人工智能学院

Data Structures and Algorithms

Stacks and Queues (栈和队列) 3

Assessment

1. Definition
2. ADT
3. Implementation
4. Application

Definition

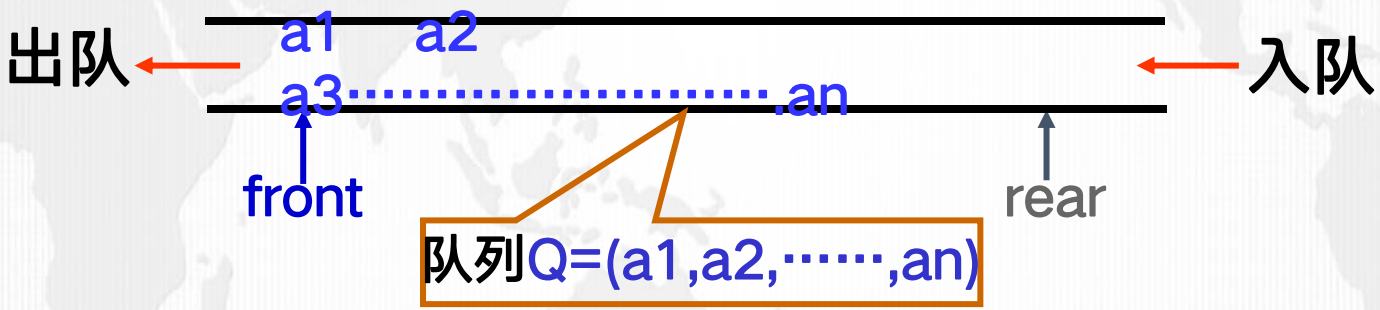
◆ 队列的定义及特点

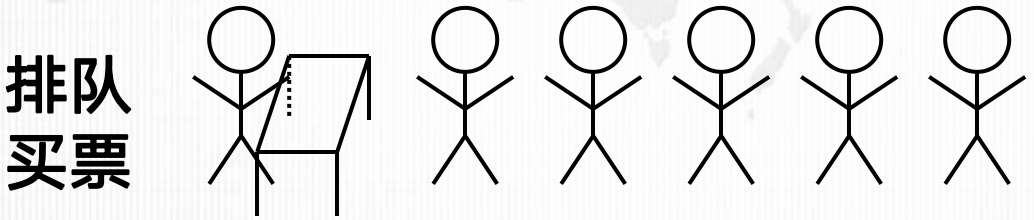
定义：队列是限定只能在表的一端进行插入，在表的另一端进行删除的线性表

队尾(rear)——允许插入的一端

队头(front)——允许删除的一端

队列特点：先进先出(FIFO)





抽象数据类型队列的定义

ADT Queue

{ 数据对象：$D=\{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$

数据关系：$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

(约定a_1端为队头，a_n端为队尾)

基本操作：

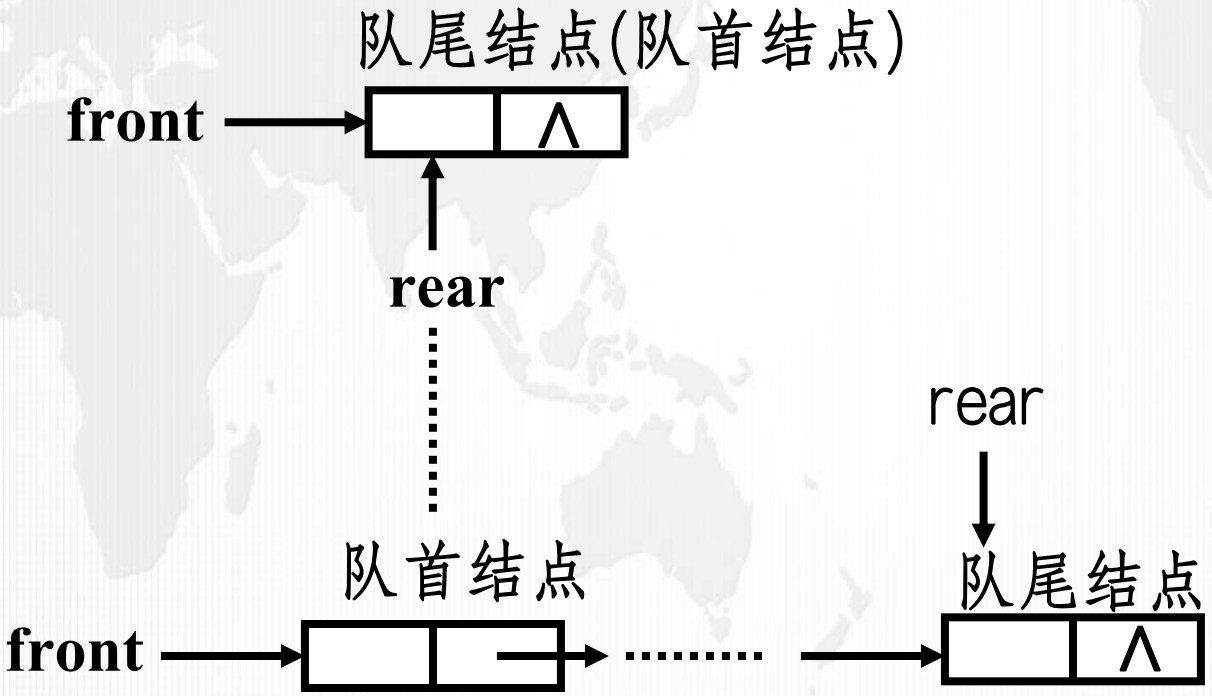
InitQueue(*Q);	构造一个空队列Q
DestroyQueue(*Q);	销毁队列Q
ClearQueue(*Q);	将队列Q清为空队列
QueueEmpty(Q);	判断队列Q是否为空
QueueLength(Q);	返回队列Q 中元素个数
GetHead(Q, *e);	取队列Q的队头元素
EnQueue(*Q, e);	元素e入队
DeQueue(*Q, *e);	出队

} ADT Queue

队列的存储结构——链式队列

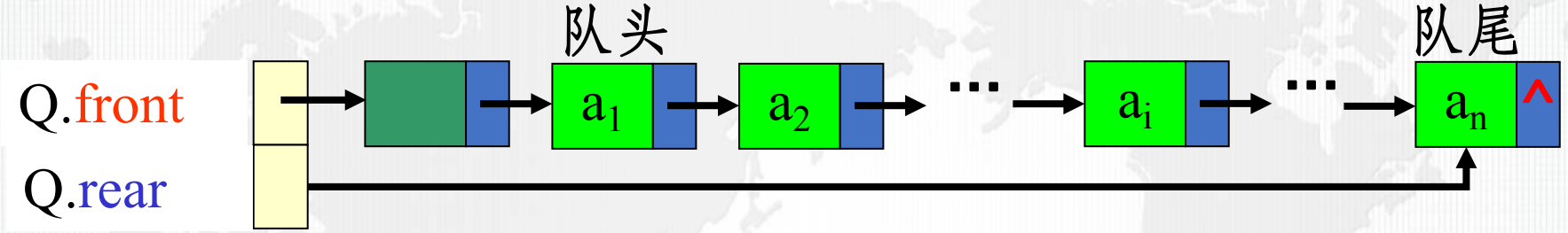
◆ 链式队列基本架构：

空队列：front = rear = NULL



队列的存储结构——链式队列

◆ 链式队列基本架构：

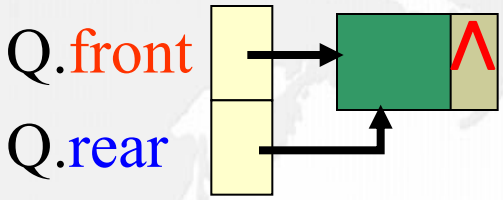



```
10  typedef struct QNode {
11      QElemType data;      // 数据域
12      struct QNode *next; // 指针域
13  } QNode, *QueuePtr;
14
15  typedef struct {
16      QueuePtr front;      // 队头指针
17      QueuePtr rear;       // 队尾指针
18  } LinkQueue;
19
```

队列的存储结构——链式队列

- 构造空链队列Q：

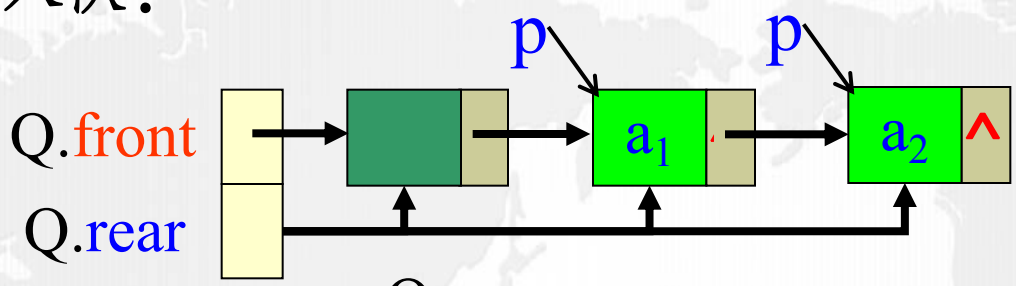
LinkQueue Q;




```
26  Status init_queue(LinkQueue *Q) {
27      Q->front = Q->rear = (QueuePtr) malloc( size: sizeof(QNode));
28      if (!Q->front) {
29          return OVERFLOW;
30      }
31      Q->front->next = NULL;
32      return OK;
33  }
```

队列的存储结构——链式队列

- 链队列入队：



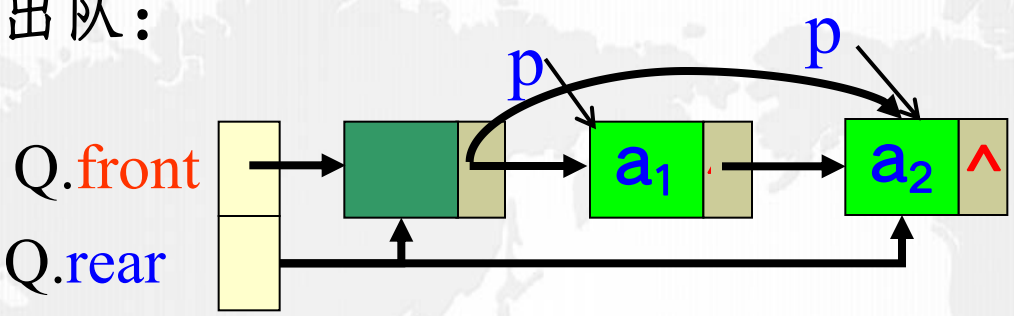
Q.rear=p;　　Q.rear->link=p;

```
35  Status enter_queue(LinkQueue *Q, QElemType  e) {
36      QuevePtr p = (QuevePtr) malloc( size: sizeof(QNode));
37      if (!p) {
38          return OVERFLOW;
39      }
40      p->data = e;
41      p->next = NULL;
42      Q->rear->next = p;
43      Q->rear = p;
44
45      return OK;
46  }
```

T(n)=O(1)

队列的存储结构——链式队列

◆ 链队列出队：




```
48  Status de_queue(LinkQueue *Q, QElemType *e) {
49      if (Q->front == Q->rear) {
50          return ERROR;
51      }
52
53      QueuePtr p = Q->front->next;
54      *e = p->data;
55      Q->front->next = p->next;
56      if (Q->rear == p) Q->rear = Q->front;
57      free(p);
58
59      return OK;
60  }
```

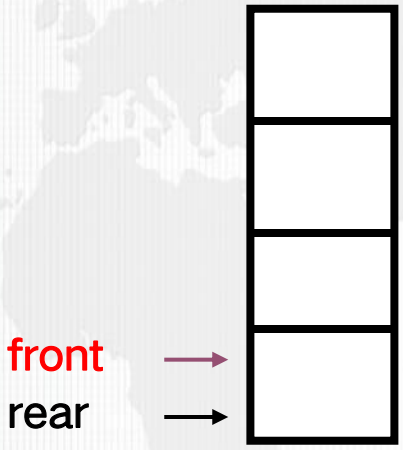
p=Q.front->link;

Q.front->link=p->link;

T(n)=O(1)

队列的存储结构——顺序队列

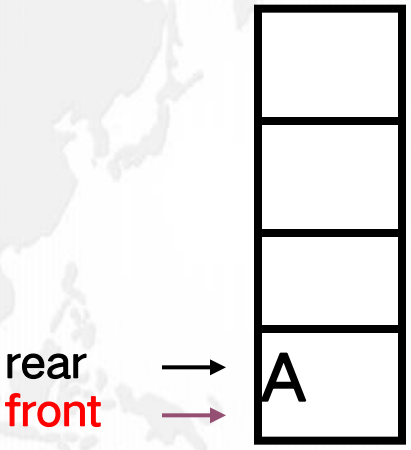
- 用数组queue[MaxSize]描述一个队列， location(i)=i-1;



队空：

front=0;

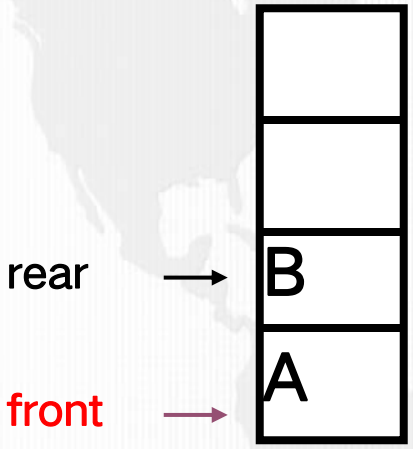
rear=-1;



插入A元素：

front=0;

rear=0;

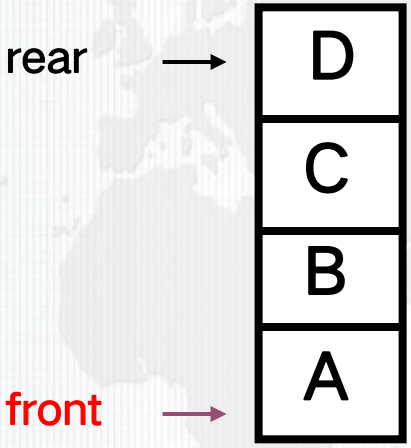


插入B元素：

front=0;

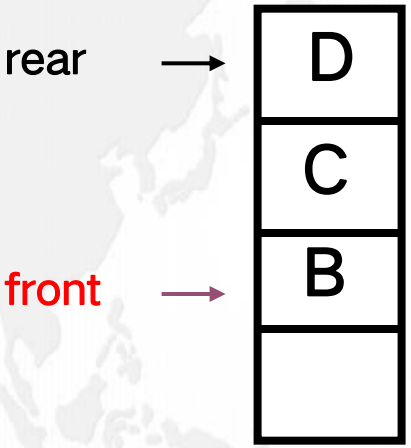
rear=1;

队列的存储结构——顺序队列



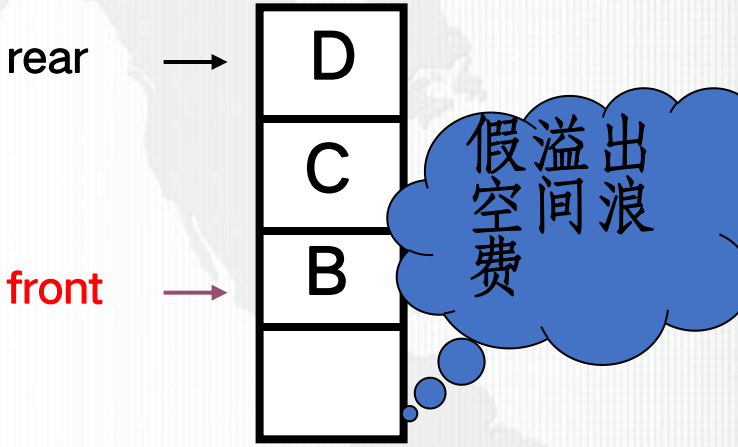
队列满：

rear=MaxSize-1



出队，

front=1;

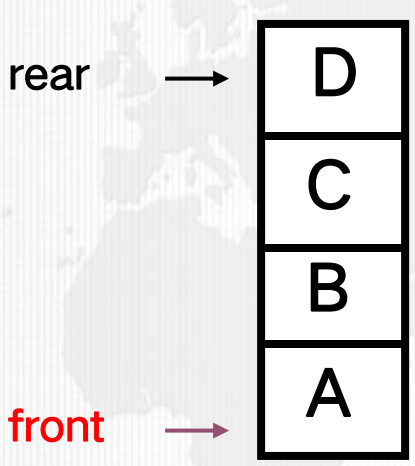


插入E：

rear=MaxSize-1;

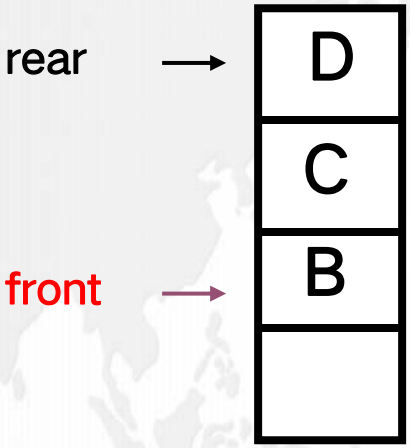
队列满，不能插入

队列的存储结构——顺序队列



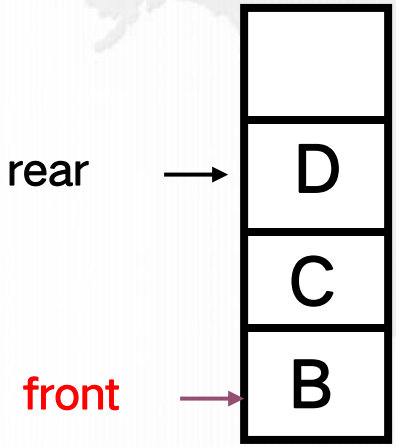
队列满：

rear=MaxSize-1



出队，

front=1;



出队后调整：

front=0;

rear=rear-1;

队列的存储结构——顺序队列

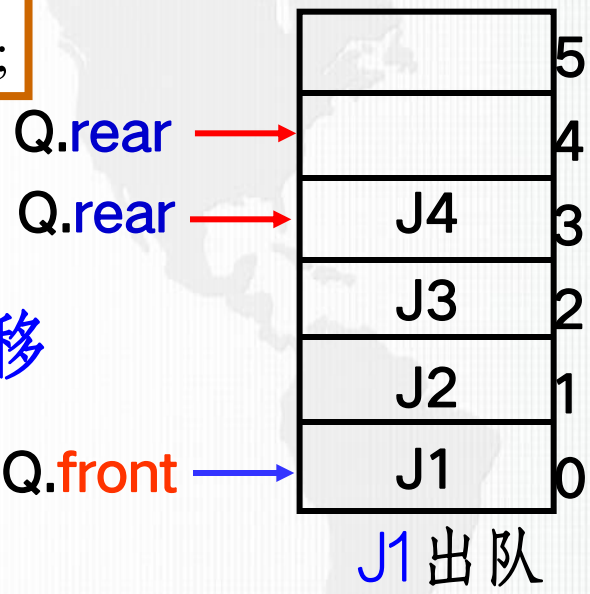
❖假溢出问题解决方案

真溢出条件：	假溢出条件：
Q.front==0; Q.rear==MAXQSIZE;	Q.front ≠ 0; Q.rear==MAXQSIZE;

方案一：

队首固定，每次出队，剩余元素下移

缺点：浪费时间



队列的存储结构——顺序队列

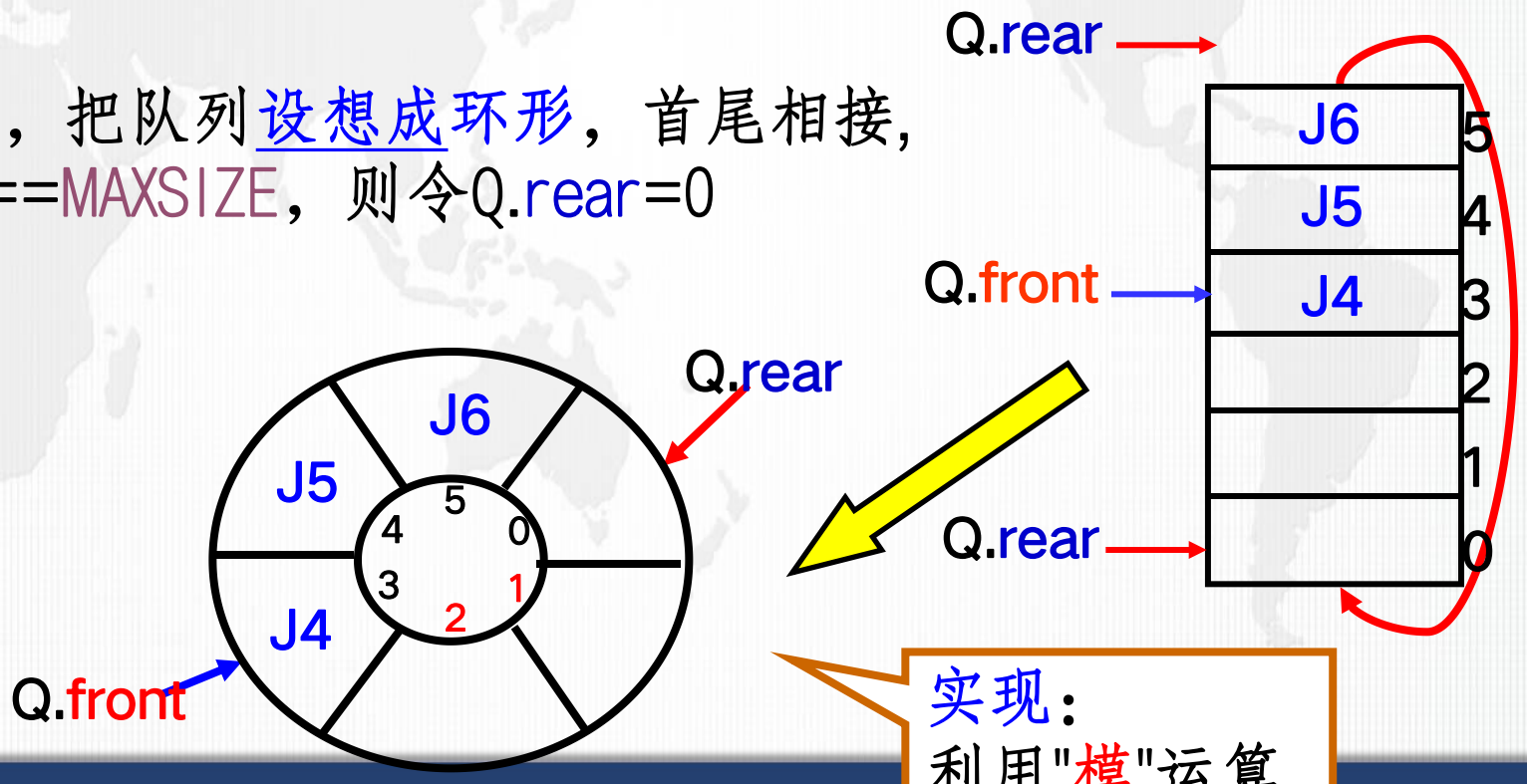
❖假溢出问题解决方案

真溢出条件：	假溢出条件：
Q.front==0;	Q.front ≠ 0;
Q.rear==MAXQSIZE;	Q.rear==MAXQSIZE;

方案二：

循环队列，把队列设想成环形，首尾相接，

若Q.rear==MAXSIZE，则令Q.rear=0



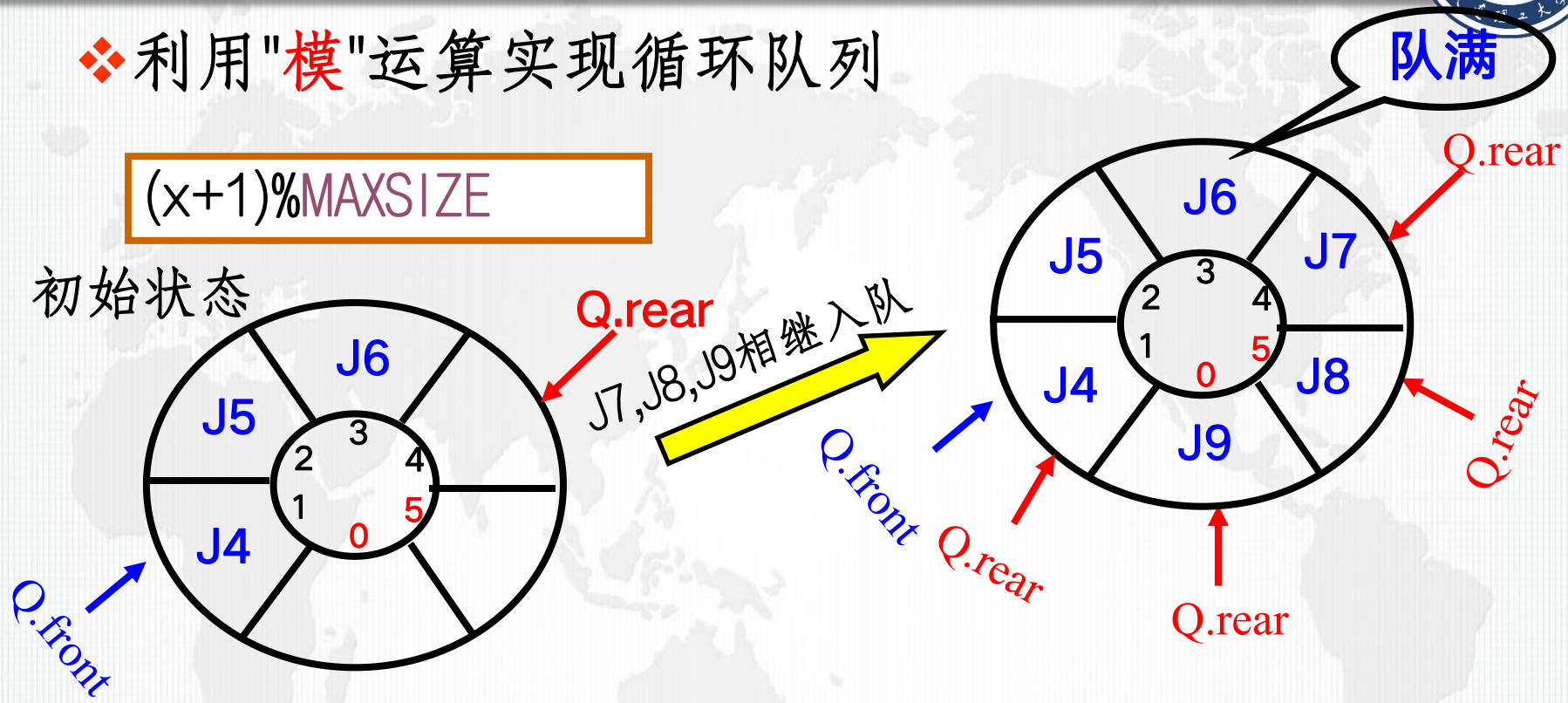
实现：

利用"模"运算

队列的存储结构——循环队列

- 利用"模"运算实现循环队列

(x+1)%MAXSIZE



"模"运算

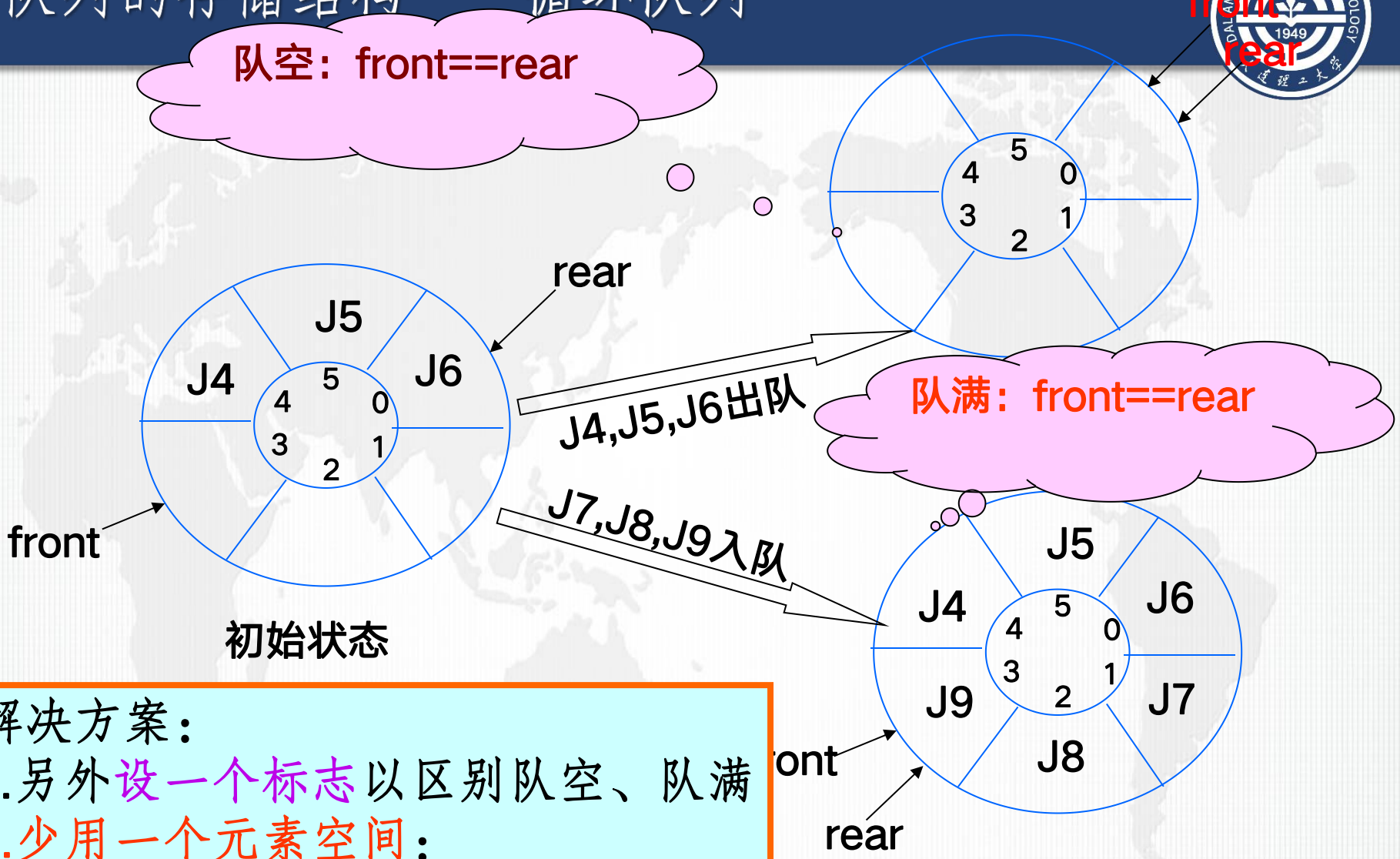
x%MAXSIZE——结果范围为

0~(MAXSIZE-1)

则， (x+1)%MAXSIZE可实现：x+1；

满足范围要求

队列的存储结构——循环队列



解决方案：

1.另外设一个标志以区别队空、队满

2.少用一个元素空间：

队空：front==rear

队满：(rear+1)%M==front

队列的存储结构——循环队列

- 构造空的循环队列Q：

```
#define MAXSIZE 100

typedef int QElemType;
typedef struct {
    QElemType *base;
    int front;
    int rear;
} SqQueue;
```

```
Status init_queue(SqQueue *Q) {
    Q->base = (QElemType *)malloc( size: MAXSIZE * sizeof(QElemType));
    if (!Q->base) {
        return OVERFLOW;
    }
    Q->front = Q->rear = 0;
    return OK;
}
```

队列的存储结构——循环队列

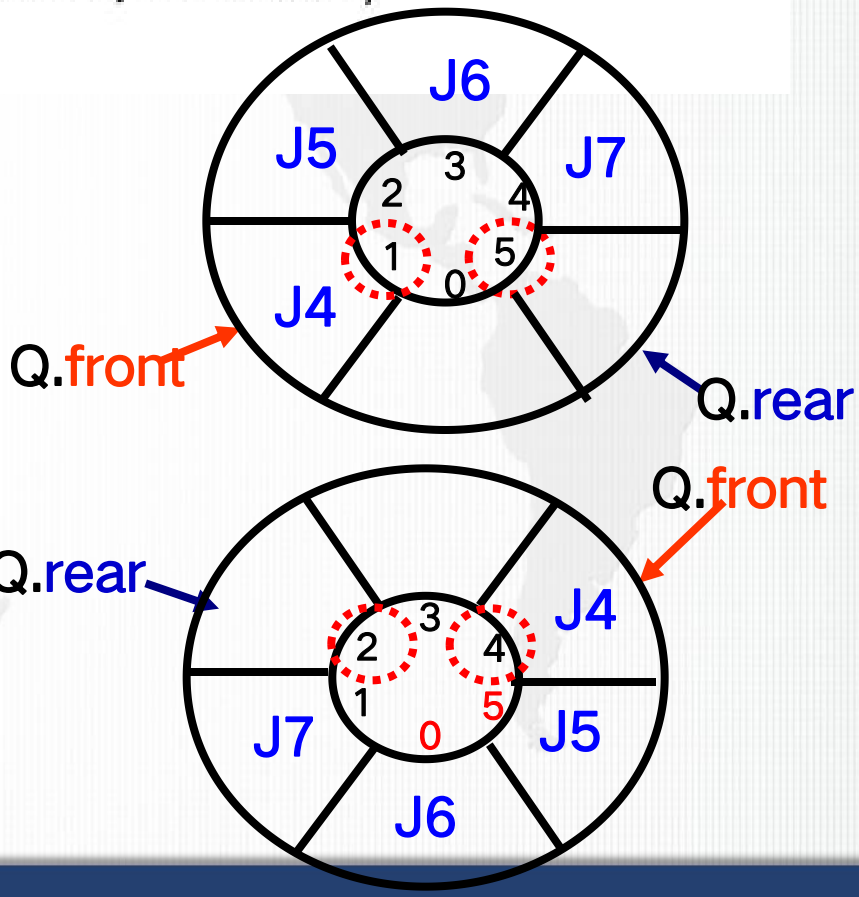
- 求循环队列中元素个数：

```
31  int queue_length(SqQueue *Q) {
32      return (Q->rear - Q->front + MAXSIZE) % MAXSIZE;
33  }
```

T(n)=O(1)

队中结点的个数：

（rear－front＋maxSize）%maxSize

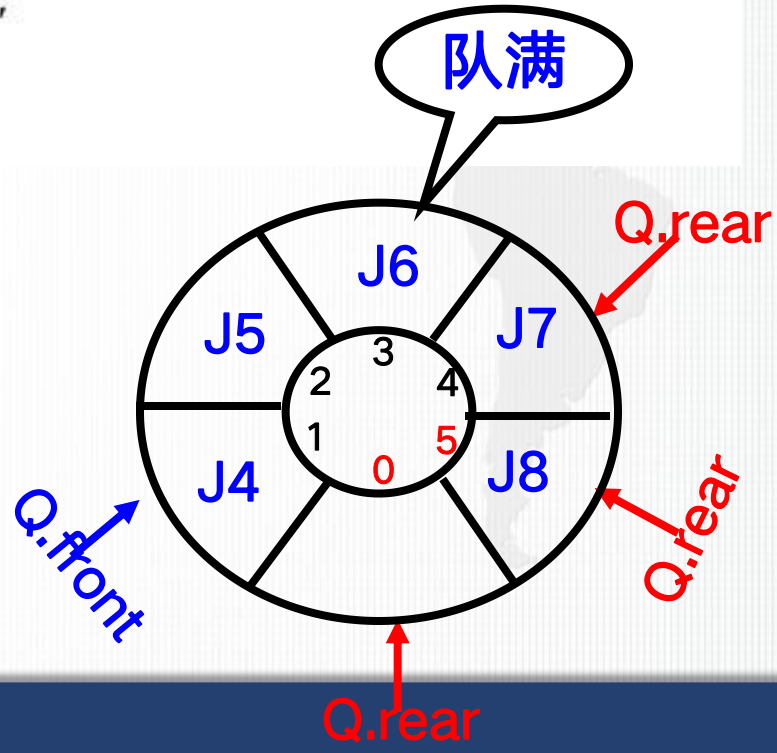


队列的存储结构——循环队列

- 循环队列入队：

```
35  Status enter_queue(SqQueue *Q, QElemType e) {
36      if ((Q->rear + 1) % MAXSIZE == Q->front) {
37          return ERROR;
38      }
39      Q->base[Q->rear] = e;
40      Q->rear = (Q->rear + 1) % MAXSIZE;
41      return OK;
42  }
```

T(n)=O(1)

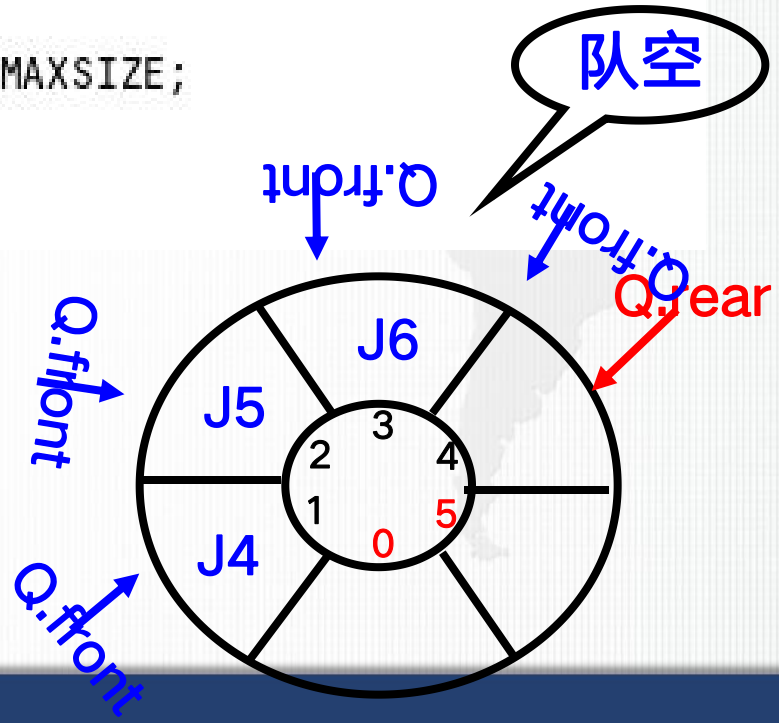


队列的存储结构——循环队列

- 循环队列出队：

```
44  Status de_queue(SqQueue *Q, QElemType *e) {
45      if (Q->front == Q->rear) {
46          return ERROR;
47      }
48      *e = Q->base[Q->front];
49      Q->front = (Q->front + 1) % MAXSIZE;
50      return OK;
51  }
```

$T(n)=O(1)$



队列应用

◆ 队列在计算机系统中的应用

队列在计算机系统中的应用非常广泛，以下仅从两个方面来简述队列在计算机系统中的作用：

1. 第一个方面是解决主机与外部设备之间速度不匹配的问题，
2. 第二个方面是解决由多用户引起的资源竞争问题。

对于第一个方面，仅以主机和打印机之间速度不匹配的问题为例做简要说明。主机输出数据 给打印机打印，输出数据的速度比打印数据的速度要快得多，解决的方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入这个缓冲区，写满后就暂停。打印机就从缓冲区中按照先进先出的原则依次取出数据并打印，打印完后再向主机发出请求。主机接到请求后再向缓冲区写入打印数据。打印数据缓冲区中所存储的数据就是一个队列。

队列应用

◆ 队列在计算机系统中的应用

队列在计算机系统中的应用非常广泛，以下仅从两个方面来简述队列在计算机系统中的作用：

1. 第一个方面是解决主机与外部设备之间速度不匹配的问题，
2. 第二个方面是解决由多用户引起的资源竞争问题。

对于第二个方面，CPU 资源的竞争就是一个典型的例子。在一个带有多终端的计算机系统上，有多个用户需要CPU各自运行自己的程序，它们分别通过各自的终端向操作系统提出占用CPU的请求。操作系统通常按照每个请求在时间上的先后 顺序，把它们排成一个队列，每次把CPU分配给队首请求的用户使用。当相应的程序运行结束或 用完规定的时间间隔后，令其出队，再把CPU分配给新的队首请 个用户的请求，又使CPU能够正常运行。

栈与队列的应用举例

◆ 练习1

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1:

输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

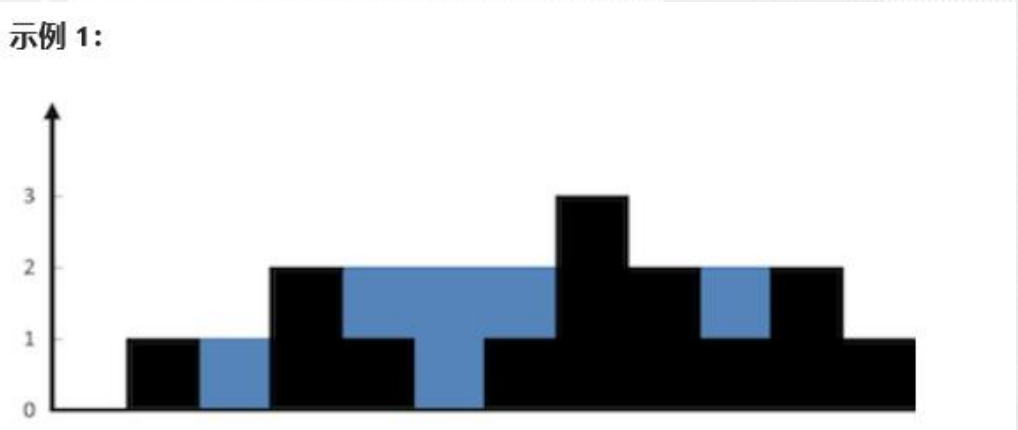
输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2:

输入：height = [4,2,0,3,2,5]

输出：9



栈与队列的应用举例

◆ 练习2（括号匹配进阶）

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度

示例 1:
输入: s = "(()"
输出: 2
解释: 最长有效括号子串是 "()"
示例 2:
输入: s = ")()())"
输出: 4
解释: 最长有效括号子串是 "()()"
示例 3:
输入: s = ""
输出: 0

线性表的应用举例

◆ 练习3（用栈和队列实现）

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

示例 1：

输入：head = [1, 2, 3, 4]

输出：[2, 1, 4, 3]

示例 2：

输入：head = []

输出：[]

示例 3：

输入：head = [1]

输出：[1]

提示：

链表中节点的数目在范围[0, 100] 内

0 <= Node.val <= 100

栈与队列的应用举例

◆ 练习1

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法一：

直观想法

直接按问题描述进行。对于数组中的每个元素，我们找出下雨后水能达到的最高位置，等于两边最大高度的较小值减去当前高度的值。

算法：

1. 初始化 ans=0
2. 从左向右扫描数组：
3. 初始化 left_max=0，right_max=0
4. height[0]的位置到当前位置寻找最大值：left_max=max(left_max,height[j])
5. 从当前位置到height末端寻找最大值：right_max=max(right_max,height[j])
6. 将min(max_left,max_right)−height[i]累加到 ans

时间复杂度较大为O（n^2）

栈与队列的应用举例

◆ 练习1

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法二：

直观想法

在暴力方法中，我们仅仅为了找到最大值每次都要向左和向右扫描一次。但是我们可以提前存储这个值。因此，可以通过动态编程解决。

算法：

1. 找到数组中从下标 i 到最左端最高的条形块高度 left_max
2. 找到数组中从下标 i 到最右端最高的条形块高度 right_max。
3. 扫描数组 height 并更新答案：
4. 累加 min(max_left[i],max_right[i])−height[i]到 ans上

复杂度为0（n）

栈与队列的应用举例

◆ 练习1

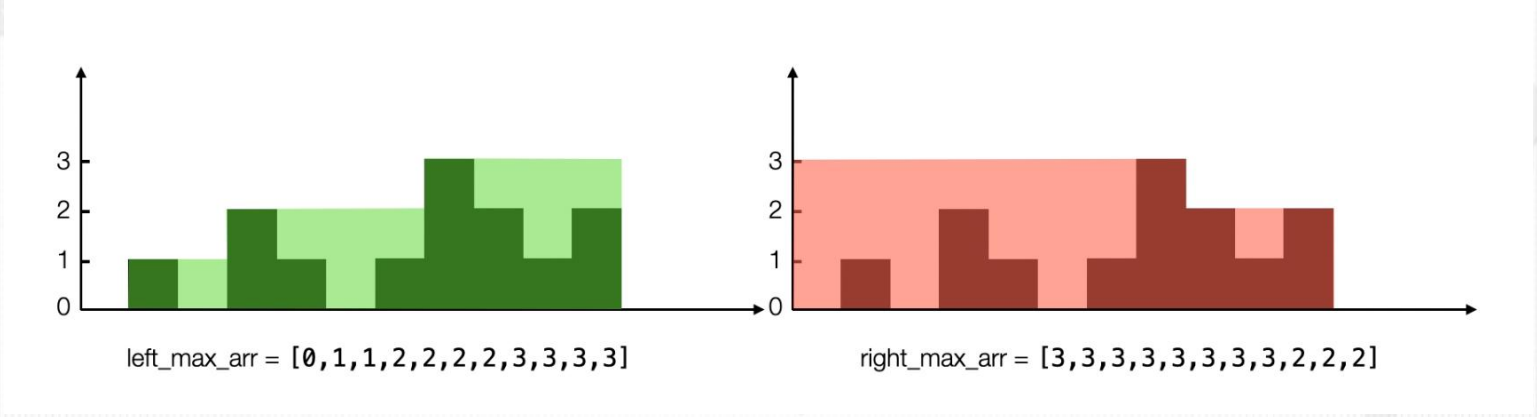
给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法二：

直观想法

在暴力方法中，我们仅仅为了找到最大值每次都要向左和向右扫描一次。但是我们可以提前存储这个值。因此，可以通过动态规划解决。

算法：



复杂度为O(n)

栈与队列的应用举例

◆ 练习1

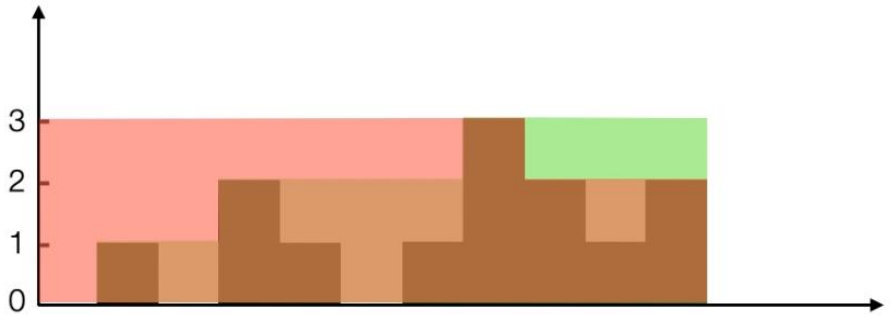
给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法二：

直观想法

在暴力方法中，我们仅仅为了找到最大值每次都要向左和向右扫描一次。但是我们可以提前存储这个值。因此，可以通过动态规划解决。

算法：



left_max_arr = [0,1,1,2,2,2,2,3,3,3,3]

right_max_arr = [3,3,3,3,3,3,3,3,2,2,2]

复杂度为O(n)

栈与队列的应用举例

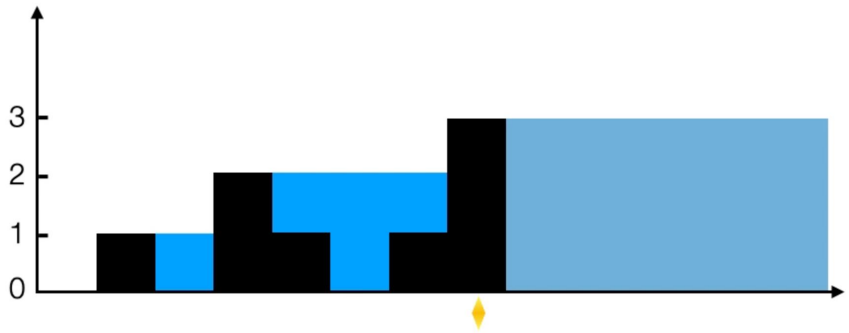
◆ 练习1

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法三：

直观想法

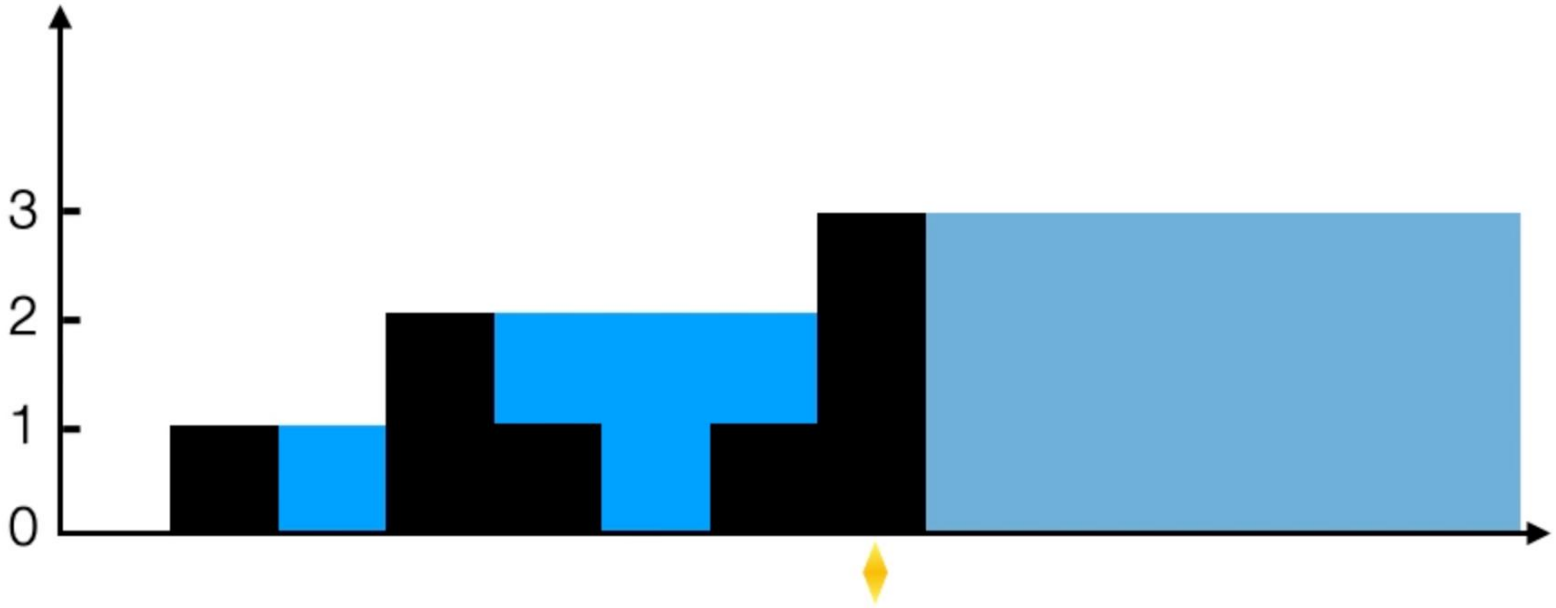
积水只能在低洼的地方形成，当后面的柱子高度比前面的低的时候，是没有办法接雨水的。所以使用单调递减栈存储可能储水的柱子，当找到一根比前面高的柱子，就可以计算接到的雨水。



height = [0,1,0,2,1,0,1,3,2,1,2,1]

栈与队列的应用举例

- 练习1



height = [0,1,0,2,1,0,1,3,2,1,2,1]

height = [0,1,0,2,1,0,1,3,2,1,2,1]

栈与队列的应用举例

◆ 练习1

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

方法三算法：

1. 使用栈来存储条形块的索引下标。
2. 遍历数组：

①当栈非空且 height[current]>height[st.top()]：

▷意味着栈中元素可以被弹出。弹出栈顶元素 top；

▷计算当前元素和栈顶的距离，准备进行填充操作distance=current-st.top()-1

▷找出界定高度

bounded_height=min(height[current],height[st.top()])-height[top]

▷往答案中累加积水量ans+=distance×bounded_height

②将当前索引下标入栈

③将 current 移动到下个位置

时间空间复杂度和方法二一样。

栈与队列的应用举例

◆ 练习2（括号匹配进阶）

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度

始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」，这样的做法主要是考虑了边界条件的处理，栈里其他元素维护左括号的下标：

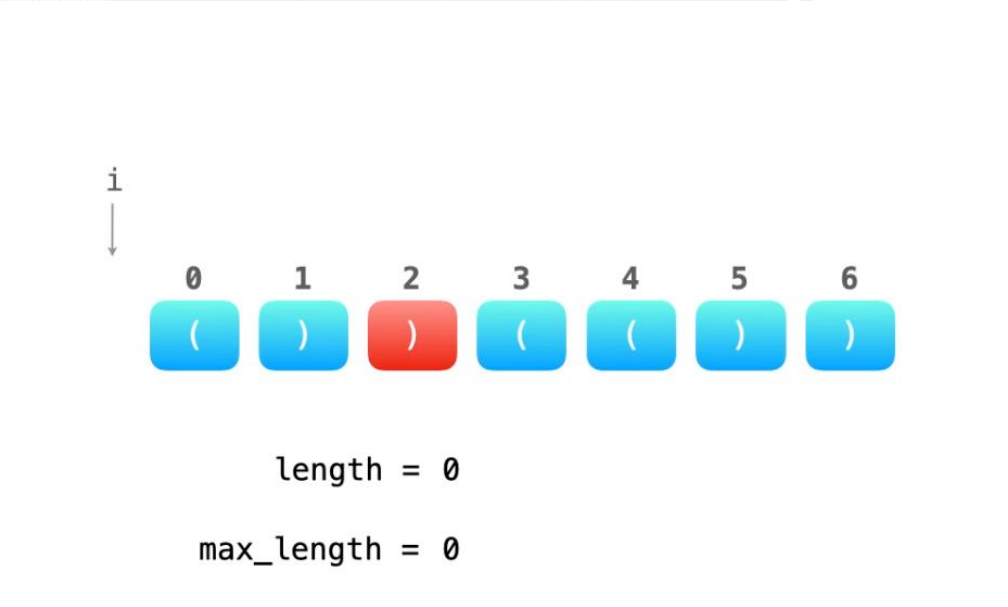
1. 对于遇到的每个‘(’ ，我们将它的下标放入栈中
2. 对于遇到的每个 ‘)’ ，我们先弹出栈顶元素表示匹配了当前右括号：
   ①如果栈为空，说明当前的右括号为没有被匹配的右括号，我们将其下标放入栈中来更新我们之前提到的「最后一个没有被匹配的右括号的下标」
   ②如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」

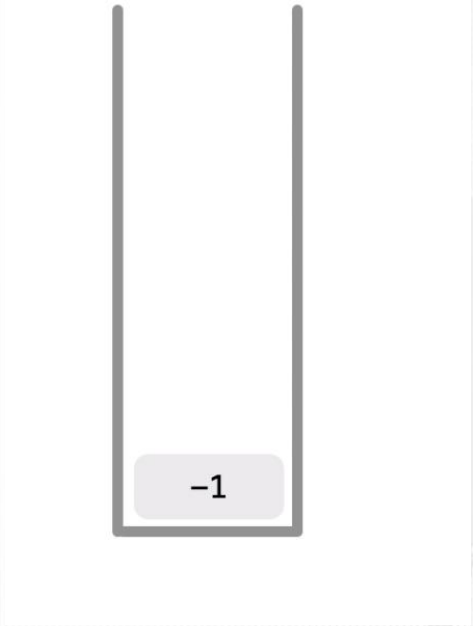
我们从前往后遍历字符串并更新答案即可。

栈与队列的应用举例

◆ 练习2（括号匹配进阶）

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度

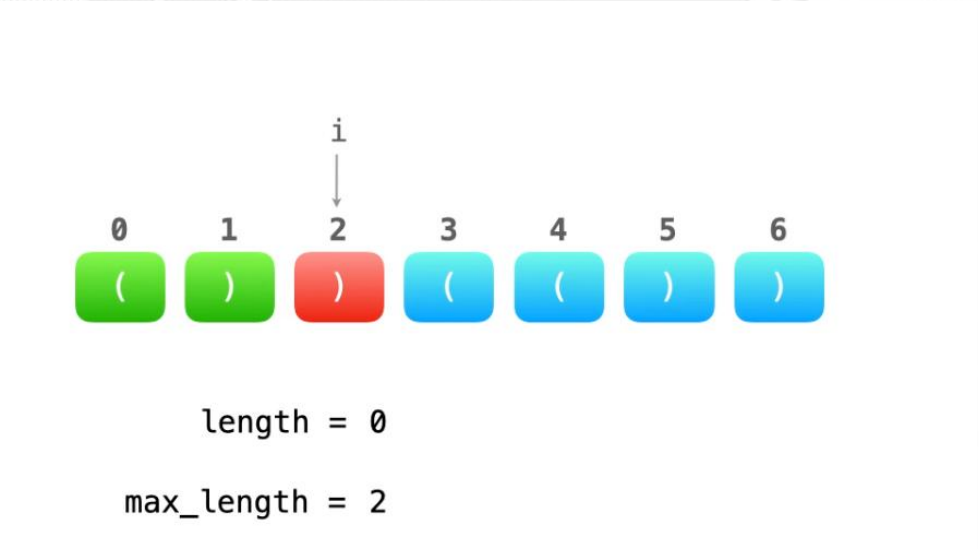


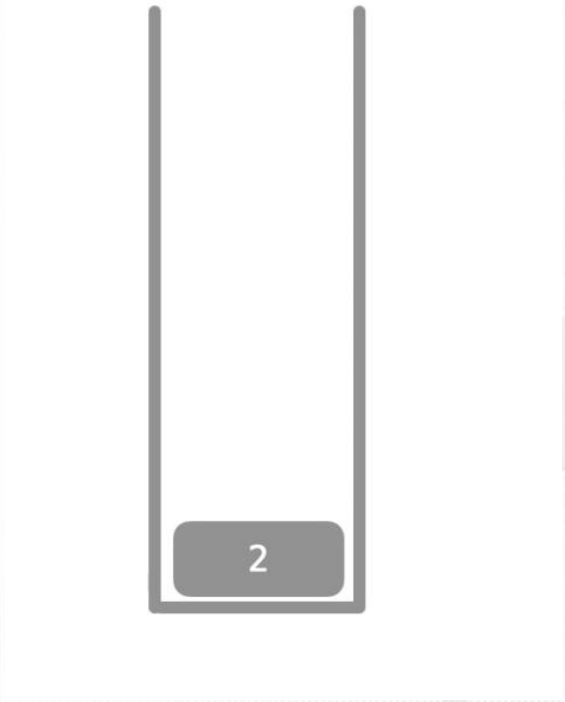


栈与队列的应用举例

◆ 练习2（括号匹配进阶）

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度





线性表的应用举例

◆ 练习3（用栈和队列实现）

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

直观思想：

实现两两交换，后面的一个和前面的一个交换，可以联想到“栈”后入先出的特点

两两交换之后整体链表的生成与存储可以利用“队列”存储

利用队列实现，遍历完原链表后还需要一次遍历生成新链表

算法：

1. 遍历给定链表，拆解出当前遍历的元素（使 next 为空）
2. 将拆解出的元素入栈，入栈前判断栈中是否有两个元素
3. 如果栈中有两个元素，先将元素依次出栈，然后入队
4. 经过一轮链表的遍历，就得到了满足结果的“链表节点”队列，每个节点 next 均为空
5. 遍历队列组成新链表即可，此时链表元素依旧是原链表的元素，内存地址没有改变

Homework

- 选择题全部
- leetcode 题目全部AC

Thanks!

See you in the next session!