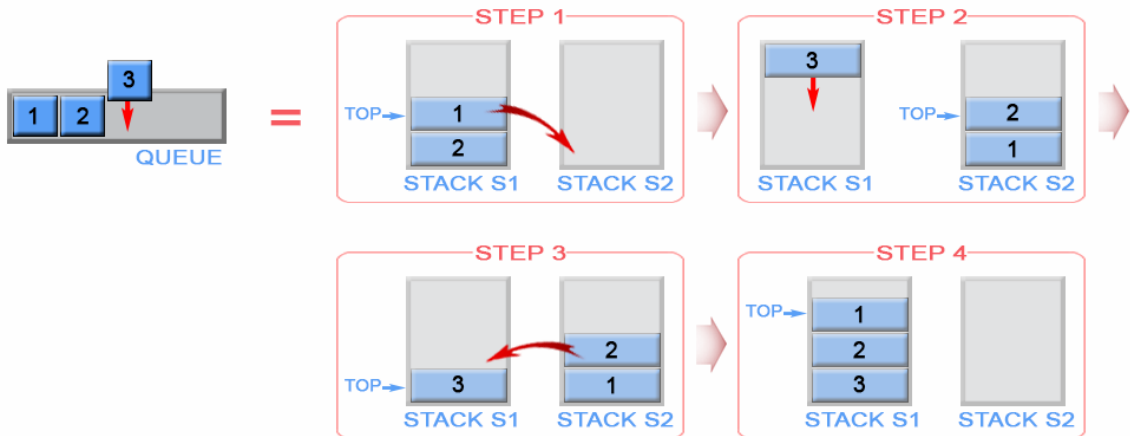


# Leetcode 栈&队列 专题

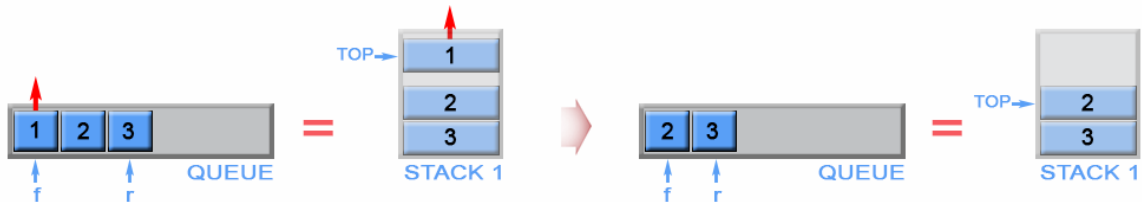
## 232.用栈实现队列

栈的顺序为后进先出，而队列的顺序为先进先出。使用两个栈实现队列，元素入栈时，利用两个栈把元素放入栈底，这样就实现了，先进先出。[此方法是入栈是利用两个栈，出栈时直接pop；也可入栈时直接push,出栈时利用两个栈，把最底下的元素取出来]



Pushing element "3" to the queue

对应的出队直接pop即可



Popping element "1" from the queue

```
1 //思路:队列是先进先出的，而栈是先进后出的，所以利用两个栈来实现队列
2 //入队列时需要把元素放到栈底(这样才能后出)，于是把s1中元素先反转放在s2中，再入栈，然后把
3 //s2中元素再反转放回来
4 //出队列时直接pop即可
4 #define MAXSIZE 100
5 struct Stack{ //用数组来实现栈
6     int data[MAXSIZE];
7     int top;
8 };
9 typedef struct { //队列定义为双栈
10     struct Stack s1; //s1为主栈
11     struct Stack s2; //s2用来反转
12 } MyQueue;
13
14 /** Initialize your data structure here. */
15 MyQueue* myQueueCreate() {
```

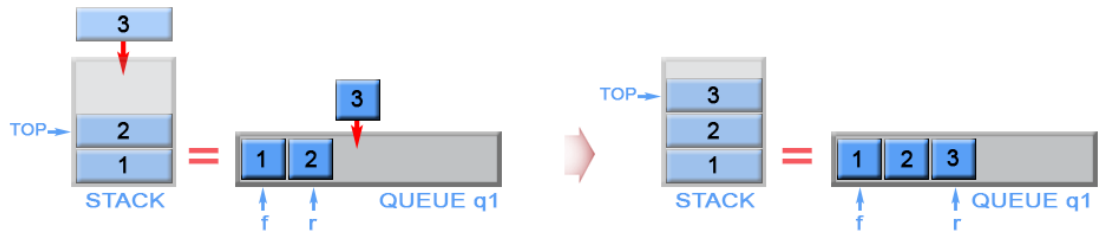
```

16     MyQueue* TempQueue= (MyQueue*)malloc(sizeof(MyQueue));
17     TempQueue->S1.top=-1;           //不能用TempQueue->S1->top=-1; 会报错
18     TempQueue->S2.top=-1;
19     return TempQueue;
20 }
21
22 /** Push element x to the back of queue. */
23 void myQueuePush(MyQueue* obj, int x) {
24     if(obj->S1.top<MAXSIZE) //判断队列是否满了
25     {
26         while(obj->S1.top!=-1) //把S1中元素反转放入S2中
27         {
28             obj->S2.data[++(obj->S2.top)]=obj->S1.data[(obj->S1.top)--];
29         }
30         obj->S1.data[++(obj->S1.top)]=x; //把新元素压入S1中
31         while(obj->S2.top!=-1) //再把S2中元素反转压入S1中
32         {
33             obj->S1.data[++(obj->S1.top)]=obj->S2.data[(obj->S2.top)--];
34         }
35     }
36 }
37
38 /** Removes the element from in front of queue and returns that element. */
39 int myQueuePop(MyQueue* obj) {
40     if(obj->S1.top!=-1)
41         return obj->S1.data[(obj->S1.top)--];
42     return NULL;
43 }
44
45 /** Get the front element. */
46 int myQueuePeek(MyQueue* obj) {
47     if(obj->S1.top!=-1)
48         return obj->S1.data[obj->S1.top];
49     return NULL;
50 }
51
52 /** Returns whether the queue is empty. */
53 bool myQueueEmpty(MyQueue* obj) {
54     if(obj->S1.top===-1) return true;
55     return false;
56 }
57
58 void myQueueFree(MyQueue* obj) {
59     free(obj);
60 }

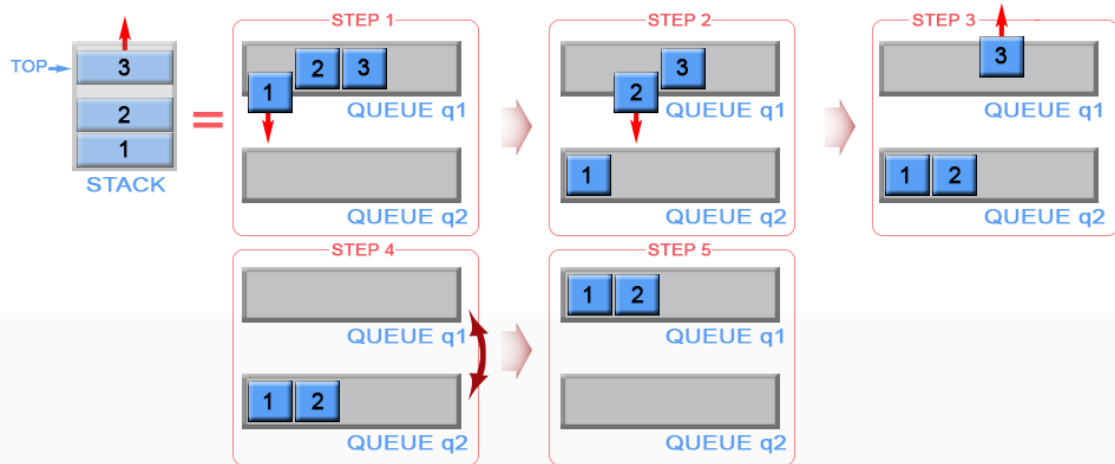
```

## 225.用队列实现栈

与用栈实现队列相似，用了两个栈，这里用到的方法，在入栈的时候直接加入，出栈时，则将主队列q1中除了最后一个值外的其他值，一个个放入队列q2,然后把q1最后一个pop出去,再把q2中元素移回q1



Pushing element "3" to the stack



Popping element "3" from the stack

自己定义的这个队列，由一个数组(malloc出来的),和front,rear,size组成，front指第一个元素的位置，rear指最后一个元素的后一个位置。且这个队列为循环队列。

```

1  #define LEN 20
2  //先定义自己的队列，以及队列的基础函数
3  struct Queue{                      //队列的结构体 其中data指向"头地址"，访问后面的元素用
    下标即可(类似数组)
4      int *data;
5      int front;
6      int rear;                      //rear指的是队列最后一个元素的后一个位置
7      int size;
8  };
9  struct Queue *initQueue(int length) //初始化队列
10 {
11     struct Queue *obj=(struct Queue*)malloc(sizeof(struct Queue));
12     obj->data=(int*)malloc(length*sizeof(int));
13     obj->front=0;
14     obj->rear=0;
15     obj->size=length;
16     return obj;
17 }
18 void enqueue(struct Queue *obj,int x) //入队列时， 先将rear后移一位，然后把值填充
    进去
19 {
20     if(obj->front != (obj->rear +1)%obj->size) //确定队列未满
21     {
22         obj->data[obj->rear]=x;
23         obj->rear=(obj->rear +1)%obj->size; //循环队列 若到达队列最后则rear返回到
        第0个元素
24     }

```

```

25 }
26 int dequeue(struct Queue* obj)
27 { //此题不会出现队列空时dequeue的情况
28     int output=obj->data[obj->front];
29     obj->front=(obj->front+1)%obj->size;
30     return output;
31 }
32 int IsEmpty(struct Queue*obj)
33 {
34     return (obj->front==obj->rear);
35 }
36 //用双队列实现栈
37 typedef struct {
38     struct Queue *q1,*q2;
39 } MyStack;
40
41 /** Initialize your data structure here. */
42 MyStack* myStackCreate() {
43     MyStack *obj=(MyStack*)malloc(sizeof(MyStack));
44     obj->q1=initQueue(LEN);
45     obj->q2=initQueue(LEN);
46     return obj;
47 }
48
49 /** Push element x onto stack. */
50 //入栈时，直接把元素放入q1队列的后面，再把top+1即可
51 void myStackPush(MyStack* obj, int x) {
52     enqueue(obj->q1,x);
53 }
54
55 /** Removes the element on top of the stack and returns that element. */
56 //出队时，把q1除最后一个元素外，移到q2中，再让q1剩下的那个元素出队，再把q2中元素移回q1
57 int myStackPop(MyStack* obj) {
58     int output;
59     while(obj->q1->front +1 != obj->q1->rear) //此题中不会出现循环的情况
60         enqueue(obj->q2,dequeue(obj->q1));
61     output=dequeue(obj->q1);
62     while(obj->q2->front != obj->q2->rear)
63         enqueue(obj->q1,dequeue(obj->q2));
64     return output;
65 }
66
67 /** Get the top element. */
68 int myStackTop(MyStack* obj) {
69     return obj->q1->data[obj->q1->rear-1];
70 }
71
72 /** Returns whether the stack is empty. */
73 bool myStackEmpty(MyStack* obj) {
74     return IsEmpty(obj->q1);
75 }
76
77 void myStackFree(MyStack* obj) { //一层层free
78     free(obj->q1->data);
79     obj->q1->data=NULL;
80     free(obj->q1);
81     obj->q1=NULL;
82     free(obj->q2->data);

```

```

83     obj->q2->data=NULL;
84     free(obj->q2);
85     obj->q2=NULL;
86     free(obj);
87     obj=NULL;
88 }

```

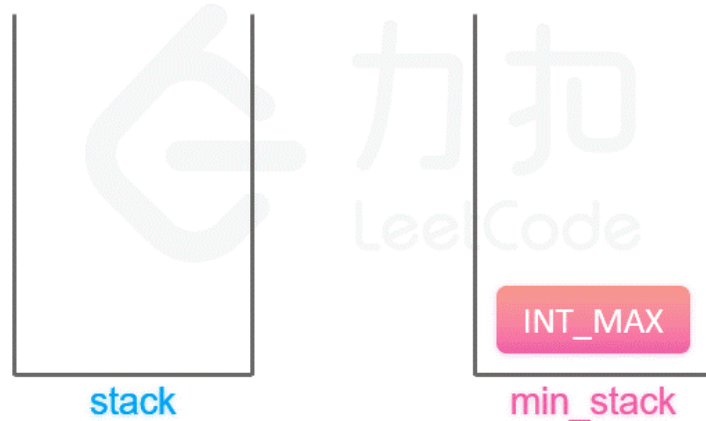
## 155.最小栈

题目描述:设计一个支持push,pop,top操作，并且能在常数时间内检索到最小元素的栈。

### 方法:辅助栈

基于栈先进后出的本质，上方的元素未出栈时，下面的元素是动不了的。(即如果在元素a入栈时，栈里有其他元素b,c,d，那么只要a在栈中，b,c,d就一定也在)。所以每次元素x入栈的时候，就把当前栈中的最小值存储在辅助栈中(进入辅助栈的最小值即 x和当前辅助栈顶的值中的最小值)。这样要找当前栈中最小值直接top辅助栈即可。pop时两个栈一起pop。

或者可以只用一个栈，就是在栈中定义两个数组，一个存储栈中的值，一个存储最小值。两个方法本质是一样的。



```

1  //思路:基于栈后进先出的特点，利用辅助栈来记录每次push后栈中的最小值
2  #define MAX 10000
3  typedef struct{
4      int data[MAX];
5      int top;
6  } MyStack;
7  typedef struct {
8      MyStack stackAll;           //储存栈的全部值
9      MyStack stackMin;          //储存主栈对应的最小值
10 } MinStack;
11
12 /** initialize your data structure here. */
13
14 MinStack* minStackCreate() {
15     MinStack *obj=(MinStack*)malloc(sizeof(MinStack));
16     obj->stackAll.top=-1;

```

```

17     obj->stackMin.top--1;
18     return obj;
19 }
20
21 void minStackPush(MinStack* obj, int x) {    //push是主栈直接进，辅助栈判断下输入
值和栈顶值的大小，进小的那个
22     if(obj->stackAll.top==MAX) return;    //栈满了
23     int min;
24     obj->stackAll.data[++obj->stackAll.top]=x;
25     if(obj->stackMin.top==1)
26         obj->stackMin.data[++obj->stackMin.top]=x;
27     else
28     {
29         min=(x<obj->stackMin.data[obj->stackMin.top])? x:obj-
>stackMin.data[obj->stackMin.top];
30         obj->stackMin.data[++obj->stackMin.top]=min;
31     }
32 }
33
34 void minStackPop(MinStack* obj) {
35     if(obj->stackAll.top==1) return;
36     obj->stackAll.top--;
37     obj->stackMin.top--;
38 }
39
40 int minStackTop(MinStack* obj) {
41     return obj->stackAll.data[obj->stackAll.top];
42 }
43
44 int minStackGetMin(MinStack* obj) {
45     return obj->stackMin.data[obj->stackMin.top];
46 }
47
48 void minStackFree(MinStack* obj) {
49     free(obj);
50 }

```

## 20.有效的括号

```

1 //思路:遇到左括号时存入栈中，遇到右括号时就和栈顶的符号对比，如果匹配则pop，继续遍历；如果
不匹配，则最终结果无效
2 bool isValid(char * s){
3     if(s==NULL||s[0]=='\0') return true;    //空字符串有效
4     char* stack=(char *)malloc(strlen(s)+1);    //用字符数组来实现栈（因为一个char
的大小是1，所以可以不乘上去）
5     int top=-1;
6     for(int i=0;s[i];i++)
7     {
8         if(s[i]=='(' ||s[i]=='[' ||s[i]=='{')    //遇到左括号
9             stack[++top]=s[i];
10        else    //遇到右括号
11        {
12            if(top==1) return false;
13            else if(s[i]==')'&&stack[top]!='(') return false;


```

```

14         else if(s[i]=='&&stack[top]!='[') return false;
15         else if(s[i]=='&&stack[top]!='{') return false;
16         else top--;
17     }
18 }
19 if(top!=-1) return false;    //运行结束时，若括号匹配，则应为空栈
20 return true;
21 }

```

## 739.每日温度

根据每日  气温 列表，请重新生成一个列表，对应位置的输出是需要再等待多久温度才会升高超过该日的天数。如果之后都不会升高，请在该位置用 0 来代替。

### 方法:正序法

可以运用一个堆栈 stack 来快速知道需要经过多少天就能等到温度升高。  
从头到尾扫描一遍给定的数组 T，如果当天的温度比堆栈 stack 顶端所记录的那天温度还要高，那么就能得到结果。

对第一个温度 23 度，堆栈为空，它的下标压入堆栈；

下一个温度 24 度，高于 23 度高，因此 23 度温度升高只需 1 天时间，把 23 度下标从堆栈里弹出，把 24 度下标压入；

同样，从 24 度只需要 1 天时间升高到 25 度；

21 度低于 25 度，直接把 21 度下标压入堆栈；

19 度低于 21 度，压入堆栈；

22 度高于 19 度，从 19 度升温只需 1 天，从 21 度升温需要 2 天；

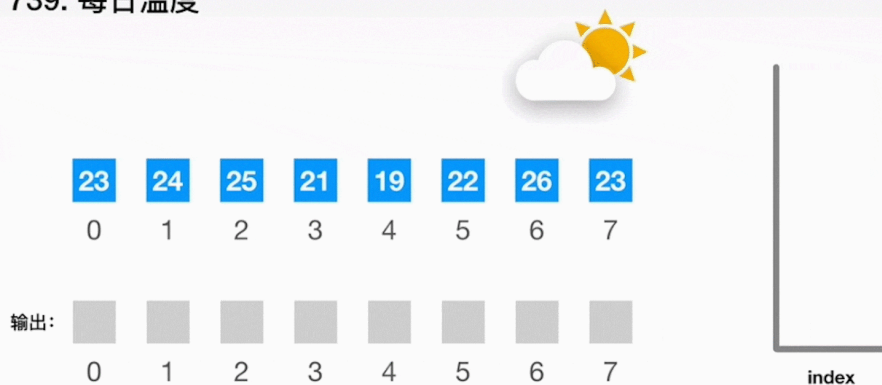
由于堆栈里保存的是下标，能很快计算天数；

22 度低于 25 度，意味着尚未找到 25 度之后的升温，直接把 22 度下标压入堆栈顶端；

后面的温度与此同理。

该方法只需要对数组进行一次遍历，每个元素最多被压入和弹出堆栈一次，算法复杂度是  $O(n)$ 。

### 739. 每日温度



每一个下标都会被压入栈一次，在遇到一个比它对应温度大的*i*时pop，并同时计算所隔天数(下标相减)

```

1 //解题思路:遍历温度数组，把每个温度的下标压入栈中，若遇到温度大于栈顶元素对应的温度时，就把
  下标的插值记录进输出的数组中
2 int* dailyTemperatures(int* T, int TSize, int* returnSize){
3     int *output=(int*)malloc(TSize*sizeof(int));

```

```

4     memset(output,0,TSize*sizeof(int));
5     int stack[TSize];
6     int top=-1;
7     for(int i=0;i<TSize;i++)
8     {
9         while(top>-1 && T[i]>T[stack[top]] )    //如果非空栈 且 当前i对应的温
          度比栈顶元素对应的温度大 就得出结果并且pop (注意要用while,因为栈中下面可能还有元素)
10        {
11            output[stack[top]]=i-stack[top] ;
12            top--;
13        }
14        stack[++top]=i;
15    }
16    *returnSize=TSize;
17    return output;
18 }

```

## 503.下一个更大元素

### 方法:单调栈

与739的方法基本相同，只不过题目要输出的是下一个最大的元素而不是下标差值，而且由于是循环数组，所以循环了两次。

依旧是把每一个元素压入栈中，遇到比栈顶大的元素就pop并且在输出数组中记录下来。(其实第二次循环可以不入栈，直接比较也可)

来自leetcode官方题解:我们首先把第一个元素  $A[1]$  放入栈，随后对于第二个元素  $A[2]$ ，如果  $A[2] > A[1]$ ，那么我们就找到了  $A[1]$  的下一个更大元素  $A[2]$ ，此时就可以把  $A[1]$  出栈并把  $A[2]$  入栈；如果  $A[2] \leq A[1]$ ，我们就仅把  $A[2]$  入栈。对于第三个元素  $A[3]$ ，此时栈中有若干个元素，那么所有比  $A[3]$  小的元素都找到了下一个更大元素（即  $A[3]$ ），因此可以出栈，在这之后，我们将  $A[3]$  入栈，以此类推。

可以发现，我们维护了一个单调栈，栈中的元素从栈顶到栈底是单调不降的。当我们遇到一个新的元素  $A[i]$  时，我们判断栈顶元素是否小于  $A[i]$ ，如果是，那么栈顶元素的下一个更大元素即为  $A[i]$ ，我们将栈顶元素出栈。重复这一操作，直到栈为空或者栈顶元素大于  $A[i]$ 。此时我们将  $A[i]$  入栈，保持栈的单调性，并对接下来的  $A[i + 1]$ ,  $A[i + 2]$  ... 执行同样的操作。

```

1  int* nextGreaterElements(int* nums, int numsSize, int* returnSize){
2      if(nums==NULL || numsSize<1)
3      {
4          *returnSize=0;
5          return NULL;
6      }
7      int* output=(int*)malloc(numsSize*sizeof(int));
8      memset(output,-1,numsSize*sizeof(int));
9      int circulate=0;    //控制循环次数
10     int stack[numsSize*2];    //记录下标，因为循环了两次所以上界应是numsSize*2
11     int top=-1;
12     for(int i=0;i<numsSize&&circulate<2;i++)    //因为是循环数组，所以遍历了两次
13     {
14         while(top>-1 && nums[i]>nums[stack[top]])
15         {
16             output[stack[top]]=nums[i];
17             top--;
18         }

```



```
19     stack[++top]=i;
20     if(i==numSize-1)
21     {
22         i=-1;          //因为结束后i++,所以这里i调为-1
23         circulate++;
24     }
25 }
26 *returnSize=numSize;
27 return output;
28 }
```