){
                if(T[j] > T[i]){
                    T[i] = j-i;                             //切记：下标相减，才是相差天数
                    flag = false;                           //此时 T[j] > T[i]
                    break;
                }

            }

            if(flag){ //此时 T[i], 后面没有比它：气温高的日子
                T[i] = 0;
            }
        }

        return T;
    }
}
```

③、复杂度（LeetCode）



2、栈 + 数组

①、（推荐）代码如下

```
class Solution {
    //一、算法思路
        //通过利用：栈（存放遍历温度下标） + 数组（存放升温距离天数）

    //二、实现细节
        //1、栈采用：Deque<Integer> stack = new LinkedList<>() --->>> 双端链表模拟，速度优于Stack
        //2、数组采用：  int[] distance;

    //三、测试用例
        //1、[73, 56, 88]
        //2、[74, 75, 71, 69, 72, 76, 73]

    //四、complexity
        //1、Time  Complexity:  O(n)    进行了for循环 n次 + 栈中元素出栈次数
        //2、Space Complexity:  O(n)    栈n + 数组n

    public int[] dailyTemperatures(int[] T) {
        //1、创建一个栈
        Deque<Integer> stack = new LinkedList<>();

        //2、创建一个数组
```

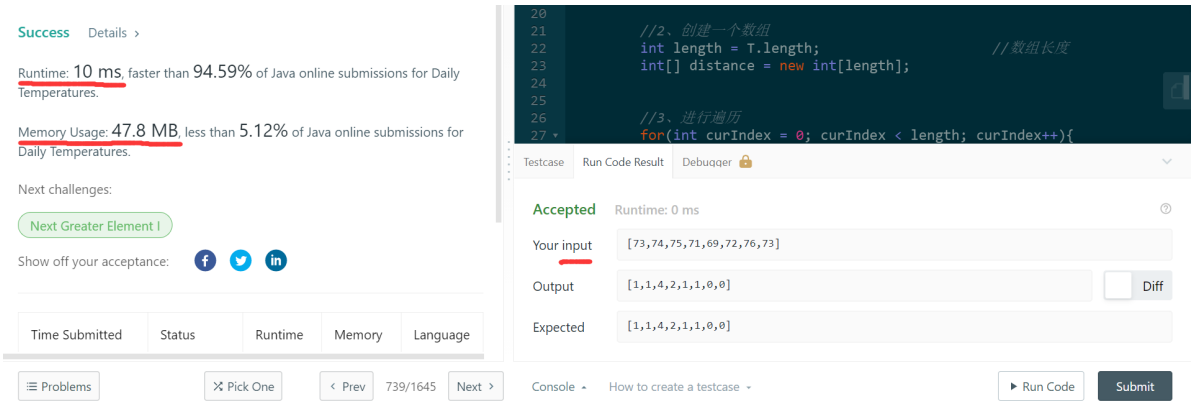
```
22          int length = T.length;                          //数组
    长度
23          int[] distance = new int[length];
24
25
26          //3、进行遍历
27          for(int curIndex = 0; curIndex < length;
    curIndex++){
28              //distance[curIndex] = 0;                  //此
    处不用进行初始化
29                                                          //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）

```java
class Solution {
    //一、算法思路
        //通过利用：栈（存放遍历温度下标） + 数组（存放升温距离天数）

    //二、实现细节
        //1、栈采用：Deque<Integer> stack = new LinkedList<>() ---->>> 双端链表模拟，速度优于Stack
        //2、数组采用：  int[] distance;

    //三、测试用例
        //1、[73, 56, 88]
        //2、[74, 75, 71, 69, 72, 76, 73]

    //四、complexity
        //1、Time  Complexity:  O(n)     进行了for循环 n次 + 栈中元素出栈次数
        //2、Space Complexity:  O(n)     栈n + 数组n

    public int[] dailyTemperatures(int[] T) {
        //1、创建一个栈
        Deque<Integer> stack = new LinkedList<>();

        //2、创建一个数组
        int length = T.length;                      //数组长度
        int[] distance = new int[length];


        //3、进行遍历
        for(int curIndex = 0; curIndex < length; curIndex++){
            //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

            while(!stack.isEmpty() && T[curIndex] > T[stack.peek()]){
               int preIndex = stack.pop();          //栈顶元素的下标
               distance[preIndex] = curIndex - preIndex;
            }

            stack.push(curIndex);
        }

        return distance;
    }
}
```

精髓所在，栈不为空（和）当前元素（大于）栈顶（继续）

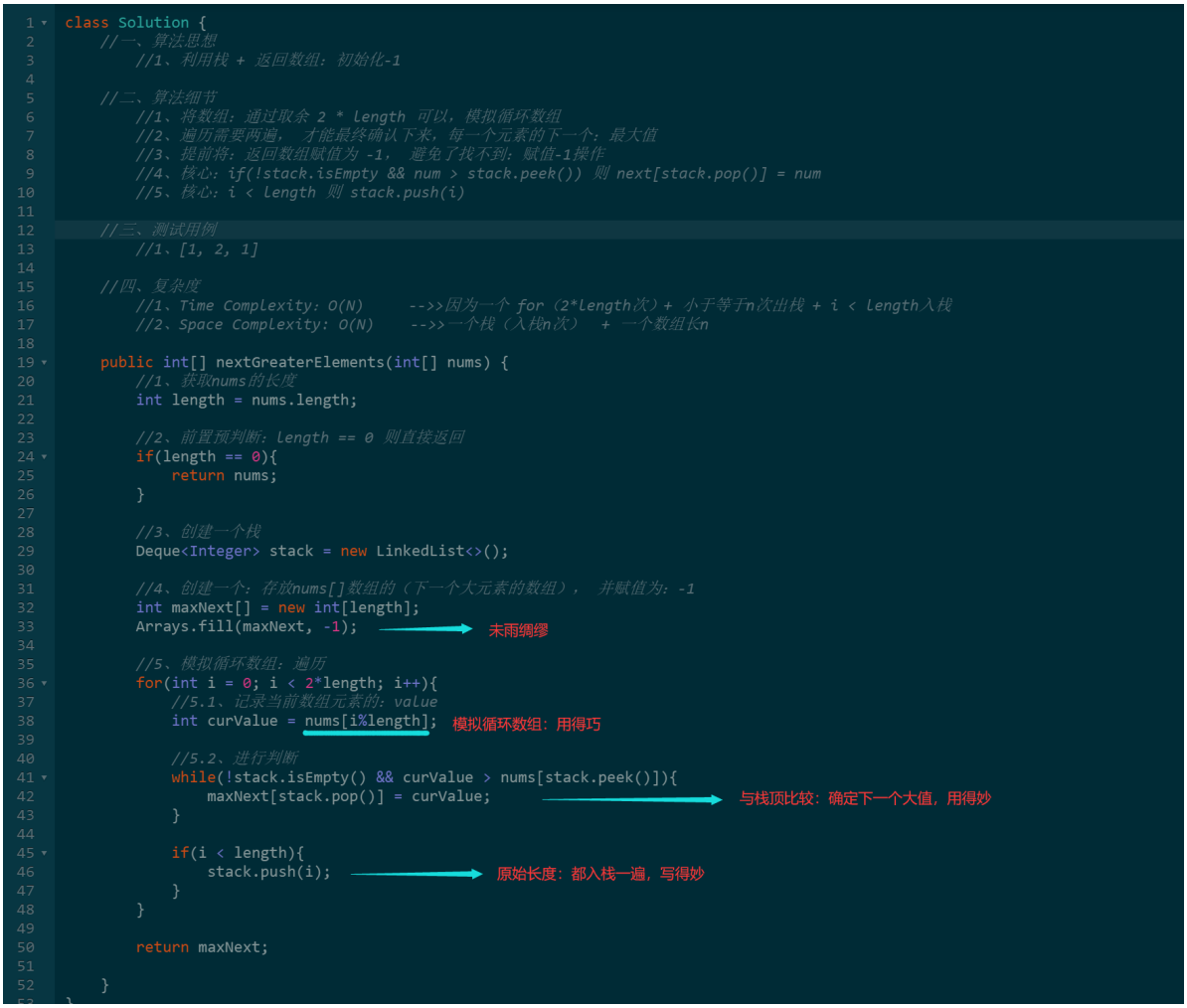
③、复杂度



503、Next Greater Element II

一、栈 + 数组

①、代码图

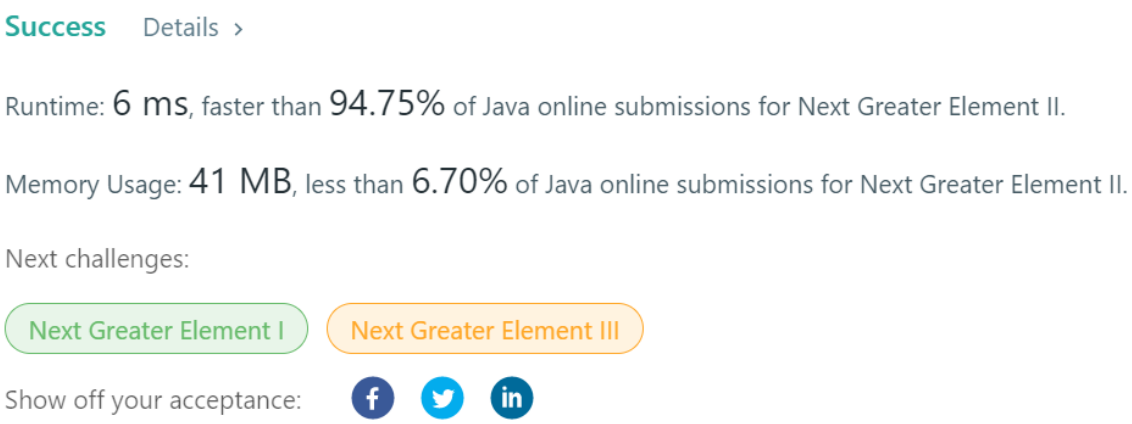


②、源代码

```
class Solution {
    //一、算法思想
        //1、利用栈 + 返回数组：初始化-1

    //二、算法细节
        //1、将数组：通过取余 2 * length 可以，模拟循环数组
        //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
        //3、提前将：返回数组赋值为 -1， 避免了找不到：赋值-1操作
        //4、核心：if(!stack.isEmpty && num > stack.peek()) 则 next[stack.pop()] = num
        //5、核心：i < length 则 stack.push(i)

    //三、测试用例
        //1、[1, 2, 1]

    //四、复杂度
```

```java
        //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
        //2、Space Complexity: O(N)     -->>一个栈（入栈n次）+ 一个数组长n

    public int[] nextGreaterElements(int[] nums) {
        //1、获取nums的长度
        int length = nums.length;

        //2、前置预判断：length == 0 则直接返回
        if(length == 0){
            return nums;
        }

        //3、创建一个栈
        Deque<Integer> stack = new LinkedList<>();

        //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
        int maxNext[] = new int[length];
        Arrays.fill(maxNext, -1);

        //5、模拟循环数组：遍历
        for(int i = 0; i < 2*length; i++){
            //5.1、记录当前数组元素的：value
            int curValue = nums[i%length];

            //5.2、进行判断
            while(!stack.isEmpty() && curValue > nums[stack.peek()]){
                maxNext[stack.pop()] = curValue;
            }

            if(i < length){
                stack.push(i);
            }
        }

        return maxNext;

    }
}
```

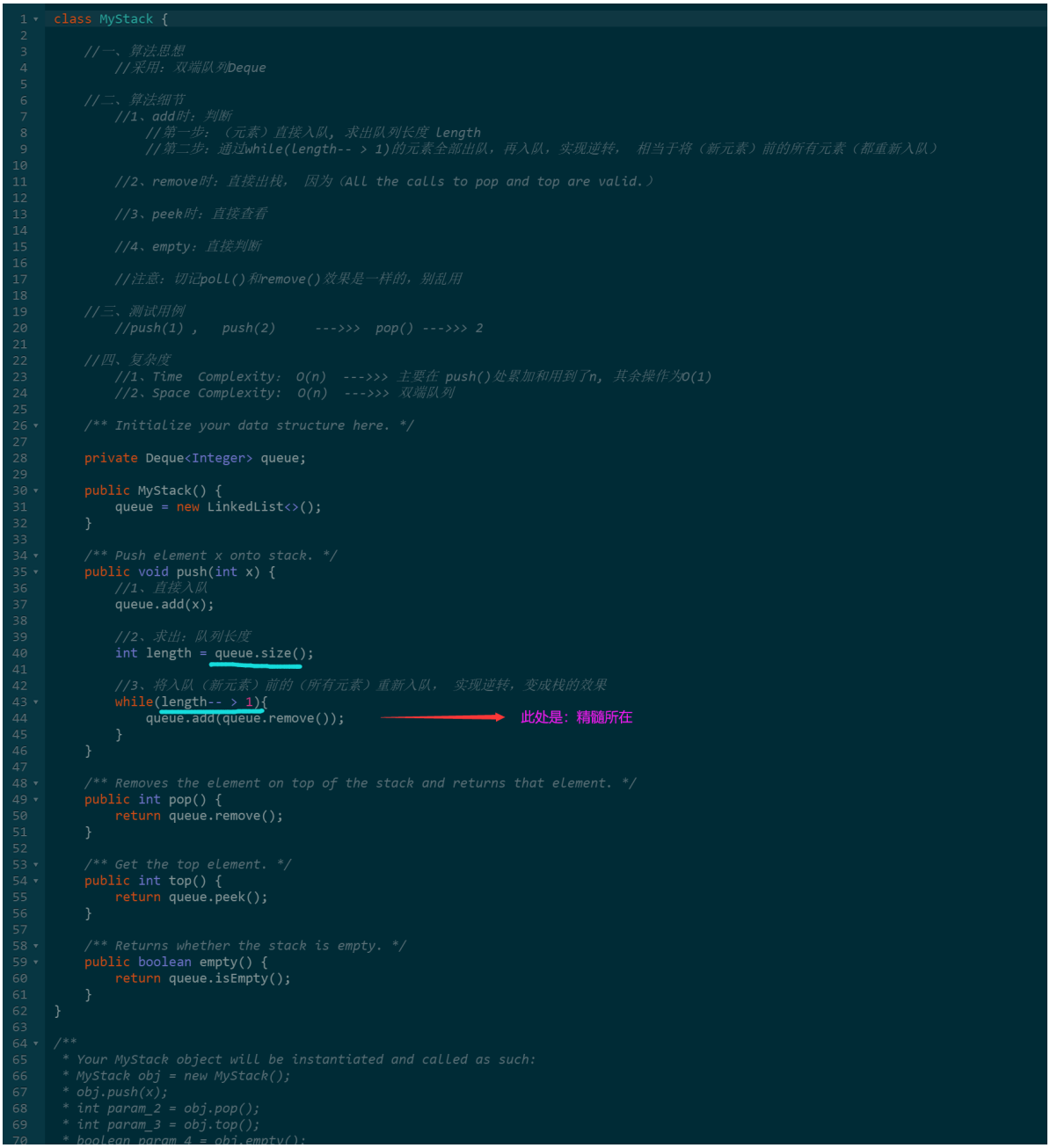
③、复杂度



225、Implement Stack using Queues

一、双端队列法

①、代码图



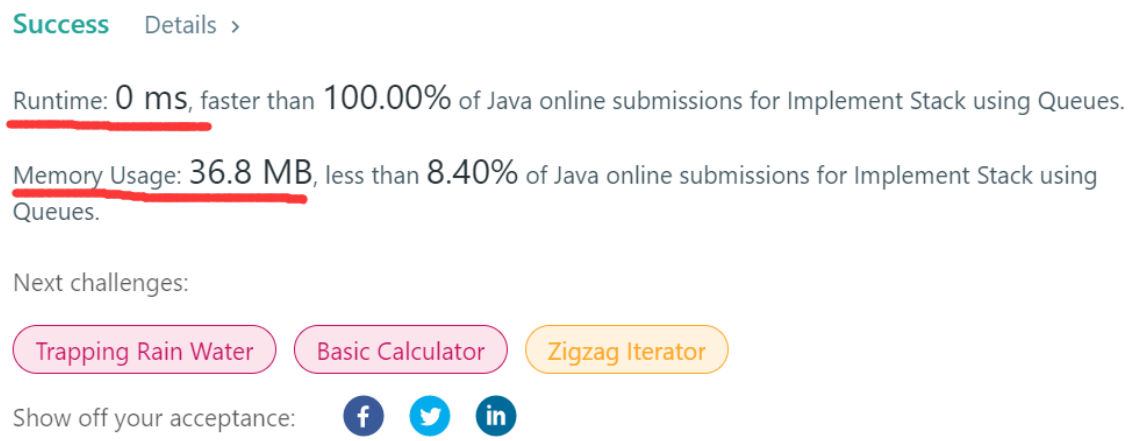
②、源代码

```
class MyStack {

    //一、算法思想
        //采用：双端队列Deque

    //二、算法细节
        //1、add时：判断
            //第一步：（元素）直接入队，求出队列长度 length
            //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）

        //2、remove时：直接出栈， 因为（All the calls to pop and top are valid.）

        //3、peek时：直接查看

        //4、empty：直接判断

        //注意：切记poll()和remove()效果是一样的，别乱用

    //三、测试用例
        //push(1) ,   push(2)     --->>>  pop() --->>> 2

    //四、复杂度
        //1、Time  Complexity:  O(n)  --->>> 主要在 push()处累加和用到了n，其余操作为O(1)
        //2、Space Complexity:  O(n)  --->>> 双端队列

    /** Initialize your data structure here. */

    private Deque<Integer> queue;

    public MyStack() {
        queue = new LinkedList<>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        //1、直接入队
        queue.add(x);

        //2、求出：队列长度
        int length = queue.size();

        //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
        while(length-- > 1){
            queue.add(queue.remove());
        }
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return queue.remove();
    }

    /** Get the top element. */
    public int top() {
        return queue.peek();
    }

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

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * boolean param_4 = obj.empty();
```

```
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



155、Min Stack (Easy)

一、栈 + Node结点法

①、代码图

```java
class MinStack {
    //一、算法思想
        //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）

    //二、算法细节
        //1、创建一个内部类：Node
             //成员变量(value)：      存放当前：入栈值x
             //成员变量(minValue) :  存放栈中：最小值

        //2、创建一个LinkedList模拟栈

        //3、在push：进行判断
             //1、当stack为空时：  直接Node入栈，
             //2、当stack不为空时：将（Node.minValue值） 与（栈顶元素.minValue） 进行比较
                 //若Node.minValue < 栈顶元素.minValue 则直接入栈
                 //否则：将Node.minValue = 栈顶元素.minValue 再入栈
            //3、stack.getMin()  --->>> 直接获取(栈顶.minValue)即可
            //4、其余：pop, peek, isEmpty 不用更改

    //三、测试用例
            //push 1 -->>> push -2 -->>> push 0     ----->>getMin = -2,  pop = 0,   getMin = -2,

    //四、复杂度
            //Time  Complexity: O(1)    --->>>  push、pop、top、getMin:  这几个方法（都是常数阶），所以为O(1)
            //Space Complexity: O(n)    --->>>  用到了：栈 + Node，  当最坏情况是，全部入栈，所以为 O(n)


    //1、创建一个栈
    private Deque<Node> stack;

    /** initialize your data structure here. */
    public MinStack() {
        stack = new LinkedList<>();
    }

    public void push(int x) {
        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
            int minValue = stack.peek().minValue;
            stack.push(new Node(x, minValue));
        }else{                                          此处为精髓：
            stack.push(new Node(x, x));                     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）

    }

    public void pop() {
        stack.pop();
    }

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

    public int getMin() {
        return stack.peek().minValue;
    }

    private class Node{
        private int value;          //存放当前：入栈值x
        private int minValue;       //存放栈中：最小值

        public Node(int value, int minValue){
            this.value = value;
            this.minValue = minValue;
        }
    }
}
```

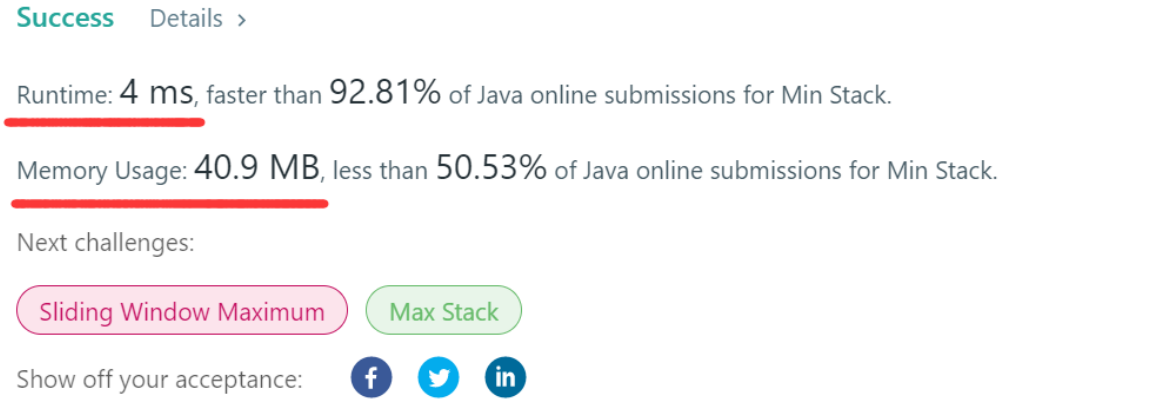
②、源代码

```java
class MinStack {
    //一、算法思想
        //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）

    //二、算法细节
        //1、创建一个内部类：Node
             //成员变量(value)：      存放当前：入栈值x
             //成员变量(minValue) :  存放栈中：最小值

        //2、创建一个LinkedList模拟栈

        //3、在push：进行判断
```

```
            //1、当stack为空时：   直接Node入栈，
            //2、当stack不为空时：将（Node.minValue值） 与（栈顶元素.minValue） 进行比较
                //若Node.minValue < 栈顶元素.minValue 则直接入栈
                //否则：将Node.minValue = 栈顶元素.minValue 再入栈
            //3、stack.getMin()  --->>> 直接获取(栈顶.minValue)即可
            //4、其余：pop, peek, isEmpty 不用更改

    //三、测试用例
            //push 1 -->>> push -2 -->>> push 0     ---->>getMin = -2,  pop = 0,   getMin = -2,

    //四、复杂度
            //Time  Complexity：O(1)    --->>>  push、pop、top、getMin：  这几个方法（都是常数阶），所以为O(1)
            //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


    //1、创建一个栈
    private Deque<Node> stack;

    /** initialize your data structure here. */
    public MinStack() {
        stack = new LinkedList<>();
    }

    public void push(int x) {
        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
            int minValue = stack.peek().minValue;
            stack.push(new Node(x, minValue));
        }else{
            stack.push(new Node(x, x));
        }

    }

    public void pop() {
```

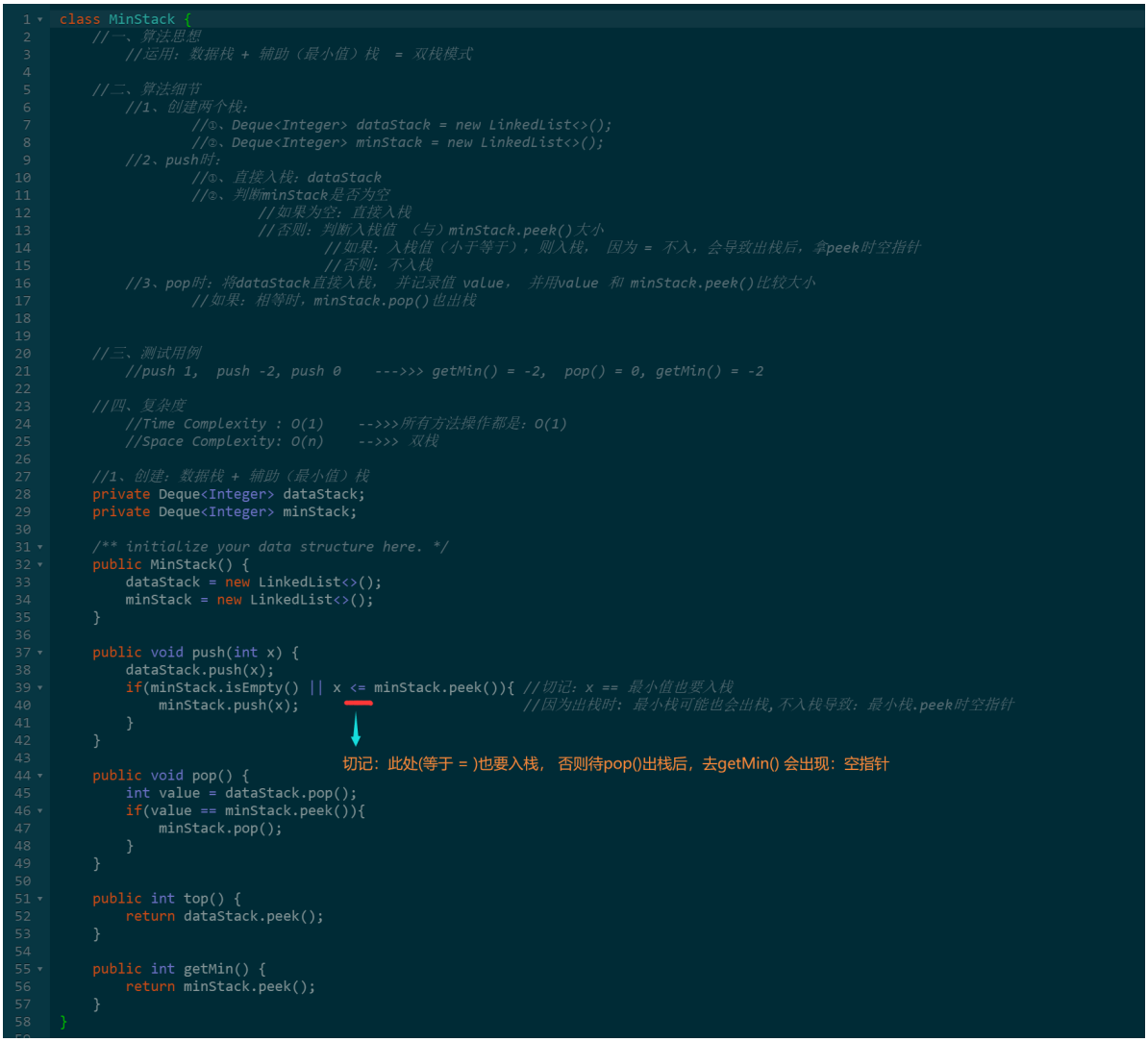
```
47          stack.pop();
48      }
49
50      public int top() {
51          return stack.peek().value;
52      }
53
54      public int getMin() {
55          return stack.peek().minValue;
56      }
57
58      private class Node{
59          private int value;          //存放当前：入栈值x
60          private 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
    such:
71   * MinStack obj = new MinStack();
72   * obj.push(x);
73   * obj.pop();
74   * int param_3 = obj.top();
75   * int param_4 = obj.getMin();
76   */
```

③、复杂度



二、数据栈 + 辅助（最小值）栈

①、代码图



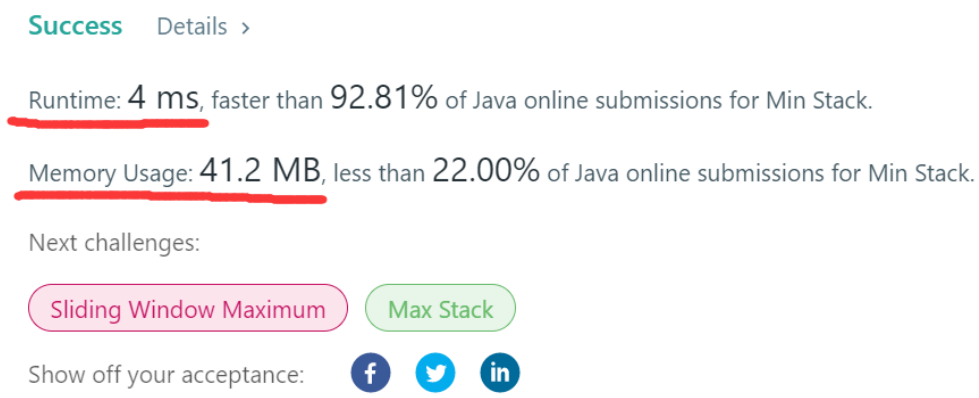
②、源代码

```
class MinStack {
    //一、算法思想
        //运用：数据栈 + 辅助（最小值）栈  = 双栈模式

    //二、算法细节
        //1、创建两个栈：
                //①、Deque<Integer> dataStack = new LinkedList<>();
                //②、Deque<Integer> minStack = new LinkedList<>();
        //2、push时：
                //①、直接入栈：dataStack
                //②、判断minStack是否为空
                        //如果为空：直接入栈
```

```
                            //否则：判断入栈值 （与） minStack.peek()大小
                                    //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
                                    //否则：不入栈
        //3、pop时：将dataStack直接入栈， 并记录值 value， 并用value 和 minStack.peek()比较大小
                    //如果：相等时，minStack.pop()也出栈


    //三、测试用例
        //push 1,  push -2, push 0    --->>> getMin() = -2,  pop() = 0, getMin() = -2

    //四、复杂度
        //Time Complexity : O(1)    -->>>所有方法操作都是：O(1)
        //Space Complexity: O(n)    -->>> 双栈

    //1、创建：数据栈 + 辅助（最小值）栈
    private Deque<Integer> dataStack;
    private Deque<Integer> minStack;

    /** initialize your data structure here. */
    public MinStack() {
        dataStack = new LinkedList<>();
        minStack = new LinkedList<>();
    }

    public void push(int x) {
        dataStack.push(x);
        if(minStack.isEmpty() || x <= minStack.peek()){ //切记：x == 最小值也要入栈
            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
        }
    }

    public void pop() {
        int value = dataStack.pop();
        if(value == minStack.peek()){
            minStack.pop();
        }
```

```
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



三、自定义：链栈

①、代码图

```java
class MinStack {
    //一、算法思路
        //采用：自己创建（一个链栈） + 头插法

    //二、算法细节
        //1、创建：链结点LinkStack（数据结构）
            //int value：存放当前入栈值
            //int minValue：存放栈最小值
            //StackNode next:  存放下一个结点
        //2、push时：若链栈 LinkStack == null   --->>> 直接入栈，  最小值为x
                //否则：用Math.min(x, LinkStack.minValue)找出最小值，  然后入栈
            //入栈：采用头插法，  通过head = LinkStack的构造方法，巧妙插入
        //3、pop时：直接head = head.next
        //4、top时：直接head.value
        //5、getMin时：直接head.getMin

    //三、测试用例
        //push 1, push -2, push 0   --->>> getMin() = -2, pop() = 0, getMin() = -2

    //四、复杂度
        //Time Complexity:  O(1)
        //Space Complexity: O(n)


    //1、创建一个：链栈的头指针
    private LinkedNode head;

    /** initialize your data structure here. */
    public MinStack() {

    }

    public void push(int x) {
        if(head == null){
            head = new LinkedNode(x, x, head);
        }else{
            head = new LinkedNode(x, Math.min(x, head.minValue), head);
        }
    }
                                                核心在于：传入当前链表，  通过构造方法，进行（头插法）操作
    public void pop() {
        head = head.next;
    }

    public int top() {
        return head.value;
    }

    public int getMin() {
        return head.minValue;
    }

    private class LinkedNode{
        private int value;       //存放当前入栈值
        private int minValue;    //存放栈最小值
        private LinkedNode next;  //存放下一个结点

        LinkedNode(int value, int minValue, LinkedNode head){
            this.value = value;
            this.minValue = minValue;
            this.next = head;     精髓
        }
    }
}
```

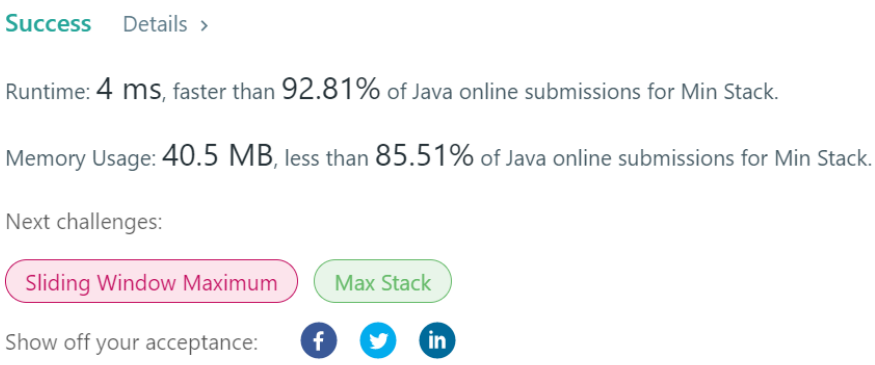
②、源代码

```java
class MinStack {
    //一、算法思路
        //采用：自己创建（一个链栈） + 头插法

    //二、算法细节
        //1、创建：链结点LinkStack（数据结构）
            //int value：存放当前入栈值
            //int minValue：存放栈最小值
            //StackNode next:  存放下一个结点
        //2、push时：若链栈 LinkStack == null   --->>> 直接入栈，  最小值为x
                //否则：用Math.min(x, LinkStack.minValue)找出最小值，  然后入栈
```

```
            //入栈：采用头插法， 通过head = LinkStack的构造方法，巧妙插入
        //3、pop时：直接head = head.next
        //4、top时：直接head.value
        //5、getMin时：直接head.getMin

    //三、测试用例
        //push 1, push -2, push 0   --->>> getMin() = -2, pop() = 0, getMin() = -2

    //四、复杂度
        //Time Complexity:  O(1)
        //Space Complexity: O(n)


    //1、创建一个：链栈的头指针
    private LinkedNode head;

    /** initialize your data structure here. */
    public MinStack() {

    }

    public void push(int x) {
        if(head == null){
            head = new LinkedNode(x, x, head);
        }else{
            head = new LinkedNode(x, Math.min(x, head.minValue), head);
        }
    }

    public void pop() {
        head = head.next;
    }

    public int top() {
        return head.value;
    }

    public int getMin() {
        return head.minValue;
    }
```

```
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;
62          }
63      }
64  }
65
```

③、复杂度



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;
```

```java
class Solution {
    //一、算法思想
        //采用：遍历字符串 + 栈

    //二、算法细节
        //1、创建一个栈：Deque<Character> stack = new LinkedList<>()
        //2、遍历字符串，通过if、else if 进行选择
            //①、遇到( , [,  {  左括号则入栈它们的（右括号）
            //②、遇到), ], } 时
                //判断：栈是否为空，  如果为空 return false
                //判断：右括号是否 == 出栈元素，  如果不等于 return false
        //3、return stack.isEmpty() 作为是否是：有效括号的结果


    //三、测试用例
        //1、([)]   --->>>  return false
        //2、{[]}   --->>>  return true
        //3、[]     --->>>  return true

    //四、复杂度
        //1、Time  Complexity：O(n)          -->> 因为要遍历length长度
        //2、Space Complexity：O(n)          -->> 用到了：栈

    //五、优缺点
        //1、advantage:   通俗易懂，  代码量少
        //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

    public boolean isValid(String s) {
        //1、创建一个栈
        Deque<Character> stack = new LinkedList<>();

        //2、获取字串长度
        int length = s.length();        //因为题目给定：1 <= s.length <= 10^4 所以不用前置判断了

        //3、遍历字符串
        for(int i = 0; i < length; i++){
            Character c = s.charAt(i);

            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();
    }
}
```

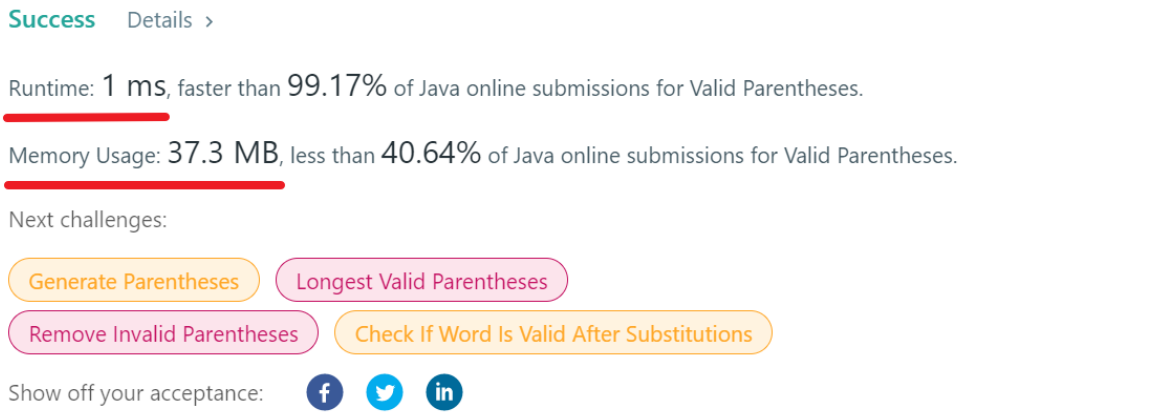
精髓在于：
1、入栈相反字符
2、匹配栈顶时为空 || 出栈元素 != 当前字符 -->> false

②、源代码

```java
class Solution {
    //一、算法思想
        //采用：遍历字符串 + 栈

    //二、算法细节
        //1、创建一个栈：Deque<Character> stack = new LinkedList<>()
        //2、遍历字符串，通过if、else if 进行选择
            //①、遇到( , [,  {  左括号则入栈它们的（右括号）
            //②、遇到), ], } 时
                //判断：栈是否为空，  如果为空 return false
                //判断：右括号是否 == 出栈元素，  如果不等于 return false
        //3、return stack.isEmpty() 作为是否是：有效括号的结果
```

```java


    //三、测试用例
        //1、([)]   --->>>  return false
        //2、{[]}   --->>>  return true
        //3、[]     --->>>  return true

    //四、复杂度
        //1、Time  Complexity: O(n)          -->> 因为要遍历length长度
        //2、Space Complexity: O(n)          -->> 用到了：栈

    //五、优缺点
        //1、advantage:   通俗易懂，  代码量少
        //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

    public boolean isValid(String s) {
        //1、创建一个栈
        Deque<Character> stack = new LinkedList<>();

        //2、获取字串长度
        int length = s.length();        //因为题目给定：1 <= s.length <= 10^4 所以不用前置判断了

        //3、遍历字符串
        for(int i = 0; i < length; i++){
            Character c = s.charAt(i);

            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();
    }
}
```

③、复杂度



二、遍历字符串 + 栈 + Map

①、代码图

```java
class Solution {
    //一、算法思想
        //采用：遍历字符串 + 栈 + Map

    //二、算法细节
        //1、创建一个栈：Deque<Character> stack = new LinkedList<>()
        //2、创建一个：Map<Character, Character> = new HashMap<>()
        //3、将()、[]、{}，放入map中，  左括号为key，右括号为value
        //4、进行遍历：当前字符到（Map查找）
              //若找到：则将Map的value入栈
              //若找不到：判断当前栈是否为空 || 当前遍历字符 != 出栈元素
                   //栈若为空：说明（右括号）找不到匹配的（左括号） -->> 直接return false
                   //当前遍历字符 != 出栈元素：  说明匹配不上 -->> 直接return false
        //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）

    //三、测试用例
        //1、([)]   --->>>  return false
        //2、{[]}   --->>>  return true
        //3、[]     --->>>  return true

    //四、复杂度
        //1、Time  Complexity：O(n)          -->> 因为要遍历length长度
        //2、Space Complexity：O(n)          -->> 最坏超过 n ，因为用到了Map

    //五、优缺点
        //1、advantage:   添加了前置判断、拓展性强（增加匹配元素时）易于添加
        //2、shortcoming: 耗费了空间

    public boolean isValid(String s) {
        //1、获取字串长度
        int length = s.length();        //因为题目给定：1 <= s.length <= 10^4 所以对此做:判断

        if(length % 2 == 1){   此处用得巧
            return false;                //奇数个括号，  不可能匹配
        }

        //2、创建一个栈
        Deque<Character> stack = new LinkedList<>();


        //3、创建一个Map，并将成对括号：放入
        Map<Character, Character> map = new HashMap<>();  //直接用map.put()而不用：匿名内部类，统一初始化
        map.put('(', ')');                                //是因为：map.put()的效率更高
        map.put('[', ']');
        map.put('{', '}');



        //4、遍历字符串
        for(int i = 0; i < length; i++){
            Character c = s.charAt(i);
            if(map.containsKey(c))       containsKey()中是有s的，注意了
                stack.push(map.get(c));
            else if(stack.isEmpty() || c != stack.pop())
                return false;
        }

        return stack.isEmpty();
    }
}
```

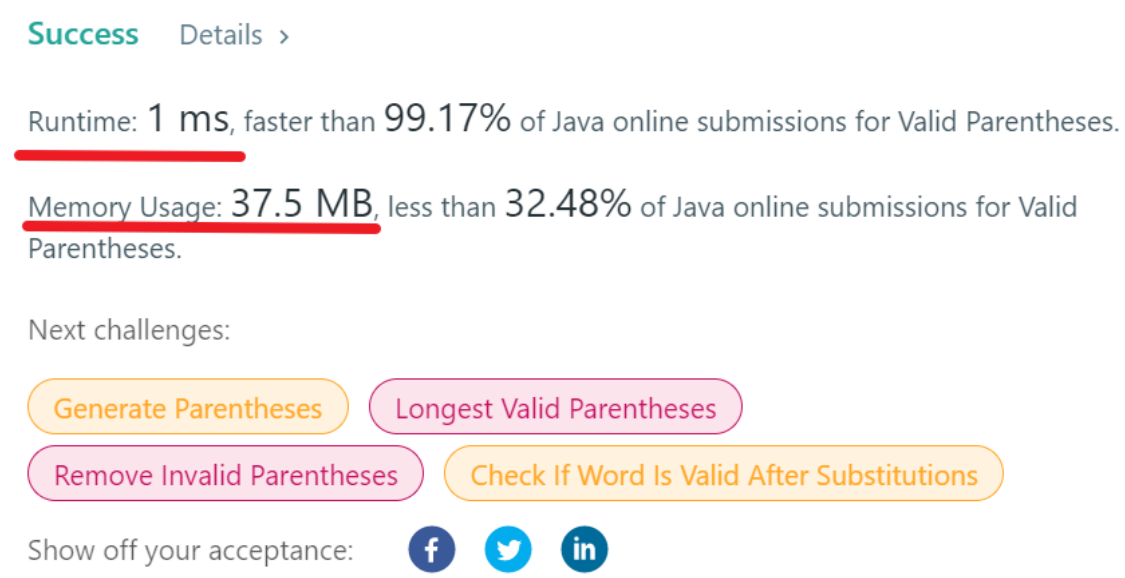
②、源代码

```java
class Solution {
    //一、算法思想
        //采用：遍历字符串 + 栈 + Map

    //二、算法细节
        //1、创建一个栈：Deque<Character> stack = new LinkedList<>()
        //2、创建一个：Map<Character, Character> = new HashMap<>()
        //3、将()、[]、{}，放入map中，  左括号为key，右括号为value
```

```
        //4、进行遍历：当前字符到（Map查找）
                //若找到：则将Map的value入栈
                //若找不到：判断当前栈是否为空 || 当前遍历字符 != 出栈元素
                        //栈若为空：说明（右括号）找不到匹配的（左括号） -->> 直接return false
                        //当前遍历字符 != 出栈元素：  说明匹配不上 -->> 直接return false
        //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）

    //三、测试用例
        //1、([)]   --->>>  return false
        //2、{[]}   --->>>  return true
        //3、[]     --->>>  return true

    //四、复杂度
        //1、Time  Complexity：O(n)          -->> 因为要遍历length长度
        //2、Space Complexity：O(n)          -->> 最坏超过 n , 因为用到了Map

    //五、优缺点
        //1、advantage:   添加了前置判断、拓展性强（增加匹配元素时）易于添加
        //2、shortcoming: 耗费了空间

    public boolean isValid(String s) {
        //1、获取字串长度
        int length = s.length();        //因为题目给定：1 <= s.length <= 10^4 所以对此做:判断

        if(length % 2 == 1){
            return false;               //奇数个括号，  不可能匹配
        }

        //2、创建一个栈
        Deque<Character> stack = new LinkedList<>();


        //3、创建一个Map，并将成对括号：放入
```

```
        Map<Character, Character> map = new HashMap<>();
//直接用map.put()而不用：匿名内部类，统一初始化
        map.put('(', ')');
//是因为：map.put()的效率更高
        map.put('[', ']');
        map.put('{', '}');



        //4、遍历字符串
        for(int i = 0; i < length; i++){
            Character c = s.charAt(i);
            if(map.containsKey(c))
                stack.push(map.get(c));
            else if(stack.isEmpty() || c != stack.pop())
                return false;
        }

        return stack.isEmpty();
    }
}
```

③、复杂度



三、哨兵 + stack

①、代码图

②、源代码

```java
class Solution {
    //1、创建Map
    private static final Map<Character, Character> map = new HashMap<>(){
        {
            put('(', ')');
            put('[', ']');
            put('{', '}');
            put('?', '?');          //作为哨兵
        }
    };

    public boolean isValid(String s) {
        //1、前置欲判断
        if(s.length() % 2 == 1 || !map.containsKey(s.charAt(0)))
            return false;

        //2、创建一个栈
        Deque<Character> stack = new LinkedList<>();
        stack.push('?');            // (?, ?)作为哨兵

        //3、遍历：判断
        for(char c : s.toCharArray()){
            if(map.containsKey(c)){
                stack.push(c);
            }else if(map.get(stack.pop()) != c){
                return false;
            }
        }

        return stack.size() == 1;
    }
}
```

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: