第三章 栈和队列

第三章　栈和队列

本章学习两种特殊的线性数据结构，它们特殊在定义的操作不同，即插入和删除操作只能在线性表的两端进行。

只能在一端进行的----- 栈

分别在两端进行的----- 队列

第三章　栈和队列

内容提要：

- 3.1栈的定义及逻辑结构
- 3.2栈的顺序存储结构及操作实现
- 3.3栈的链式存储结构及操作实现
- 3.4栈的应用举例
- 3.5队列的定义及逻辑结构
- **3.6**循环队列存储结构及操作实现
- **3.7** 队列的链式存储结构及操作实现

第三章　栈和队列

一、栈的逻辑结构

1、栈（stack)：是一种特殊的线性表（数据元素之间的关系是线性关系），其插入、删除只能在表的一端进行，另一端固定不动。

栈顶（toP）：插入、删除的一端；
栈底（b ottom）：固定不动的一端；
入栈（P ush）：又称压入，即插入一个元素；
出栈（P OP）：又称弹出，即删除一个元素；

第三章　栈和队列
一、栈的逻辑结构

2、说明：设（$a_1, a_2, a_3, \ldots, a_n$）是一个栈

1）表尾称为栈顶，表头称为栈底，即a1为栈底元素，an为栈顶元素;

2）在表尾插入元素的 操作称进栈操作，在表头删除元素的操作称为出栈操作;

3）元素按a1, a2, a3, ..., an 的次序进栈, 第一个进栈的元素一定在栈底，最后一个进栈的元素一定在栈顶, 第一个出栈的元素为栈顶元素;

栈底　　栈顶

（$a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$）

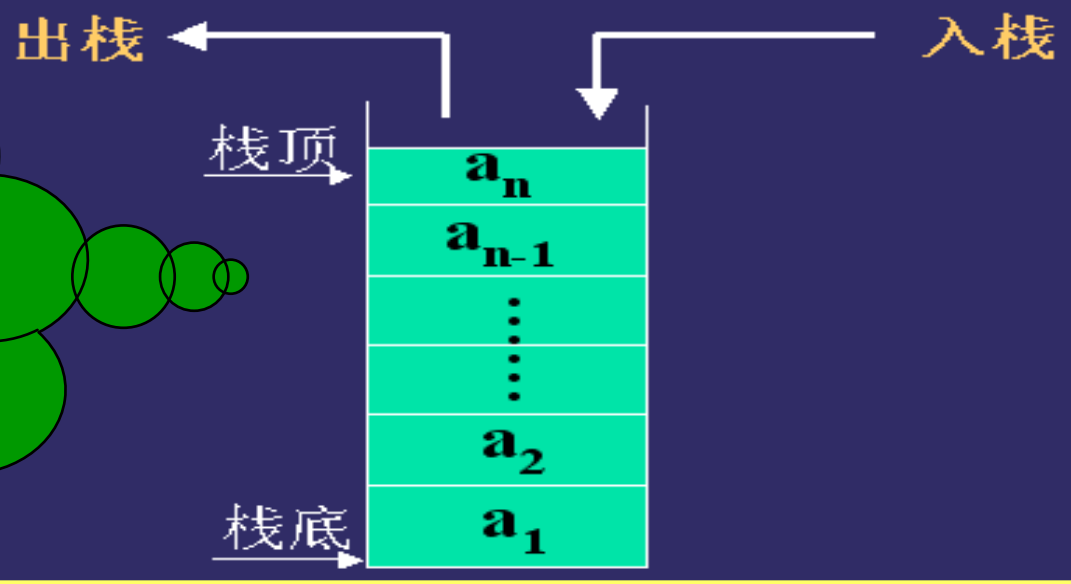
出栈　进栈

第三章　栈和队列

一、栈的逻辑结构

栈的示意图

第一个进栈的元素在栈底，最后进栈的元素在栈顶，第一个出栈的元素为栈顶元素，最后一个出栈的元素为栈底元素



栈特点：由于限制了插入删除只能在一端进行，那么元素的操作顺序有“先进后出”或“后进先出”的特点(**Last In First out-LIFO First In Last out ---FILO**)

第三章　栈和队列

一、栈的逻辑结构

例题：假设有A，B，C，D四个元素；它们入栈的次序为A—＞B—＞C—＞D出栈次序任意，请问不可能得到下面哪些出栈序列？

(1)A B C D (2)D B C A (3)C D B A (4)C B A D

(3)A C B D (6)D B A C (7)A D C B (8)C A B D

第三章　栈和队列
一、栈的逻辑结构

3、栈的基本操作

1） 初始化操作InitStack(&S)

功能：构造一个空栈S；

2） 销毁栈操作DestroyStack(&S)

功能：销毁一个已存在的栈;

3） 置空栈操作ClearStack(&S)

功能：将栈S置为空栈;

4） 取栈顶元素操作GetTop(S, &e)

功能：取栈顶元素，并用e 返回;

第三章　栈和队列
一、栈的逻辑结构

5）进栈操作Push(&S, e)
　功能：元素e进栈;

6）退栈操作Pop(&S, &e)
　功能：栈顶元素退栈，并用e返回;

7）判空操作StackEmpty(S)
　功能：若栈S为空，则返回True，否则返回False;

第三章　栈和队列

二、栈的ADT描述

ADT　Stack　{

data structure:

D={ai | ai∈D0　i=1,2,……　n>=0}

R={N}

N={<ai-1,ai> | ai-1,ai ∈D0　i=2,3,4,…… }

D0是某个数据对象

operations:

InitStack(&S)　DestroyStack(&S)　ClearStack(&S)

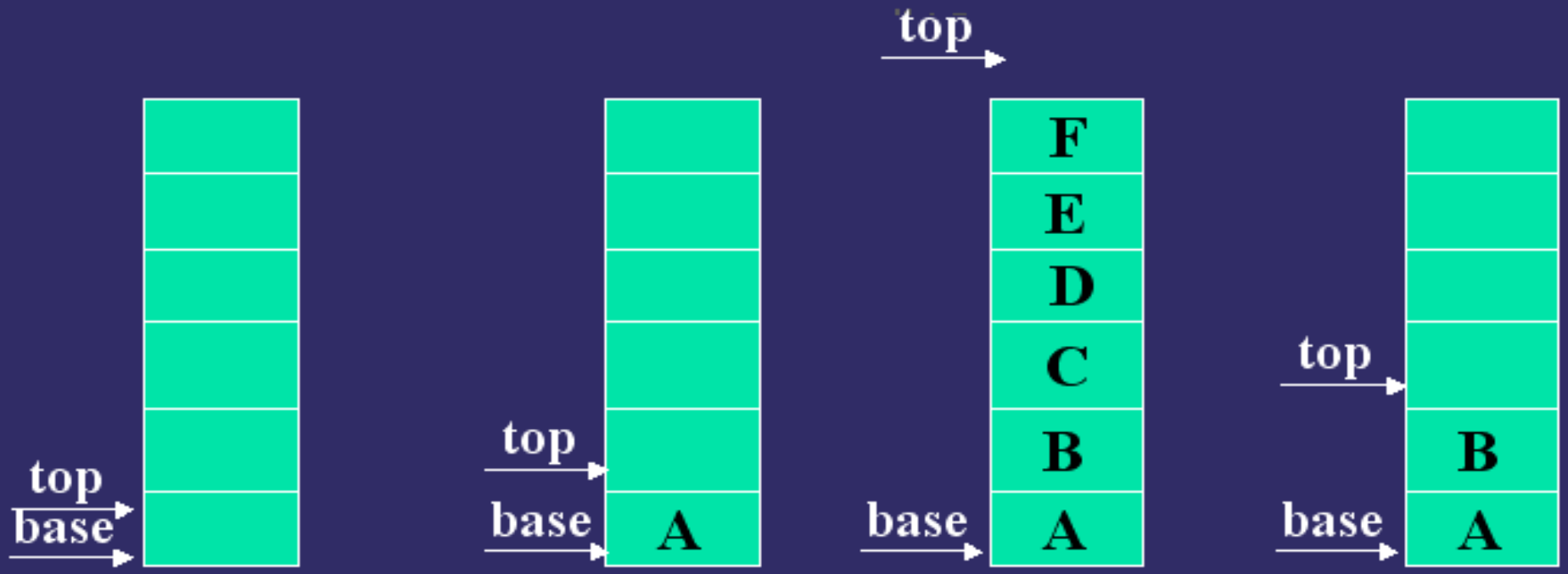
Push(&S, e)　Pop(&S, &e)　GetTop(S, &e)

}　ADT　Stack

§3.2栈的顺序存储结构及实现

1、存储方式：同一般线性表的顺序存储结构完全相同。是利用一组连续的内存单元依次存放自栈底到栈顶的数据元素，栈顶元素的位置由一个称为栈顶指针的变量指示 。

§3.2栈的顺序存储结构及实现



若base = NULL 栈结构不存在

若top 初始值＝base为空栈

top = top 一1出栈

top = top + 1 入栈

栈顶指针始终指向栈顶元素的下一个位置

§3.2栈的顺序存储结构及实现

2、特点：简单、方便，但易产生溢出。

上溢（Overflow）栈已经满，又要压入元素；

下溢（Underflow）栈已经空，还要弹出元素；

注：上溢是一种错误，使问题的处理无法进行下去；而下溢一般认为是一种结束条件，即问题处理结束。

§3.2栈的顺序存储结构及实现

3、 虚拟实现

```
#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;//栈空间基址称为栈底指针，指向栈底位置
SElemType *top //栈顶指针,约定栈顶指针指向栈顶元素的
                //下（后面）一个位置；
int stacksize;      //当前分配的栈空间大小
                    //（以sizeof(SElemType)为单位）
}SqStack;// SqStack:：结构类型名
```

§3.2栈的顺序存储结构及实现

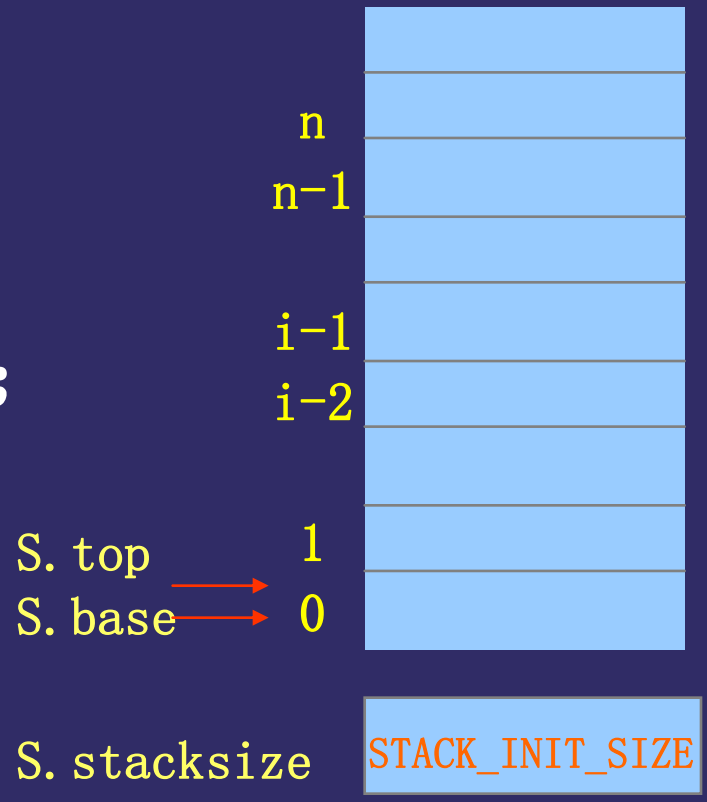
二　顺序栈基本操作的算法

1）初始化操作

InitStack_Sq(SqStack &S)

参数：S是存放栈的结构变量；

功能：建一个空栈S；



初始化操作图示

§3.2栈的顺序存储结构及实现

```
Status InitStack_Sq(SqStack &S)  {
     //构造一个空栈S
    S.base=(SElemType * )malloc(STACK_INIT_SIZE
                                * sizeof(SElemType));
    //为顺序栈动态分配存储空间
    if (! S. base) exit(OVERFLOW);    //存储分配失败
    S.top=S.base;
    S.stacksize=STACK_INIT_SIZE;
    return OK;              }//InitStack_Sq
```

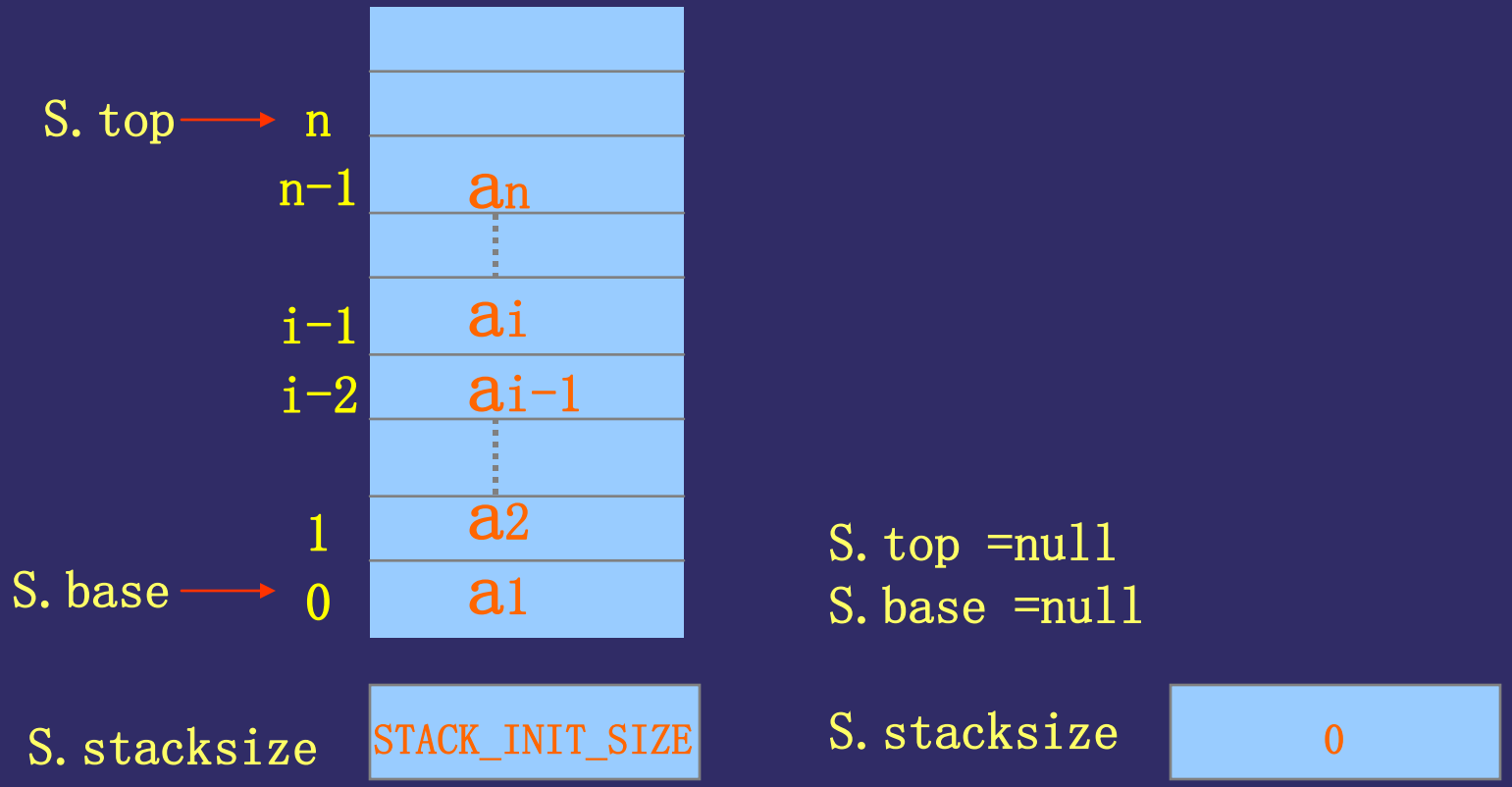
§3.2栈的顺序存储结构及实现

2） 销毁栈操作 DestroyStack_Sq(SqStack &S)

功能：销毁一个已存在的栈;

```
Status DetroyStack_Sq ( SqStack &S) {
   If (!S.base) return ERROR;  //若栈未建立（尚未分配栈空间）
    free (S.base);                    // 回收栈空间
    S.base = S.top = null;
    S.stacksize = 0;
    return OK;
        } // DetroyStack_Sq
```

§3.2栈的顺序存储结构及实现



销毁栈操作图示

§3.2栈的顺序存储结构及实现

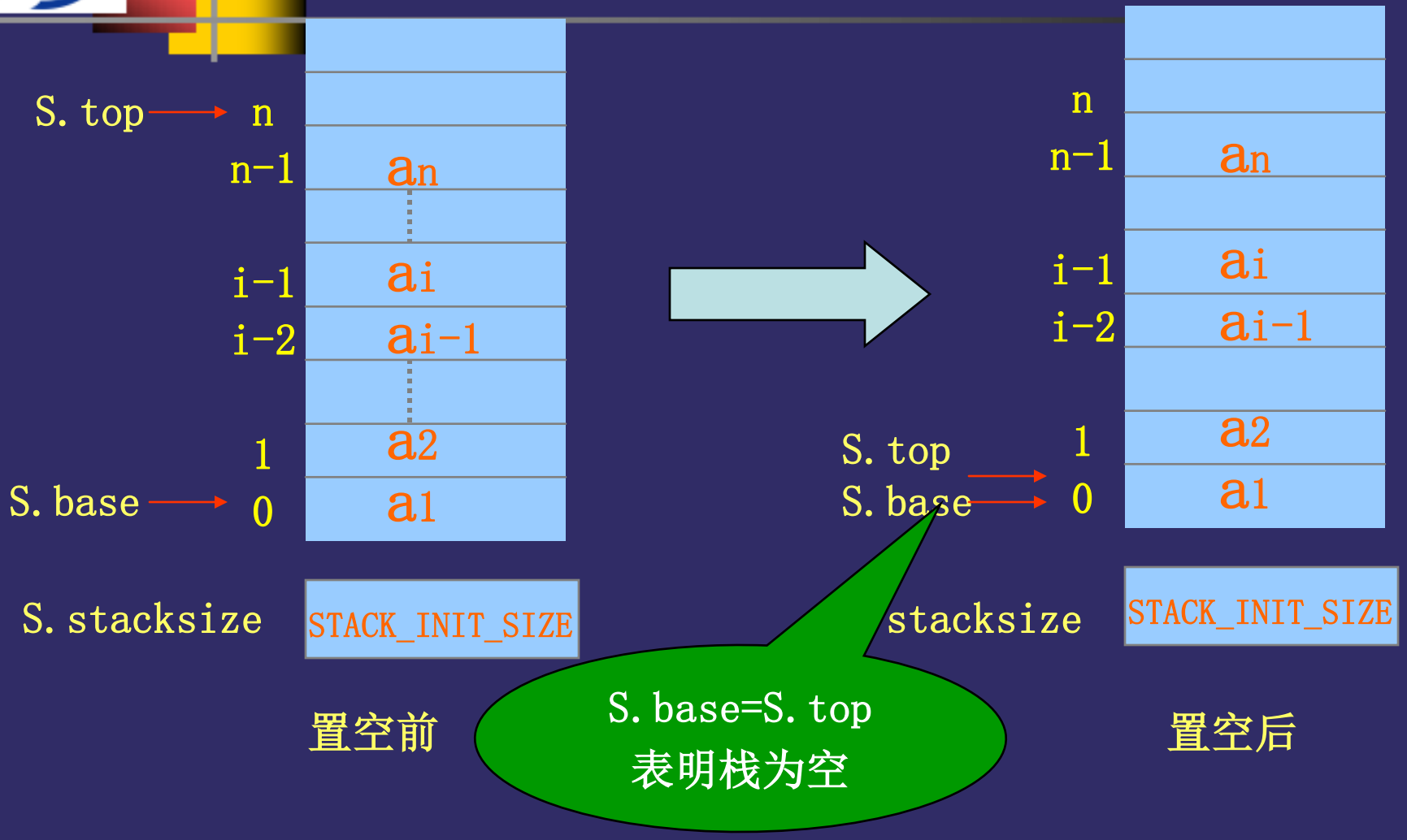
3）置空栈操作ClearStack_Sq(SqStack &S)

功能：将栈S置为空栈

算法

```
Status ClearStack_Sq ( SqStack &S) {
  If (!S.base) return ERROR;  // 若栈未建立（尚未分
                              //配栈空间）
  S.top = S.base ;
  return OK;
}// ClearStack_Sq
```

§3.2栈的顺序存储结构及实现



置空栈操作图示

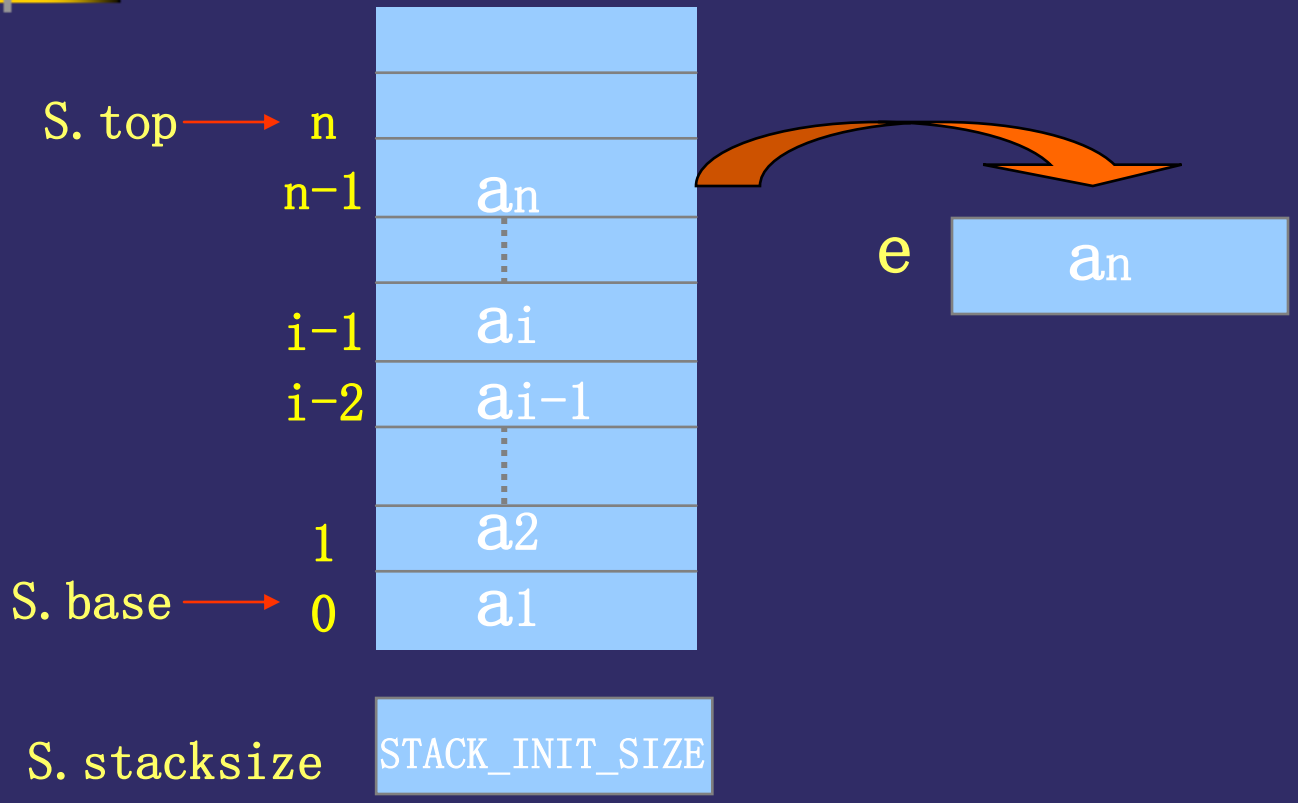
§3.2栈的顺序存储结构及实现

4） 取栈顶元素操作GetTop_Sq(SqStack S, SElemType &e)

功能：取栈顶元素，并用e返回;

```
Status GetTop_Sq(SqStack S, SelemType &e) {
    // 若栈不空，则用e返回S的栈顶元素，并返回
//OK；否则返回ERROR
    if (S.top= =S.base)  return ERROR; //若栈为空
    e = * (S.top-1);
    return OK;
}//GetTop_Sq
```

§3.2栈的顺序存储结构及实现



取栈顶元素操作图示

§3.2栈的顺序存储结构及实现

5）进栈操作Push(SqStack &S, SElemType e)

功能：元素e 进栈；

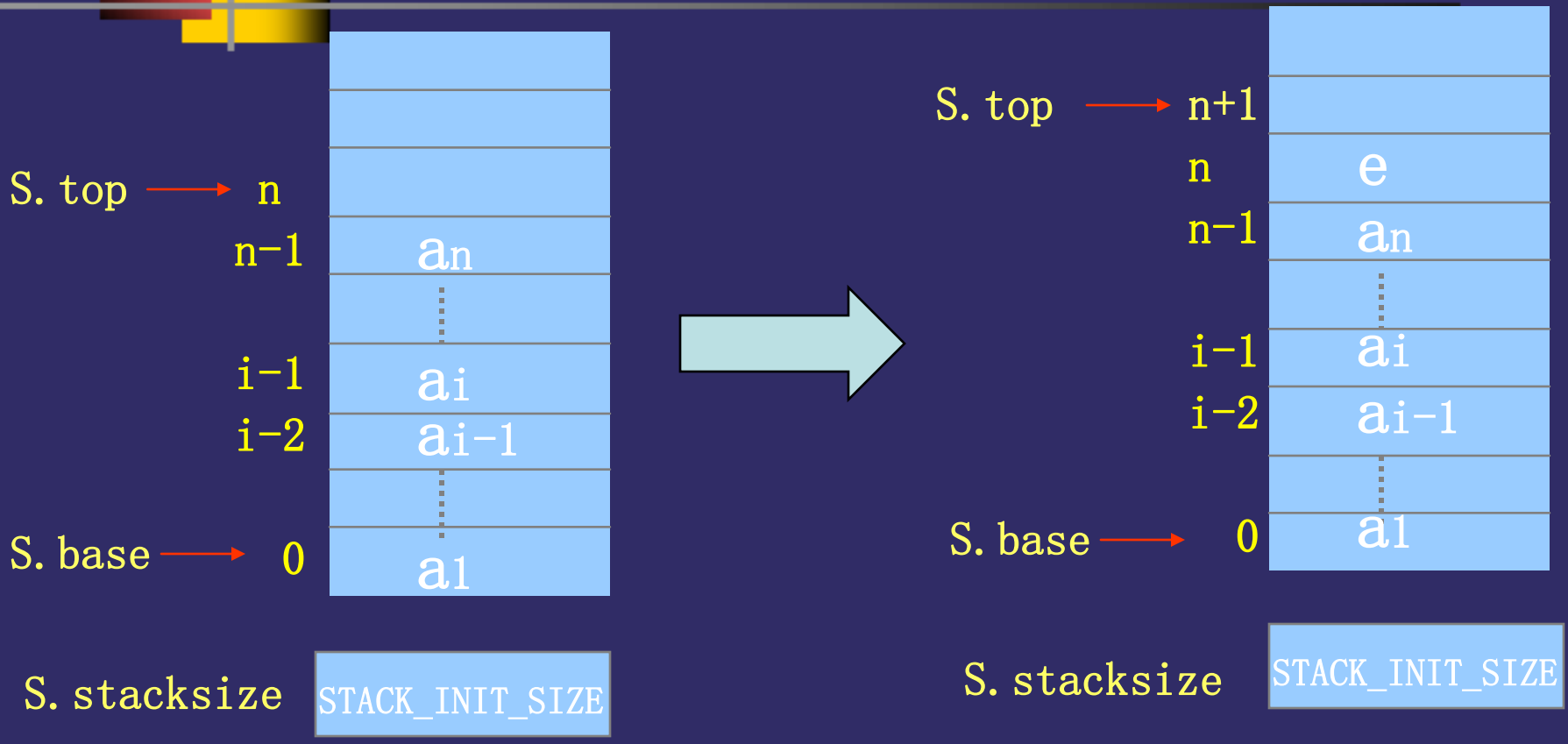
进栈操作主要步骤：

1）S是否已满， 若栈满，重新分配存储空间；

2）将元素e 写入栈顶；

3）修改栈顶指针，使栈顶指针指向栈顶元素的下（后面）一个位置；

§3.2栈的顺序存储结构及实现



e 进栈前

e 进栈后

进栈操作图示

§3.2栈的顺序存储结构及实现

进栈操作算法：

```
Status Push(SqStack &S, SElemType e) {
  //将元素e插入栈成为新的栈顶元素,要考虑上溢情况
  if (S.top-S.base>=S.stacksize) {//栈满，追加存储空间
        S.base= (SElemType * )realloc(S.base,
    (S.stacksize +STACKINCREMENT) sizeof(ElemType));
     if (! S. base) exit(OVERFLOW);  //存储分配失败
     S.top=S.base+S.stacksize;
     S.stacksize+=STACKINCREMENT;  }
  *S.top++=e;     //元素e 插入栈顶，修改栈顶指针
  return OK;     }//Push
```

```
*S.top=e;
S.top=S.top+1;
```

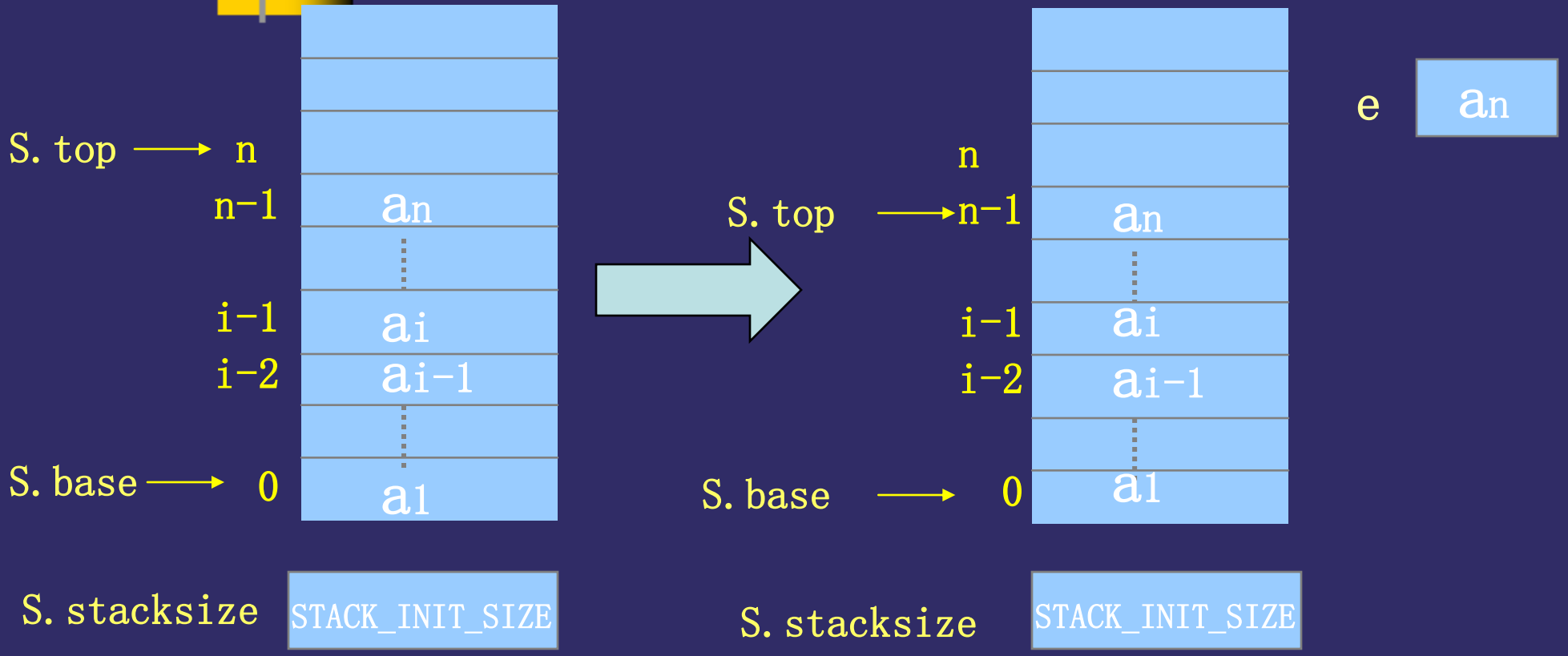
§3.2栈的顺序存储结构及实现

6）出栈操作Pop(SqStack &S, SElemType &e)
功能：栈顶元素退栈，并用e 返回;

```
Status Pop(SqStack &S, SElemType &e) {
  //若栈不空，则删除S的栈顶元素，用e 返回其值并返回OK
//否则返回ERROR
    if (S.top= = S.base) return ERROR;  //栈空，返回ERROR
    e = * --S.top;          //删除栈顶元素，用e 返回其值，并修
                            //改栈顶指针
    return OK;
} //Pop
```

→ { S.top=S.top-1;
  e=*S.top;

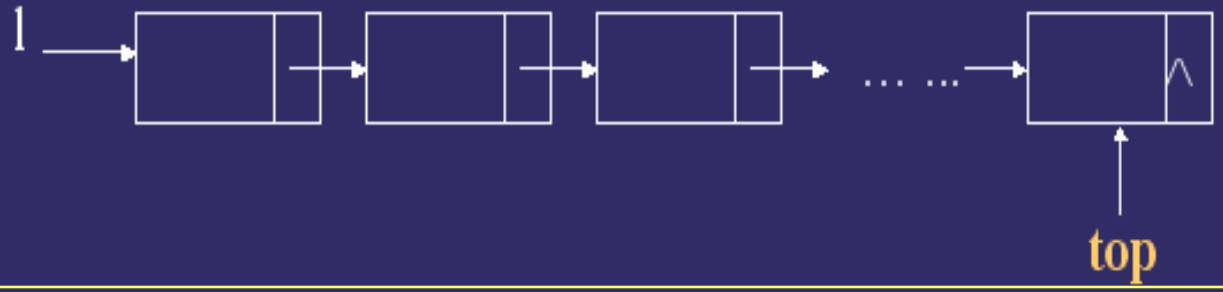
§3.2栈的顺序存储结构及实现



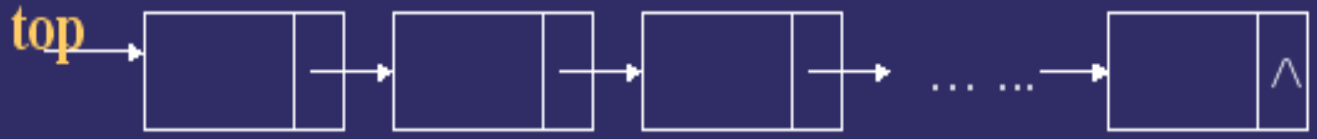
出栈操作图示

§3.3栈的链式存储结构及实现

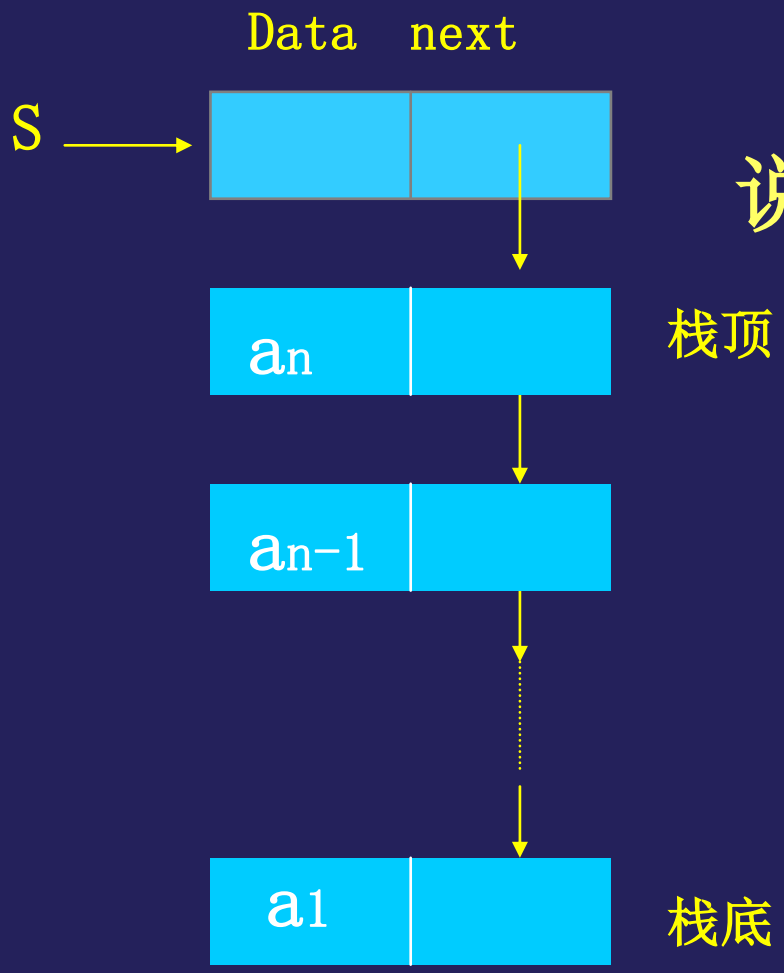
1、存储方式：同一般线性表的单链式存储结构完全相同。但是应该确定链表的哪端对应于栈顶，如果链表尾作为栈顶，则入、出栈操作的时间复杂性为O(n).



如果链表头作为栈顶，则入、出栈操作的时间复杂性为O(1)，所以，一般把链表头作为栈顶。



栈的链式存储结构，也称链栈，如图所示：



说明：

1）链式栈可用线性链表表示；
2）链式栈的栈顶指针就是链表的头指针；
3）进栈操作就是在该线性链表第一个元素结点之前插入一个新结点，出栈操作就是删除链表的第一个元素结点。

2、特点：减小溢出，提高空间利用率。只有系统没有空间了，才回溢出；

§3.3栈的链式存储结构及实现

栈链式存储结构下各运算的虚拟实现

1、栈初始化

S =(LStack *) malloc (sizeof (LStack)) ;

s . next = NULL ;

时间复杂性：O (1)

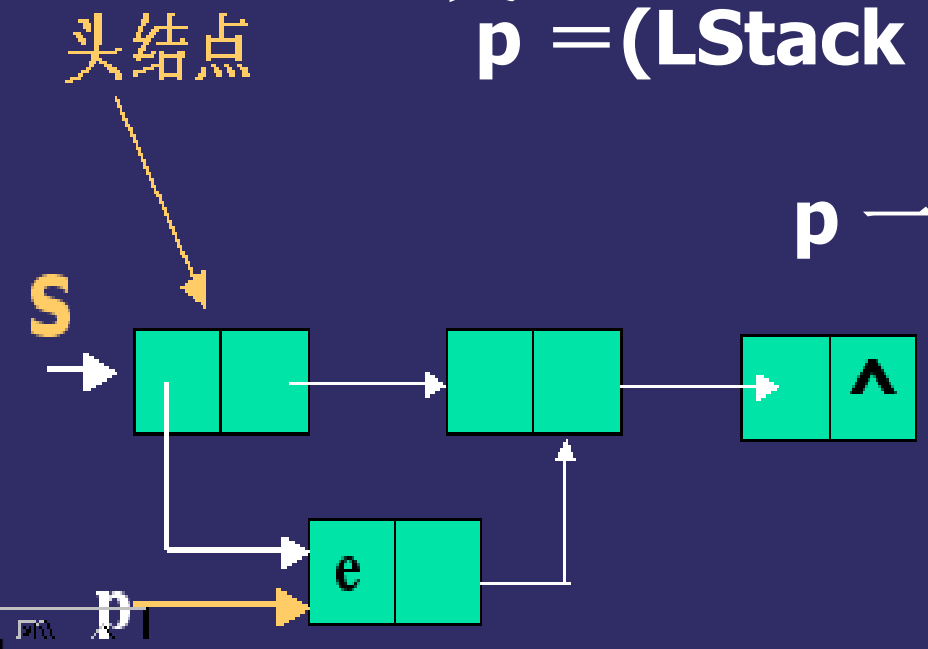
2、入栈

p =(LStack *) malloc (sizeof (Lstack));

P —>data=e ;

p —>next=S —>next ;

S —>next=P ;



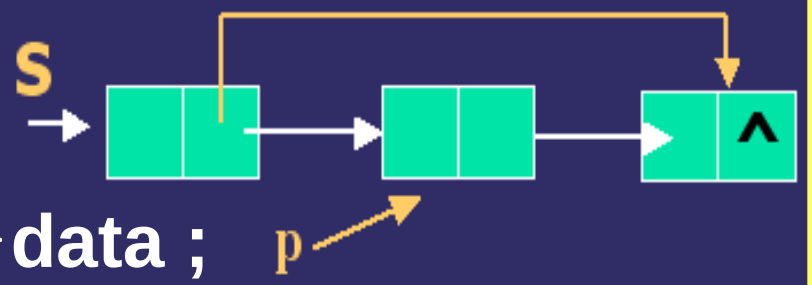
时间复杂性：O (l)

§3.3栈的链式存储结构及实现

栈链式存储结构下各运算的虚拟实现

3、出栈 //删除栈顶元素，并用e 返回其值

```
if ( S ->next=NULL )
return (OVERfLOW)//下溢
else  p = s ->next ; e=p ->data ;
 s ->next = p ->n ext ;
free(p);
```



4、判断栈是否空

```
if ( S ->next = NULL )
return TRUE;
```

§3.3栈的链式存储结构及实现

栈链式存储结构下各运算的虚拟实现

其它操作同线性表！
栈的操作是线性表操作的特例！

顺序表与链表的比较:

顺序栈需要固定空间长度，存在溢出问题，但链式栈的每个元素都有指针域，增加了空间开销。另外，求链栈的大小时间复性为O(n)

小结

1.栈是限定仅能在表尾一端进行插入、删除操作的线性表；

2. 栈的元素具有后进先出的特点；

3. 栈顶元素的位置由一个称为栈顶指针的变量，进栈、出栈操作要修改栈顶指针；

§3.4 栈的应用举例

栈结构具有后进先出的特征，使栈成为程序设计中的有用工具，本节介绍栈的几个简单应用的实例。

例1　数制转换

1）十进制N和其它进制数的转换是计算机实现计算的基本问题,其解决方法很多,其中一个简单算法基于下列原理:

N=(N div d)*d+N mod d

(其中:div为整除运算,mod为求余运算)

§3.4 栈的应用举例

2）例如 $(1348)_{10}=(2504)_8$，其运算过程如下：

$$N = (N\text{div}8)_{10} \times 8 + N \bmod 8$$

N：十进制数，div：整除运算，mod：求余运算；

	N	N div 8	N mod 8	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

§3.4 栈的应用举例

由上述求解过程可以看出，在计算过程中是从低位到高位顺序产生八进制数的各个数位，而显示时按照我们的习惯是从高位在前，低位在后，即按从高位到低位的顺序输出，即后计算出的（高）位数先输出，具有后进先出的特点，因此可将计算过程中得到的八进制数顺序进栈，出栈序列打印输出的就是对应的八进制数。

3） 算法

```
void conversion( ){  //对于输入的任意一个非负十
//进制整数，打印输出与其等值的八进制数
      InitStack(S);                  //建空栈
      scanf(“%d”, N);    //输入一个非负十进制整数
      while(N) {                 // N不等于零循环
            Push(S, N % 8);  // N/8第一个余数进栈
            N=N/8    };          //整除运算
      while(! StackEmpty) {
      //输出存放在栈中的八制数位。
            Pop(S, e);
            Printf(“%d”, e);
         }
 } //conversion
```

3.4 栈的应用举例

例2 表达式求值

1） 表达式的构成 操作数+运算符+界符（如括号）

2） 表达式的求值：

例 5+6× (1+2)- 4

按照四则运算法则，上述表达式的计算过程为：

5+6× (1+2)- 4=5+6× 3- 4 = 5+18-4= 23-4=19

表达式中运算符的运算顺序为： + ， × ， + ， -

如何确定运算符的运算顺序？

3.2 栈的应用举例

3） 算符优先关系表

表达式中任何相邻运算符θ_1、θ_2的优先关系有：

$\theta_1<\theta_2$：θ_1的优先级低于θ_2

$\theta_1=\theta_2$：θ_1的优先级等于θ_2

$\theta_1>\theta_2$：θ_1的优先级高于θ_2

由四则运算法则，可得到如下的算符优先关系表：

3.2 栈的应用举例

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

注： θ1 θ2是相邻算符，θ1在左 θ2在右

算符间的优先关系表

3.4 栈的应用举例

4) 算符优先算法

我们再来分析一下表达式5+4× (1+2)- 6 计算过程：

5+4× (1+2)- 6

从左向右扫描表达式：

遇操作数——保存；

遇运算符号θ_j——与前面的刚扫描过的运算符θ_i比较

若$\theta_i<\theta_j$ 则保存θ_j，（因此已保存的运算符的优先关系为 $\theta_1<\theta_2<\theta_3<\theta_4$ 。。。）

若$\theta_i>\theta_j$ 则说明θ_i是已扫描的运算符中优先级最高者，可进行运算；

若$\theta_i = \theta_j$ 则θ_i为（，θ_j为），说明括号内的式子已计算完，需要消去括号；

后面也许有优先级更大的算符

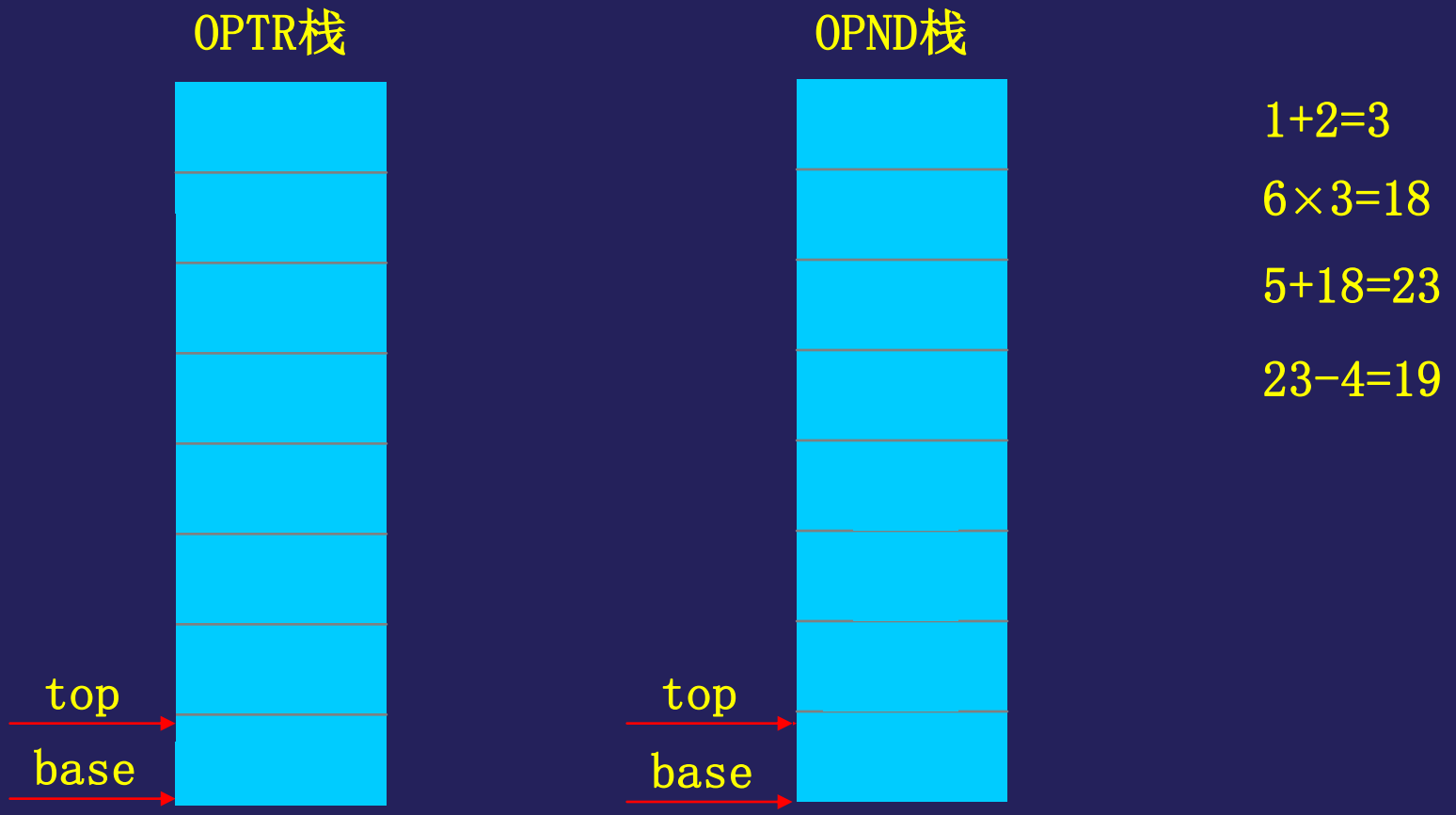
3.4 栈的应用举例

我们看到：进行运算的算符θi是当前扫描过的运算符中优先级最高者，同时，也是到当前最后被保存的运算符，由此可见，可以利用两个栈分别保存扫描过程中遇到的操作数和运算符。

3.4 栈的应用举例

表达式求值示意图：5+6×(1+2)-4=19

读入表达式过程：5+6×(1+2)-4#



3.4 栈的应用举例

在算符优先算法中，建立了两个工作栈。一个是OPTR栈，用以保存运算符,一个是OPND栈，用以保存操作数或运算结果。

```
operandType EvaluateExpression( ){
  //算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和运算数栈，OP为运算符集合。
  InitStack(OPTR);    Push (OPTR, #);
  InitStack(OPND);   c=qetchar( );
  While(c!=‘ #’  ||  GetTop(OPTR)!=‘#’){
      if (! In (c, OP)) {Push(OPND,  c); c=getchar( )}
  //不是运算符则进栈,In(c, OP)判断c是否是运算符的函数
        else
```

续
```
    switch (Precede(GetTop(OPTR), c) {
        case <:      // 新输入的算符c优先级高，c进栈
            Push(OPTR, c);  c=getchar( );  break;
        case =:      // 脱括号并接收下一字符
            Pop(OPTR, x);  c=getchar( ); break;
        case >:  //新输入的算符c优先级低，即栈顶算符优先权
                 //高,出栈并将运算结果入栈OPND
            Pop(OPTR, theta);
            Pop(OPND, b);  Pop(OPND, a);
            Push(OPND, Operate(a, theta, b));
            break;
    } // switch
  } //while
  return GetTop(OPND);
} //EvaluateExpression
```

栈与递归(补充)

由上看到：应用中如果处理数据处理过程具有后进先出的特性，可利用栈来实现数据处理过程。下面我们将看到可以用栈来实现递归。

1．什么是递归

递归是一个很有用的工具，在数学和程序设计等许多领域中都用到了递归。递归定义：简单地说，一个用自己定义自己的概念,称为递归定义。

例 **n!= 1 2 3 4 (n-1) n**

n!递归定义 **n!= 1** 当 **n=0**时

n!= n (n-1)! 当**n> 0**时

用(n-1)!定义n!

栈与递归

2.递归函数：一个直接调用自己或通过一系列调用间接调用自己的函数称为递归函数。

```
A（  ）{
   …

   A( ) ;

   …
}
```

A 直接调用自己

```
B( ) {
   …

   C( );

   …
}
```

```
C( ) {
      …

      B( );

      …
   }
```

B间接调用自己

栈与递归

递归是程序设计中的有用工具，下面举例说明递归算法的编写和执行过程。

2．递归算法的编写

1）将问题用递归的方式描述（定义）

2）根据问题的递归描述（定义）编写递归算法

问题的递归描述（定义）

递归定义包括两项

基本项（终止项）：描述递归终止时问题的求解；

递归项：将问题分解为与原问题性质相同，但规模较小的问题；

栈与递归

例1　编写求解 阶乘n! 的递归算法

首先给出阶乘n! 的递归定义

n!的递归定义

基本项：　n!=1　　当 n=1

递归项：　n!=n (n-1)!　　当 n> 1

有了问题的递归定义，很容易写出对应的递归算法：

```
int fact (int n) {//算法功能是求解并返回n的阶乘
    If（n=1） return（1）；
    Else return（n*fact (n-1)）；
}//fact
```

栈与递归

例2. 编写求解Hanci塔问题的递归算法

有三个各为X，Y，Z的塔座，在X项有n个大小不同依小到大编号为1，2…n的圆盘。 现要求将X上的n个圆盘移至Z上，并仍以同样顺序叠放， 圆盘移动时必须遵守下列原则：

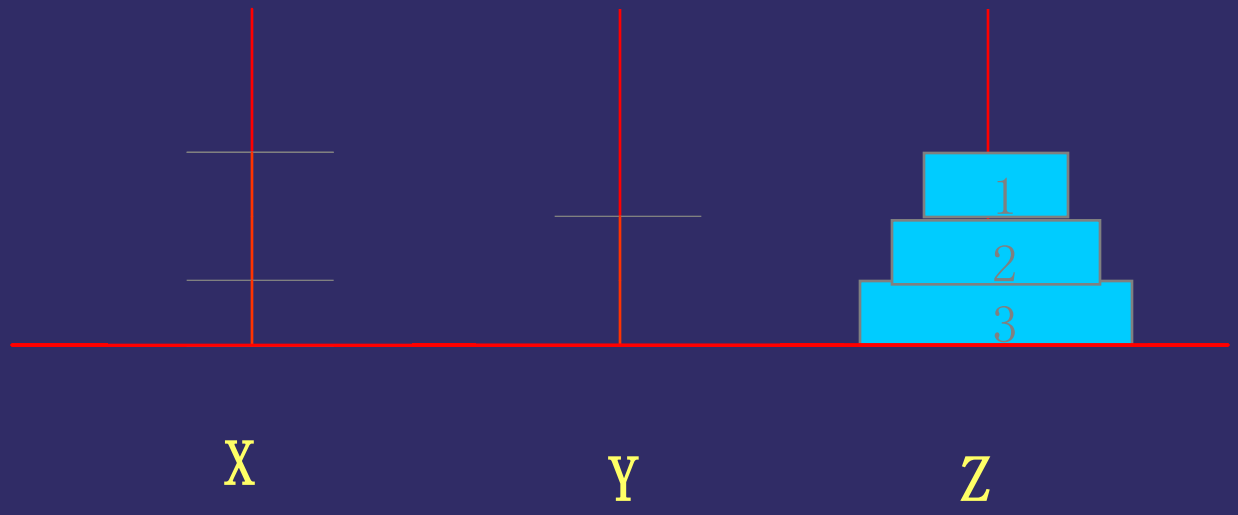
1）每次移动一个盘子；

2）圆盘可以放在X，Y，Z中的任一塔座上；

3）任何时刻都不能将较大的圆盘压放在较小圆盘之上；

例 n=3时圆盘移动的过程如下图所示：

栈与递归



栈与递归

首先给出求解Hanci塔问题 的递归描述（定义）

基本项： n=1时，将n号圆盘从X移至Z；

递归项： n>1时，

将X上1——n-1号圆盘移至Y；

将X上的n号圆盘从X移至Z；

将Y上1——n-1号圆盘从Y移至Z；

将规模为n的问题的求解归结为规模为n-1的问题的求解

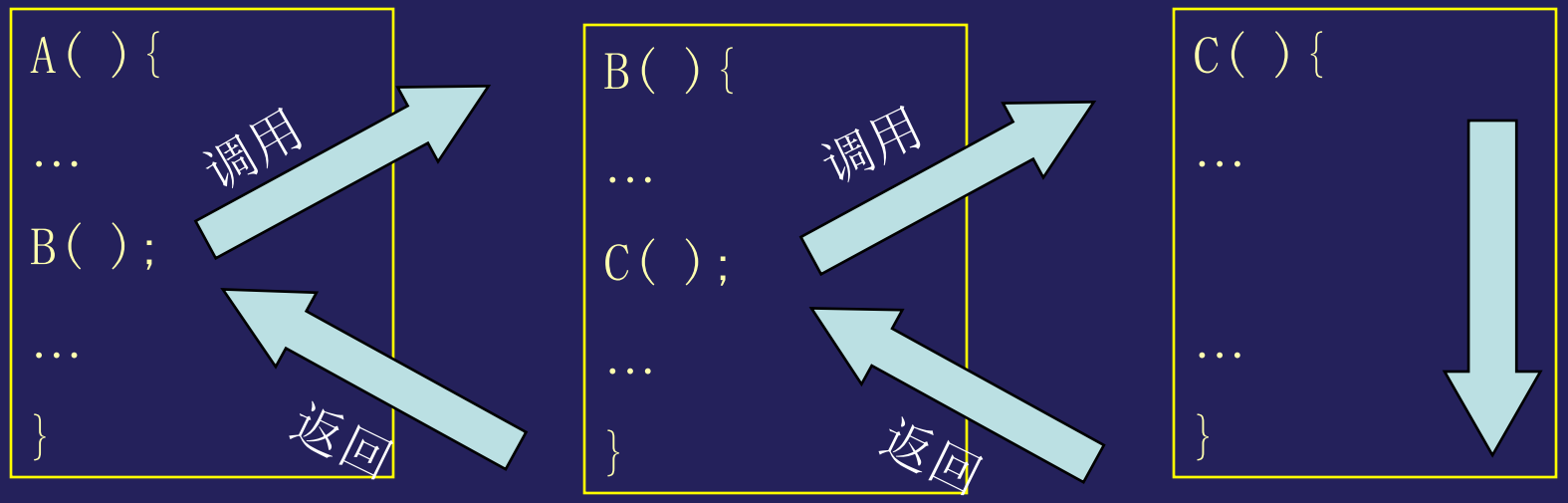
有了问题的递归定义，很容易写出对应的递归算法：

```
void hanoi (int n, char x, char y, char z)
//将塔座x上按直径由小到大且自上而下编号为1至n的n个圆盘按规则搬到塔座z上，y可用作辅助塔座。
{
    if (n= =1)
     move(x,1,z);      //将编号为1的圆盘从x移动z
    else {
            hanoi(n-1, x, z, y);
//将x上编号为1至n-1的圆盘移到y, z作辅助塔
     move(x, n, z);    //将编号为 n的圆盘从x移到z
            hanoi(n-1, y, x, z);
 //将y上编号为1至n-1的圆盘移到z，x作辅助塔
    }
}
```

栈与递归

3 递归函数的实现

先看一般函数的调用机制如何实现的。



函数调用顺序 A ⟶ B ⟶ C

函数返回顺序 C ⟶ B ⟶ A

后调用的函数先返回

函数调用机制可通过栈来实现

计算机正是利用栈来实现函数的调用和返回的

栈与递归

我们看一下n=3 阶乘函数fact(n)的执行过程

```
Main( ) {
   int n=3, y;
L  y= fact(n);
}
```

调用 ↓ ↑ 返回 6

```
int fact (n) {
    If（n=1） return（1）;
    Else
L1      return（n*fact (n-1)）;
}//fact
```

n=3

调用 →

```
int fact (int n) {
    If（n=1） return（1）;
    Else
L1      return（n*fact (n-1)）;
}//fact
```

n=2

调用 →

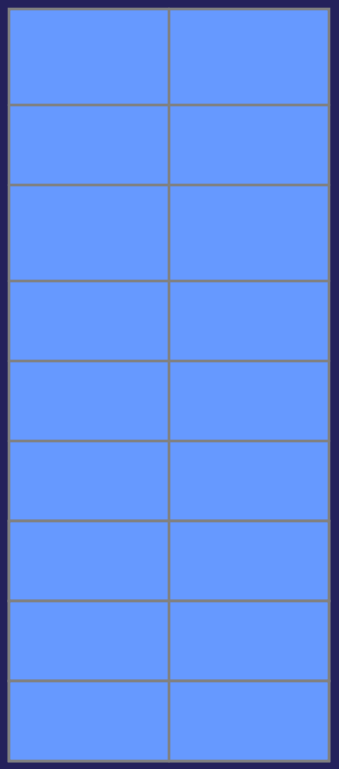
```
int fact (int n) {
    If（n=1） return（1）;
    Else
L1      return（n*fact (n-1)）;
}//fact
```

n=1

调用 ↑ ↓ 返回 1

← 返回2

返回地址 实参



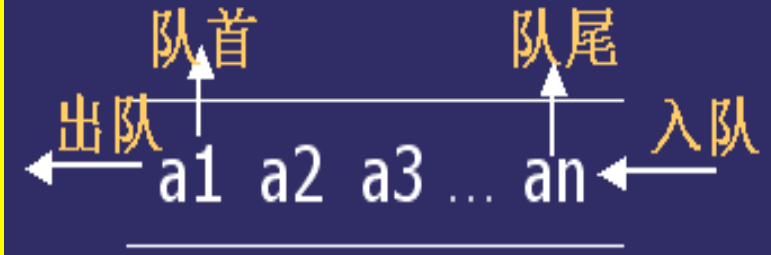
注意递归调用中栈的变化

§3.5队列的逻辑结构及操作实现

一　队列的逻辑结构

1、队列（Queue）： 是一种特殊的线性表(数据元之间的关系是线性关系.其插入、删除分别在表的两端进行，一端只能插入、另一端只能删除。)

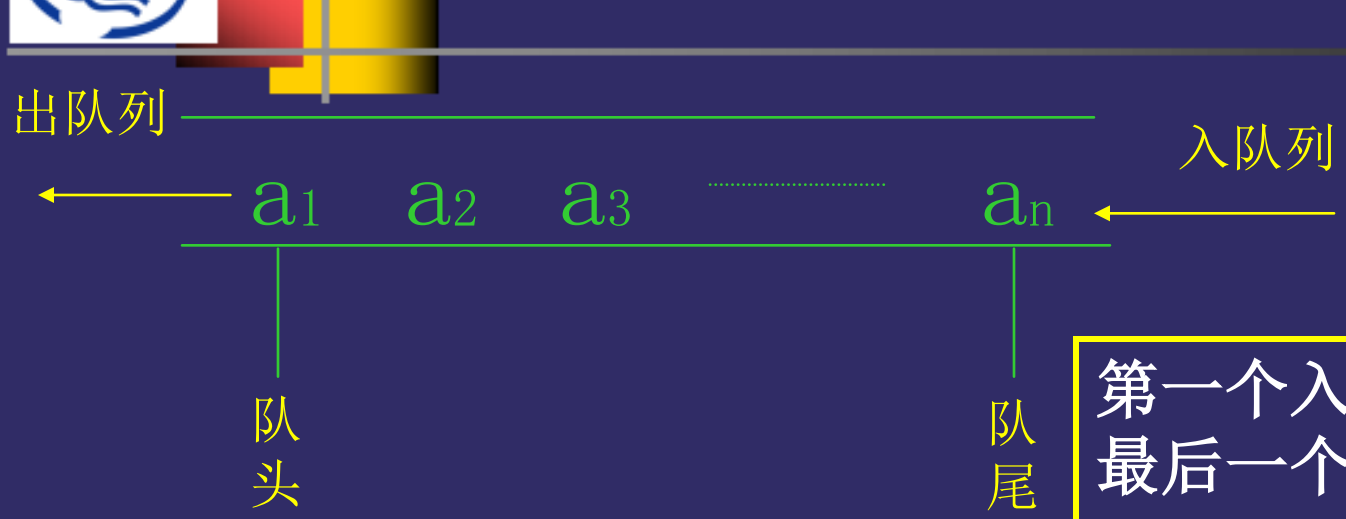
队首（**front**）：进行删除的一端；
队尾（**rear**）：进行插入的一端；
入队：在队尾插入一个元素；
出队：在队首删除一个元素；



2、队列的特点： 由于限制了插入、删除分别在两端进行，那么元素的操作顺序有“先进先出”或“后进后出”的特点（**First In First Out--FIFO Last In Last Out -- LILO**）

§3.5队列的逻辑结构及操作实现

一　队列的逻辑结构



队列的示意图

队列的特点
先进先出

第一个入队的元素在队头，
最后一个入队的元素在队尾，
第一个出队的元素为队头元素，
最后一个出队的元素为队尾元素

队列类似于日常的排队，新来的人站在队尾，队头的人进行事务处理后离队。

队列通常设置两个变量分别指示队头元素位置和队尾元素的位置，这两个变量分别称为队头指针、队尾指针；

§3.5队列的逻辑结构及操作实现

一　队列的逻辑结构

二　队列的基本操作

1）初始化操作InitQueue(&Q)

功能：构造一个空队列Q；

2）销毁操作DestroyQueue(&Q)

功能：销毁已存在队列Q；

3）置空操作ClearQueue(&Q)

功能： 将队列Q置为空队列 ；

4）判空操作QueueEmpty(Q)

功能：若队列Q为空，则返回True，否则返回False；

§3.5队列的逻辑结构及操作实现

一　队列的逻辑结构

5）取队头元素操作GetHead(Q,&e)

功能：取队头元素，并用e返回；

6）入队操作EnQueue(&Q, e)

功能：将元素e插入Q的队尾；

7）出队操作DeQueue(&Q，&e)

功能：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；

§3.5队列的逻辑结构及操作实现

二 队列ADT描述

ADT Queue

{ data structure:

D={ai | ai∈D0 i=1,2,..... n>=0}

R={N}

N={<ai-1,ai> | ai-1,ai ∈D0 i=2,3,4,..... }

D0是某个数据对象

operations:

InitQueue(&Q) DestroyQueue(&Q) ClearQueue(&Q)

QueueEmpty(Q) GetHead(Q,&e) EnQueue(&Q, e)

}

§3.5队列的逻辑结构及操作实现

三 队列ADT应用举例

打印机队列管理：在一台打印机共享使用时，任何时刻它只能处理一个用户的打印请求。打印任务的组织就用一个队列来组织（先请求的，先处理）。

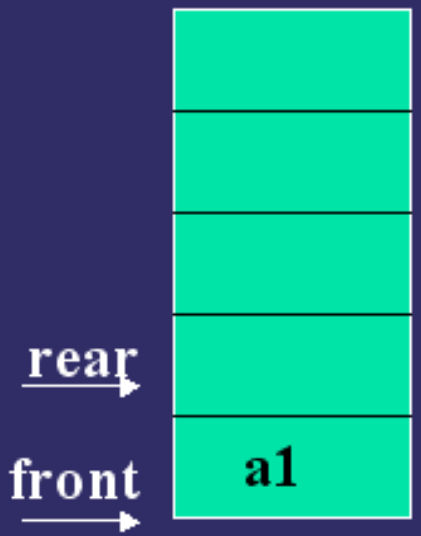
当用户发出打印请求时，把打印任务*入队*；
当打印机空闲时，从打印队列中*出队*一个任务；
当打印机阻塞时,系统管理员可以*清空队列.*

§3.5队列的逻辑结构及操作实现

四 队列的顺序存储结构

1、存储方式：同一般线性表的顺序存储结构完全相同。但是由于在两端操作，设两个指示器，(rear 和front ）分别指示队列的尾和首。

特别约定:头指针始终指向队列头元素，而尾指针始终指向 队列尾元素的下一位置！



空队列：rear = front = 0

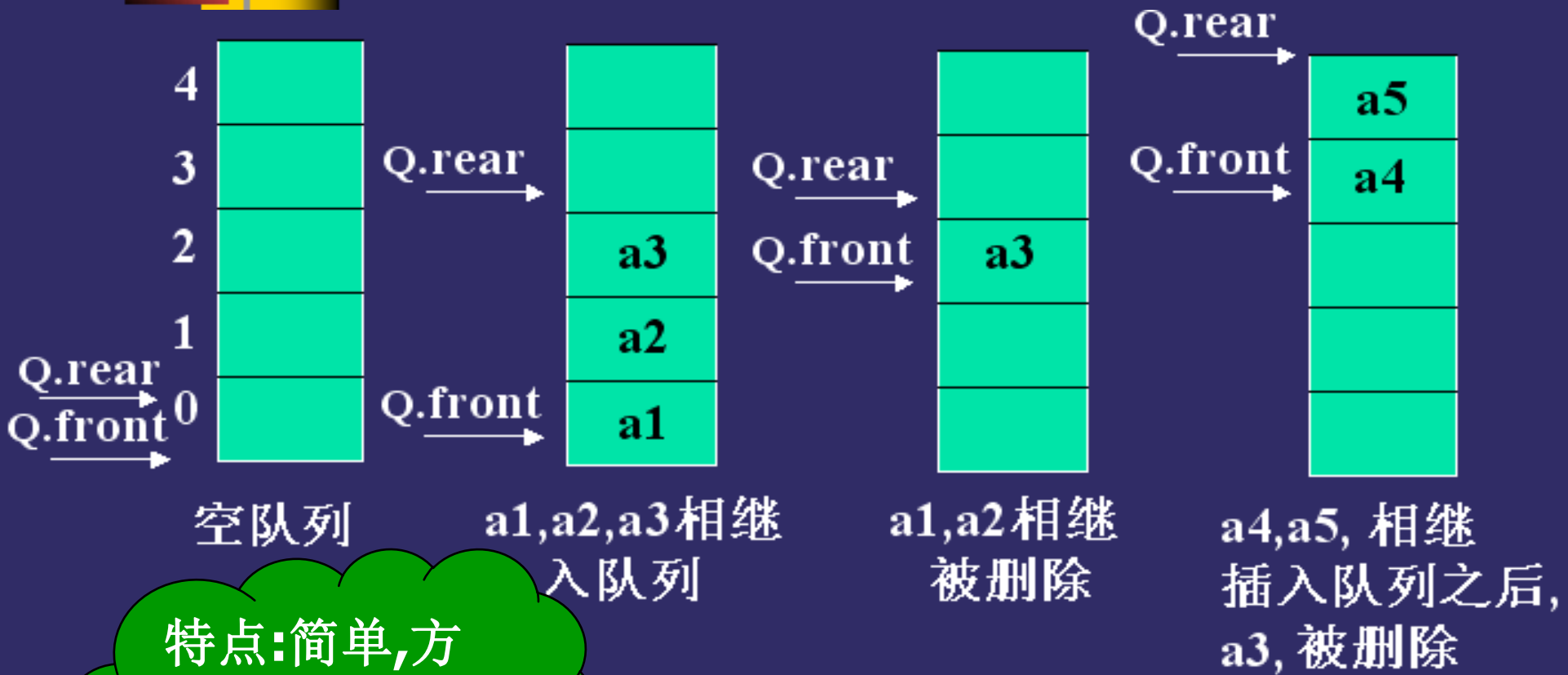
入队：rear =rear + 1

出队：front = front + 1

队列空：front = rear

§3.5队列的逻辑结构及操作实现

四 队列的顺序存储结构



特点:简单,方便,但是易产生*假*溢出.

§3.5队列的逻辑结构及操作实现

四 队列的结构定义

```
// ------- 队列顺序结构的定义--------
#define MAXQSIZE 100
typedef  struct
 {
        QElemType   base[MAXQSIZE];   // 静态数组
                int   front;                    // 队列头指针
                int   rear;                     // 队列尾指针
 }    SqQueue;
```

只是称为指针，实现时不一定用指针变量

§3.5队列的逻辑结构及操作实现

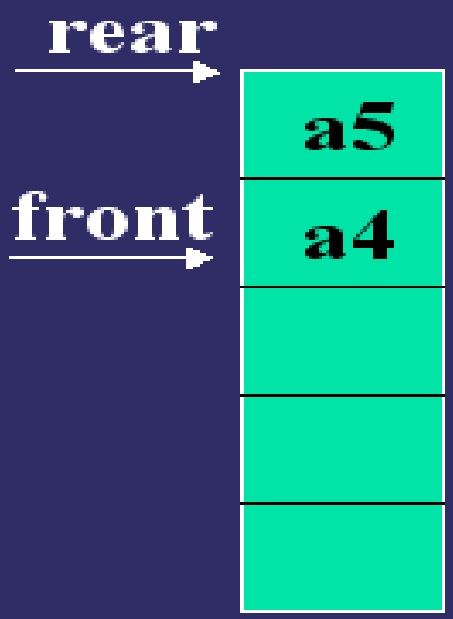
五 队列的简单操作

(1）初始化 Q .front = Q.rear =0 ;
(2）入队Q .base[Q .rear]＝e ;
Q .rear + + ;
(3）出队Q .front + + ;
(4）判空Q .front = = Q .rear ;
(5）求队长Q .rear-Q .front

§3.5 队列的顺序存储表示及操作的实
六 顺序队列存在的问题及解决

可以看出：当入队一个元素时，可能会出现假溢出现象.即：不能入队，但空间并没有使用完！

解决的办法：



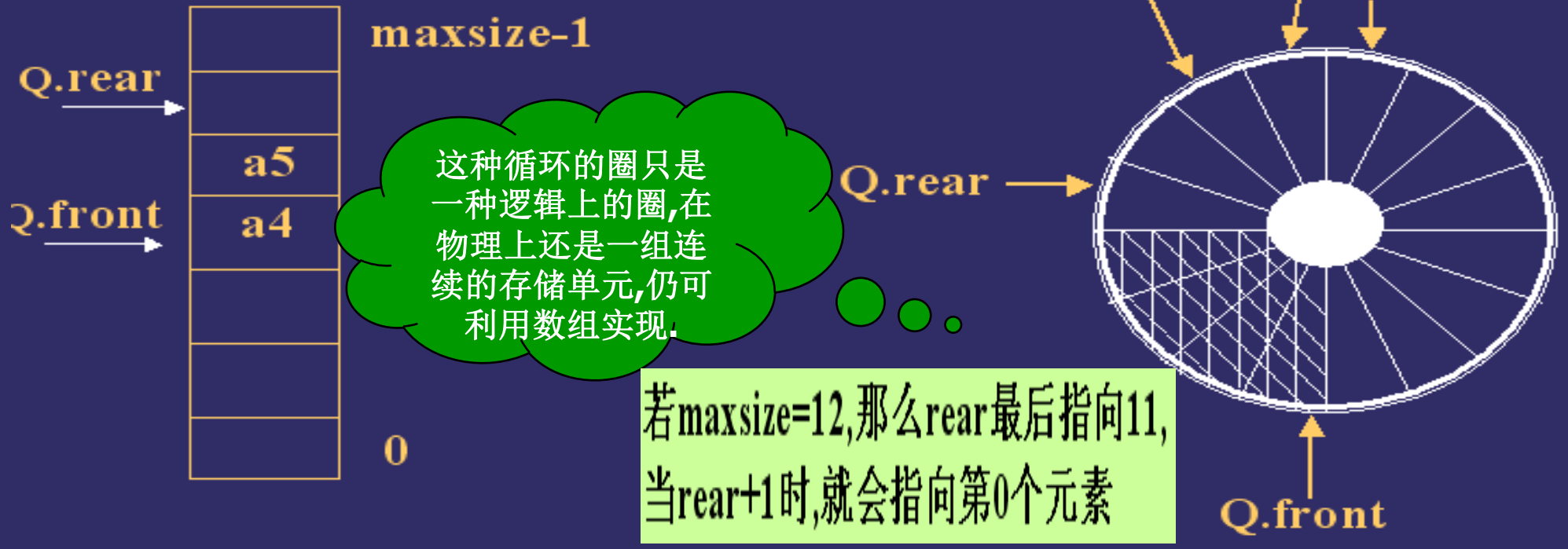
（1）平移，一旦发生假溢出，把队列中所有元素移到队列开头．效率低．

（2）使用动态数组，当产生假溢出或真溢出时，都重新分配一个更大的空间．（不可取）

（3）使用环数组，即将顺序存储的空间视为一个环空间。

§3.6 循环队列存储结构及操作实现

1．方式：将顺序队列臆造为一个环状的空间，如右图所示，称之为循环队列.指针和队列元素之间的关系不变



§3.6 循环队列存储结构及操作实现

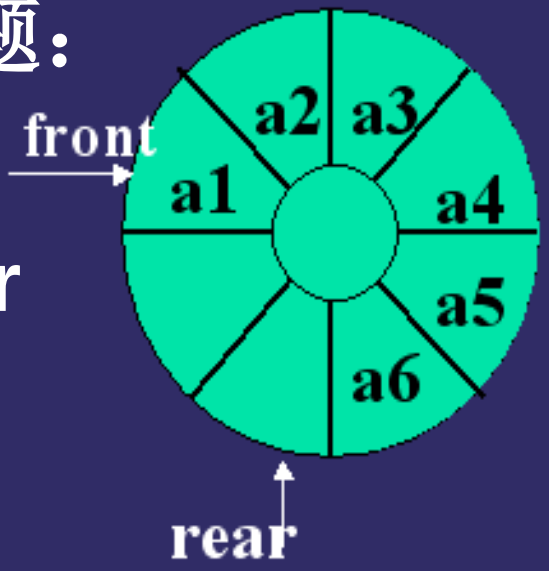
循环队列可充分利用空间，但仍存在问题：

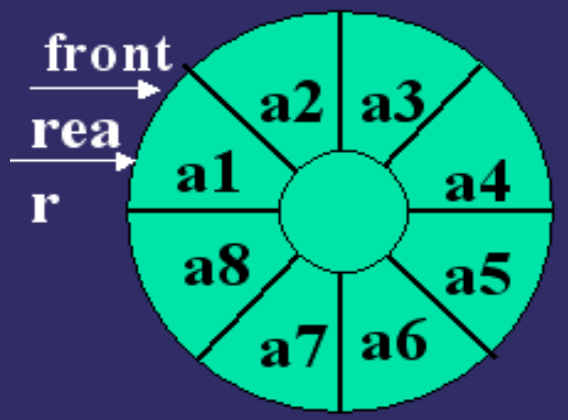
我们知道：队列为空时**front = rear**

右图中，继续入队**a7 ,a8** ，则**front=rear**

即：队列为空或队列为满时，都是

front = rear ，如何区分？





§3.6 循环队列存储结构及操作实现

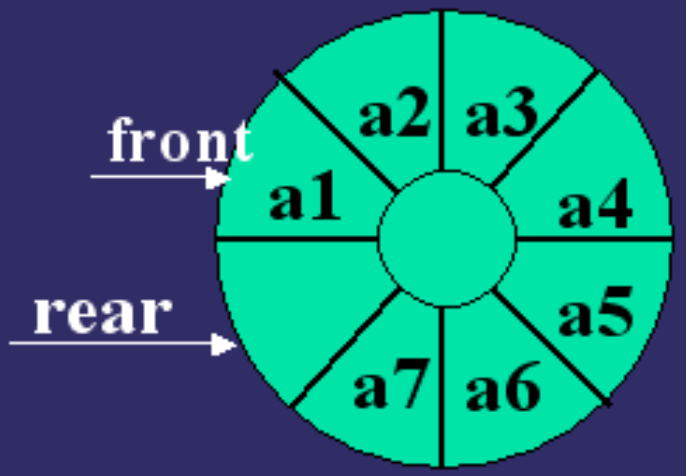
两种方法

1. 设置标志位以区别队列是“空”还是“满”

   因出队而相等，则为空；

   因入队而相等，则为满；

2. 少用一个元素的空间，约定 rear+1=front 时，就认为队满.



§3.6 循环队列存储结构及操作实现

```
#define  MAXSIZE 100  //  最大队列长度
   typedef struct{
      QElemType  *base;     //初始化时动态分配
                            //存储空间的基址
      int front;    //队头指针，指向队头元素
      int rear;     //队尾指针，指向队尾元素的下
                    //一个位置
}SqQueue;
```

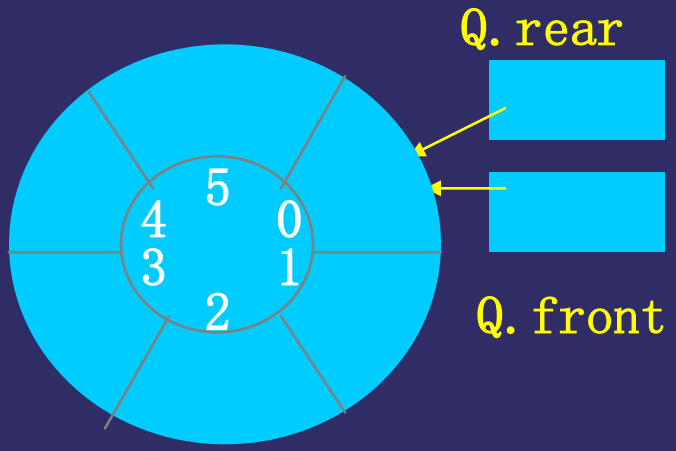
§3.6 循环队列存储结构及操作实现

循环队列的基本操作算法

1）初始化操作InitQueue_Sq(SqQueue &Q)

参数：Q是存放队列的结构变量；

功能：建一个空队列Q；



建一个空队列Q

§3.6 循环队列存储结构及操作实现

算法：

```
Status InitQueue_Sq(SqQueue &Q) {
        //构造一个空队列Q
        Q.base=(ElemType * )malloc
   (MAXOSIZE *sizeof (ElemType));
        if (!Q.base) exit (OVERFLOW);
      //存储分配失败
        Q.front = Q.rear=0;
        Return OK;
   } // InitQueue_Sq
```

§3.6 循环队列存储结构及操作实现

入队操作EnQueue_Sq(SqQueue &Q, QElemType e)

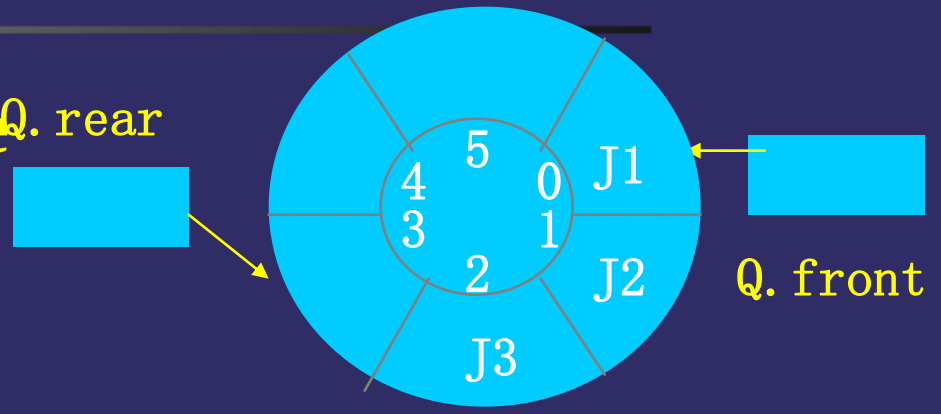
功能：将元素e插入队尾；

入队操作主要步骤：

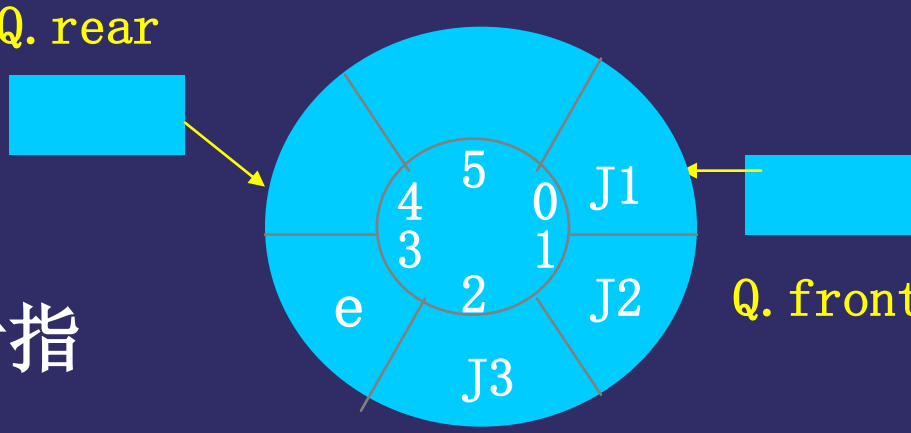
1）Q是否已满， 若满，返回ERROR；否则转2）；

2）将元素e 写入队尾；

3）修改队尾指针，使队尾指针指向队尾元素的下一个位置；



元素e入队前



元素e入队后

§3.6 循环队列存储结构及操作实现

入队操作算法

```
Status EnQueue_Sq(SqQueue  &Q, QElemType  e )
      //将元素e插入队尾
  if ((Q.rear+1)%MAXSIZE= =Q.front) return
ERROR ;
      Q.base[Q.rear] = e ;     // 将元素e插入队尾
      Q.rear= (Q.rear+1)%MAXSIZE;
 // 修改队尾指针
      return OK;   } // EnQueue_Sq
```

求余运算

§3.6 循环队列存储结构及操作实现

出队操作DeQueue_Sq (SqQueue &Q, QElemType &e)

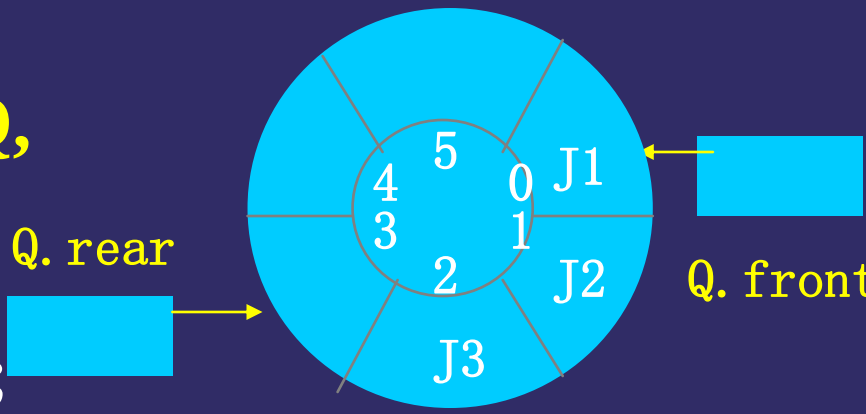
功能：删除队头元素，用e返回其值；

出队操作主要步骤：

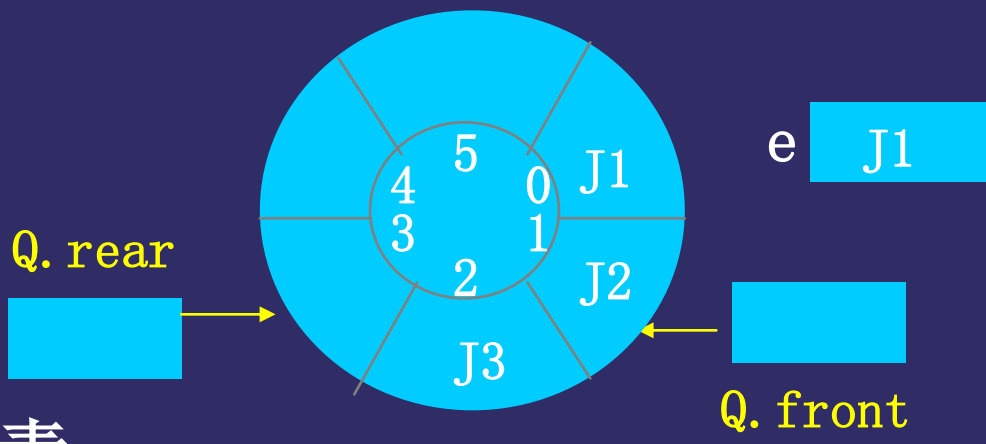
1）Q是否为空， 若空，返回ERROR；否则转2）；

2）将队头元素赋值给e；

3）修改队头指针，删除队头元素；



出队操作前



出队操作后

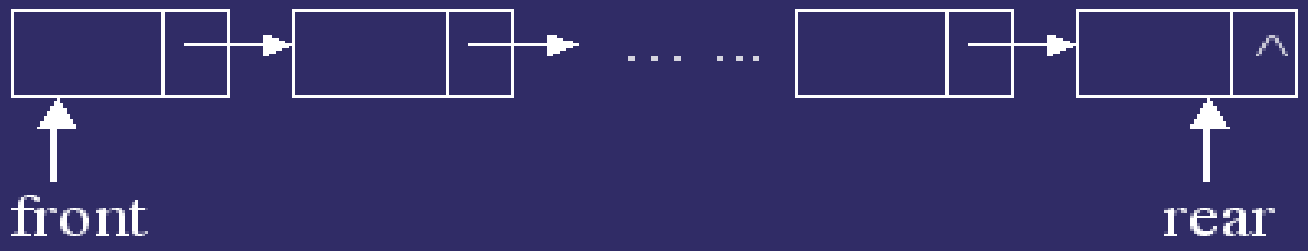
§3.6 循环队列存储结构及操作实现

出队操作算法

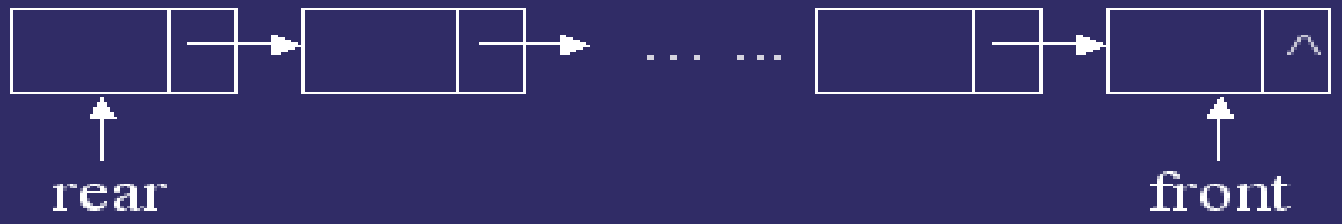
```
Status DeQueue_Sq(SqQueue  &Q, QElemType &e )
      //删除队头元素，用e返回其值,并返回OK；否
      //则返回ERROR
 if ((Q.rear= =Q.front) return ERROR ;
 e =Q.base[Q.front] ;
 Q.front= (Q.front+1)%MAXSIZE; // 修改队头指针
      return OK;
}  // EnQueue_Sq
```

§ 3.7 队列的链式存储结构及操作实现

1、存储方式：同一般线性表的单链式存储结构完全相同。但是由于插入、删除在两端进行，需要两个指针**front** 、**rear** 分别指向队列的两端。有两种不同的链法：

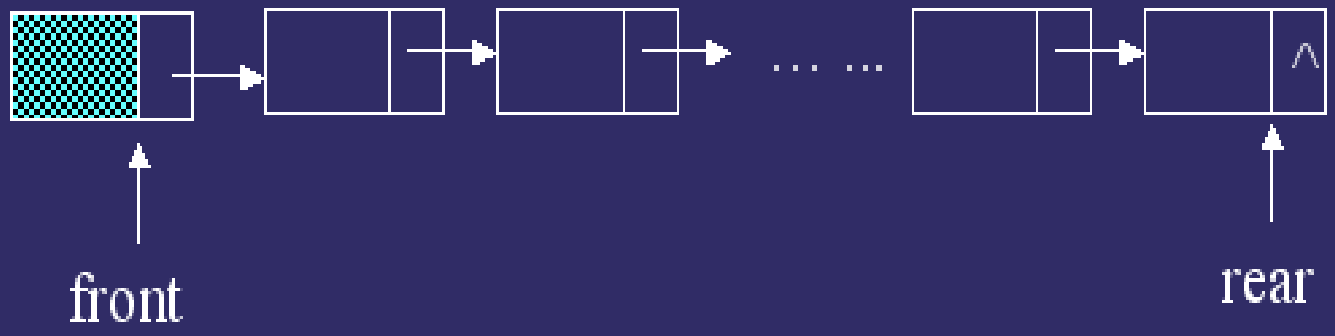


哪个好?



§ 3.7 队列的链式存储结构及操作实现

入队都很容易，但是出队差别很大！！
应该用第1 种链法。为了处理空队列，可以加上头结点。



2 、特点：没有假溢出，提高了空间利用率。
只有系统没有空间了，才会溢出；

§ 3.7 队列的链式存储结构及操作实现

3　链队列的虚拟实现

结构类型；

```
typedef struct QNode{ //链队列结点的类型定义
    QElemType  data;//用于存放队列元素
    struct QNode *next;//用于存放元素直接后继
                       //结点的地址
}QNode,*QueuePtr;
typedef struct{//链队列的表头结点的的类型定义
QueuePtr front;  //队头指针，指向链表的头结点
QueuePtr rear;   //队尾指针，指向队尾结点
}LinkQueue;
```

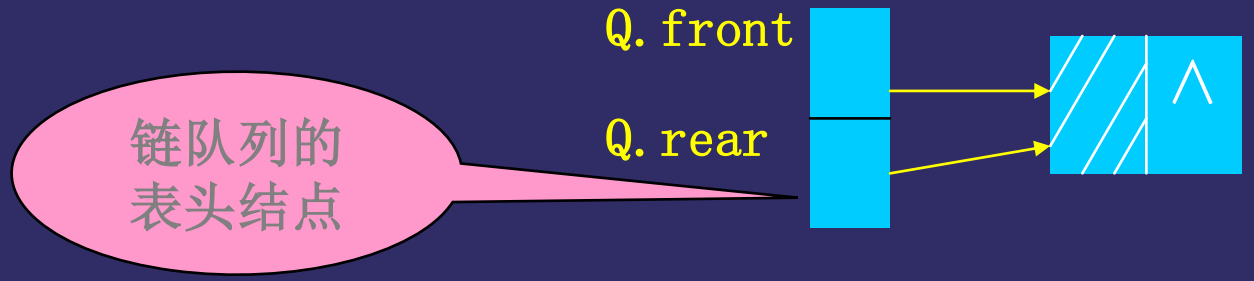
表示链队列的一个结点

指针变量用于存放链队列结点的地址

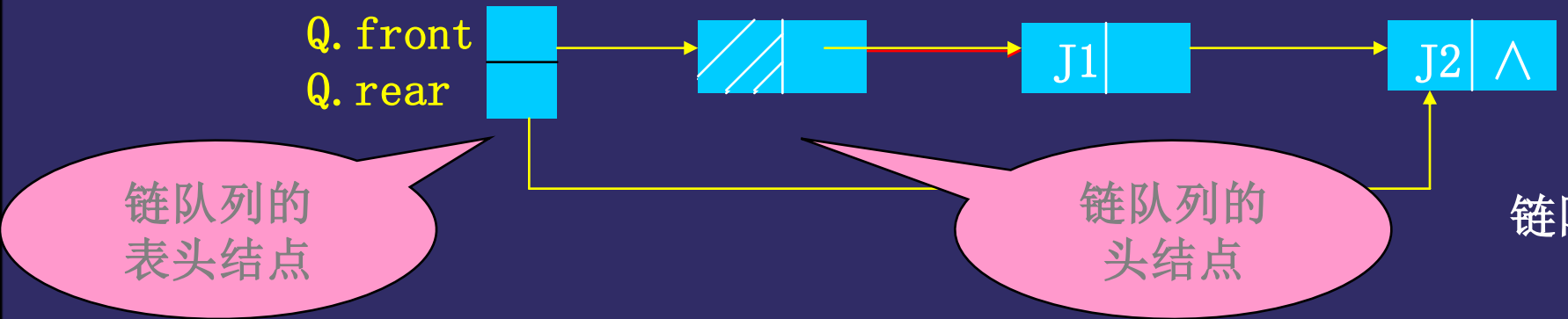
结构类型用于存放队头指针、队尾指针

§3.7 队列的链式存储结构及操作实现

设Q为**LinkQueue**类型的变量，Q为链队列的表头结点，用于存储队列队头指针和队尾指针；初始建队时，令**Q.front=Q.rear=null;**



空链队列



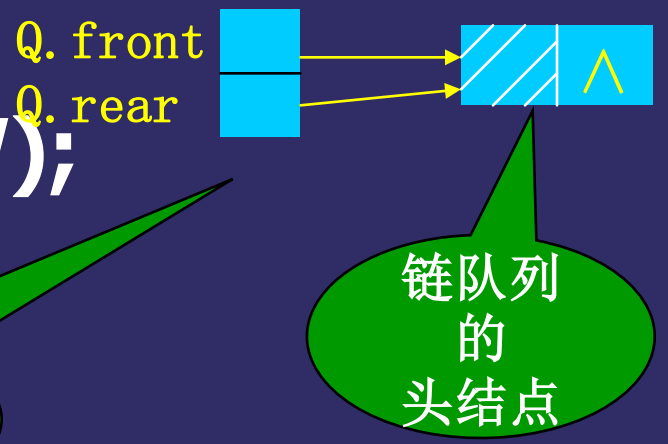
链队列图示

§3.7 队列的链式存储结构及操作实现

链队列基本操作算法

1）初始化操作InitQueue_L(LinkQueue &Q)

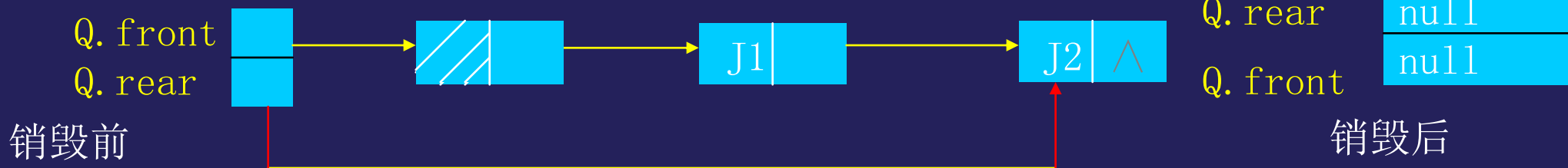
```
Status InitQueue_L(LinkQueue &Q) {
    //建一个空队列
Q.from=Q.rear=(QueuePtr)malloc(
sizeof(QNode));  //为链队列的头结点
分配空间
    if (!Q.front) exit(OVERFLOW);
    Q.front->next=NULL;
     return OK;
} //InitQueue_L
```



空链队列

2) 销毁操作DestroyQueue_L(LinkQueue &Q)

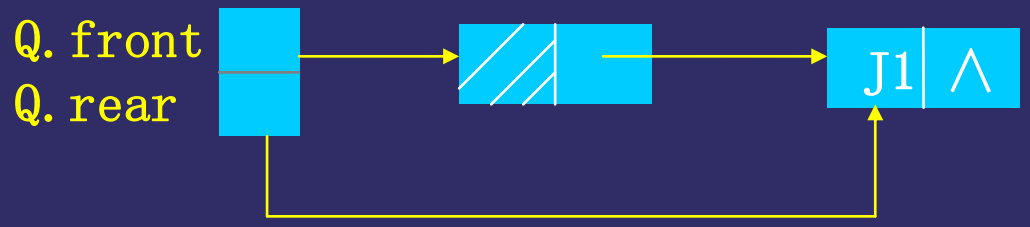
```
Status DestroyQueue_L(LinkQueue &Q) {
        //销毁队列Q
       if (!Q.front) return ERROR; // 链队列不存在
       while(Q.front->next) {   // 回收链队列的所有元素结点空间
           p=Q.front->next;
          Q.front->next=p->next;
          free(p);
       }
       free(Q.front );                    // 回收链队列头结点空间
      Q.front=Q.rear=null;
       return OK;
}//DestroyQueue_L
```



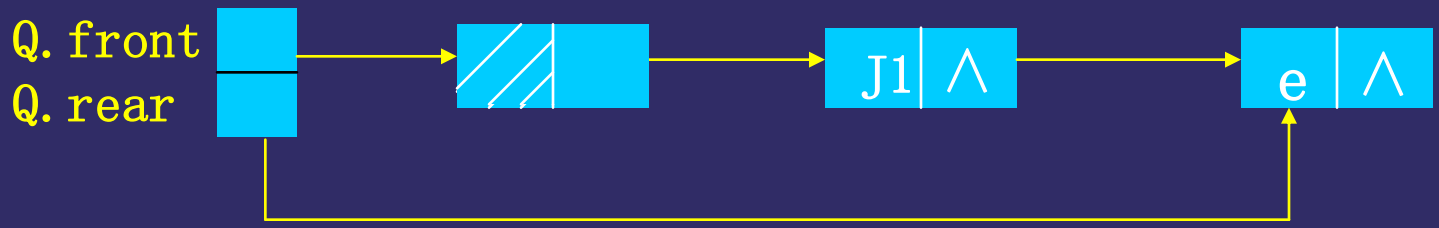
§ 3.7 队列的链式存储结构及操作实现

3）入队操作EnQueue_L(LinkQueue &Q, QElemType)

入队操作图示

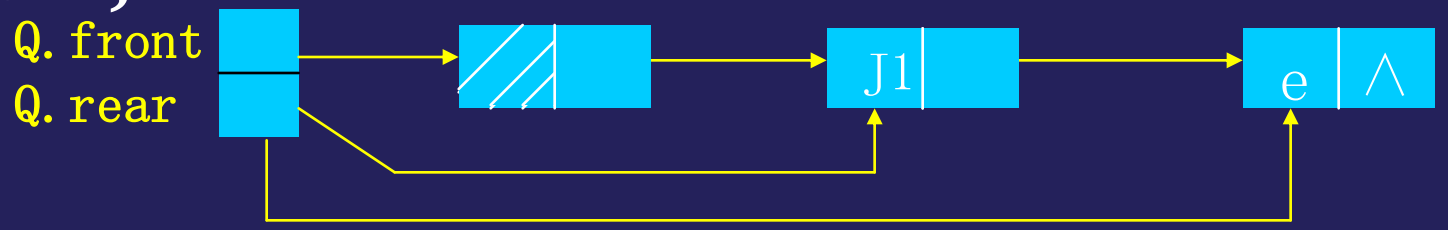


e入队前



e入队后

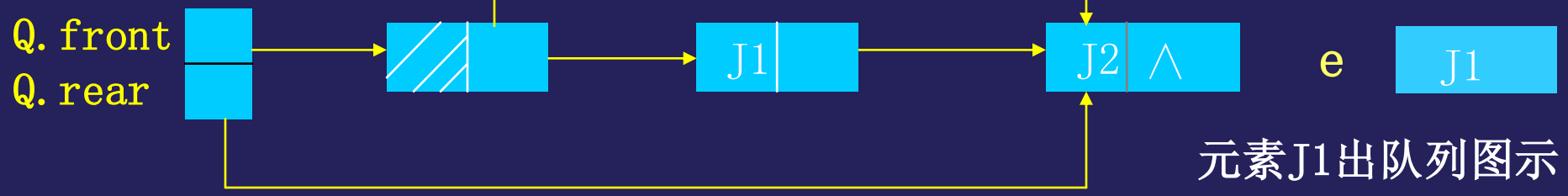
```
Status EnQueue_L(LinkQueue &Q, QElemType ) {
    //在链队列Q队尾，插入新的队尾结点
    p=(QueuePtr)malloc(sizeof(QNode));  //为新元素e分
                                        //配结点空间
    if (!p) exit (OVERFLOW);    //存储分配失败
    p->date=e; p->next=NULL;
    Q.rear->next=p;   //修改队尾指针，插入新队尾结点
    Q.rear=p;
     return  OK;
}
```



在链队列Q队尾，插入新的元素e

4）出队操作DeQueue_L(LinkQueue &Q, QElemType&e)

```
Status DeQueue_L(LinkQueue &Q, QElemType &e) {
    //若队列不空，则删除Q的队头元素结点，用e返回其值，并返回OK；否则返回ERROR
    if (Q.front = =Q.rear) return ERROR;
        //若队列空，返回ERROR
    p=Q.front->next;            // p指向队头元素结点
    e=p->data;
    Q.front->next=p->next; // 修改链队列头结点指针
if(Q.rear= =p)Q.rear=Q.front;
// 对于链队列只有一个元素结点的情况,要同时修改队尾指针
    free(p);
    return OK;
}
```



元素J1出队列图示