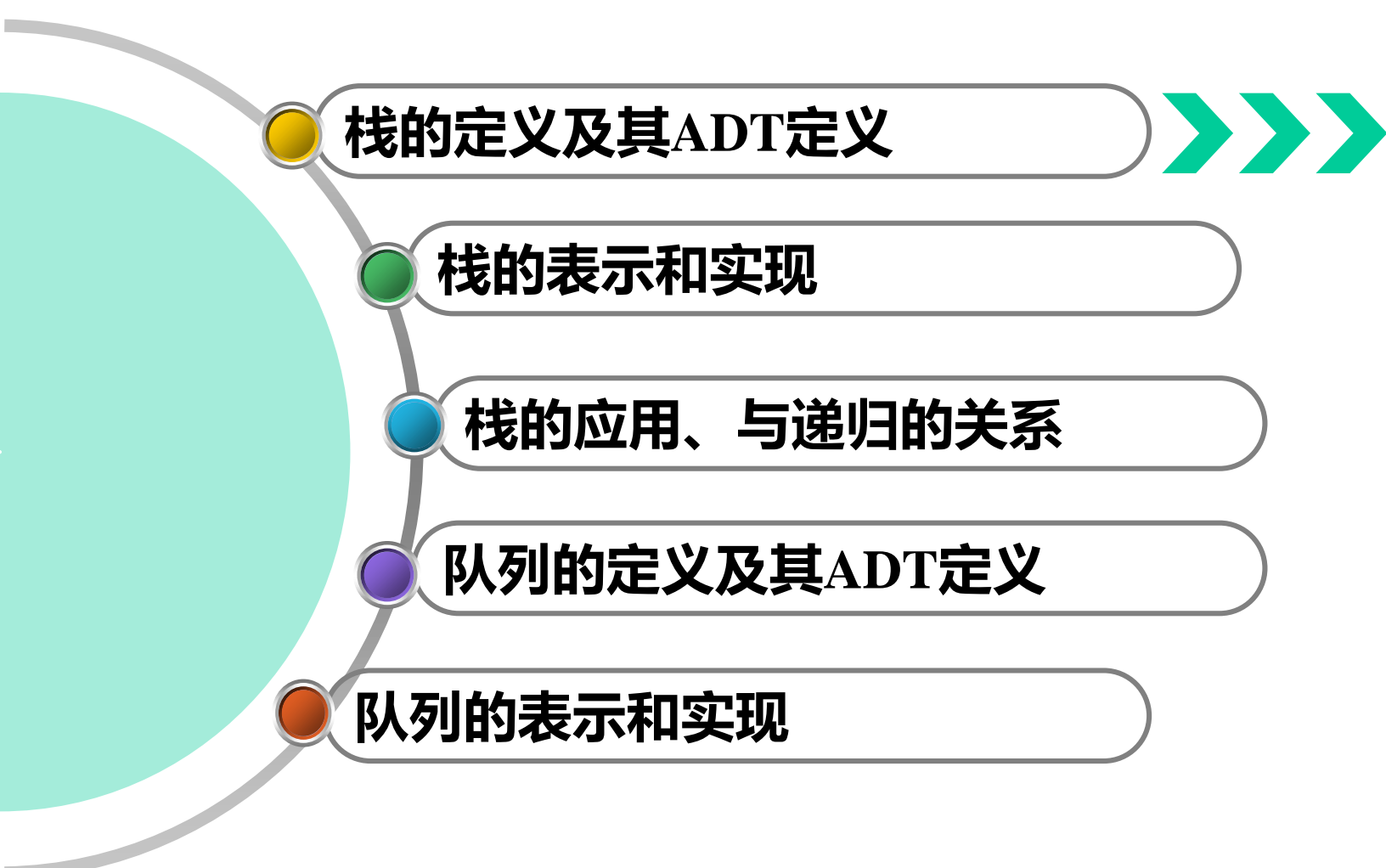


DS—第三章

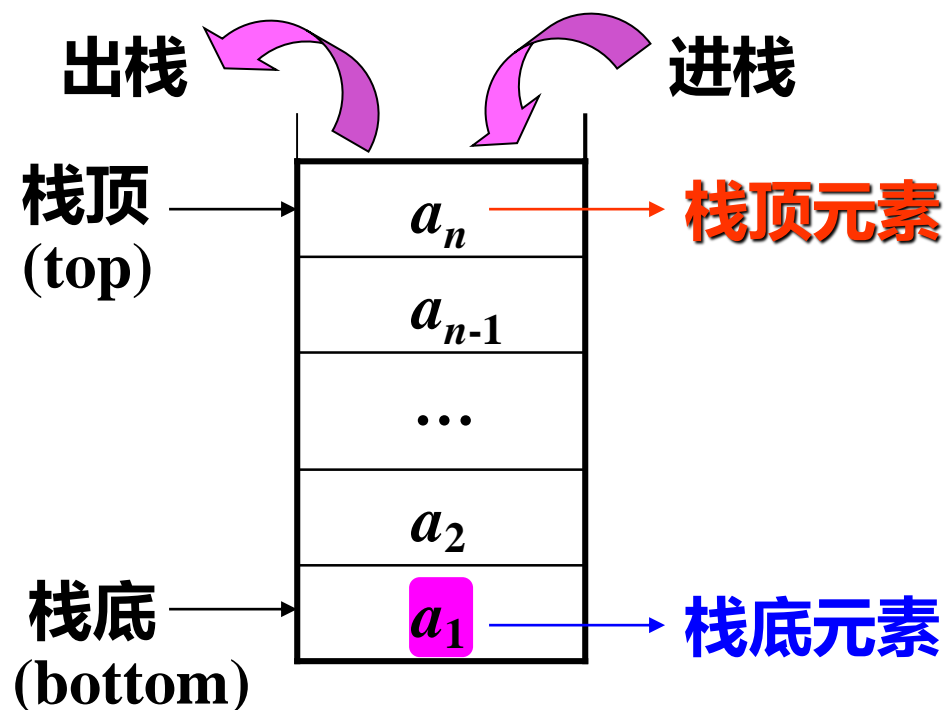
栈和队列-- *Stacks & Queues*





● 栈的定义

栈：线性表 { 限定仅在表尾进行插入或删除操作。
后进先出 (LIFO结构)。



练习:栈的逻辑特性

【例1】、设输入序列1、2、3、4，则下述序列中（ ）不可能是出栈序列。【中科院中国科技大学2005】

A. 1、2、3、4

B. 4、3、2、1

C. 1、3、4、2

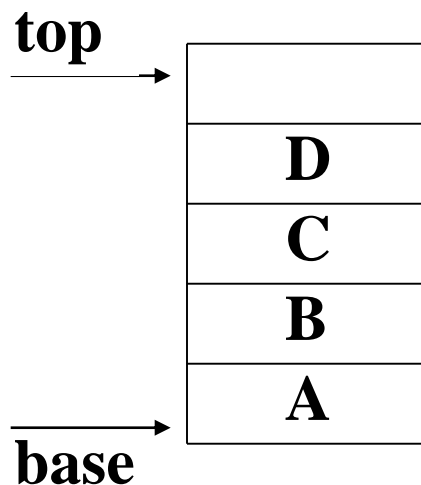
D. 4、1、2、3

答案：D.



● 顺序栈

顺序栈：利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针 top 指示栈顶元素在顺序栈中的位置。



栈满
 若再进行元素“出栈”操作，将产生“**下溢**”。

```
#define STACK_INIT_SIZE 100 // 栈存储空间的初始分配量
```

```
#define STACKINCREMENT 10 // 栈存储空间的分配增量
```

```
typedef struct {
```

```
    SelemType *base; // 栈底指针，它始终指向栈底的位置。
```

```
    SelemType *top; // 栈顶指针。
```

```
    int stacksize; // 当前分配的栈可使用的最大存储容量。
```

```
} Sqstack;
```

注：base 的值为 NULL，表明栈结构不存在。

书上基本操作
的实现

● 栈的基本操作在顺序栈中的实现

```
#define maxs 9;
```

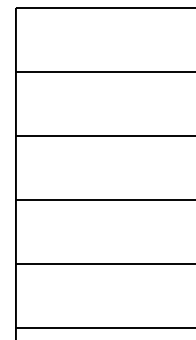
```
main()
```

```
{ int stack[maxs]; } InitStack
```

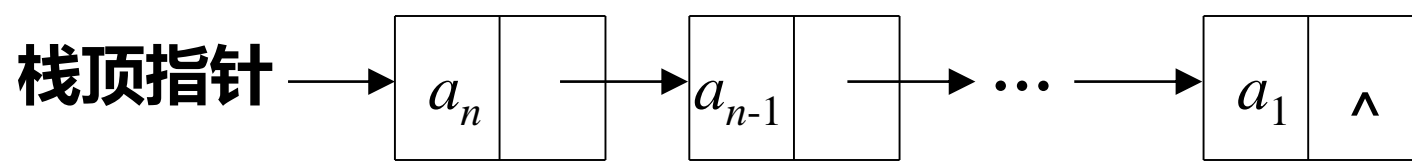
```
Status GetTop (SqStack S, SElemType &e) {  
    if (S.top == S.base) return ERROR;
```

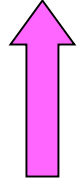
```
Status Pop (SqStack &S, SElemType &e) {    T_SIZE * sizeof(SElemtype));  
    if (S.top == S.base) return ERROR;  
    e = * -- S.top;  
    return OK;  
} // Pop
```

```
        (S.stacksize + STACKINCREMENT) * sizeof(SElemtype));  
    if (!S.base) exit (OVERFLOW);  
    S.top = S.base + S.stacksize;  
    S.stacksize += STACKINCREMENT; }  
    * S.top ++ = e;    return OK;  
} // Push
```



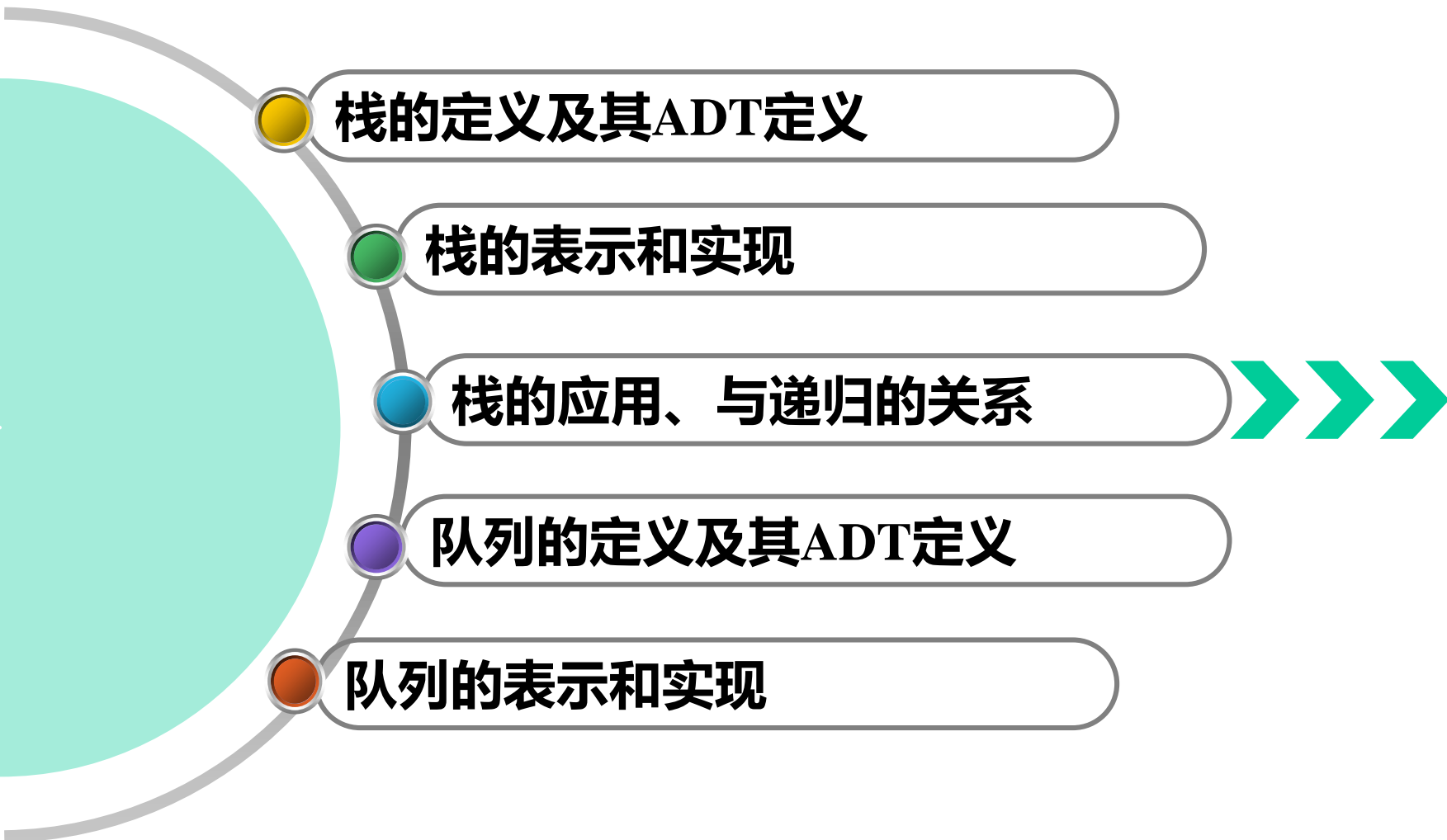
链栈





注意：链栈中指针的方向





3.2.1 数制转换

十进制数 N 和其他 d 进制数 M 的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法是逐次除以基数 d 取余法，它基于下列原理：

$$N = (N \operatorname{div} d) * d + N \operatorname{mod} d$$

具体作法为：首先用 N 除以 d ，得到的余数是 d 进制数 M 的最低位 M_0 ，接着以前一步得到的商作为被除数，再除以 d ，得到的余数是 d 进制数 M 的次最低位 M_1 ，依次类推，直到商为 0 时得到的余数是 M 的最高位 M_s （假定 M 共有 $s + 1$ 位）。

例： $(1348)_{10}=(2504)_8$, 其运算过程如下：

	N	$N \text{ div } 8$	$N \text{ mod } 8$	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

void conversion ()
{
 int stack[4];
 int top=0;
 int N;
 scanf(“%d”, N);
 while (N) {
 stack[top]=N%8;
 top++;
 N=N/8;
 }
 for(top=top-1; top>=0; top--)
 printf(“%d”,stack[top]);
}

InitStack(S)

Push(S, N%S)

While (!Stackempty(S)) {
 Pop(S, e);
 printf(“%d”, e);
}

top

bottom

0

2

5

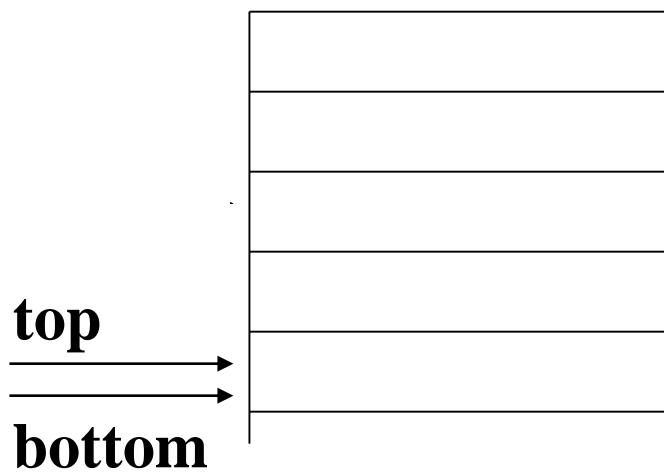
0

4

3.2.2 括号匹配的检验

假设表达式中允许括号嵌套，则检验括号是否匹配的方法可用“**期待的急迫程度**”这个概念来描述。

例： [([] [])]
 1 2 3 4 5 6 7 8



可能出现的不匹配的情况：

盼来的右括号**不是**所“期待”的；

到来的是“**不速之客**”

（**右括号多**）；

到结束也未盼来所“期待”的括号

（**左括号多**）。

算法的设计思想：

- 1) 凡出现**左括号**，则**进栈**；
- 2) 凡出现右括号，首先检查栈是否空。
若栈空，则表明该“右括号”多余；
否则和栈顶元素比较，
若相匹配，则“左括号出栈”，
否则表明不匹配。
- 3) 表达式检验结束时，
若栈空，则表明表达式中匹配正确，
否则表明“左括号”有多余的。

3.2.3 行编辑程序

功能：接受用户从终端输入的数据并存入用户的数据区。

做法 { 接受一个字符即存入数据区。（**差**！难纠错。）
设一个输入缓冲区，接受完一行字符后再存入用户的数据区。（**好**！可及时纠错。）

纠错办法 { # 退格符，表示前一个字符无效。
@ 退行符，表示整行字符均无效。

例：接受的字符为：

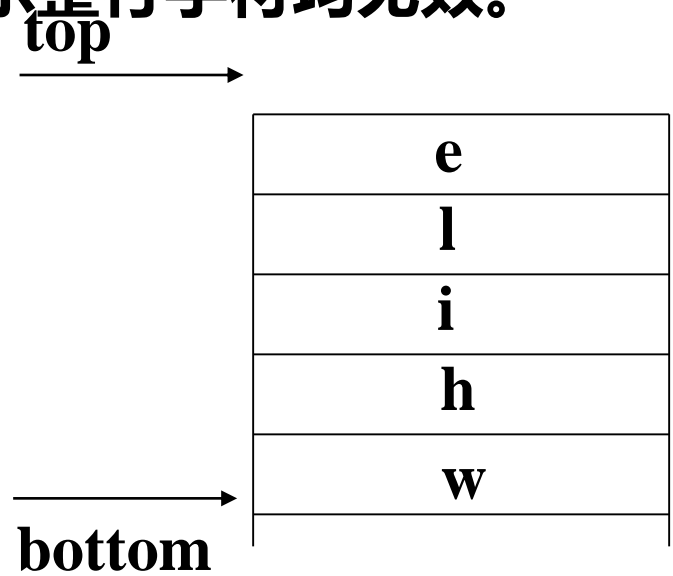
whli###ile

outch@putch

实际有效的为：

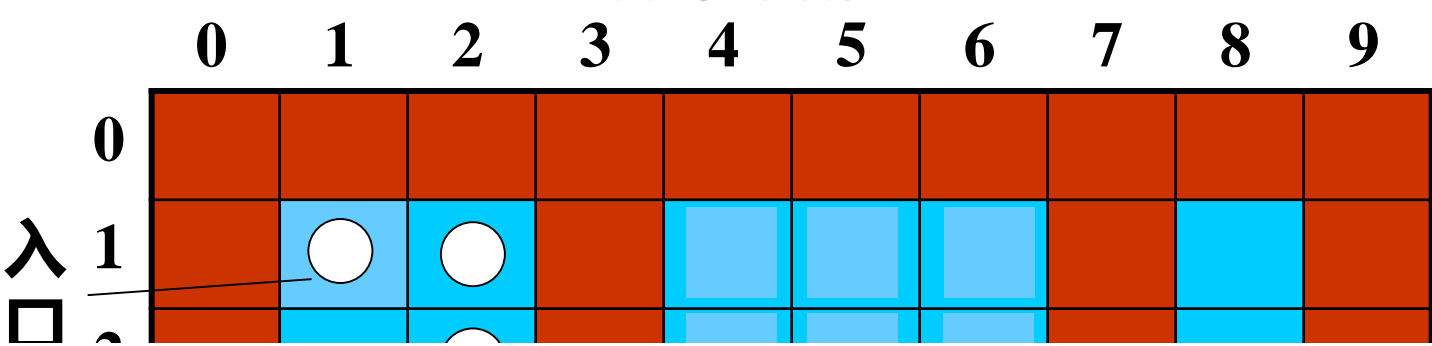
while

putch



```
void LineEdit( ) {  
    InitStack(S);  
    ch=getchar();  
    while (ch != EOF) { //EOF为全文结束符  
        while (ch != EOF && ch != '\n') {  
            switch (ch) {  
                case '#' : Pop(S, c); break;  
                case '@': ClearStack(S); break; // 重置S为空栈  
                default : Push(S, ch); break;  
            }  
            ch = getchar(); // 从终端接收下一个字符  
        }  
        将从栈底到栈顶的字符传送至调用过程的数据区 ;  
        ClearStack(S); // 重置S为空栈  
        if (ch != EOF) ch = getchar();  
    }  
    DestroyStack(S);  
}
```

3.2.4 迷宫求解 穷举求解



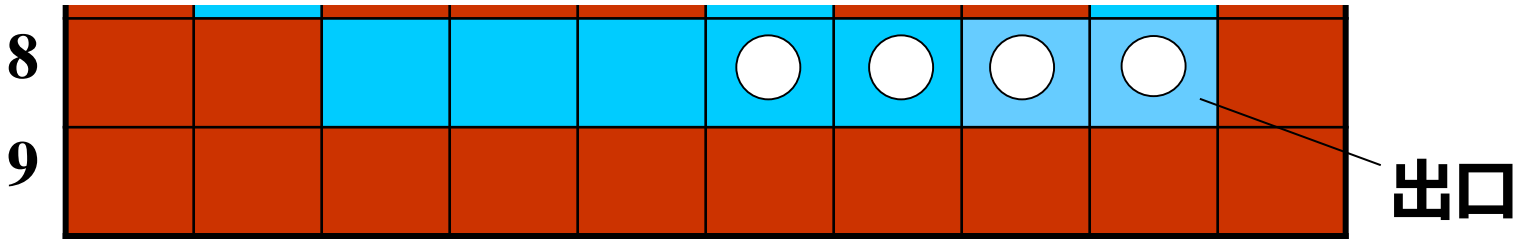
88
87
86
85
75
65
64
63
53
52
51
41
31
32
22
12
11

求迷宫路径算法的基本思想：

若当前位置“可通”，则纳入路径，继续前进；

若当前位置“不可通”，则后退，换方向（按东南西北的顺序）继续探索；

若四周“均无通路”，则将当前位置从路径中删除出去。



出口

3.2.5 表达式求值

运算规则 { 先乘除，后加减；
 从左算到右；
 先括号内，后括号外；

例：求表达式 $4+2\times 3-10/5$ 的值。

计算顺序为： $4+2\times 3-10/5=4+6-10/5=10-10/5=10-2=8$

8

操作数或结果

#

运算符

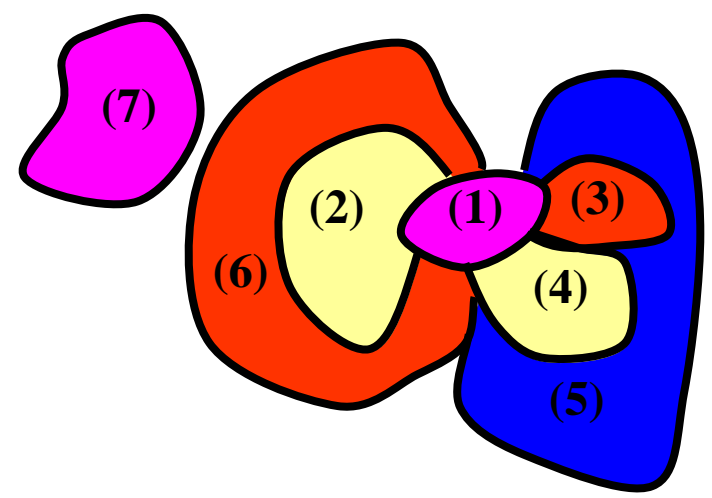
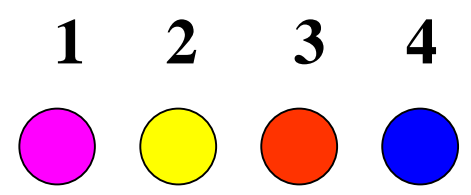
补充：地图四染色问题

“四染色”定理是计算机科学中著名定理之一，即可以用不多于四种颜色对地图着色，使相邻的行政区域不重色。

算法思想：从第一号行政区开始逐一染色，每一个区域逐次用颜色 1、2、3、4 进行试探。若当前所取的色数与周围已染色的行政区不重色，则用栈记下该行政区的色数，否则依次用下一色数进行试探；若出现用 1 至 4 色均与相邻区域发生重色，则需退栈回溯，修改当前栈顶的色数，再进行试探。直至所有行政区域都已分配合适的颜色。

例：已知 7 个行政区域地图，对其进行染色。

	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0



1	2	3	4	5	6	7
1	2	3	2	4	3	1

课堂作业

- 1、若入栈序列是 a, b, c, d, e , 则不可能的出栈序列是 ()。
(A) $edcba$ (B) $decba$ (C) $dceab$ (D) $abcde$
- 2、判定一个栈 ST(最多元素为 m_0) 为空的条件是 ()。
(A) $ST.top \neq ST.base$ (B) $ST.top == ST.base$
(C) $ST.top \neq ST.base + m_0$ (D) $ST.top == ST.base + m_0$
- 3、判定一个栈 ST(最多元素为 m_0) 为满的条件是 ()。
(A) $ST.top \neq ST.base$ (B) $ST.top == ST.base$
(C) $ST.top \neq ST.base + m_0$ (D) $ST.top == ST.base + m_0$

3.3 栈与递归的实现

递归：一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。

例：阶乘函数
$$Fact(n) = \begin{cases} 1 & \text{若 } n = 0 \\ n \cdot Fact(n-1) & \text{若 } n > 0 \end{cases}$$

相应的 C 语言函数是：

```
float fact(int n)
{
    float s;
    if (n == 0)
        s = 1;
    else
        s = n * fact(n - 1);
    return (s);
}
```

若求 5!，则有

```
main()
{
    printf("5!=%f\n", fact(5));
}
```


当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：

1. 将实参等传递给被调用函数，**保存返回地址（入栈）**；
2. 为被调用函数的局部变量**分配存储区**；
3. 将**控制转移**到被调用函数的入口。

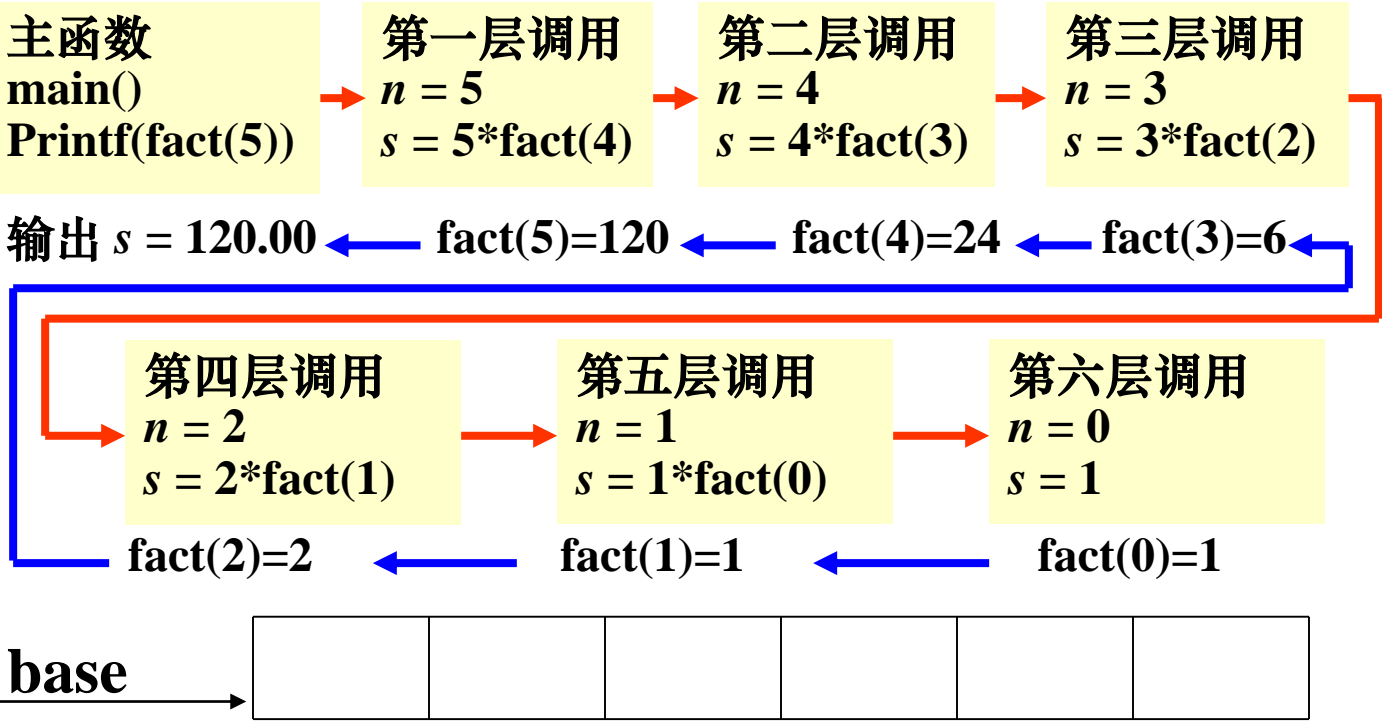
从被调用函数返回调用函数之前，应该完成：

1. **保存**被调函数的**计算结果**；
2. **释放**被调函数的**数据区**；
3. 按被调函数保存的返回地址（**出栈**）将**控制转移**到调用函数。

多个函数嵌套调用的规则是：**后调用先返回**。

此时的内存管理实行“**栈式管理**”。

递归调用执行过程： `printf("5!=%f\n",fact(5));`



```
float fact(int n)
{float s;
 if (n == 0)  n = 0
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

```
float fact(int n)
{float s;
 if (n == 0)  n = 1
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

```
float fact(int n)
{float s;
 if (n == 0)  n = 5
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

```
float fact(int n)
{float s;
 if (n == 0)  n = 4
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

```
float fact(int n)
{float s;
 if (n == 0)  n = 3
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

```
float fact(int n)
{float s;
 if (n == 0)  n = 2
  s =1;
 else
  s = n*fact(n -1);
 return (s);
}
```

思考

```
fac(int n)
{ if(n==0)
    return 1;
  else
    { s=n*fac(n-1);
      printf("%d",n);
      return s;
    }
}
```

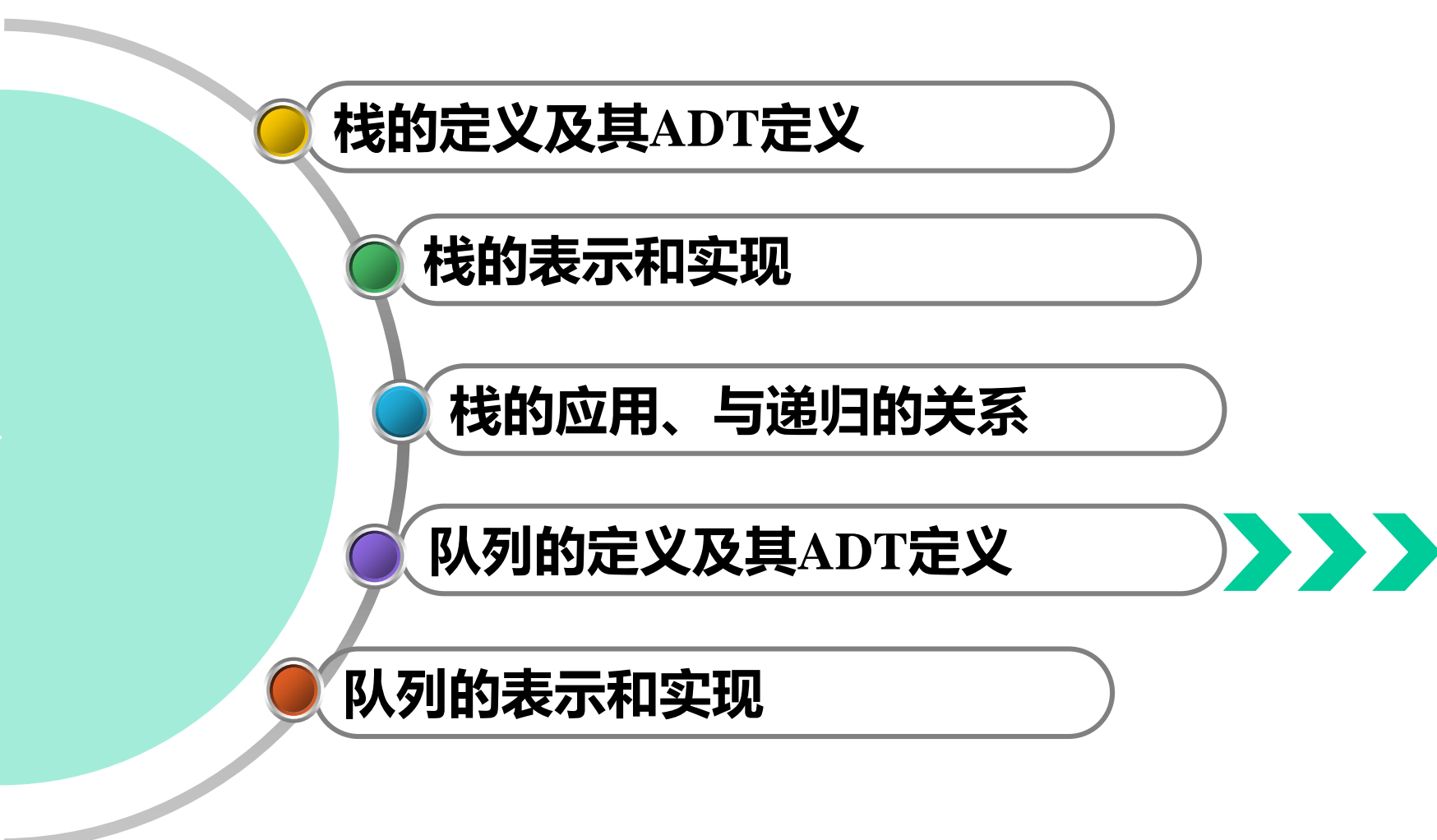
当n=4时，输出结果为

1 , 2 , 3 , 4

```
fac(int n)
{ if(n==0)
    return 1;
  else
    { printf("%d",n);
      s=n*fac(n-1);
      return s;
    }
}
```

当n=4时，输出结果为

4 , 3 , 2 , 1



3.4.1 抽象数据类型队列的定义

● 队列的定义

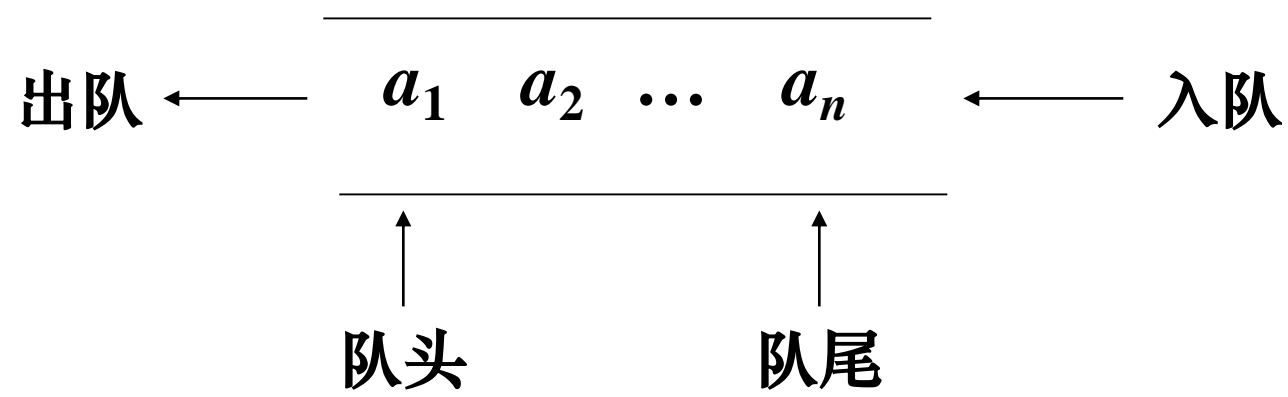
队列：线性表
(queue)

限定在表的一端插入、另一端删除。
先进先出 (FIFO结构)。

队尾

队头

下图是队列的示意图：



当队列中没有元素时称为空队列。

● 队列的抽象数据类型的定义

ADT Queue {

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

数据关系: $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 2, \dots, n \}$

约定其中 a_1 端为队列头, a_n 端为队列尾。

基本操作:

InitQueue(&Q)

操作结果: 构造一个空队列 Q。

DestroyQueue(&Q)

初始条件: 队列 Q 已存在。

操作结果: 队列 Q 被销毁, 不再存在。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回 TRUE，
否则返回 FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q, &e)

初始条件：Q 为非空队列。

操作结果：用 e 返回 Q 的队头元素。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

EnQueue(&Q, e)

初始条件：队列 Q 已存在。

操作结果：插入元素 e 为 Q 的新的队尾元素。

DeQueue(&Q, & e)

初始条件：Q 为非空队列。

操作结果：删除 Q 的队头元素，并用 e 返回其值。

} ADT Queue

● 双端队列

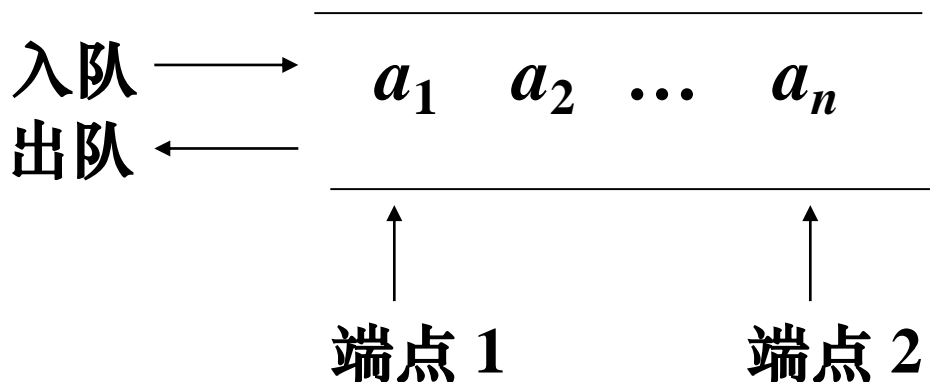
双端队列：线性表
(double-ended queue)

限定插入和删除在表的两端进行。

先进先出 (FIFO结构)。

端点 1
端点 2

下图是双端队列的示意图：



输出受限的双端队列：一个端点可插入和删除，
另一个端点仅可插入。

输入受限的双端队列：一个端点可插入和删除，
另一个端点仅可删除。

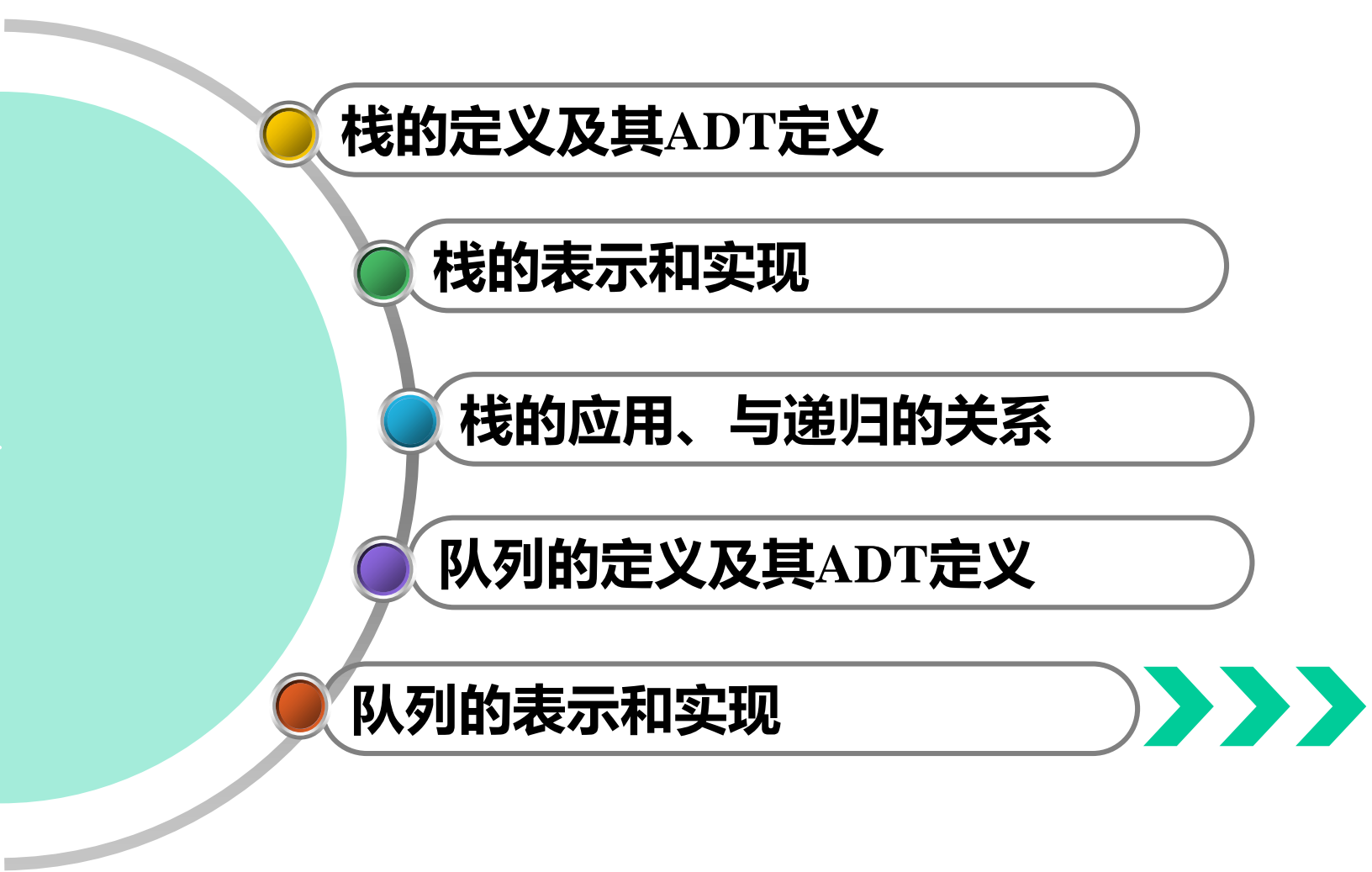
某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作，若元素a,b,c,d,e依次入队列后，再进行出队操作，则不可能得到的顺序是（ ）。

A . bacde

B. dbace

C. dbcae

D. ecbad



3.4.2 链队列——队列的链式表示和实现

链队列：用链表表示的队列。

是限制仅在表头删除和表尾插入的单链表。

一个链队列由一个头指针和一个尾指针唯一确定。

（因为仅有头指针不便于在表尾做插入操作）。

为了操作的方便，也给链队列添加一个头结点，因此，空队列的判定条件是：**头指针和尾指针都指向头结点。**

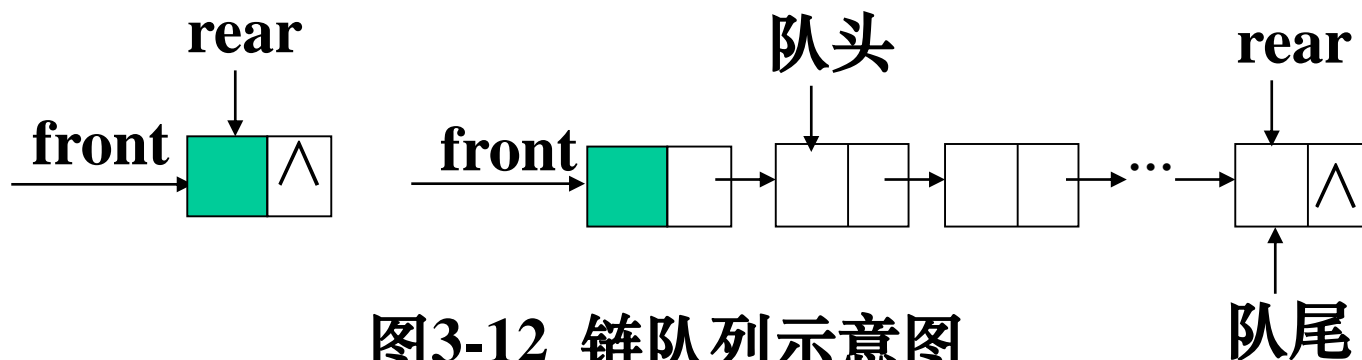


图3-12 链队列示意图

用 C 语言定义链队列结构如下：

```
typedef struct QNode
{
    QElemtype    data;
    struct QNode *next;
} Qnode, *QueuePtr; // 定义队列的结点

typedef struct
{
    QueuePtr front; // 队头指针
    QueuePtr rear;  // 队尾指针
}LinkQueue;
```

队列的基本操作在链队列中的实现：

队列的初始化：

Status InitQueue (LinkQueue &Q)

{ // 构造一个空队列 Q

Q.front = Q.rear = (QueuePtr) malloc (sizeof(QNode));

if (!Q.front) exit (OVERFLOW); // 存储分配失败

Q.front -> next = NULL;

return OK;

}

销毁队列:

Status DestroyQueue (LinkQueue &Q)

```
{ while (Q.front)
    { Q.rear = Q.front -> next;
      free (Q.front);
      Q.front = Q.rear;
    }
  return OK;
}
```

Q.rear= null

Q.front = null

插入操作在链队列中的实现

Status EnQueue (LinkQueue &Q, QElemType e)

{ // 插入元素 e 为 Q 的新的队尾元素

p = (QueuePtr) malloc (sizeof (QNode));

if (!p) exit (OVERFLOW);

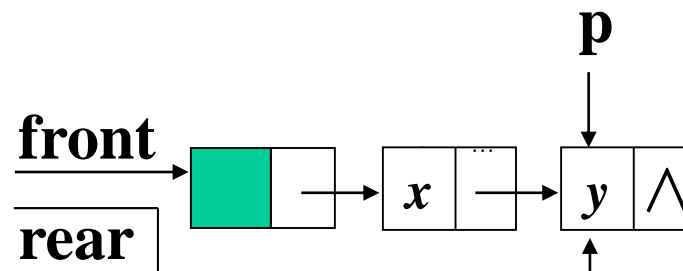
p -> data = e; p -> next = NULL;

Q.rear -> next = p;

Q.rear = p;

return OK;

}



删除操作在链队列中的实现

Status DeQueue (LinkQueue &Q, QElemType &e)

{ if (Q.front == Q.rear) return ERROR;

p = Q.front -> next;

e = p -> data;

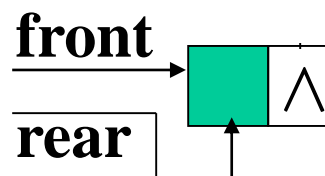
Q.front -> next = p -> next;

if (Q.rear == p) Q.rear = Q.front;

free (p);

return OK;

}



课堂作业

- 1、栈的特点是 () , 队列的特点是 () 。
- 2、线性表、栈和队列都是 () 结构 , 线性表可以在 () 位置插入和删除元素 , 栈只能在 () 插入和删除元素 , 队列只能在 () 插入元素和 () 删除元素。
- 3、设栈 S 和队列 Q 的初始状态皆为空 , 元素 a_1, a_2, a_3, a_4, a_5 和 a_6 依次通过一个栈 , 一个元素出栈后即进入队列 Q , 若 6 个元素出队列的顺序是 $a_3, a_5, a_4, a_6, a_2, a_1$ 则栈 S 至少应该容纳 () 个元素。
 (A) 3 (B) 4 (C) 5 (D) 6
- 4、若队列的入队序列是 1, 2, 3, 4 , 则出队序列是 () 。
 (A) 4,3,2,1 (B) 1,2,3,4 (C) 1,4,3,2 (D) 3,2,4,1

3.4.3 循环队列—队列的顺序表示和实现

是限制仅在表头删除和表尾插入的顺序表。
利用一组**地址连续**的存储单元**依次存放**队列中的数据元素。
因为：队头和队尾的位置是变化的，**所以：**设头、尾指针。

头尾指针 {

初始化时的初始值均应置为 0。

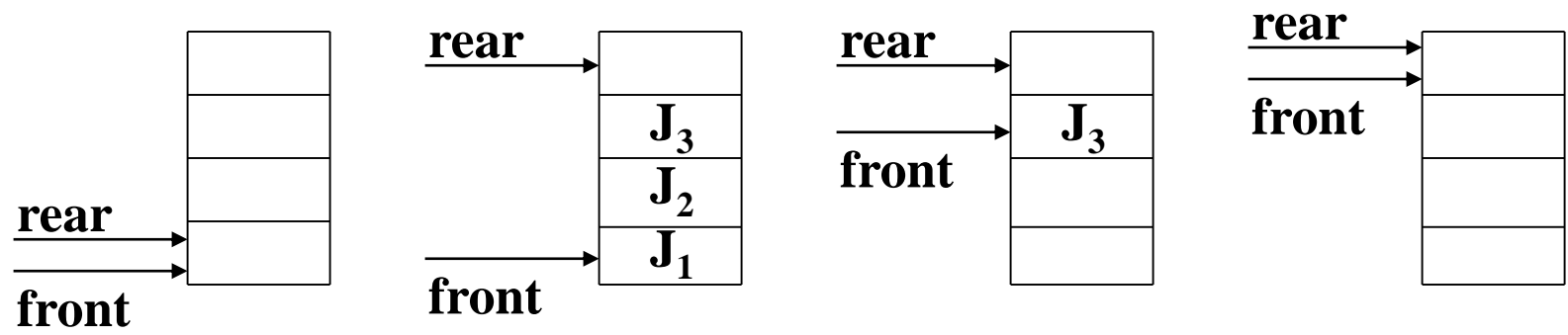
入队，尾指针增 1

出队，头指针增 1

头尾指针相等时队列为空

在非空队列里，头指针始终指向队头元素

尾指针始终指向队尾元素的下一位置。



头、尾指针和队列元素之间的关系

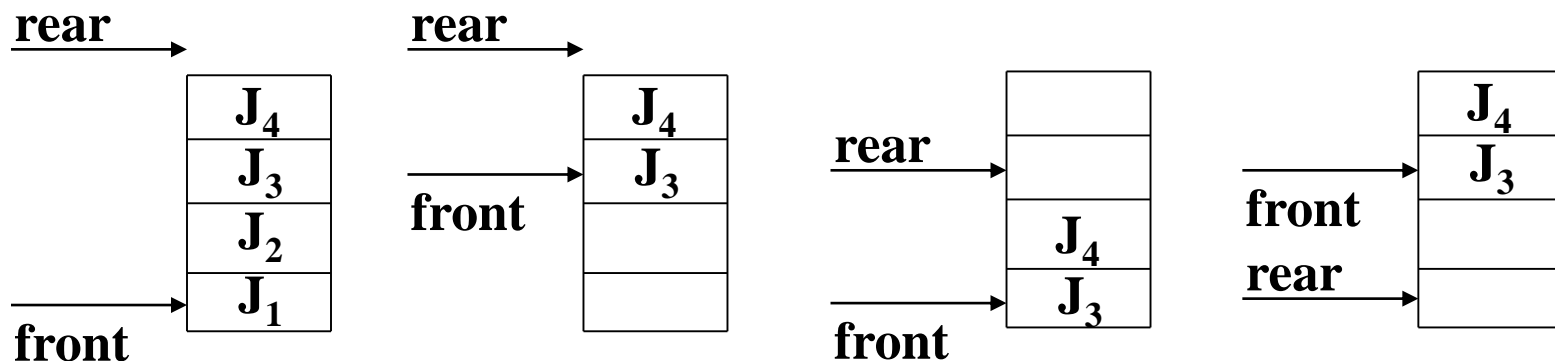
在顺序队列中，当尾指针已经指向了队列的最后一个位置的下一位置时，若再有元素入队，就会发生“**溢出**”。

“**假溢出**”——队列的存储空间未滿，却发生了溢出。

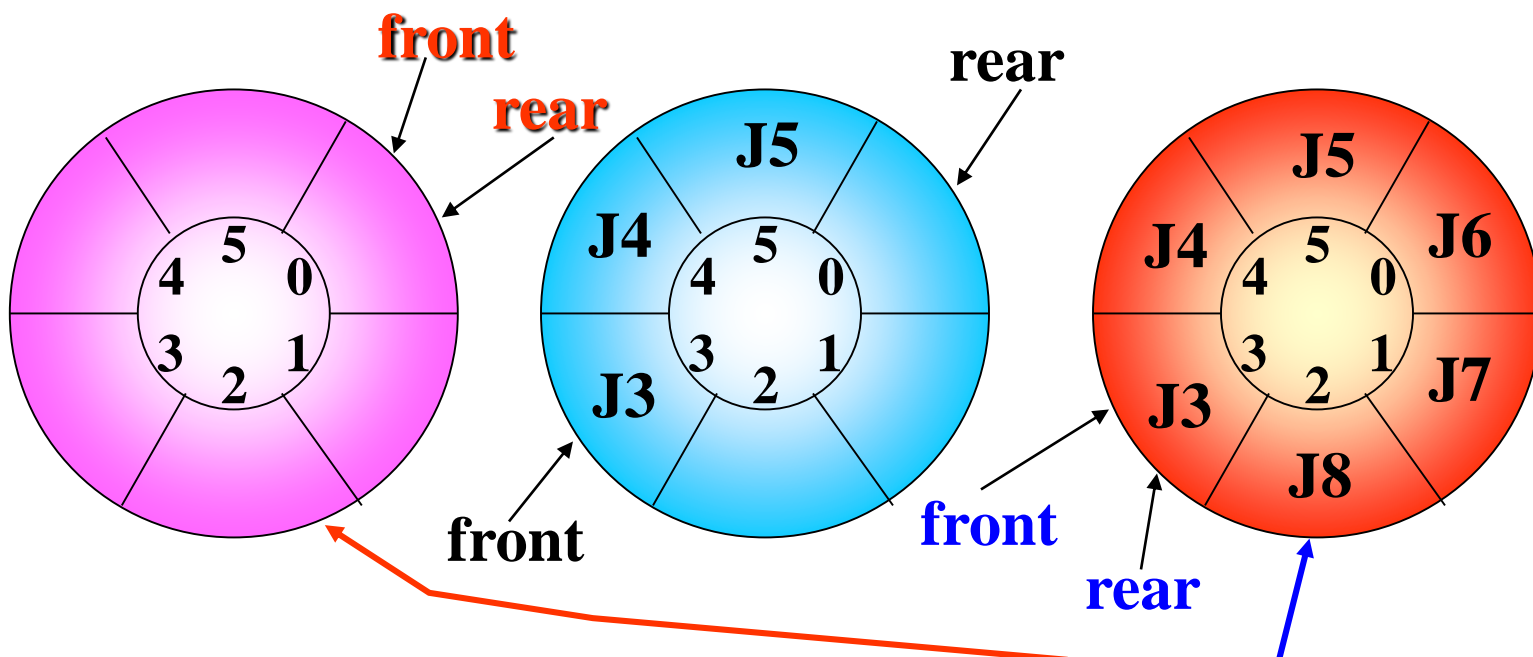
解决“假溢出”的问题有两种可行的方法：

- (1)、**平移元素**：把元素平移到队列的首部。**效率低**。
- (2)、**将新元素插入到第一个位置上，构成循环队列**，入队和出队仍按“先进先出”的原则进行。

操作效率、空间利用率高。



循环队列的三种状态：



注：仅凭 $\text{front} = \text{rear}$ 不能判定队列是空还是满。

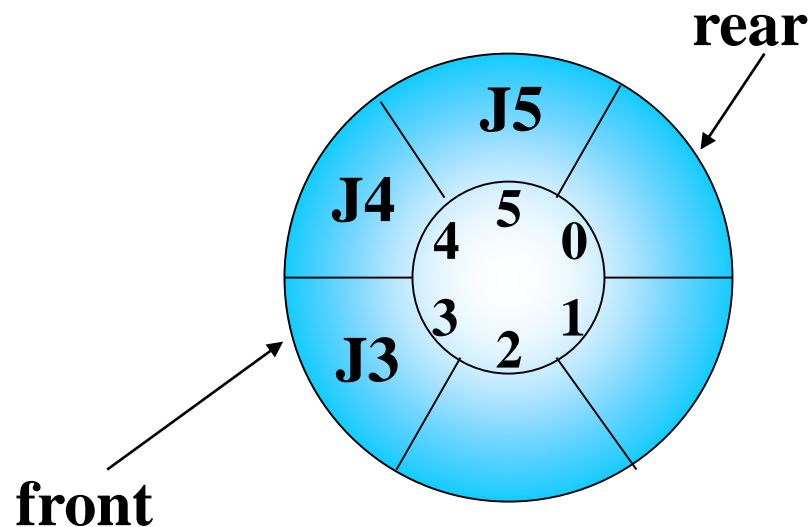
解决办法： (1)、另设一个布尔变量以区别队列的空和满(使用一个计数器记录队列中元素的总数。);

(2)、少用一个元素的空间，约定入队前测试尾指针在循环意义下加 1 后是否等于头指针，若相等则认为队满；

队列的顺序存储结构:

```
#define MAXQSIZE 100      //最大队列长度

typedef struct {
    QElemType *base;      // 预分配存储空间基址
    int front;             // 头指针，若队列不空，
                           // 指向队列头元素
    int rear;              // 尾指针，若队列不空，
                           // 指向队列尾元素 的下一个位置
} SqQueue;
```



循环意义下的加 1 操作可以描述为：

```
if (rear + 1 >= MAXQSIZE)
```

```
    rear = 0;
```

```
else
```

```
    rear ++;
```

利用模运算可简化为： $\text{rear} = (\text{rear} + 1) \% \text{MAXQSIZE}$

队列的基本操作在循环队列中的实现：

队列的初始化：

Status InitQueue (SqQueue &Q)

{ // 构造一个空队列Q

Q.base = (QElemType *) malloc

(MAXQSIZE *sizeof (QElemType));

if (!Q.base) exit (OVERFLOW); // 存储分配失败

Q.front = Q.rear = 0;

return OK;

}

求循环队列的长度

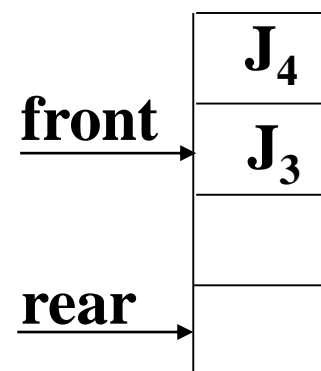
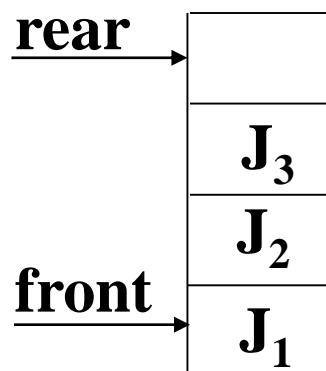
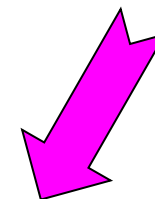
```
Status QueueLength (SqQueue Q) {
```

```
// 返回队列 Q 的元素个数
```

```
return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
```

```
}
```

考虑到循环队列



插入操作在循环队列中的实现

```
Status EnQueue (SqQueue &Q, QElemType e) {
```

```
    // 插入元素 e 为 Q 的新的队尾元素
```

```
    if ((Q.rear + 1) % MAXQSIZE == Q.front)
```

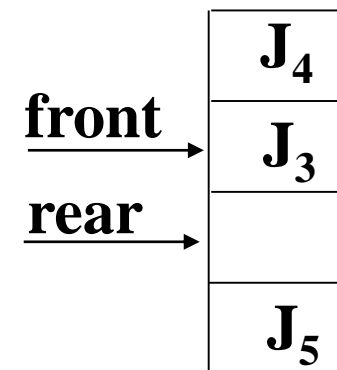
```
        return ERROR;           // 队列满
```

```
    Q.base[Q.rear] = e;
```

```
    Q.rear = (Q.rear + 1) % MAXQSIZE;
```

```
    return OK;
```

```
}
```



删除操作在循环队列中的实现

```
Status DeQueue (SqQueue &Q, ElemType &e) {
```

```
    // 若队列不空，则删除 Q 的队头元素，
```

```
    // 用e 返回其值，并返回 OK；否则返回 ERROR
```

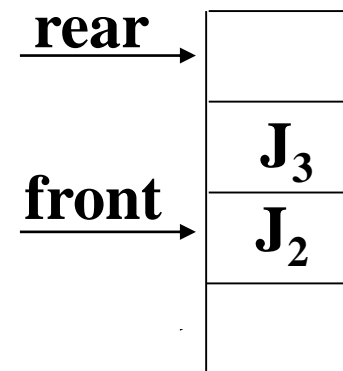
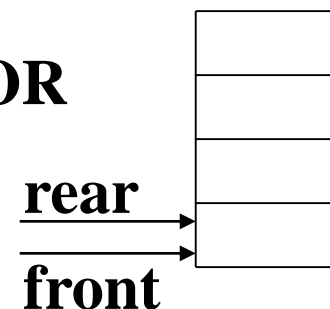
```
    if (Q.front == Q.rear) return ERROR;
```

```
    e = Q.base[Q.front];
```

```
    Q.front = (Q.front + 1) % MAXQSIZE;
```

```
    return OK;
```

```
}
```



队列的应用

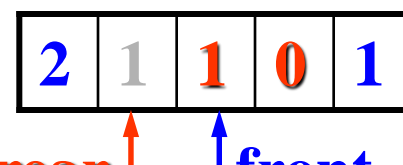
补例 1：杨辉三角的计算

利用循环队列的计算过程：

假设只计算 2 行，

队列的最大容量为 5。

1			1	1		
2			1	2	1	
3		1	3	3	1	
4	1	4	6	4	1	



```
do {
    DeQueue(Q, s);    // s 为二项式系数表第 k 行中“左上方”的值
    GetHead(Q, e);    // e 为二项式系数表第 k 行中“右上方”的值
    cout<<e;          // 输出 e 的值
    EnQueue(Q, s+e);  // 计算所得第 k+1 行的值入队列
} while (e!=0);
```

补例 2: CPU 资源的竞争问题

在多用户计算机系统中，各个用户需要使用 CPU 运行自己的程序，它们分别向操作系统提出使用 CPU 的请求，操作系统按照每个请求在时间上的先后顺序，将其排成一个队列，每次把CPU分配给队头用户使用，当相应的程序运行结束，则令其出队，再把CPU分配给新的队头用户，直到所有用户任务处理完毕。

补例 3：主机与外部设备之间速度不匹配的问题。

以主机和打印机为例来说明，主机输出数据给打印机打印，主机输出数据的速度比打印机打印的速度要快得多，若直接把输出的数据送给打印机打印，由于速度不匹配，显然不行。解决的方法是设置一个打印数据缓冲区，主机把要打印的数据依此写到这个缓冲区中，写满后就暂停输出，继而去做其它的事情，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，这样利用队列既保证了打印数据的正确，又使主机提高了效率。

课堂练习

- 1、循环队列用数组 $A[0, m-1]$ 存放其元素值，已知其头尾指针分别是 $front$ 和 $rear$ ，则当前队列中的元素个数是（ ）。
- (A) $(rear-front+m)\%m$ (B) $rear-front+1$
(C) $rear-front-1$ (D) $rear-front$
- 2、以数组 $Q[0... m - 1]$ 存放循环队列中的元素，变量 $rear$ 和 $qulen$ 分别指示循环队列中队尾元素的实际位置和当前队列中元素的个数，队列第一个元素的实际位置是（ ）。
- (A) $1+rear - qulen$ (B) $rear - qulen + m$
(C) $m - qulen$ (D) $1 + (rear - qulen + m) \% m$