

数据结构与算法

Data Structures and Algorithms

张岩



海量数据计算研究中心



哈工大计算机科学与技术学院



2019/10/18

第2章 线性表





学习目标

- 掌握线性表的逻辑结构，线性表的顺序存储结构和链式存储结构的描述方法；熟练掌握线性表在顺序存储结构和链式存储结构的结构特点以及相关的查找、插入、删除等基本操作的实现；并能够从时间和空间复杂性的角度综合比较两种存储结构的不同特点
- 掌握栈和队列的结构特性和描述方法，熟练掌握栈和队列的基本操作的实现，并且能够利用栈和队列解决实际问题
- 掌握串的结构特性以及串的基本操作，掌握针对字符串进行操作的常用算法和模式匹配算法
- 掌握多维数组的存储和表示方法，掌握对特殊矩阵进行压缩存储时的下标变换公式，了解稀疏矩阵的压缩存储表示方法及适用范围
- 了解广义表的概念和特征





本章主要内容

- 2.1 线性表的逻辑结构
- 2.2 线性表的存储结构
- 2.3 栈 (Stack)
- 2.4 队列 (Queue)
- 2.5 串 (String)
- 2.6 数组 (Array)
- 2.7 广义表 (Generalized List)
- 本章小结



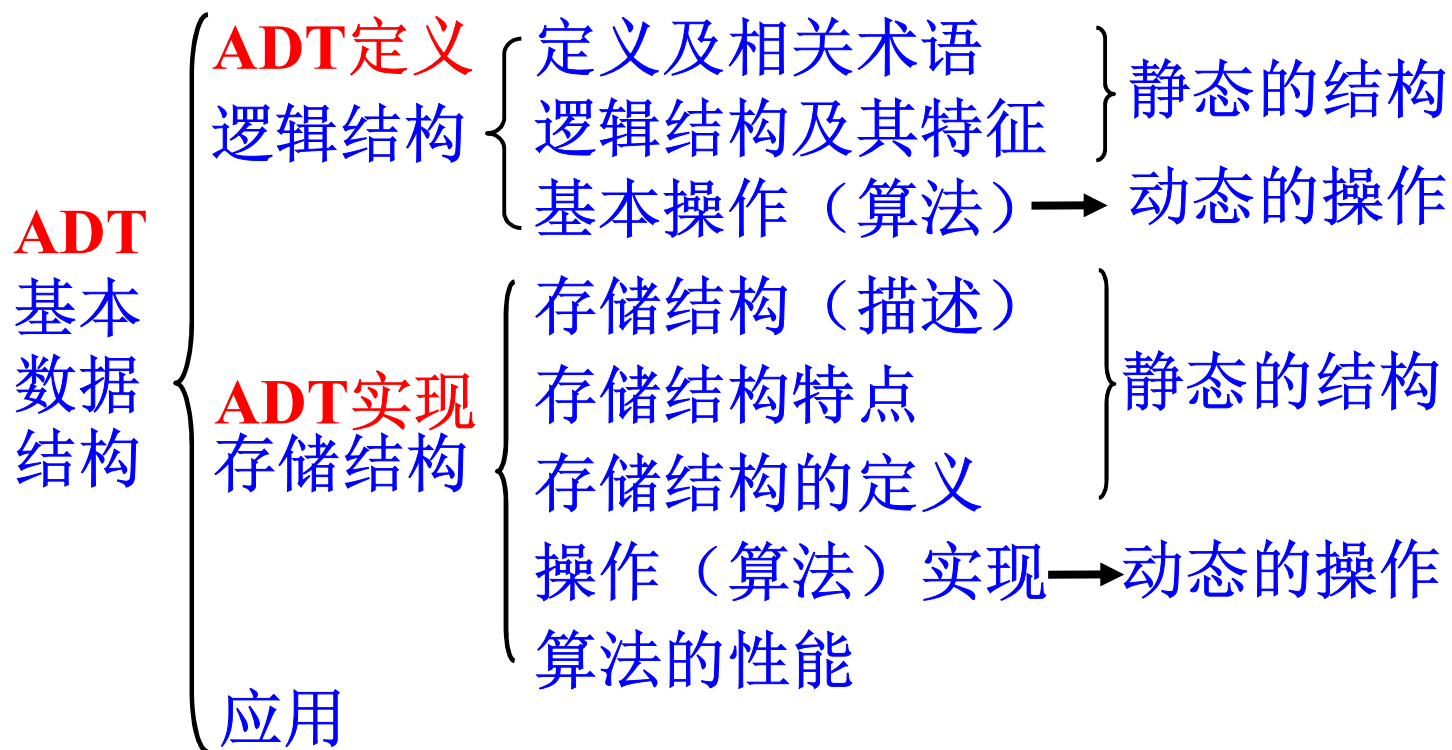


本章的知识点结构

基本的数据结构（ADT）

■ 线性表、栈、队列、串、（多维）数组、广义表

知识点结构





2.1 线性表的逻辑结构

线性表的定义:

■ 是由 n ($n \geq 0$) 个性质 (类型) 相同的元素组成的序列。

■ 记为:

$$L = (a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

■ a_i ($1 \leq i \leq n$) 称为数据元素;

● 下角标 i 表示该元素在线性表中的位置或序号。

■ n 为线性表中元素个数, 称为线性表的长度;

● 当 $n=0$ 时, 为空表, 记为 $L = ()$ 。

■ 图示表示:

● 线性表 $L = (a_1, a_2, \dots, a_i, \dots, a_n)$ 的图形表示如下:





2.1 线性表的逻辑结构(Cont.)

➤ **逻辑特征:** $L = (a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

■ **有限性:**

● 线性表中数据元素的个数是有穷的。

■ **相同性:**

● a_i 为线性表中的元素元素类型相同

■ **相继性:**

● a_1 为表中**第一个**元素，无**前驱**元素； a_n 为表中**最后一个**元素，无**后继**元素；

● 对于 $\dots a_{i-1}, a_i, a_{i+1} \dots (1 < i < n)$ ，称 a_{i-1} 为 a_i 的**直接前驱**， a_{i+1} 为 a_i 的**直接后继**。

● 中间不能有缺项。





2.1 线性表的逻辑结构(Cont.)

↓ 定义在线性表的操作（算法）：

■ 设L是类型为LIST线性表实例，x 的型为ElemType的元素实例，p 为位置变量。所有操作描述为：

- ① Insert(x, p, L)
- ② Delete(p, L)
- ③ Locate(x, L)
- ④ Retrieve(p, L)
- ⑤ Previous(p, L)
- ⑥ Next(p, L)
- ⑦ MakeNull(L)
- ⑧ First(L)
- ⑨ END (L)





2.1 线性表的逻辑结构(Cont.)

➡ ADT应用举例:

■ 设计函数 **DeleteVal** (**LIST** &**L**, **ElemType** **d**) , 其功能为删除 **L** 中所有值为 **d** 的元素。

■ **void DeleteVal(LIST &L, ElemType d)**

```
{   position p ;  
    p = First( L ) ;  
    while ( p != End( L ) )  
    {   if ( Same( Retrieve( p, L ), d ) )  
        Delete(p, L) ;  
        else  
            p = Next(p, L) ;  
    }  
}
```





2.2 线性表的存储结构----顺序表

➤ 2.2.1 顺序表:

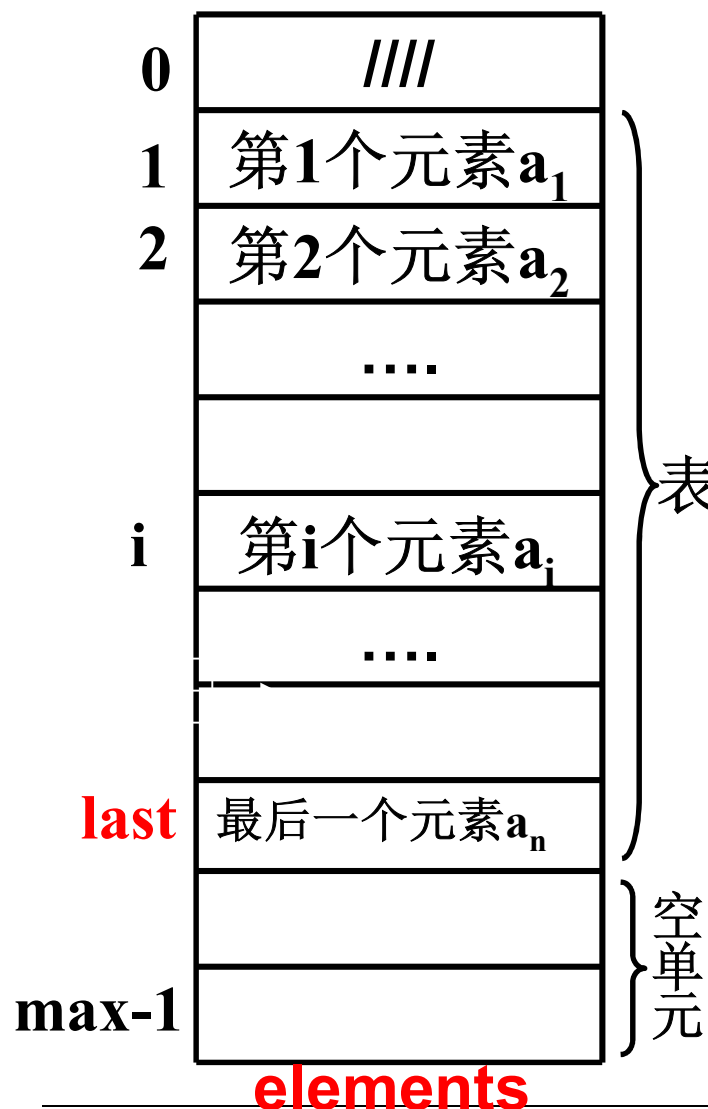
■ 把线性表的元素按照**逻辑顺序**依次存放在**数组**的**连续**单元内;

■ 再用一个**整型量**表示最后一个元素所在单元的下标, 即**表长**。

➤ 存储结构特点:

■ 元素之间逻辑上的**相继关系**, 用物理上的**相邻关系**来表示 (用**物理上的连续性**刻画逻辑上的相继性)

■ 是一种**随机访问存取**结构, 也就是可以随机存取表中的任意元素, 其存储位置可由一个**简单直观的公式**来表示。





2.2线性表的存储结构--顺序表(Cont.)

➤存储结构定义:

■类型定义:

```
#define max 100
```

```
struct LIST{
```

```
    ElemType elements[max];
```

```
    int last;
```

```
};
```

■位置类型:

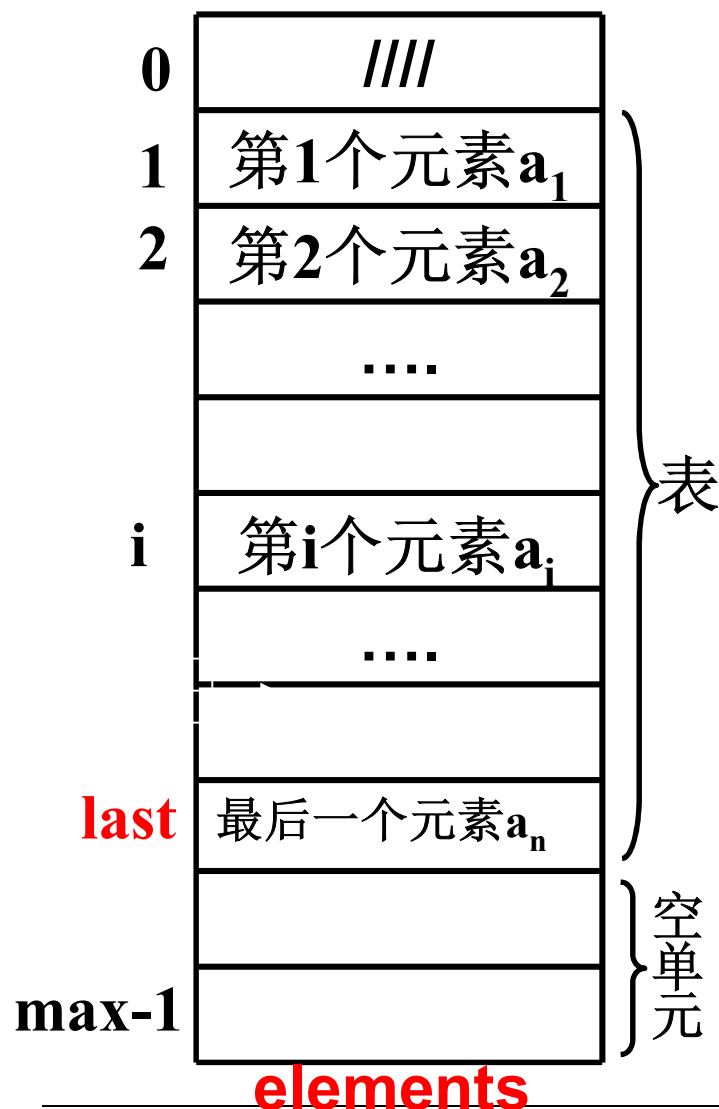
```
typedef int position;
```

■线性表的实例L: LIST L;

■元素和长度:

L.elements[p] // L的第p个元素

L.last L的长度, 最后元素的位置





2.2线性表的存储结构--顺序表(Cont.)

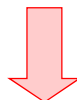
➡ 操作的实现----插入操作

■ 操作接口 `void Insert (ElemType x, position p, LIST &L)`

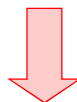
■ 插入前: $(a_1, \dots, a_{p-1}, a_p, \dots, a_n)$

■ 插入后: $(a_1, \dots, a_{p-1}, x, a_p, \dots, a_n)$

a_{p-1} 和 a_p 之间的逻辑关系发生了变化



顺序存储要求存储位置反映逻辑关系



存储位置要反映这个变化





2.2线性表的存储结构--顺序表(Cont.)

➡ 例: (35, 12, 24, 42), 在 $p=2$ 的位置上插入33。

1	2	3	4	5		M-1	last
a_1	a_2	a_3	a_4				4 5
35	33	12	24	42			

↑
33

➡ 什么时候不能插入?

■ 表满时: $L.last \geq Max$



注意边界条件

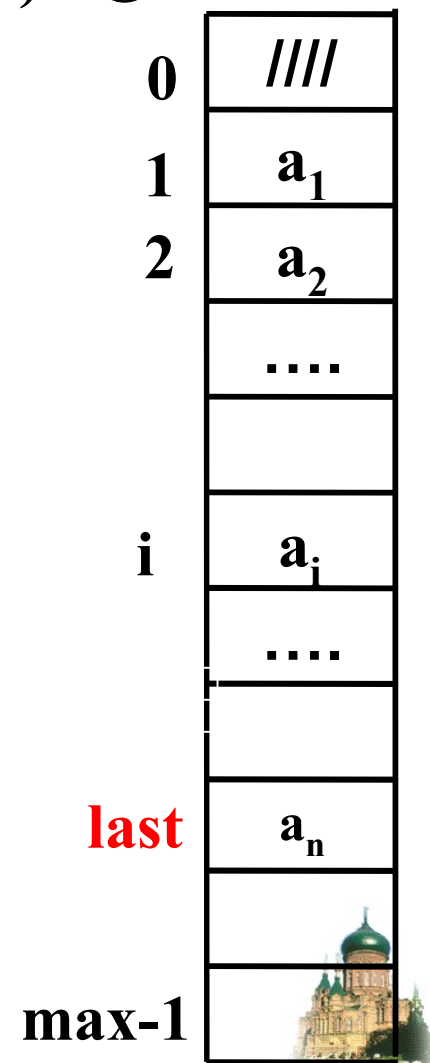
■ 合理的插入位置: $1 \leq p \leq L.last+1$





2.2线性表的存储结构--顺序表(Cont.)

```
void Insert ( ElemType x, position p, LIST &L)// ①
{ position q ;
  if (L.last >= Max - 1)
    cout<< “ 表满 ” ;
  else if (( p > L.last +1 ) || ( p < 1) )
    cout<< “ 指定位置不存在 ” ;
  else {
    for ( q = L.last; q >= p; q -- )
      L.elements[ q+1 ] = L.elements[ q ] ;
    L.elements[ p ] = x ;
    L.last = L.last + 1 ;
  }
}
```





2.2线性表的存储结构--顺序表(Cont.)

时间性能分析

■ 基本语句?

■ 最好情况 ($i=n+1$):

● 基本语句执行0次, 时间复杂度为 $O(1)$ 。

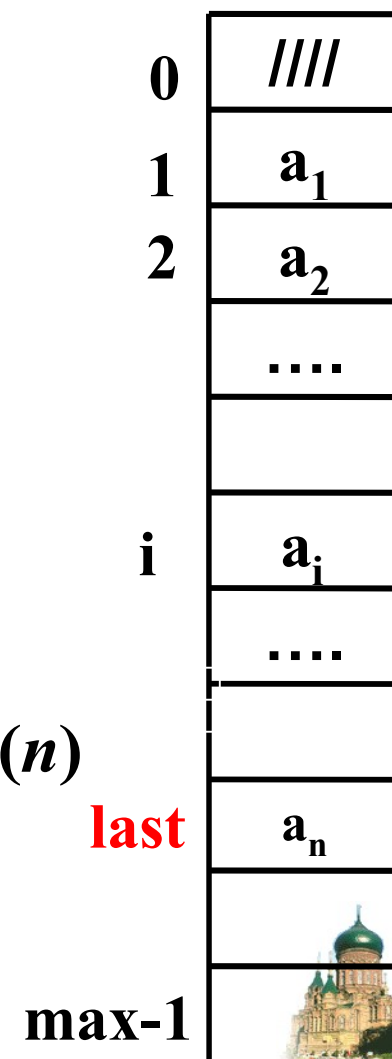
■ 最坏情况 ($i=1$):

● 基本语句执行 n 次, 时间复杂度为 $O(n)$ 。

■ 平均情况 ($1 \leq i \leq n+1$):

$$\sum_{i=1}^{n+1} p_i (n - i + 1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{n}{2} = O(n)$$

■ 时间复杂度为 $O(n)$ 。





2.2线性表的存储结构--顺序表(Cont.)

操作的实现----删除操作

■ 操作接口: `void Delete(position p, LIST &L)`

■ 删除前: $(a_1, \dots, a_{p-1}, a_p, a_{p+1}, \dots, a_n)$

■ 删除后: $(a_1, \dots, a_{p-1}, a_{p+1}, \dots, a_n)$

➡ 例: $(35, 33, 12, 24, 42)$, 删除 $p=2$ 的数据元素。

1	2	3	4	5	last	
a_1	a_2	a_3	a_4	a_5		
35	12	24	24	42		

➡ 分析边界条件?

➡ 操作算法的实现

➡ 时间性能分析



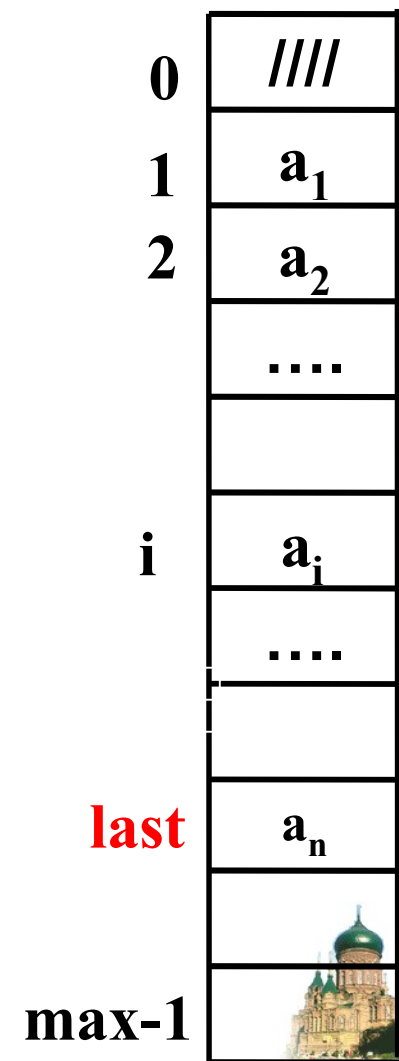


2.2线性表的存储结构--顺序表(Cont.)

```
➡ void Delete( position p, LIST &L) //②
{   position q ;
    if ( ( p > L.last ) || ( p < 1 ) )
        cout<< “指定位置不存在” ;
    else{
        L.last = L.last - 1;
        for ( q = p ; q <= L.last ; q ++ )
            L.elements[ q ] = L.elements[ q + 1 ];
    }
}
```

➡ 最好、最坏和平均移动元素个数:

➡ 时间复杂性: $O(n)$



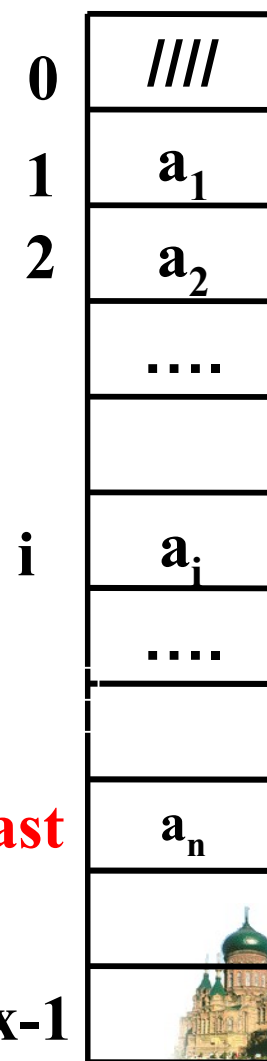


2.2线性表的存储结构--顺序表(Cont.)

其他操作的实现

```
position Locate ( ElemType x , LIST L ) //③
{ position q ;
  for ( q = 1; q <= L.last ; q++ )
    if ( L.elements[ q ] == x )
      return ( q ) ;
  return ( L.last + 1 );
} //时间复杂性:  $O(n)$ 
```

```
ElemType Retrieve ( position p , LIST L ) //④
{ if ( p > L.last )
  cout<< “指定位置不存在” ;
  else
    return ( L.elements[ p ] );
} //时间复杂性:  $O(1)$ 
```





2.2线性表的存储结构--顺序表(Cont.)

其他操作的实现

■ position Previous(position p , LIST L) //⑤
{ if ((p <= 1) || (p > L.last+1))

cout<< “前驱位置不存在”;//return 0;

else

return (p - 1);

} //时间复杂性: $O(1)$

■ position Next(position p , LIST L) //⑥

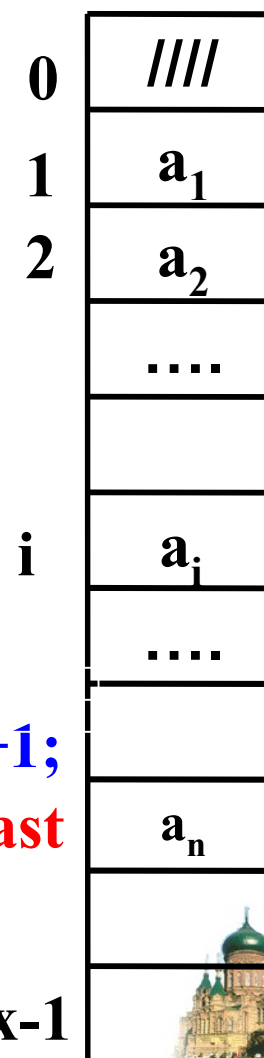
{ if ((p < 1) || (p >= L.last))

cout<< “前驱位置不存在” ;//return L.last+1;

else

return (p + 1);

} //时间复杂性: $O(1)$





2.2线性表的存储结构--顺序表(Cont.)

其他操作的实现

■ **position MakeNull(LIST &L) //⑦**

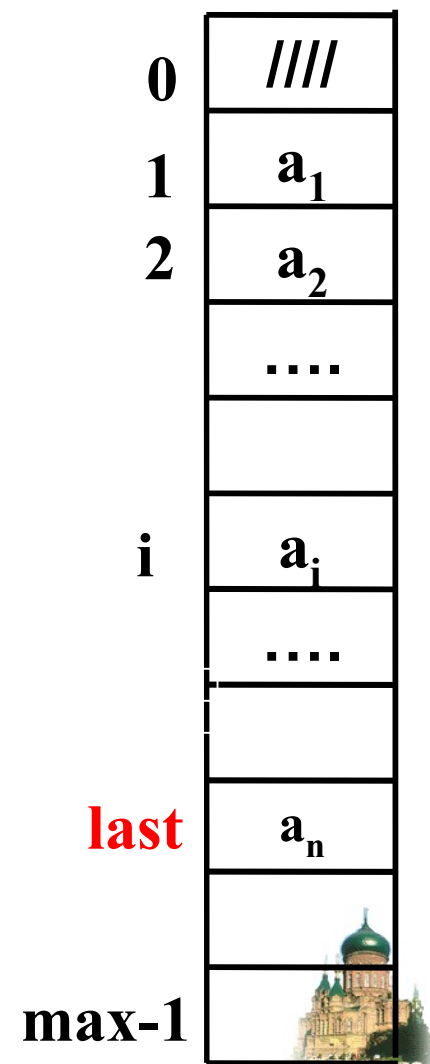
```
{ L.last = 0 ;  
  return ( L.last +1 );  
} //时间复杂性:  $O(1)$ 
```

■ **position First(LIST L) //⑧**

```
{ if ( L.last>0 ) return ( 1 );  
  else cout<<“表为空”;//return 0;  
} //复杂性:  $O(1)$ 
```

■ **position End(LIST L) //⑨**

```
{ return( L.last + 1 );  
} //  $O(1)$ 
```





2.2 线性表的存储结构--链接表

2.2.2 单链表

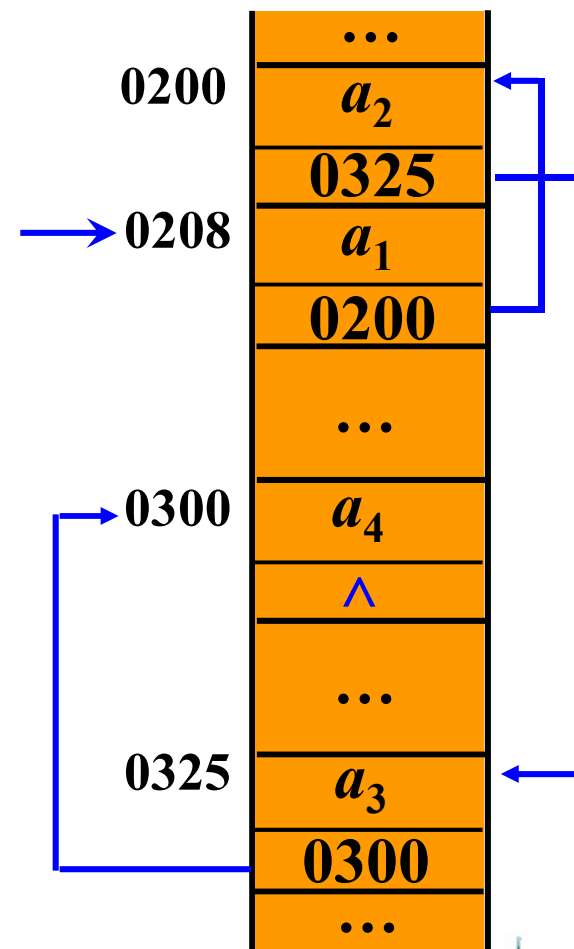
单链表:

- 一个线性表由若干个结点组成，每个结点均含有两个域：存放元素的信息域和存放其后继结点的指针域，这样就形成一个单向链接式存储结构，简称单链表或单链表。

- 例： (a_1, a_2, a_3, a_4) 的存储示意图

存储结构特点

- 逻辑次序和物理次序不一定相同；
- 元素之间的逻辑关系用指针表示；
- 需要额外空间存储元素之间的关系
- 非随机访问存取结构（顺序访问）





2.2线性表的存储结构--链接表(Cont.)

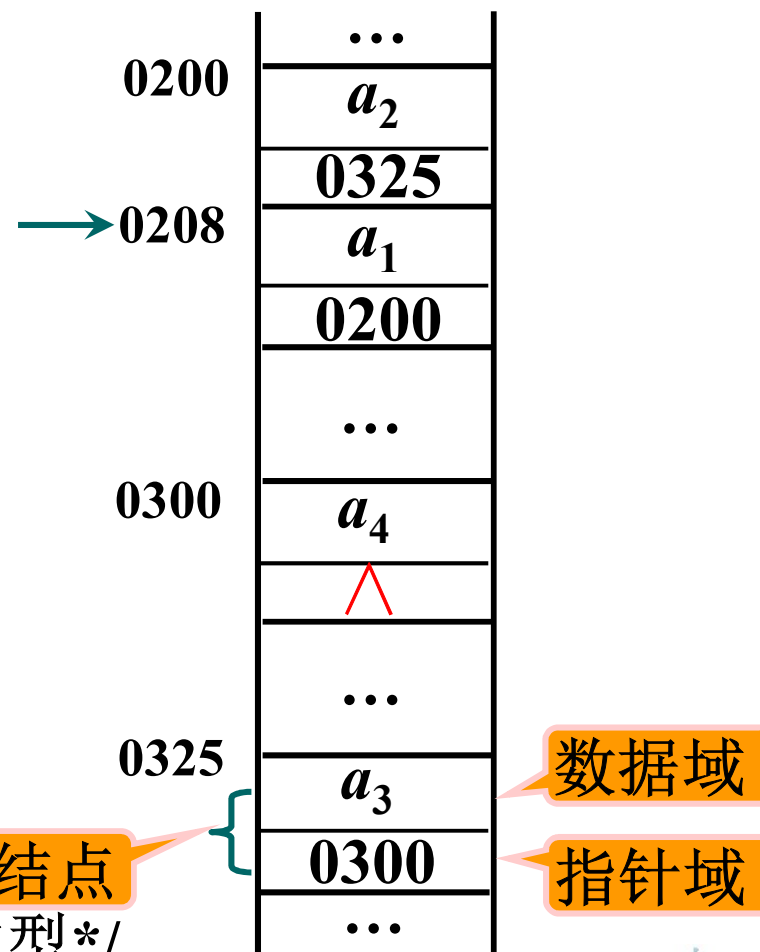
➡ 存储结构定义:

■ 结点结构:

数据域	指针域
data	next

■ 存储结构类型定义

```
struct celltype {  
    ElemType data ;  
    celltype *next ;  
}; /*结点型*/  
typedef celltype *LIST; /*线性表的型*/  
typedef celltype *position; /*位置型*/
```



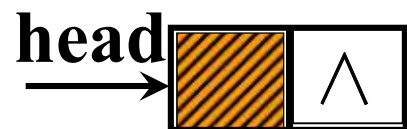
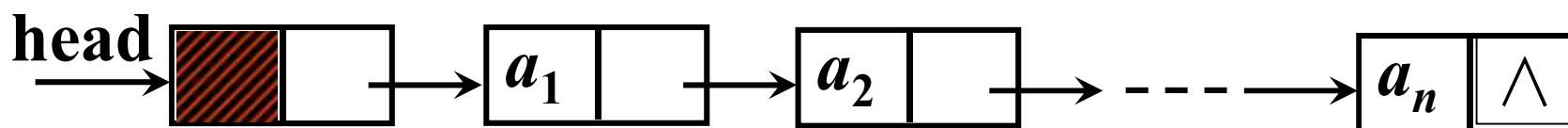


2.2线性表的存储结构--链表(Cont.)

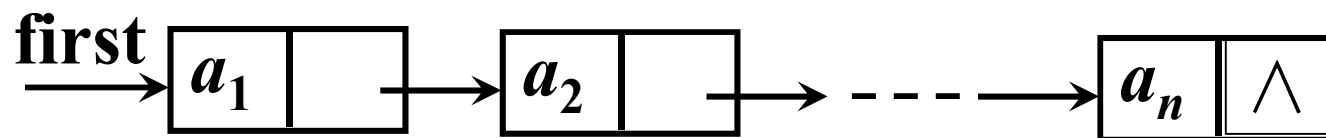
➤ 存储结构定义:

■ 单链表图示:

● 带表头结点的单链表



● 不带头结点的单链表



first==NULL;

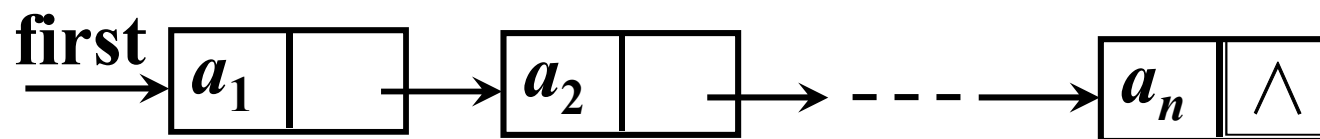
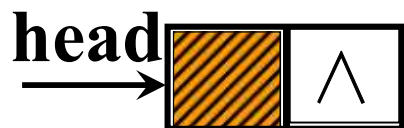
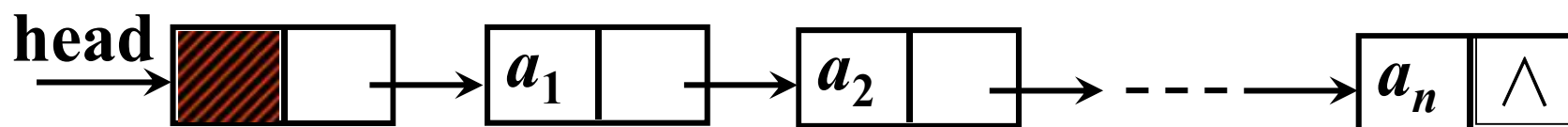




2.2线性表的存储结构--链接表(Cont.)

■ 表头结点的作用:

- 空表和非空表表示统一
- 在任意位置的插入或者删除的代码统一
- 注意: 是否带表头结点在存储结构定义中无法体现, 由操作决定



`first==NULL;`

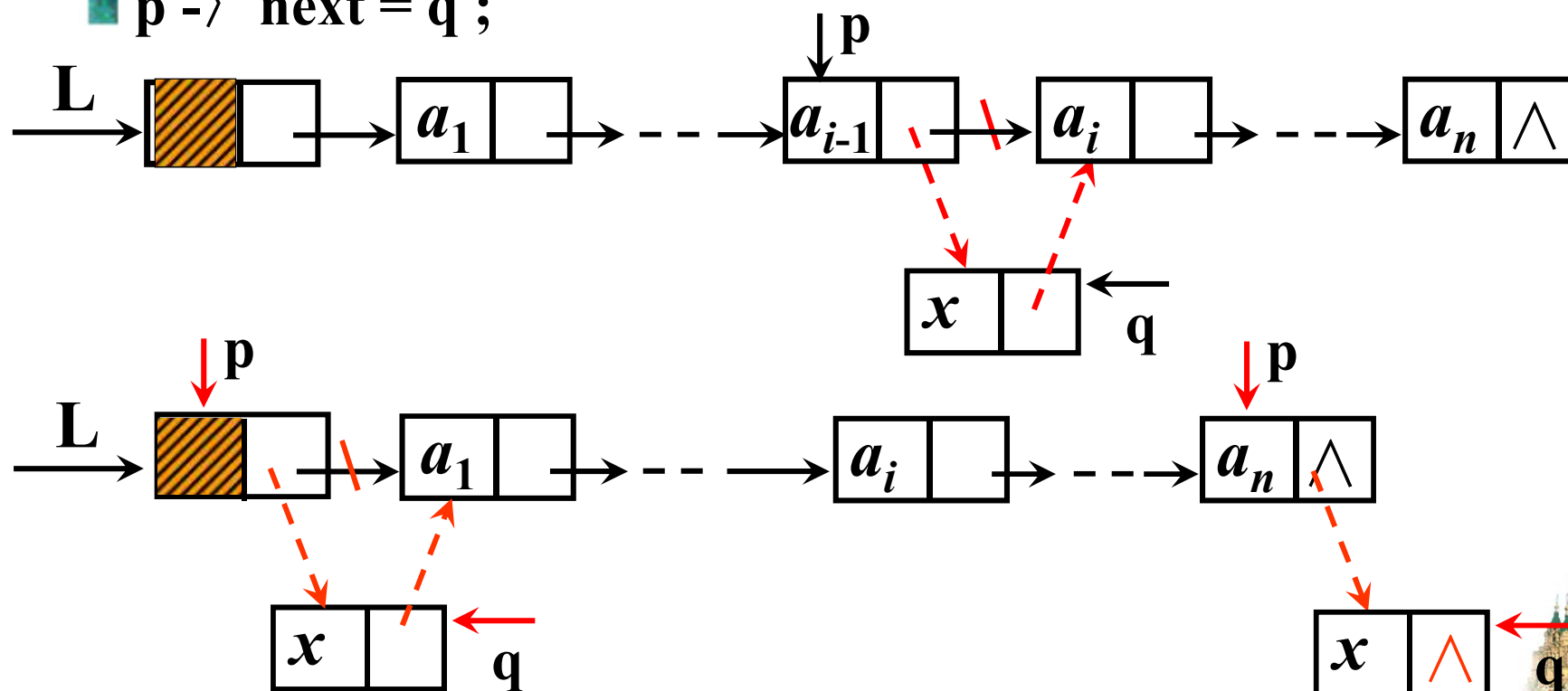




2.2 线性表的存储结构--链接表(Cont.)

操作的实现---- ①插入操作

- $q = \text{new celltype};$
- $q \rightarrow \text{data} = x;$
- $q \rightarrow \text{next} = p \rightarrow \text{next};$
- $p \rightarrow \text{next} = q;$





2.2线性表的存储结构--链接表(Cont.)

↓ 操作的实现---- ①插入操作

```
void Insert ( ElemType x, position p, LIST &L ) //①
{
    position q ;
    q = new celltype ;
    q →data = x ;
    q →next = p →next ;
    p →next = q ;
} //时间复杂性:  $O(1)$ 
```





2.2 线性表的存储结构--链接表(Cont.)

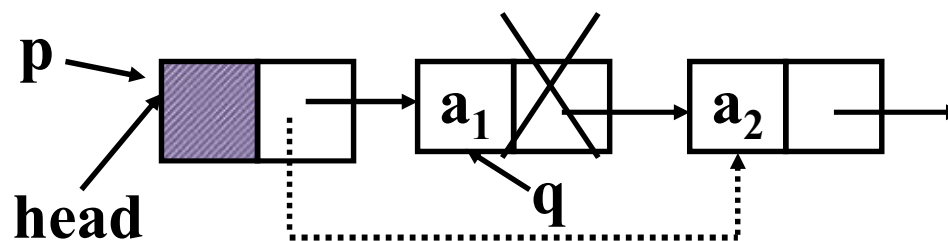
➔ 操作的实现---- ②删除操作

- $q = p \rightarrow \text{next};$
- $p \rightarrow \text{next} = q \rightarrow \text{next};$
- $\text{delete } q;$

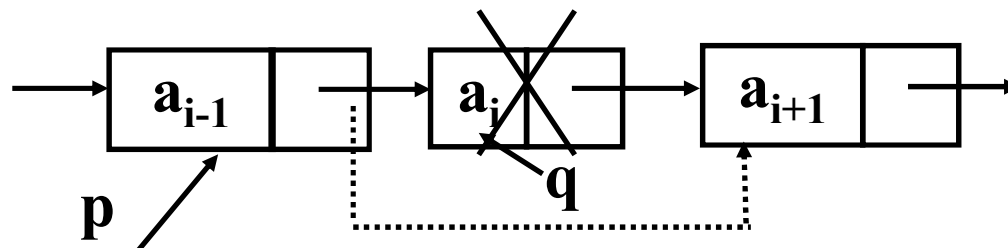
void Delete (position p, LIST &L)

```
{ position q;  
  if ( p → next != NULL ) {  
    q = p → next;  
    p → next = q → next;  
    delete q;  
  }  
}
```

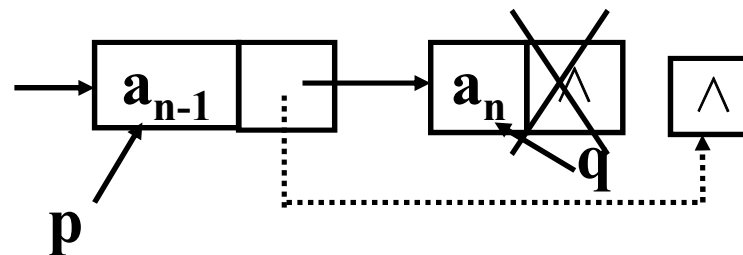
时间复杂性: $O(1)$



(a) 删除第一个元素



(b) 删除中间元素



(c) 删除表尾元素



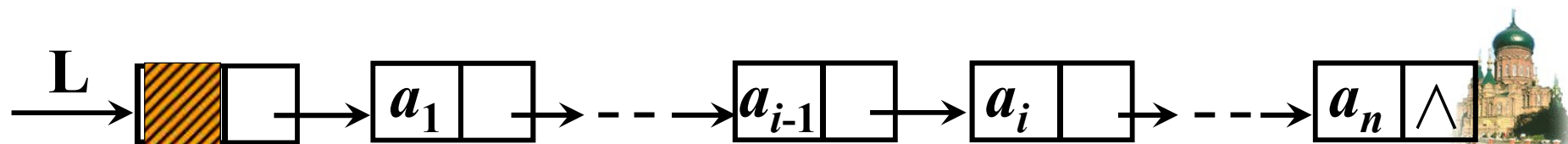


2.2线性表的存储结构--链接表(Cont.)

➡ 操作的实现---- ③Locate (ElemType x, LIST L)

■ position Locate (Elementtype x, LIST L)

```
{ position p ;  
  p = L ;  
  while ( p→next != NULL )  
    if ( p→next→data == x )  
      return p ;  
  else  
    p = p→next ;  
  return p ;  
} //时间复杂性:  $O(n)$ 
```





2.2线性表的存储结构--链接表(Cont.)

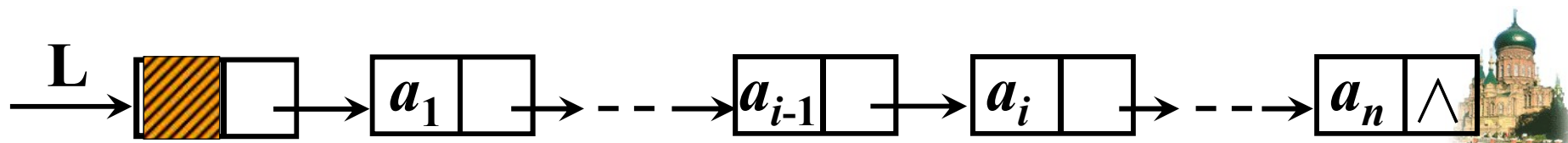
➡ 操作的实现---- ④ElemType Retrieve(position p , LIST L)

■ ElemType Retrieve (position p , LIST L)

{

 return (**p →next →data**);

} //时间复杂性: $O(1)$



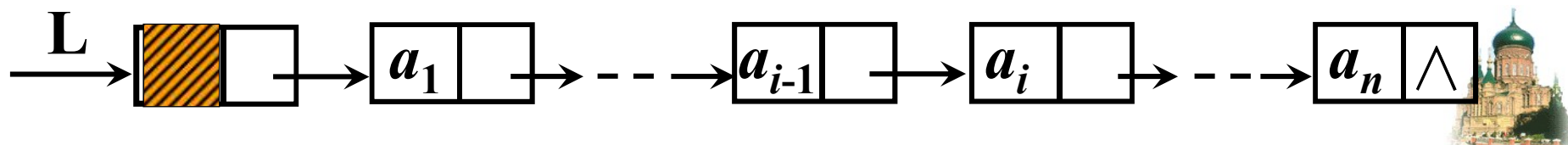


2.2线性表的存储结构--链接表(Cont.)

➡ 操作的实现---- ⑤ position Previous(position p, LIST L)

■ position Previous (position p, LIST L)

```
{ position q ;  
  if ( p == L→next )  
    cout << “不存在前驱位置” ;  
  else {  
    q = L ;  
    while ( q→next != p ) q = q→next ;  
    return q ;  
  }  
} //时间复杂性:  $O(n)$ 
```





2.2线性表的存储结构--链接表(Cont.)

➡ 操作的实现---- ⑥ position Next (position p, LIST L)

■ position Next (position p, LIST L)

```
{ position q ;
```

```
  if ( p→next == NULL )
```

```
    cout<< “不存在后继位置” ;
```

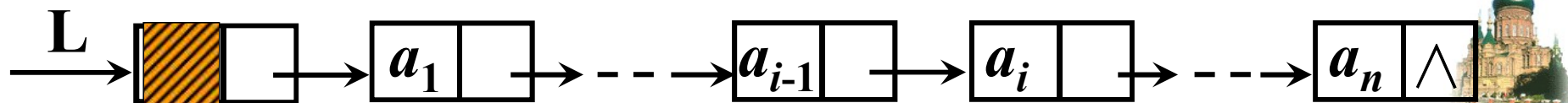
```
  else {
```

```
    q = p→next;
```

```
    return q ;
```

```
  }
```

```
} //时间复杂性:  $O(1)$ 
```





2.2线性表的存储结构--链接表(Cont.)

➡ 操作的实现---- ⑦ position MakeNull (LIST &L)

■ position MakeNull (LIST &L)

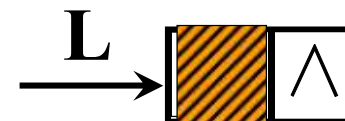
{

L = new celltype ;

L→next = NULL;

return L ;

} //时间复杂性: $O(1)$



➡ 操作的实现---- ⑧ position First (LIST L)

■ position First (LIST L)

{

return L;

} //时间复杂性: $O(1)$





2.2线性表的存储结构--链接表(Cont.)

➡ 操作的实现---- ⑨ position End (LIST L)

■ position End (LIST L)

```
{ position q ;
```

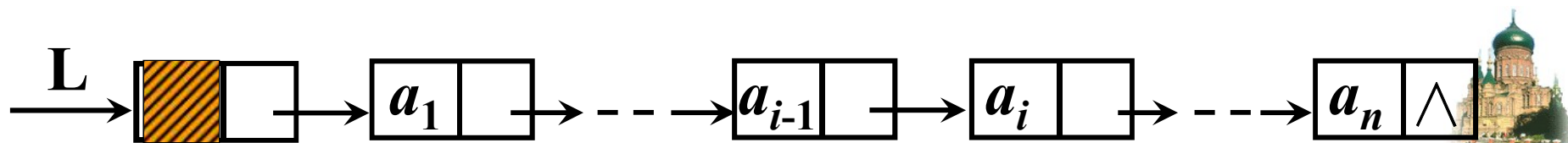
```
  q = L ;
```

```
  while ( q→next != NULL )
```

```
    q = q→next ;
```

```
  return ( q ) ;
```

```
} //时间复杂性:  $O(n)$ 
```



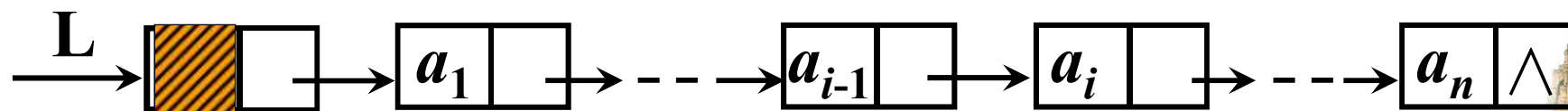


2.2线性表的存储结构--链接表(Cont.)

- 例：设计一个算法，遍历线性表，即按照线性表中元素的顺序，依次访问表中的每一个元素，每个元素只能被访问一次。

```
struct celltype {  
    ElemType data ;  
    celltype *next ;  
}; //结点型  
typedef celltype *LIST;  
/*线性表的型*/  
typedef celltype *position;  
/*位置型*/
```

```
void Travel( LIST L )  
{  
    position p ;  
    p = L→next ;  
    while ( p != NULL ) {  
        cout << p→data ;  
        p = p→next ;  
    }  
}
```





2.2线性表的存储结构--链表(Cont.)

顺序表与链表的比较

顺序表 单链表的比较		
比较参数	顺序存储	链式存储
表的容量	固定，不易扩充	灵活，易扩充
存取操作	随机访问存取	顺序访问存取
时间	插入删除费时间	访问元素费时间
空间	估算表长度，浪费空间	实际长度，节省空间





2.2 线性表的存储结构--静态链表

2.2.3 静态链表

➡ **静态链表：**链表的数组表示

■ 举例：

● 线性表： $L = (a, b, c)$

● 线性表： $M = (d, e)$

● 线性表： $av=9$ --空闲表(可用空间表)

■ 把线性表的元素存放在数组的单元中（不一定按逻辑顺序连续存放），每个单元不仅存放元素本身，而且还要存放其后继元素所在的数组单元的下标（游标）。

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5
SPACE		





2.2 线性表的存储结构--静态链表(Cont.)

➡ 结点形式:

数据域	游标域
data	next

➡ 存储结构定义:

■ 类型定义:

```
typedef struct {  
    ElemType data ;  
    int next ;  
} spacestr; /*结点类型*/  
spacestr SPACE[ maxsize ] ;/*存储池*/  
typedef int position, Cursor ;  
Cursor L, M, av;
```

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5
	SPACE	





2.2 线性表的存储结构--静态链表(Cont.)

➡ 操作的实现--- ①可用空间初始化

```
void Initialize()
```

```
{
```

```
    int j;
```

```
    /* 依次链接池中结点 */
```

```
    for (j=0; j<maxsize-1; j++ )
```

```
        SPACE[j].next=j+1;
```

```
    /* 最后一个结点指针域为空 */
```

```
    SPACE[j].next = -1;
```

```
    /* 标识线性表 */
```

```
    av=0;
```

```
}
```

av=0 →

	data	next
0	/--/	1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		11
11		12
12		-1

SPACE





2.2 线性表的存储结构--静态链表(Cont.)

➡ 操作的实现--- ②可用空间的分配操作

Cursor GetNode() //p=new spacestr;

```
{ Cursor p;
```

```
    if (SPACE[av].next == -1) //可用空间表为空
```

```
        p = -1;
```

```
    else {
```

```
        p = SPACE[av].next ;
```

```
        SPACE[av].next = SPACE[ p ].next ;
```

```
    }
```

```
    return p;
```

```
} /* 从存储池SPACE中删除结点*/
```

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5

SPACE





2.2 线性表的存储结构--静态链表(Cont.)

➡ 操作的实现--- ③可用空间的回收操作

```
void FreeNode(Cursor q) //delete q;
```

```
{
```

```
    SPACE [ q ].next =
```

```
        SPACE[av].next ;
```

```
    SPACE[av].next = q ;
```

```
} /* 放回存储池SPACE中*/
```

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5

SPACE





2.2 线性表的存储结构--静态链表(Cont.)

操作的实现--- ④插入操作

```
void Insert ( ElemType x, position p, spacestr  
*SPACE )
```

```
{   position q ;  
    q = GetNode( ) ;  
    SPACE[ q ].data = x ;  
    SPACE[ q ].next=SPACE[ p ].next ;  
    SPACE[ p ].next = q ;  
}  
// q = new celltype ;  
// q→data = x ;  
// q→next = p→next ;  
// p→next = q ;
```

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5

SPACE





2.2 线性表的存储结构--静态链表(Cont.)

操作的实现--- ⑤删除操作

```
void Delete(position p, spacestr *SPACE)
{
    position q;
    if ( SPACE[ p ].next != -1 ) {
        q = SPACE[ p ].next;
        SPACE[ p ].next = SPACE[ q ].next;
        FreeNode( q );
    }
}
```

```
// p→next !=NULL
// q = p→next;
// p→next = q→next;
// delete q;
```

	data	next
av→0		9
1	d	6
2	c	-1
M→3		1
4	a	10
5		8
6	e	-1
L→7		4
8		-1
9		11
10	b	2
11		12
12		5

SPACE





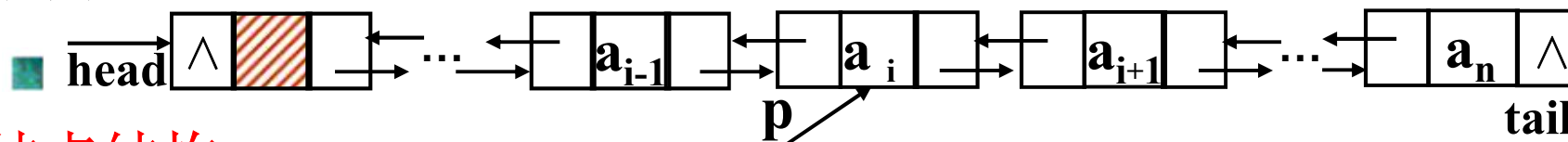
2.2 线性表的存储结构--双向链表

2.2.4 双向链表

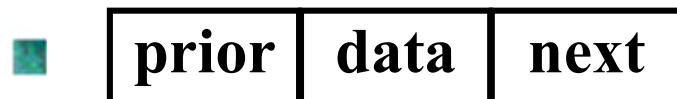
双链表:

- 在单链表的各结点中再设置一个指向其前驱结点的指针域

示例:



结点结构:



优点:

- 实现双向查找（单链表不易做到）
- 表中的位置 i 可以用指示含有第 i 个结点的指针表示。

缺点: 空间开销大。

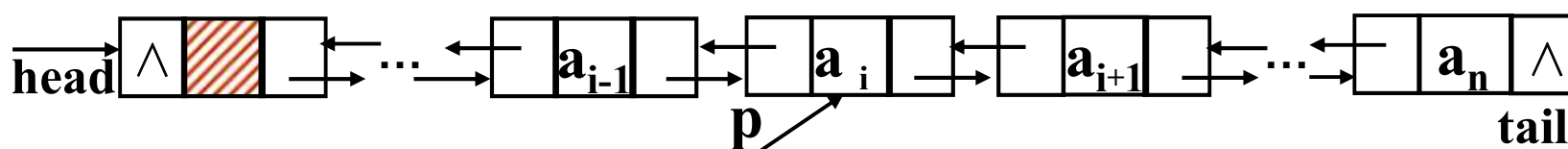
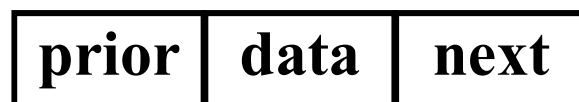




2.2 线性表的存储结构--双向链表(Cont.)

➡ 存储结构定义:

```
struct dcelltype {  
    ElemType data ;  
    dcelltype *next, *prior ;  
}; /* 结点类型 */  
/* 表和位置的类型 */  
typedef dcelltype *DLIST ;  
typedef dcelltype *position ;
```





2.2 线性表的存储结构--双向链表(Cont.)

➡ **插入操作：** 在带头结点的表中, 在位置 p 插入元素 x :

```
void Insert( ElemType x, position p, DLIST &L )
```

```
{   s = new dcelltype;
```

```
    s->data = x;
```

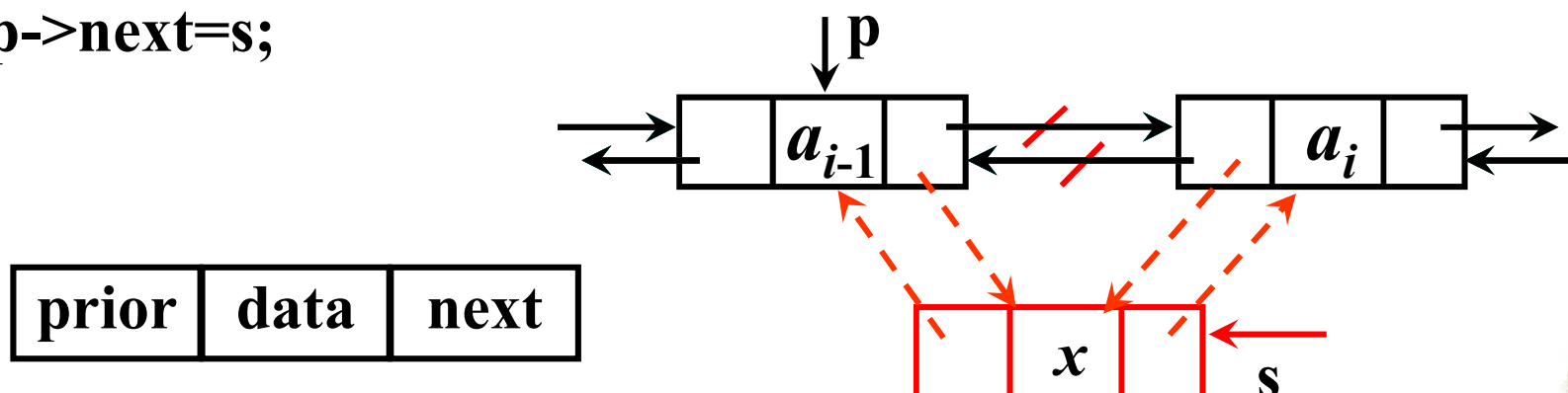
```
    s->prior=p;
```

```
    s->next=p->next;
```

```
    p->next->prior=s;
```

```
    p->next=s;
```

```
}
```





2.2 线性表的存储结构--双向链表(Cont.)

➡ **删除操作**：在**不带头结点**的表中,删除位置**p**的元素：

```
void Delete( position p, DLIST &L)
```

```
{   if (p->prior!=NULL)
```

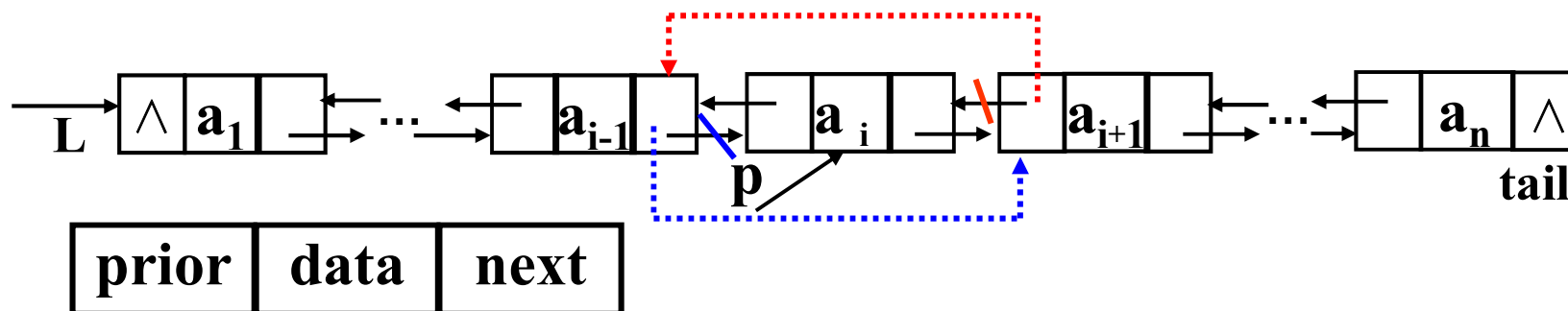
```
    p->prior->next = p->next;
```

```
    if (p->next!=NULL)
```

```
        p->next->prior = p->prior;
```

```
    delete p;
```

```
}
```





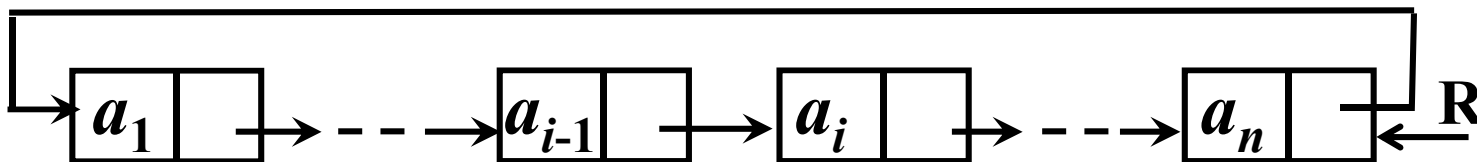
2.2 线性表的存储结构--单向环形链表

2.2.5 单向环形链表

➡ 单向环形链表

- 在(不带表头结点)的单向链表中, 使末尾结点的指针域指向头结点, 得到一个环形结构; 用指向末尾结点的指针标识这个表。

- 示例: 空表 $R=NULL$



➡ 结点结构:

- 同单链表

➡ 存储结构定义:

- 同单链表



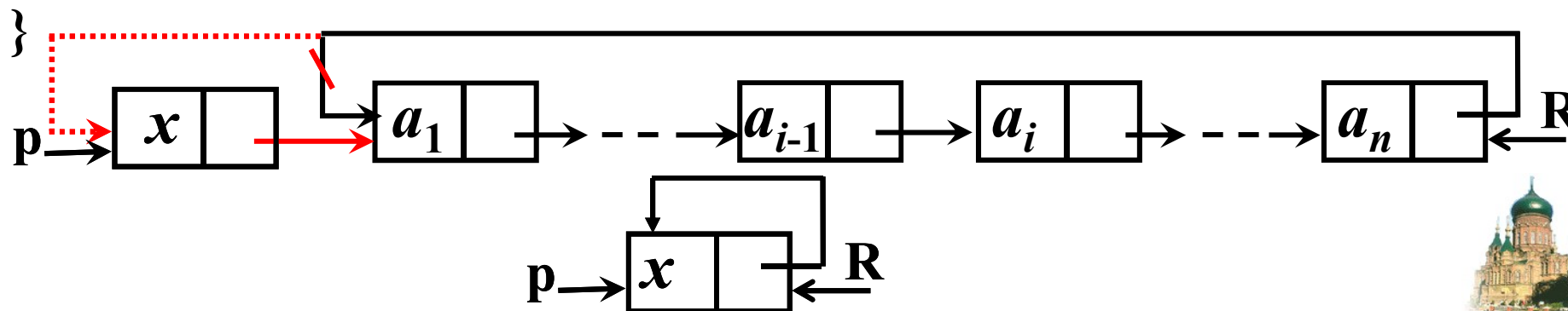


2.2 线性表的存储结构--单向环形链表(Cont.)

➡ 在表左端插入结点 $\text{LInsert}(x, R) \rightarrow \text{Insert}(x, \text{First}(R), R)$

```
void LInsert( Elementtype x , LIST &R )
```

```
{   celltype *p ;  
    p = new celltype ;  
    p→data = x ;  
    if ( R == NULL )  
        { p→next = p ; R = p ; }  
    else  
        { p→next = R→next ; R→next = p ; }
```

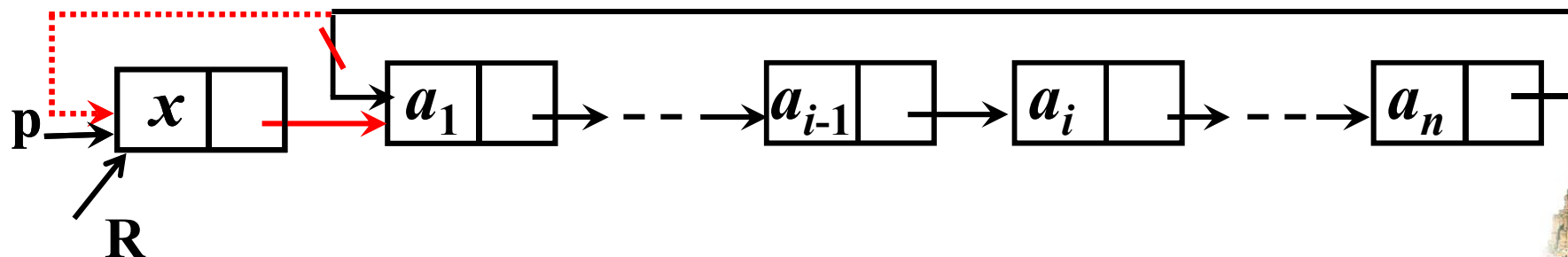
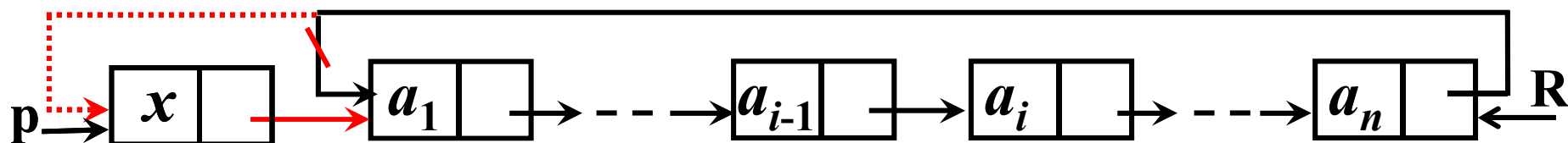




2.2 线性表的存储结构--单向环形链表(Cont.)

➡ 在表右端插入结点 $\text{RInsert}(x, R) \rightarrow \text{Insert}(x, \text{End}(R), R)$

```
void RInsert( ElemType x , LIST R )  
{  
    LInsert ( x , R ) ;  
    R = R→next ;  
}
```





2.2 线性表的存储结构--单向环形链表(Cont.)

➡ 用循环链表求解约瑟夫问题

■ 约瑟夫问题:

n 个人围成一个圆圈, 首先, 第1个人从1开始, 顺时针报数, 报到第 m 个人, 令其出列。然后再从下一个人开始, 从1顺时针报数, 报到第 m 个人, 再令其出列, ..., 如此下去, 直到圆圈中只剩一个人为止。此人即为优胜者。

■ 算法基本思路?

■ 用什么数据结构?





2.2 线性表的存储结构--单向环形链表(Cont.)

➡ 用循环链表求解约瑟夫问题

■ 约瑟夫问题的解法:

```
void Josephus ( List &Js, int n, int m)
{   celltype *p=Js, *pre=NULL;
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<m-1; j++) {
            pre=p; p=p->next;
        }
        cout<< "出列的人是" <<p->data<<endl;
        pre->next =p->next; delete p;
        p=pre->next;
    }
}
```





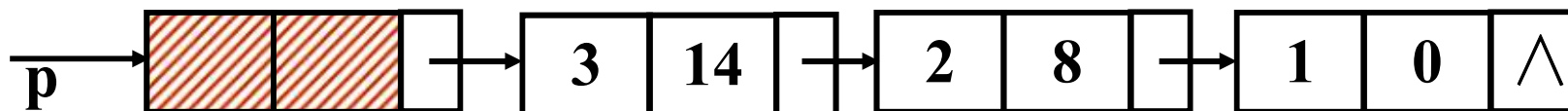
2.2 线性表的存储结构--一元多项式

2.2.6 多项式的代数运算

➤ 多项式: $p(x) = 3x^{14} + 2x^8 + 1$

➤ 存储表示: 采用单链表表示

➤ 示例:



➤ 存储结构定义:

```
struct polynode {
```

```
    int coef; //系数
```

```
    int exp;  //指数
```

```
    polynode *link; //指向下一项的指针
```

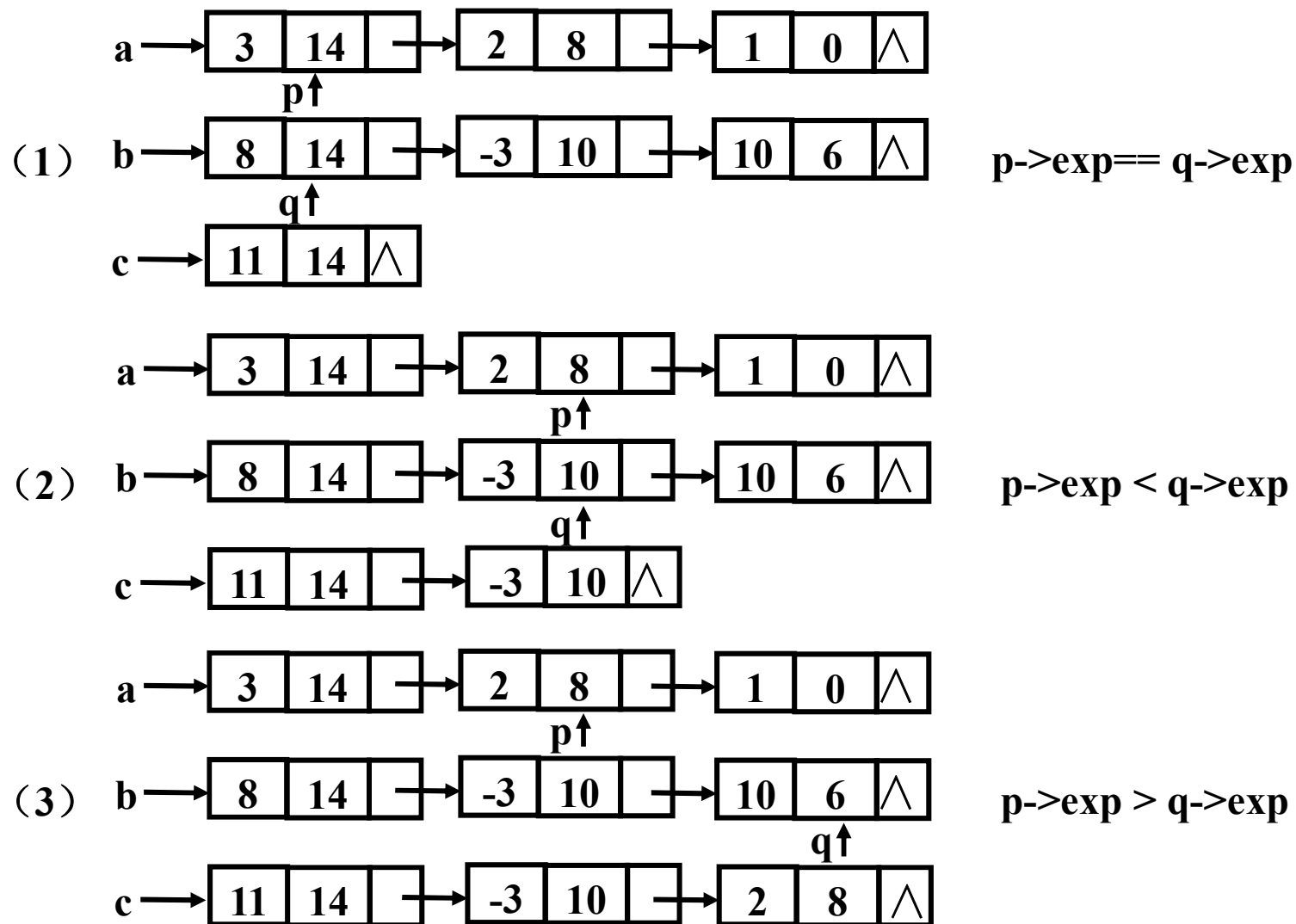
```
}; //结点类型
```

```
typedef polynode *polypointer; //多项式的类型
```



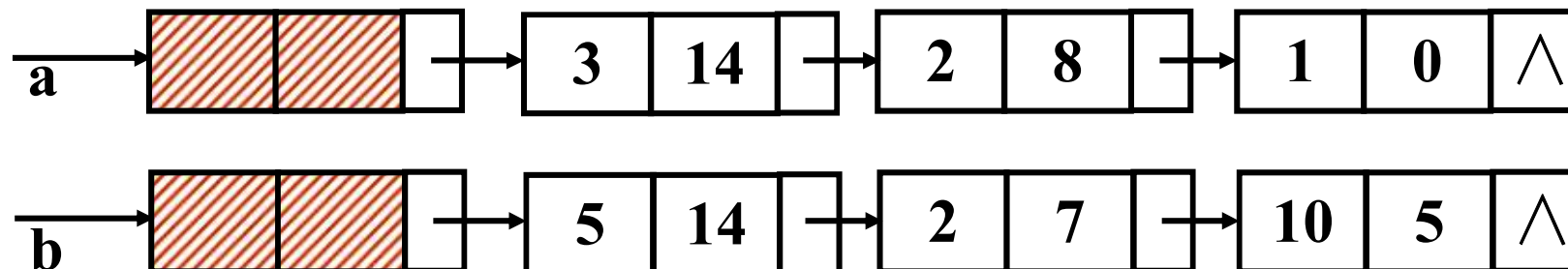


2.2 线性表的存储结构--一元多项式(Cont.)





2.2 线性表的存储结构--一元多项式(Cont.)

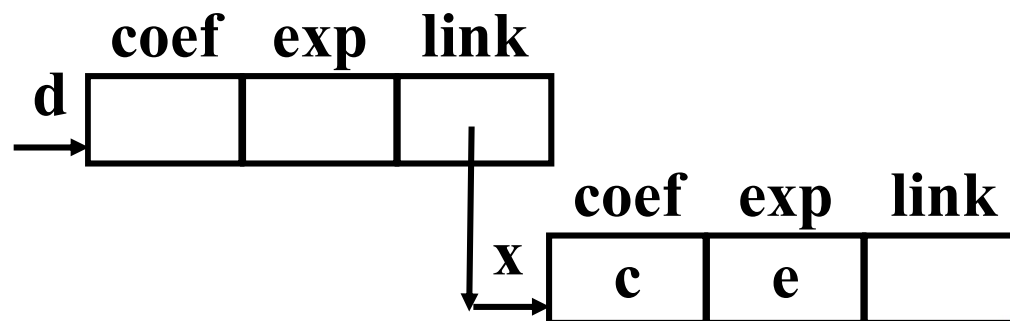


➔ 算法Attch(c, e, d):

- 建立一个新结点，其系数coef = c，指数exp = e；并把它链到 d 所指结点之后，返回该结点指针。

polypointer Attch (int c , int e , polypointer d)

```
{ polypointer x ;  
  x = new polynode ;  
  x→coef = c ;  
  x→exp = e ;  
  d→link = x ;  
  return x ;  
}
```





2.2 线性表的存储结构--一元多项式(Cont.)

多项式加法:

```
polypointer PolyAdd ( polypointer a , polypointer b )
{
    polypointer p, q, d, c;
    int y ;
    p = a→link; q = b→link;
    c = new polynode; d = c ;
    while ( (p != NULL) && (q != NULL) )
        switch ( Compare ( p→exp, q→exp ) )
        {
            case '=' :
                y = p→coef + q→coef ;
                if ( y ) d = Attch( y, p→exp, d );
                p = p→link ; q = q→link ;
                break;
            case '>':
                d = Attch( p→coef, p→exp, d );
                p = p→link ;
                break;
            case '<':
```





2.2 线性表的存储结构--一元多项式(Cont.)

➡ 多项式加法:

```
d = Attch( q→coef, q→exp, d );  
q = q→link ;  
break;
```

```
}  
while ( p != NULL )  
{  
    d = Attch( p→coef, p→exp, d );  
    p = p→link ;  
}  
while ( q !=NULL )  
{  
    d = Attch( q→coef, q→exp, d );  
    q = q→link ;  
}  
d→link = NULL ;  
p = c; c = c→link;  
delete p;  
return c;
```

```
}
```





2.2 线性表的存储结构--一元多项式(Cont.)

时间复杂性:

- $O(m+n)$ 其中, m 和 n 分别是两个多项式最高次幂

一元多项式链表的建立?

一元多项式的减法

- 如何利用加法运算?

一元多项式的乘法

- 如何利用加法运算并有效地减少空间以提高效率

一元多项式的除法

- 给出商多项式和余多项式





2.3 特殊的线性表--栈

➤ **栈**：限定仅在一端进行插入和删除操作的线性表。

➤ **空栈**：不含任何数据元素的栈。

➤ **栈顶和栈底**

■ 允许插入（入栈、进栈、压栈）和删除（出栈、弹栈）的一端称为**栈顶**，另一端称为**栈底**。

➤ **栈的示意图**

➤ **栈的操作**

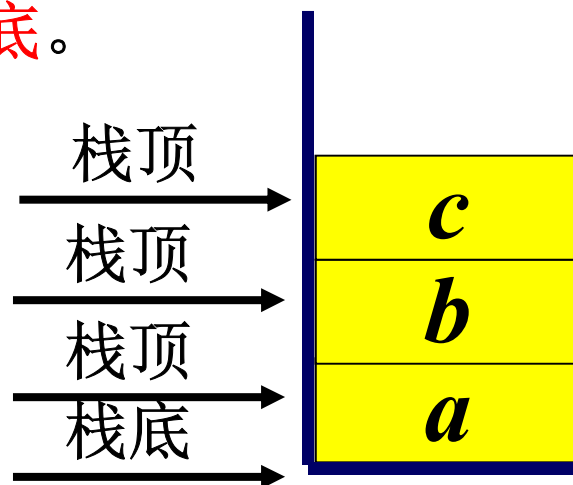
■ ① **MakeNull (S)**

■ ② **Top (S)**

■ ③ **Pop (S)**

■ ④ **Push (x , S)**

■ ⑤ **Empty (S)**



栈的特性：后进先出

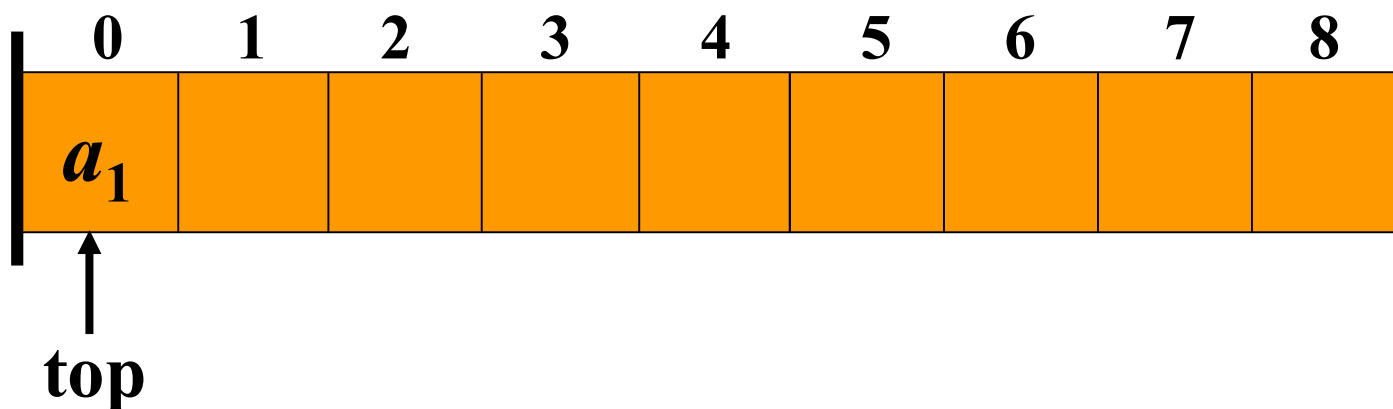




2.3.1 栈的数组实现--顺序栈

➡ 栈的顺序存储结构及实现

❓ 如何改造数组实现栈的顺序存储？



确定用数组的哪一端表示栈底。

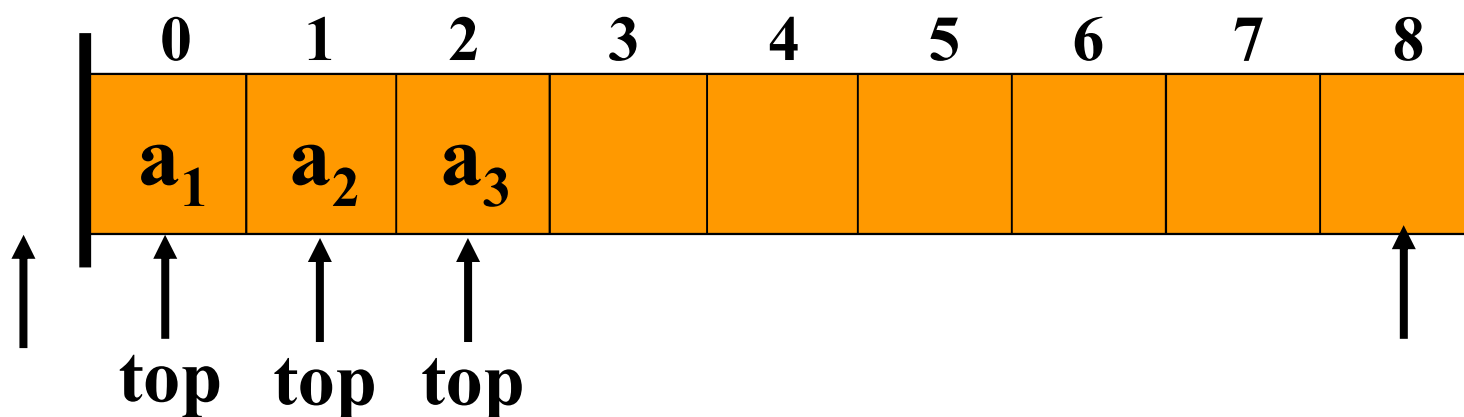
附设指针**top**指示栈顶元素在数组中的位置。





2.3.1 栈的数组实现--顺序栈(Cont.)

➡ 栈的顺序存储结构及实现



进栈: top 加1

栈空: $\text{top} = -1$

出栈: top 减1

栈满: $\text{top} = \text{max}-1$





2.3.1 栈的数组实现--顺序栈(Cont.)

➡ 栈的顺序存储结构定义

```
typedef struct {  
    ElemType elements[max];  
    int top ;  
} STACK ;
```

■ STACK S ;

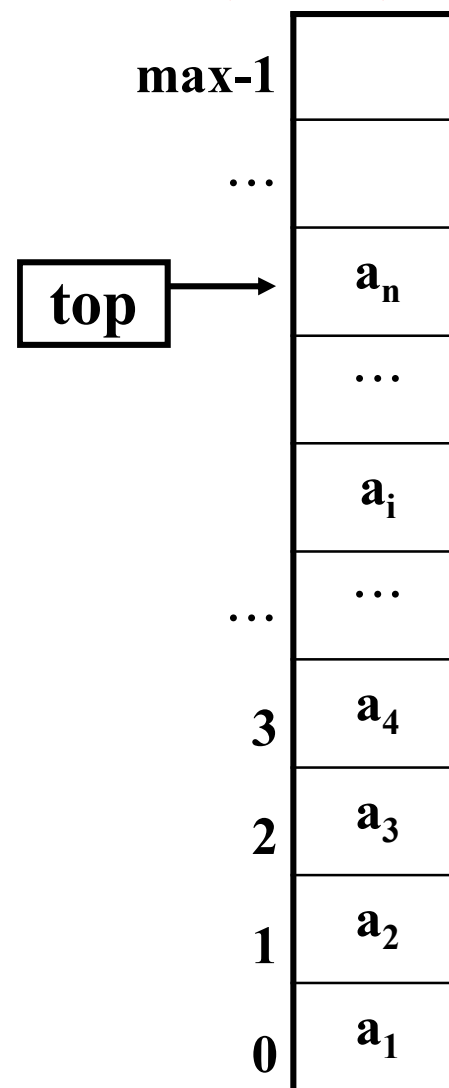
■ 栈的容量: max

■ 栈顶指针: S.top

■ 栈顶元素: S.elements[S.top] ;

■ 栈空: S.top = -1 ;

■ 栈满: S.top = max-1 ;



顺序栈示意图



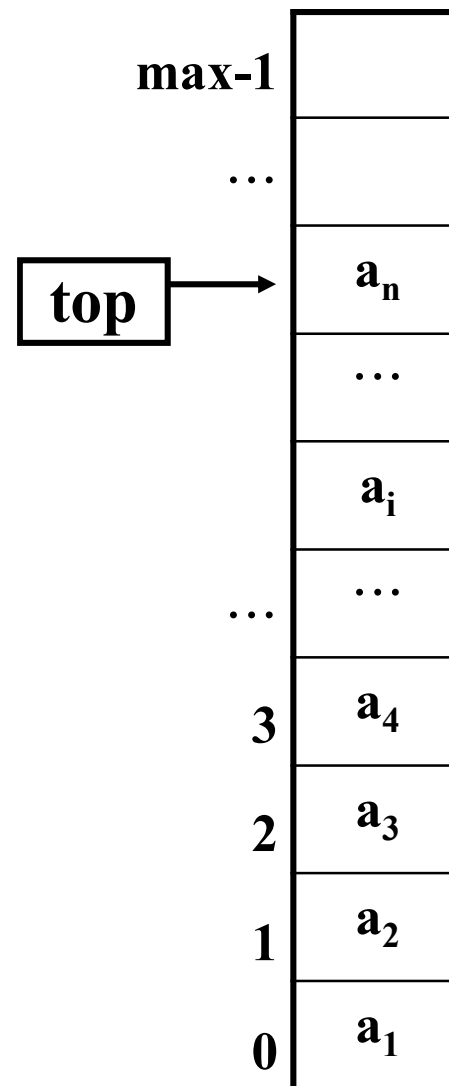


2.3.1 栈的数组实现--顺序栈(Cont.)

➡ 栈的操作的实现

① **void MakeNull(STACK &S)**
 { **S.top = -1 ;** }

② **boolean Empty(STACK S)**
 { **if (S.top < 0)**
 return TRUE
 else
 return FALSE ;
 }



顺序栈示意图

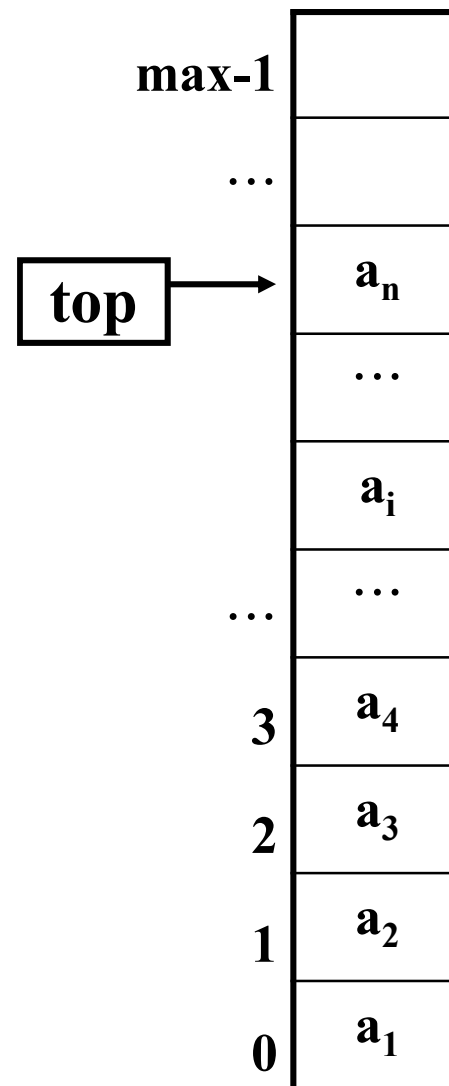




2.3.1 栈的数组实现--顺序栈(Cont.)

```
③ ElemTtype Top( STACK S )
{ if ( Empty( S )
    return NULLES;
  else
    return ( S.elements[ S.top ] );
}

④ void Pop( STACK &S )
{
  if ( Empty (S ) )
    cout<< “栈空” ;
  else
    S.top = S.top - 1 ;
}
```



顺序栈示意图

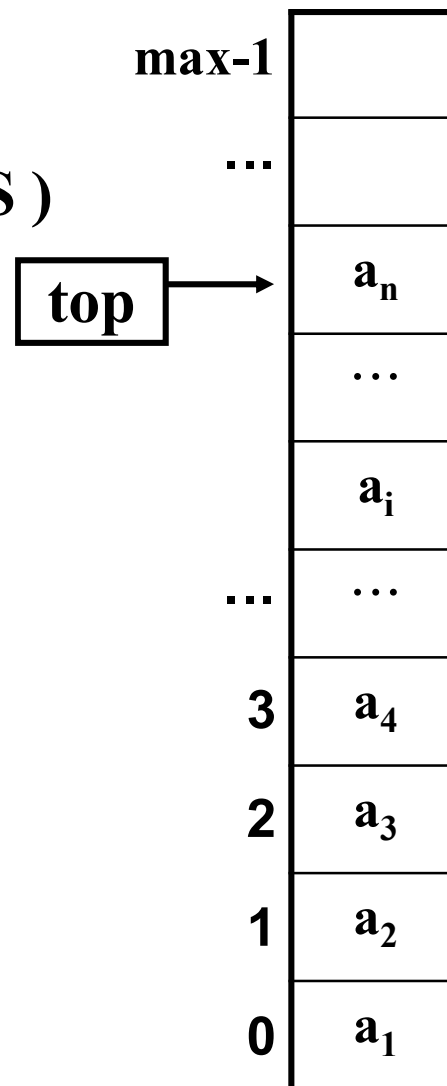




2.3.1 栈的数组实现--顺序栈(Cont.)

➡ 栈的操作的实现

```
⑤ void Push ( ElemTtype  x, STACK  &S )  
    {  
        if ( S.top == max - 1 )  
            cout<< “栈满” ;  
        else  
        {  
            S.top = S.top + 1 ;  
            S.elements[ S.top ] = x ;  
        }  
    }
```



顺序栈示意图



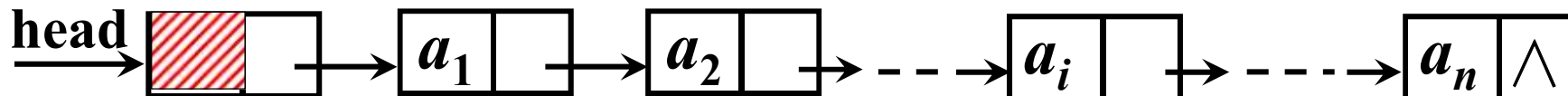


2.3.2 栈的指针实现--链栈

➡ 栈的链接存储结构及实现

■ 链栈：栈的链接存储结构

① 如何改造链表实现栈的链接存储？



① 将哪一端作为栈顶？ 将链表首端作为栈顶，方便操作。

① 链栈需要加头结点吗？ 表头结点的作用与链表相同。

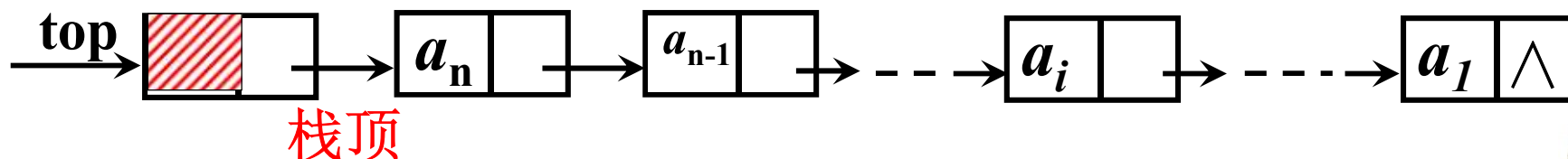
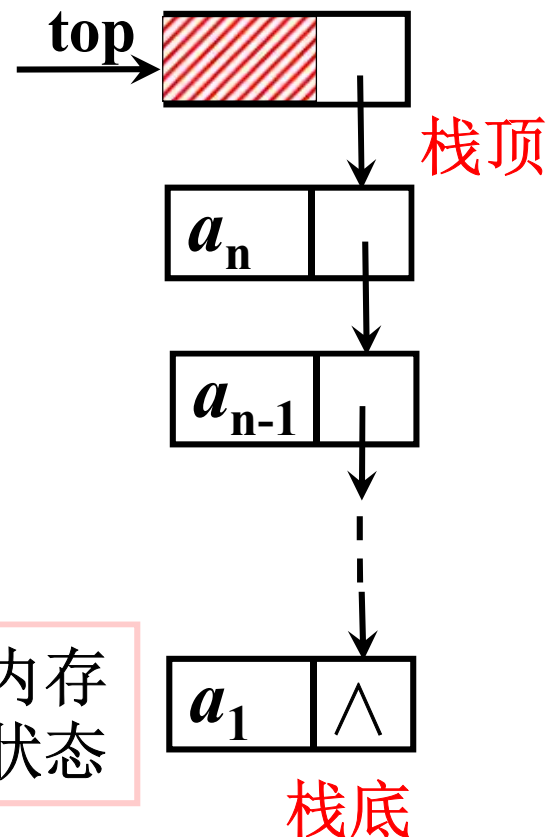




2.3.2 栈的指针实现--链栈(Cont.)

➡ 栈的链式存储结构定义

```
struct node{  
    ElemType data;  
    node *next;  
}; //结点的"型"  
typedef node *STACK; //栈的"型"
```



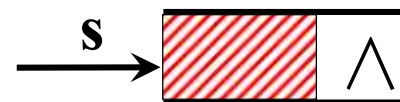


2.3.2 栈的指针实现--链栈(Cont.)

➡ 栈的操作的实现

① STACK MakeNull()

```
{  STACK s;  
    s=new node;  
    /*s=(node *)malloc(sizeof(node));*/  
    s->next=NULL;  
    return s;  
}
```



② boolean Empty(STACK stk)

```
{    if (stk->next)  
        return FALSE;  
    else  
        return TRUE;  
}
```





2.3.2 栈的指针实现--链栈(Cont.)

➡ 栈的操作的实现

③ void Push(Elementtype elm, STACK stk)

{

STACK s;

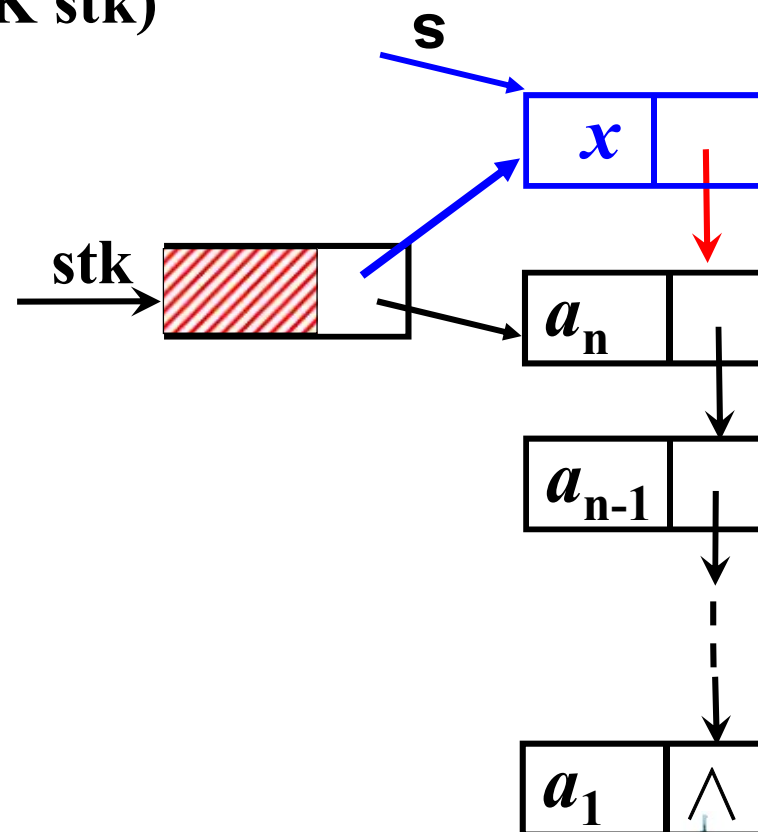
s=new node;

s->data=elm;

s->next=stk->next;

stk->next=s;

}



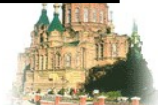
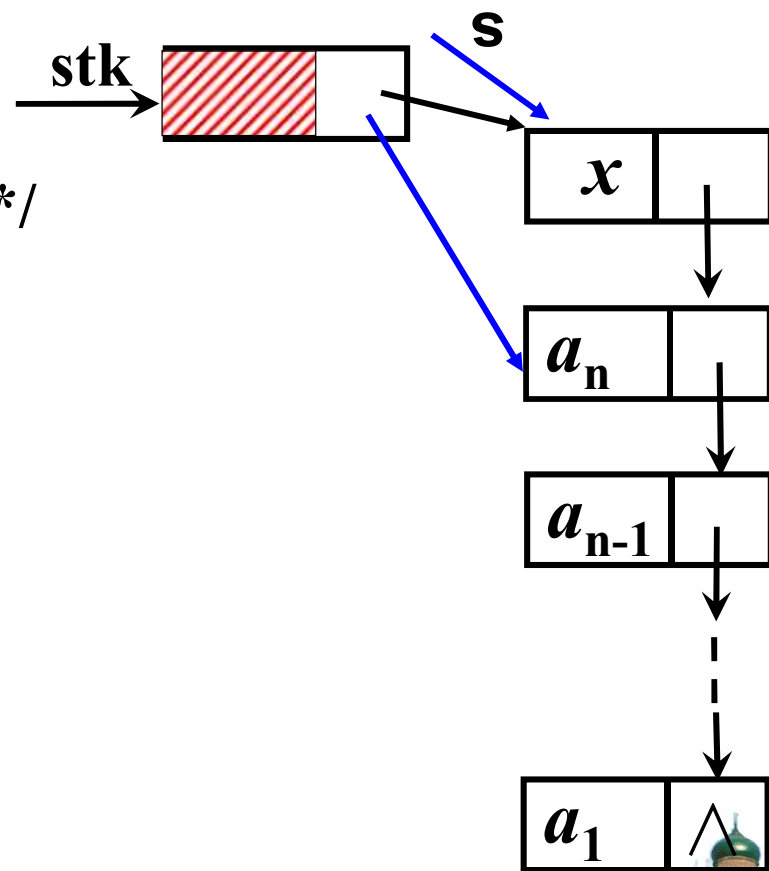


2.3.2 栈的指针实现--链栈(Cont.)

➡ 栈的操作的实现

④ void Pop(STACK stk)

```
{  STACK s;  
    if (stk->next){/*stk->next!=NULL*/  
        s=stk->next;  
        stk->next=s->next;  
        delete s; /* free(s) */  
    }  
}
```



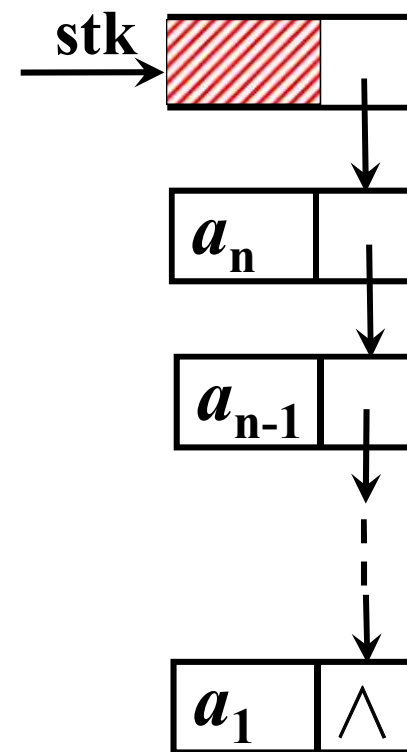


2.3.2 栈的指针实现--链栈(Cont.)

➡ 栈的操作的实现

⑤ ElemType Top(STACK stk)

```
{  if (stk->next)
    return (stk->next->data);
    else
    return NULLES;
}
```





2.3.3 栈与递归调用

➤ 递归调用的定义

- 子程序（或函数或方法）**直接调用自己**或通过一系列调用语句**间接调用自己**。是一种**描述问题**和**解决问题**的基本方法。

➤ 递归的基本思想

- 把一个不能或不好求解的**大问题**转化为一个或几个**小问题**，再把这些小问题进一步分解成更小的小问题，**直至每个小问题都可以直接求解**。

➤ 递归的要素

- **递归边界条件**：确定递归到何时终止，也称为**递归出口**；
- **递归模式**：大问题是如何分解为小问题的，也称为**递归体**





2.3.3 栈与递归调用(Cont.)

➡ 递归调用举例

■ 求阶乘的函数

$$n! = \begin{cases} 1 & , \text{ 当 } n = 0 \text{ 或 } n = 1 \text{ 时} \\ n * (n-1)! & , \text{ 当 } n \geq 1 \text{ 时} \end{cases}$$

■ 递归算法

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

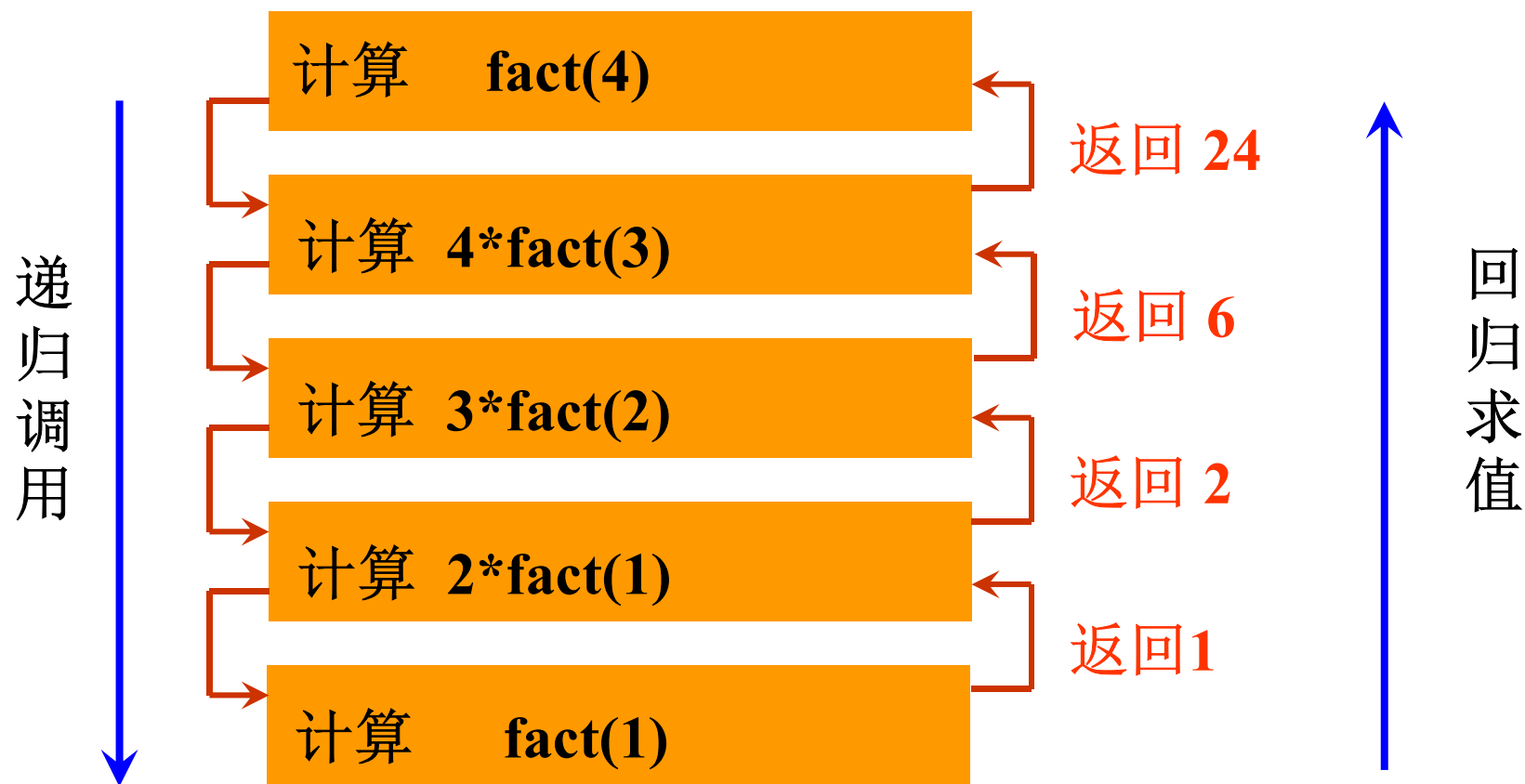




2.3.3 栈与递归调用(Cont.)

递归调用举例

求解阶乘 $n!$ 的过程

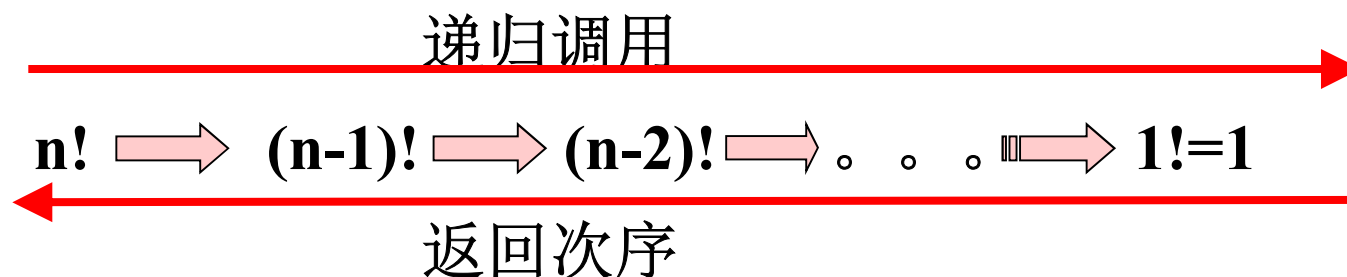




2.3.3 栈与递归调用(Cont.)

➡ 递归过程与递归工作栈

- 递归过程在实现时，需要自己直接或间接调用自己。
- 层层向下递归，返回次序正好相反：





2.3.3 栈与递归调用(Cont.)

➡ 递归过程与递归工作记录

- 每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。

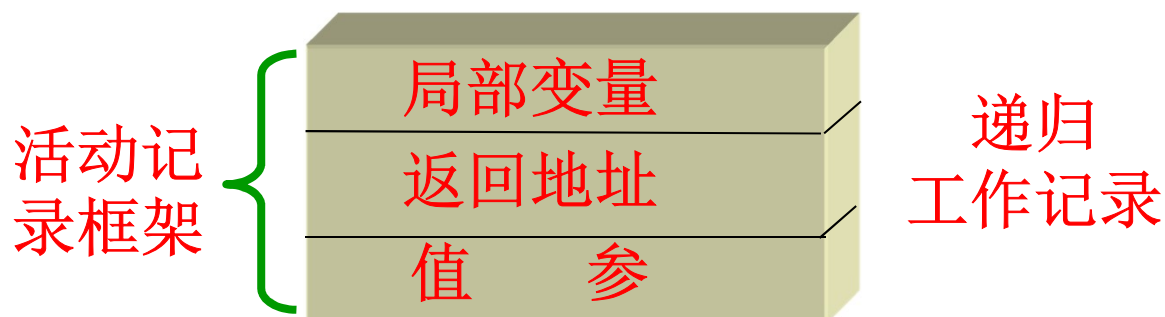




2.3.3 栈与递归调用(Cont.)

➡ 递归函数的内部执行过程

- (1) **建栈**：运行开始时，首先为递归调用**建立**一个工作**栈**，其结构包括值参、局部变量和返回地址；
- (2) **压栈**：每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址**压栈**；
- (3) **出栈**：每次递归调用结束后，将栈顶元素**出栈**，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

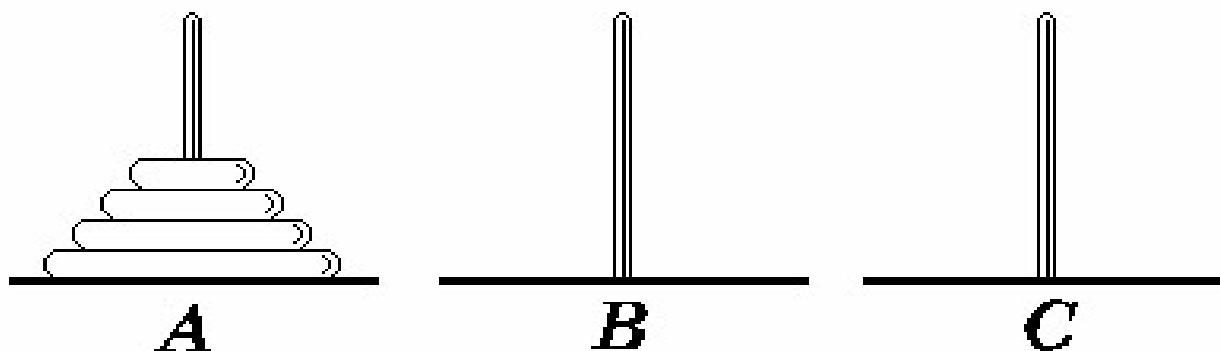




2.3.3 栈与递归调用(Cont.)

➡ 汉诺塔问题——递归的经典问题

- 在世界刚被创建的时候有一座钻石宝塔（塔A），其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。





2.3.3 栈与递归调用(Cont.)

➡ 汉诺塔问题的递归求解:

- 如果 $n = 1$ ，则将这一个盘子直接从 塔A移到塔 C 上。
- 否则，执行以下三步：
 - 将塔A上的 $n-1$ 个碟子借助塔C先移到塔B上；
 - 把塔A上剩下的一个碟子移到塔C上；
 - 将 $n-1$ 个碟子从塔B借助于塔A移到塔C上。





2.3.3 栈与递归调用(Cont.)

➡ 汉诺塔问题的递归求解:

```
void Hanoi(int n, char A, char B, char C)
```

```
{
```

```
    if (n==1) Move(A, C);
```

```
    else {
```

```
        Hanoi(n-1, A, C, B);
```

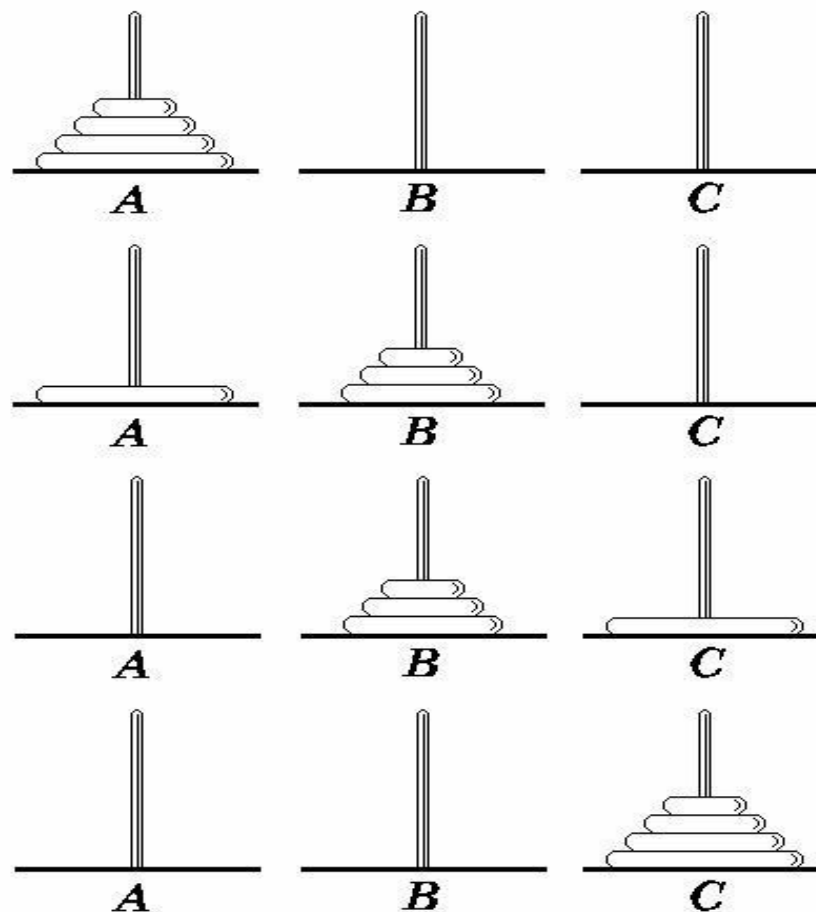
```
        Move(A, C);
```

```
        Hanoi(n-1, B, A, C);
```

```
    }
```

```
}
```

➡ 时间复杂度?





2.3.3 栈与递归调用(Cont.)

➡ 递归函数的运行轨迹

- 写出函数当前调用层执行的各语句，并用有向弧表示**语句的执行次序**；
- 对函数的每个递归调用，写出对应的函数调用，从调用处画一条有向弧指向被调用函数入口，表示**调用路线**，从被调用函数末尾处画一条有向弧指向调用语句的下面，表示**返回路线**；
- 在返回路线上标出本层调用所得的函数值。







2.3.4 栈的应用

➡ **数制转换**----是计算机实现计算的基本问题。

- 方法：除留余数法。例如对输入的任意非负十进制整数,打印输出与其等值的八进制数。

```
void main()
{   STACK s=NEWSTACK();
    cin>>n;
    while(n){
        Push(n%8,s);
        n/=8;
    }
    while(! Empty(s)) {
        cout<<Top(s);
        POP(s) ;
    }
}
```

}//时间复杂度?

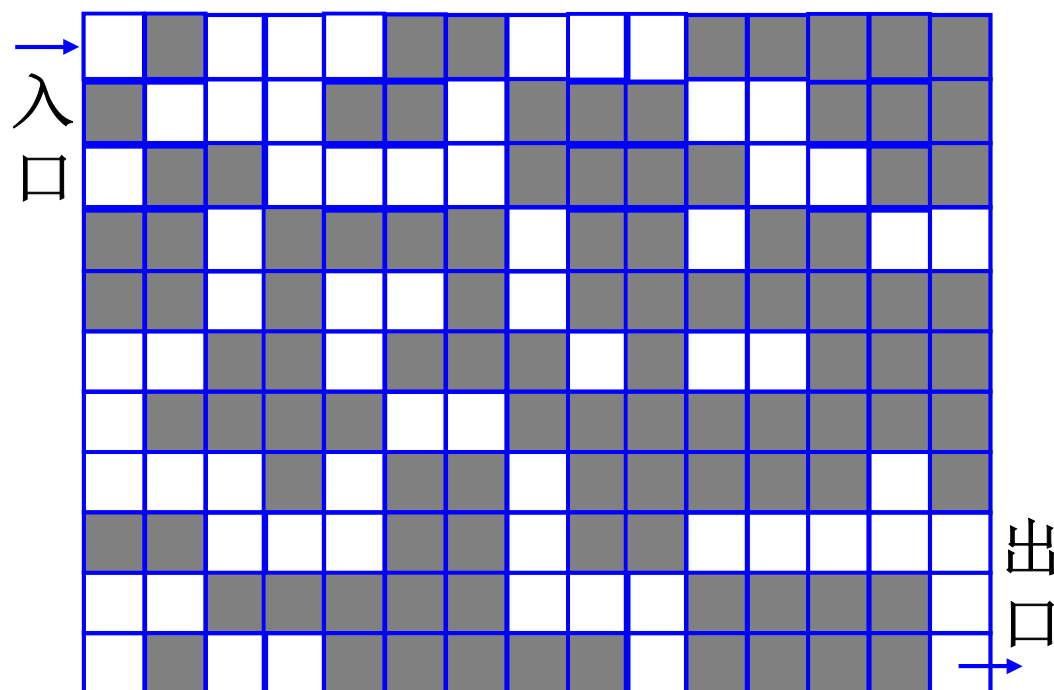




2.3.4 栈的应用(Cont.)

➡ 迷宫求解问题

- 一个迷宫可用下图所示矩阵[m,n]表示，0表示能通过，1表示不能通过。现假设老鼠从左上角[1,1]进入迷宫，设计算法，寻求一条从右下角[m,n]出去的简单路径。



迷宫示例

```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 1 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 1 1 0 1 1 1 1 1 0 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0
```

11×15→m × n





2.3.4 栈的应用(Cont.)

➡ 迷宫求解问题

■ 分析:

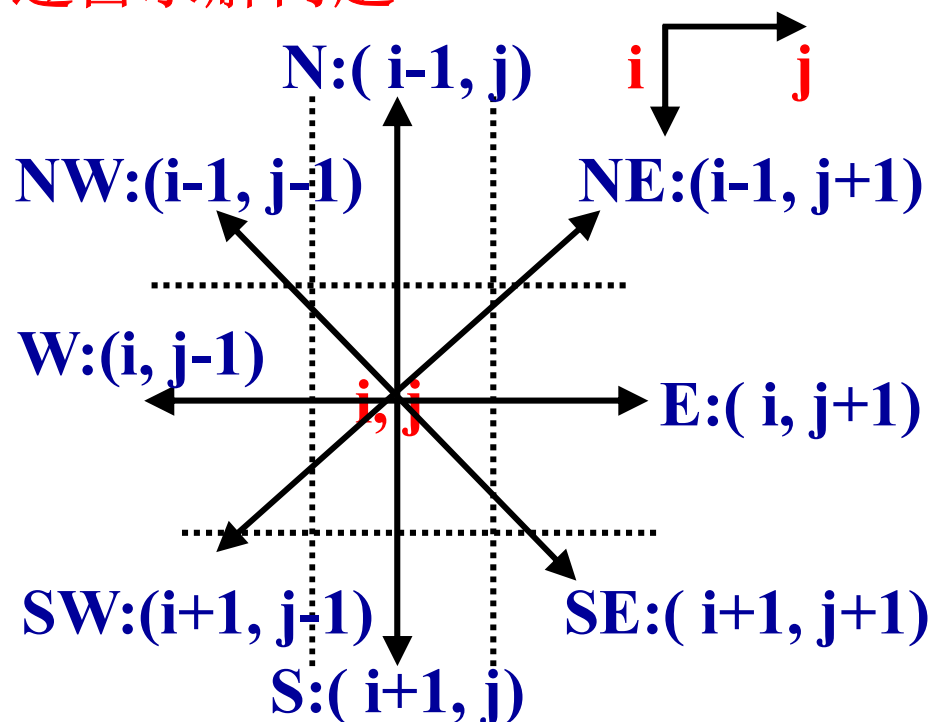
- 迷宫用二维数组 $\text{maze}[i, j]$ ($1 \leq i \leq m, 1 \leq j \leq n$) 表示, 入口 $\text{maze}[1, 1] = 0$; 老鼠在任意位置可用 (i, j) 坐标表示;
- 位置 (i, j) 周围有8个可能走通的方向, 分别记为: E, SE, S, SW, W, NW, N, NE。
- 方向 v 按从正东开始且顺时针分别记为1-8, $v=1, \dots, 8$; 设二维数组 move 记下八个方位的增量;





2.3.4 栈的应用(Cont.)

➡ 迷宫求解问题



v	i	j	说明
1	0	1	E
2	1	1	SE
3	1	0	S
4	1	-1	SW
5	0	-1	W
6	-1	-1	NW
7	-1	0	N
8	-1	1	NE

➡ 如从 (i, j) 到 (g, h) 且 $v = 2$ (东南) 则有:

■ $g = i + \text{move}[2, 1] = i + 1;$

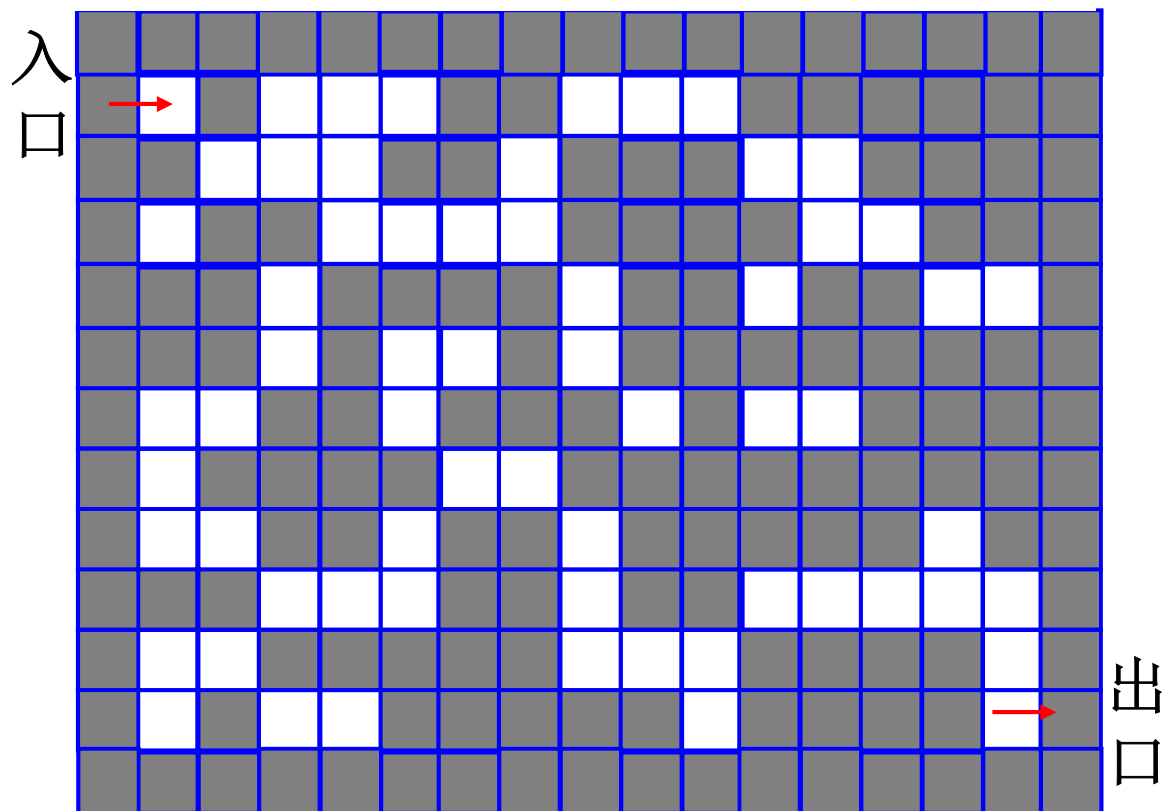
■ $h = j + \text{move}[2, 2] = j + 1;$





2.3.4 栈的应用(Cont.)

- 为避免时时监测边界状态，可把二维数组 $\text{maze}[1:m, 1:n]$ 扩大为 $\text{maze}[0:m+1, 0:n+1]$ ，且令 0 行和 0 列、 $m+1$ 行和 $n+1$ 列的值为 1。



迷宫示例





2.3.4 栈的应用(Cont.)

- 采用**试探**的方法，当到达某个位置且周围八个方向走不通时需要回退到上一个位置，并换一个方向继续试探；
为解决回退问题，需设一个栈，当到达一个新位置时将 (v, i, j) 进栈，回退时退栈。
- 每次换方向寻找新位置时，需测试该位置以前是否已经经过，对已到达的位置，**不能重复试探**，为此设矩阵 **mark**，其初值为0，一旦到达位置 (i, j) 时，置 $mark[i, j] = 1$;





2.3.4 栈的应用(Cont.)

■ 算法描述

- (1) 老鼠在(1, 1)进入迷宫，并向正东 ($v=1$) 方向试探
- (2) 监测下一方位(g, h)。若(g, h)=(m, n)且 $\text{maze}[m, n]=0$ ，则老鼠到达出口，输出走过的路径；程序结束。
- (3) 若(g, h) \neq (m, n)，但(g, h)方位能走通且第一次经过，则记下这一步，并从(g, h)出发，再向东试探下一步。否则仍在(i, j)方位换一个方向试探。
- (4) 若(i, j)方位周围8个方位阻塞或已经过，则需退一步，并换一个方位试探。若(i, j)=(1, 1)则到达入口，说明迷宫走不通。





2.3.4 栈的应用(Cont.)

■ 算法求精

```
void GETMAZE ( maze , mark ,move ,s )
{ (i, j, v) = (1,1,1);  mark[1, 1] = 1;  top = 0;
  do { g = move[v, 1]; h = move[v, 2];
    if (( g == m) && ( h == n) && (maze[m, n] == 0 ))
      { output( S ); return ; }
    if ((maze[g, h] ==0) && mark[g, h] == 0))
      { mark[g, h] = 1; Push( i, j, v, s ); (i, j,v) = (g, h,1) ; }
    else if ( v < 8 )
      v = v + 1 ;
    else { while (( s.v == 8) && (!Empty(s))) POP( s );
      if ( top > 0 )
        (i, j, v++) = Pop( s );  } ;
    } while (( top ) && ( v != 8 ));
  cout << “路径不存在! ” ;
}
```





2.3.4 栈的应用(Cont.)

➡ **表达式求值**：编译器和科学计算器设计中最基本的一个问题

➡ **表达式的三种形式** 表达式： $\begin{cases} \text{前缀表达式(波兰式)} \\ \text{中缀表达式} \\ \text{后缀表达式(逆波兰式)} \end{cases}$

$$\text{例如, } (a+b)*(a-b) = \begin{cases} * + a b - a b \\ (a+b)*(a-b) \\ a b + a b - * \end{cases}$$

■ 高级语言中，采用类似自然语言的**中缀表达式**，但计算机对中缀表达式的处理是很困难的，而对**后缀或前缀表达式**则显得**非常简单**。

■ 后缀表达式的特点：

- 在**后缀表达式**中，**变量（操作数）**出现的**顺序**与**中缀表达式顺序**相同。
- 后缀表达式中**不需要括号**规定计算顺序，而由运算操作符）的**位置**来确定运算顺序。





2.3.4 栈的应用(Cont.)

	+	-	*	/	()
+	>	>	<	<	<	>
-	>	>	<	<	<	>
*	>	>	>	>	<	>
/	>	>	>	>	<	>
(<	<	<	<	<	=
)	>	>	>	>		>

表达式求值—算符优先法

I. 将中缀表达式转换成后缀表达式（操作符栈）

$$(a+b)*(a-b) \Rightarrow a \ b \ + \ a \ b \ - \ *$$

- 对 中缀表达式 从左至右依次扫描，因操作数的顺序保持不变，所以，当遇到操作数时直接输出；
- 为调整运算顺序，设立一个栈用以保存操作符，扫描到操作符时，将操作符压入栈中：
 - 进栈的原则是保持栈顶操作符的优先级要高于栈中其他操作符的优先级；
 - 否则，将栈顶操作符依次退栈并输出，直到满足要求为止；
- 当遇到“(”进栈；当遇到”)”时，退栈输出直到“(”为止。





2.3.4 栈的应用(Cont.)

➡ 表达式求值—算符优先法

II. 由后缀表达式计算表达式的值（操作数栈）

$$(a+b)*(a-b) \Rightarrow a \ b \ + \ a \ b \ - \ *$$

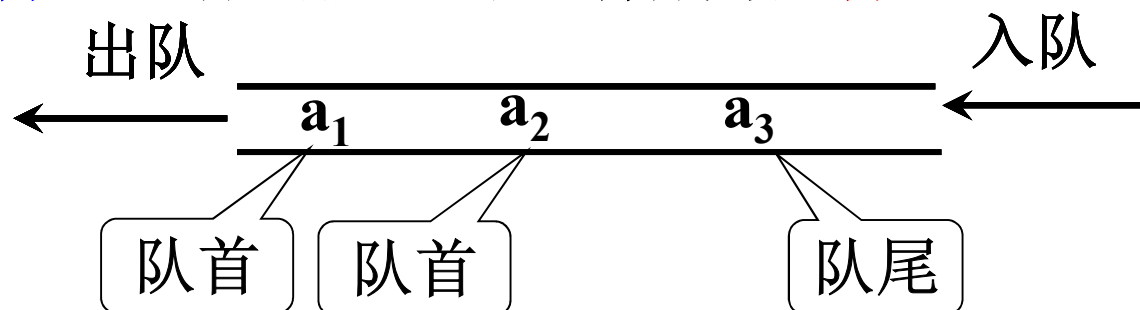
- ➡ 对后缀表达式从左至右依次扫描，与 I 相反，遇到操作数时，将操作数进栈保存；
- ➡ 当遇到操作符时，从栈中退出两个操作数并作相应运算，将计算结果进栈保存；直到表达式结束，栈中唯一元素即为表达式的值。
- ➡ 思考题：合并 I 和 II 的算符优先法





2.4 特殊的线性表--队列

- **队列**：只允许在一端进行插入操作，而另一端进行删除操作的线性表。
- **空队列**：不含任何数据元素的队列。
- **队尾和队首**：允许插入（也称入队、进队）的一端称为队尾，允许删除（也称出队）的一端称为队首。



队列的操作特性：先进先出

- **队列的操作**：

■ **MakeNull(Q)**、**Front(Q)**、**EnQueue(x, Q)**、**DeQueue(Q)**、**Empty(Q)**

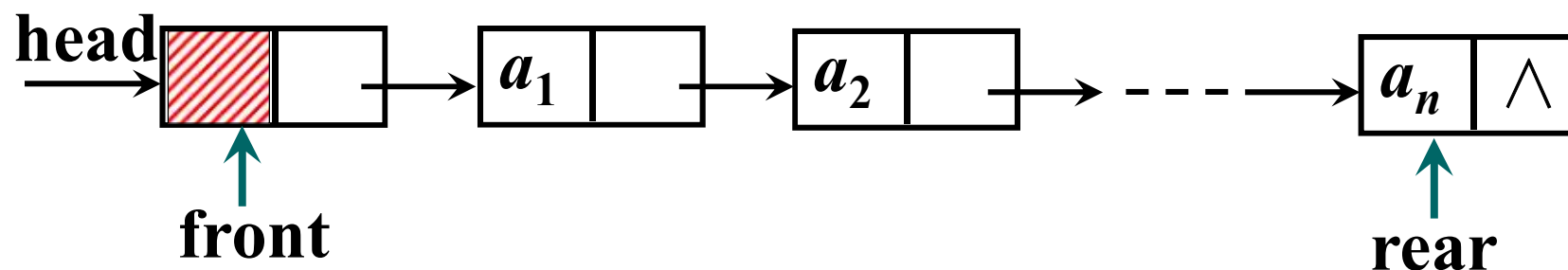




2.4.1 队列的指针实现

➤ 队列的链接存储结构及实现

- 链队列：队列的链接存储结构
- 如何改造单链表实现队列的链接存储？



- 队首指针即为链表的头结点指针
- 增加一个指向队尾结点的指针

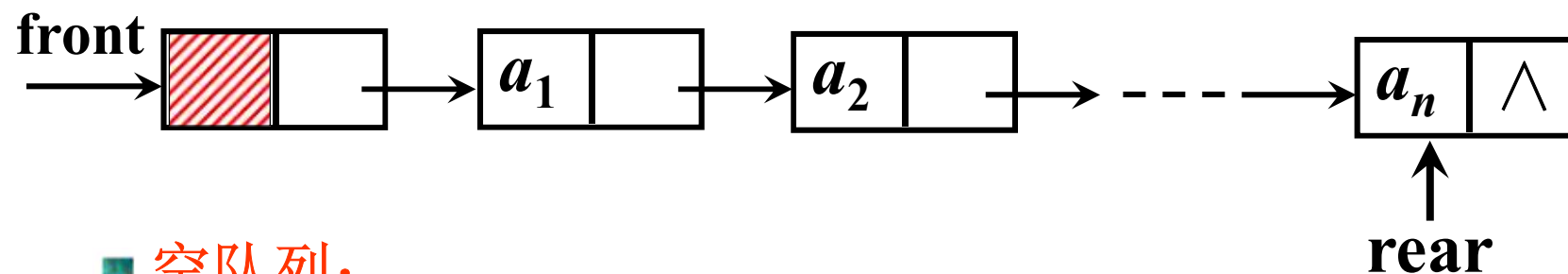




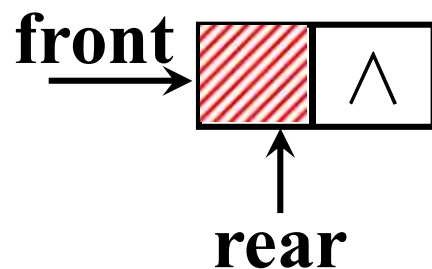
2.4.1 队列的指针实现(Cont.)

➡ 队列的链接存储结构及实现

■ 非空队列:



■ 空队列:





2.4.1 队列的指针实现(Cont.)

➡ 队列的链接存储结构及实现

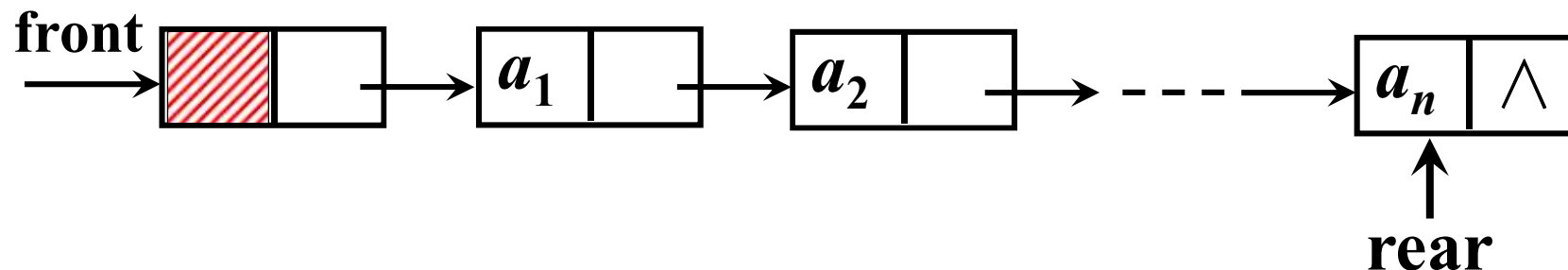
■ 存储结构定义

//结点的类型:

```
struct celltype {  
    ElemType data;  
    celltype *next;  
};
```

队列的 类型:

```
struct QUEUE {  
    celltype *front;  
    celltype *rear;  
};
```





2.4.1 队列的指针实现(Cont.)

➡ 队列的链接存储结构及实现

■ 操作的实现----初始化和判空

① **void MakeNull(QUEUE &Q)**

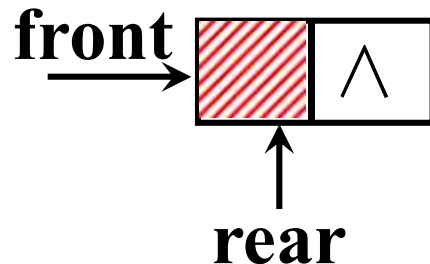
{

Q.front = new celltype ;

Q.front→next = NULL ;

Q.rear = Q.front ;

}



② **Boolean Empty(QUEUE &Q)**

{ if (Q.front == Q.rear)

return TRUE ;

else

return FALSE ;

}

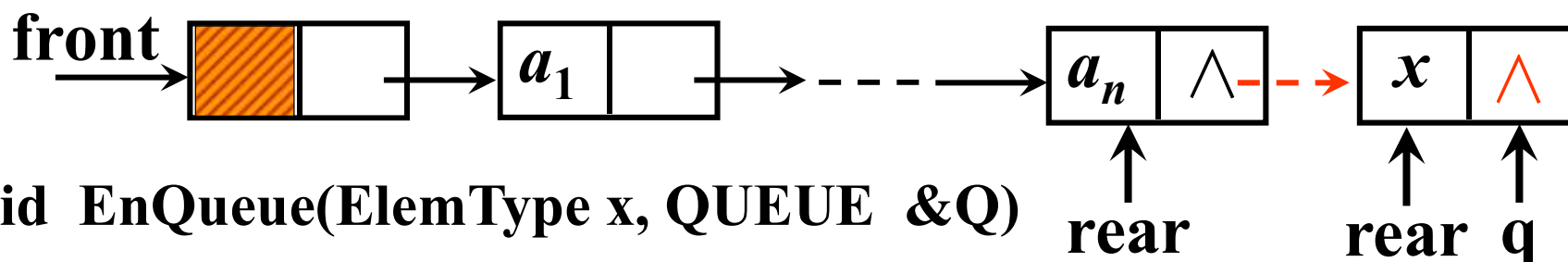




2.4.1 队列的指针实现(Cont.)

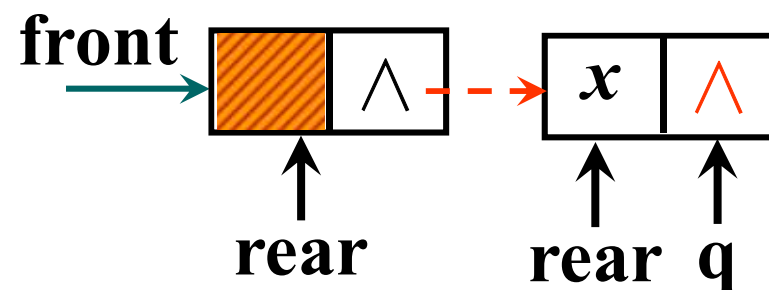
➡ 队列的链接存储结构及实现

■ 操作的实现----入队



③ void EnQueue(ElemType x, QUEUE &Q)

```
{  q=new cwltype;
   q->data=x ;
   q->next=NULL;
   //q->next=Q.rear->next;
   Q.rear->next=q;
   Q.rear=q;
}
```



④ 如果没有头结点会怎样?

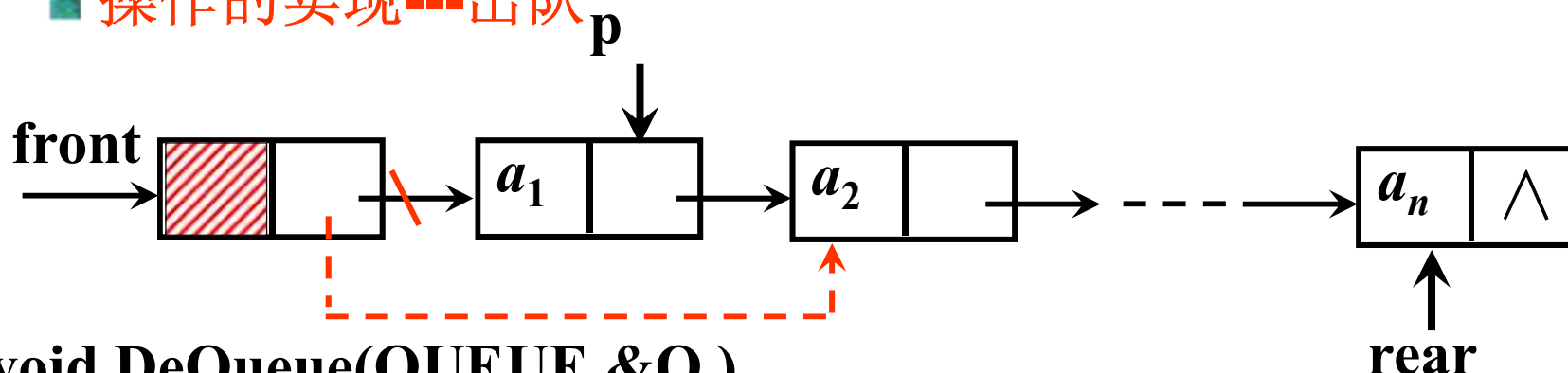




2.4.1 队列的指针实现(Cont.)

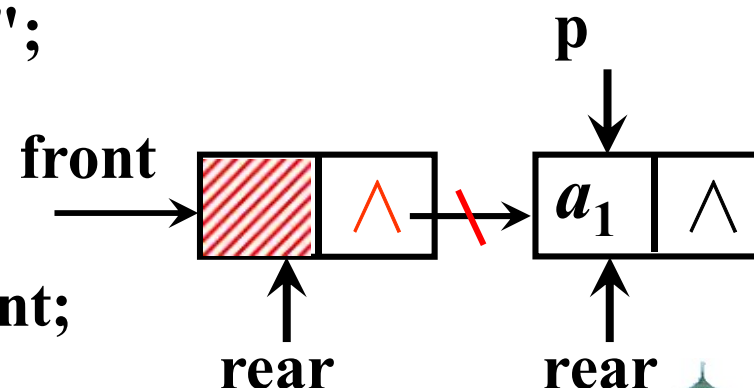
➤ 队列的链接存储结构及实现

■ 操作的实现---出队



```
void DeQueue(QUEUE &Q )
```

```
{  if (Q.rear==Q.front) cout<<"队空";  
    p=Q.front->next;  
    Q.front->next=p->next;  
    if (p->next==NULL) Q.rear=Q.front;  
    delete p;  
}
```



考虑边界情况：队列中只有一个元素？

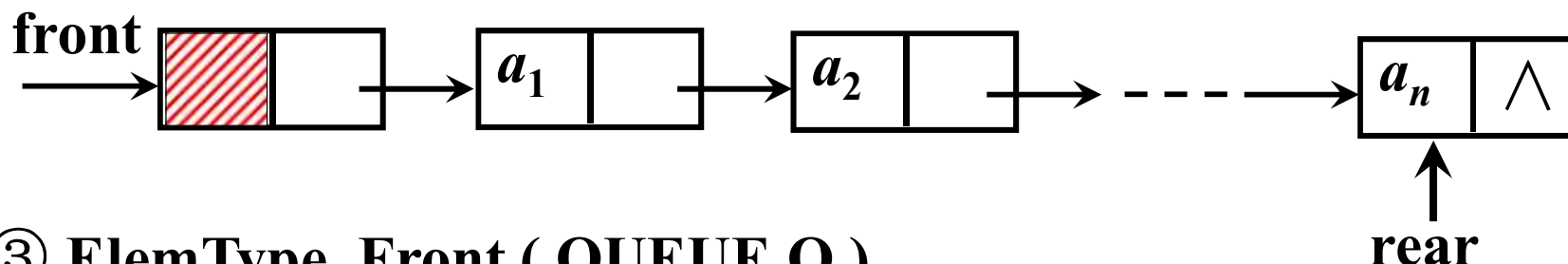




2.4.2 队列的指针实现(Cont.)

➡ 队列的链接存储结构及实现

■ 操作的实现---返回队首元素

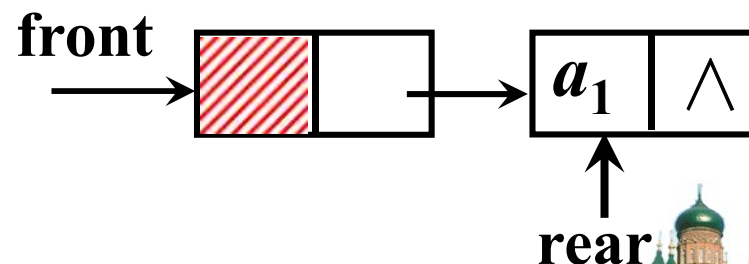


③ ElemType Front (QUEUE Q)

```
{ if ( Q.front→next )
```

```
    return Q.front→next→data ;
```

```
}
```



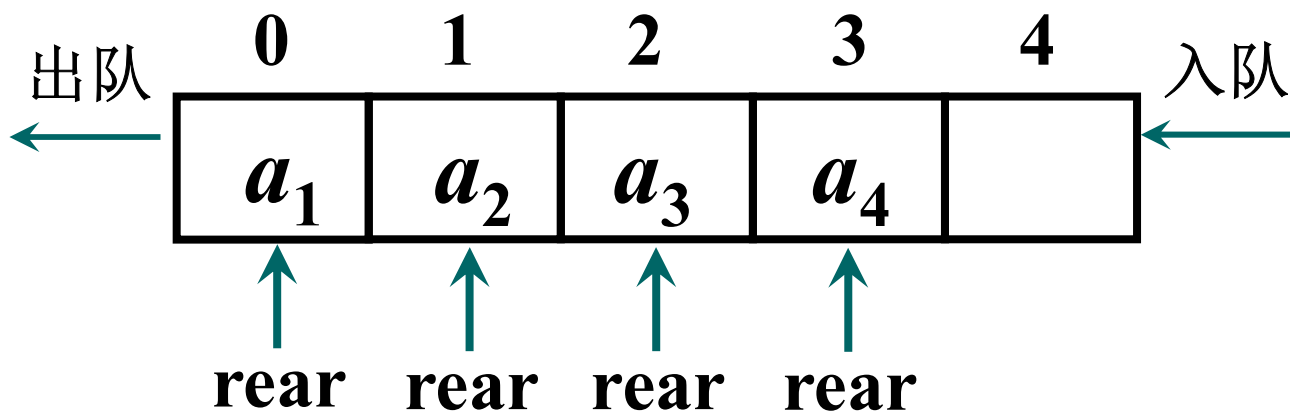


2.4.3 队列的数组实现

队列的顺序存储结构及实现

■ 如何改造数组实现队列的顺序存储？

● $a_1a_2a_3a_4$ 依次入队



入队操作时间性能为 $O(1)$



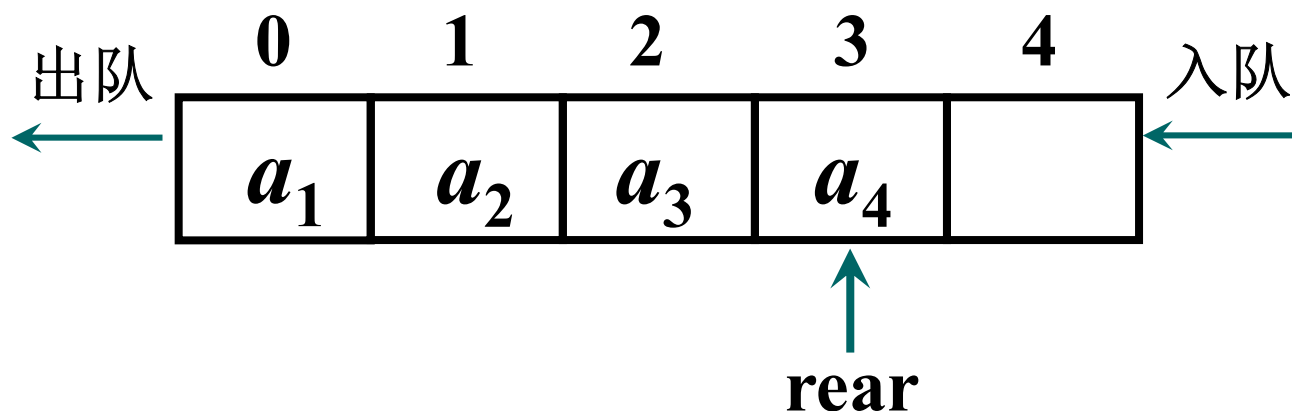


2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何改造数组实现队列的顺序存储?

● $a_1 a_2$ 依次出队



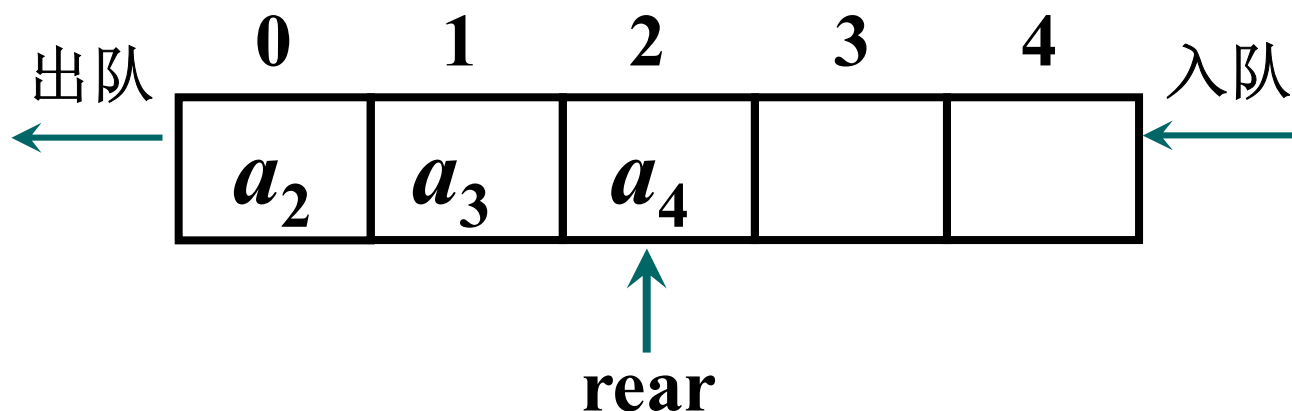


2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何改造数组实现队列的顺序存储?

● $a_1 a_2$ 依次出队



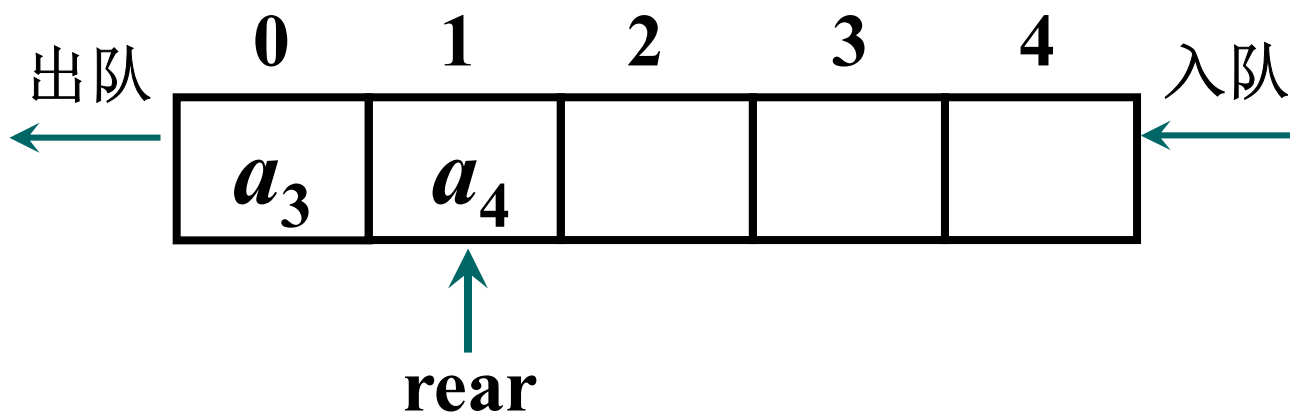


2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何改造数组实现队列的顺序存储？

● $a_1 a_2$ 依次出队



出队操作时间性能为 $O(n)$





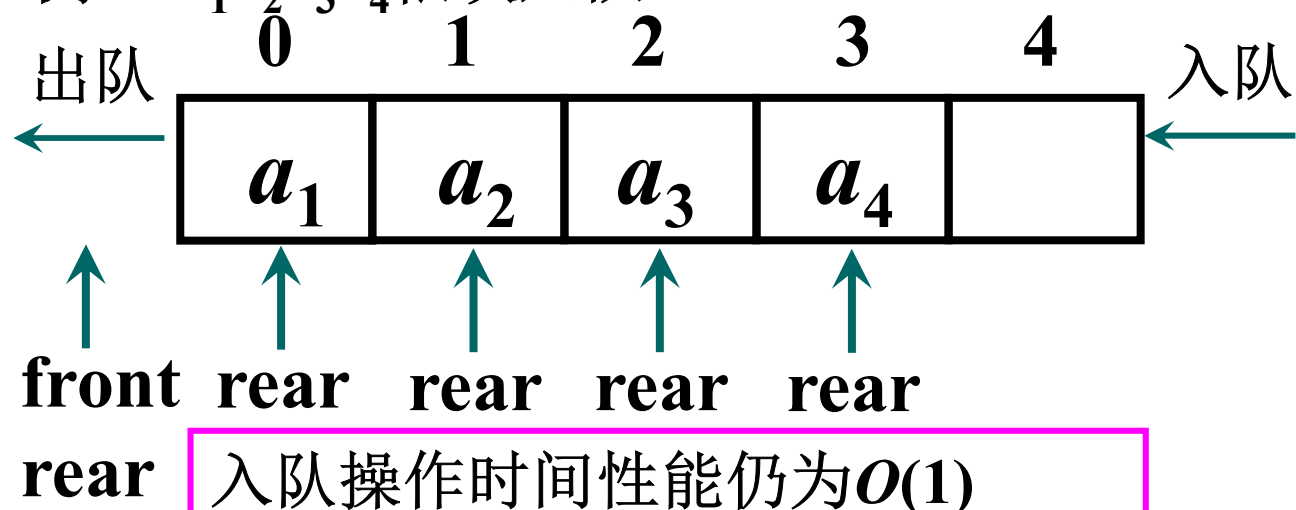
2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何改进出队的时间性能？

- 所有元素不必存储在数组的前 n 个单元；
- 只要求队列的元素存储在数组中连续单元；
- 设置队头、队尾两个指针。

● 例： $a_1 a_2 a_3 a_4$ 依次入队



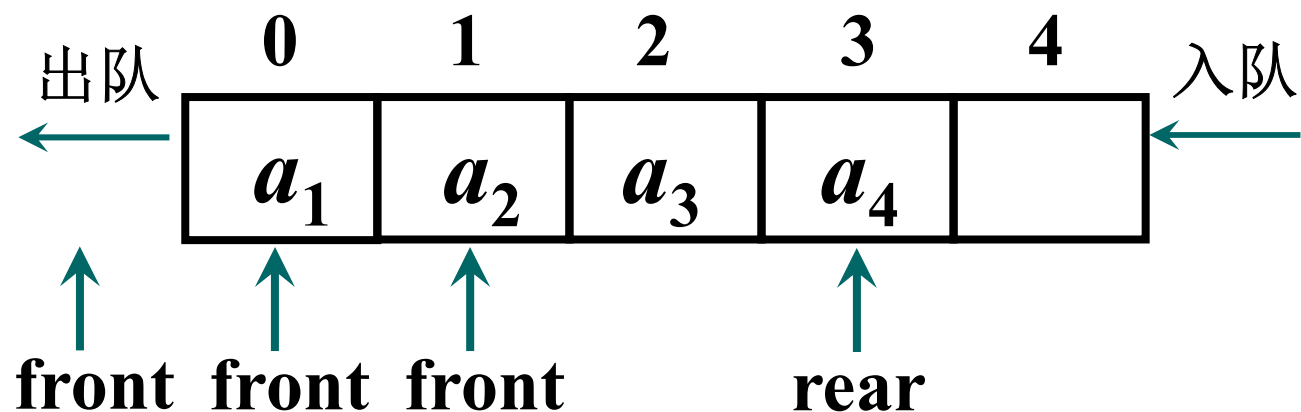


2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何改进出队的时间性能?

● $a_1 a_2$ 依次出队



出队操作时间性能提高为 $O(1)$

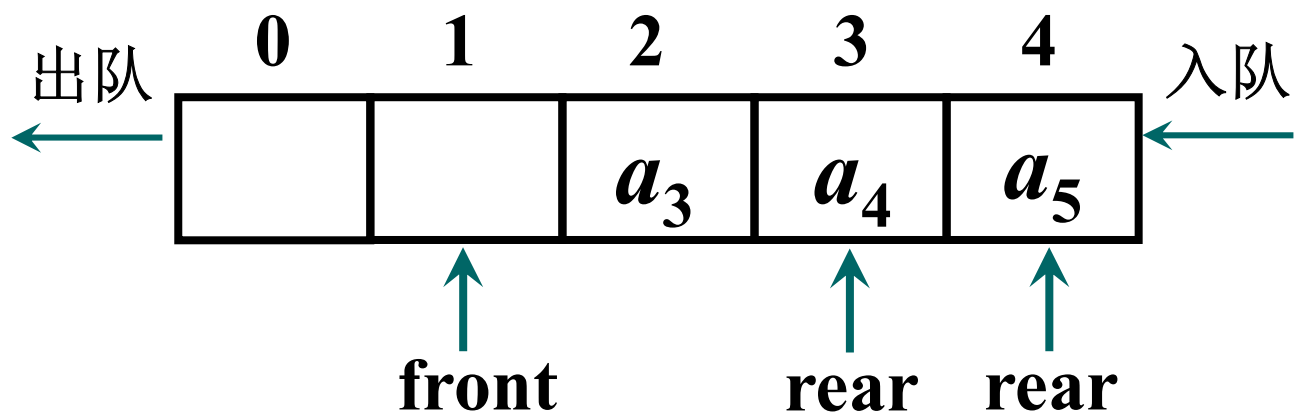




2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 队列的移动有什么特点？



■ 继续入队会出现什么情况？

■ 假溢出：

- 当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，但此时数组的低端还有空闲空间，这种现象叫做**假溢出**。

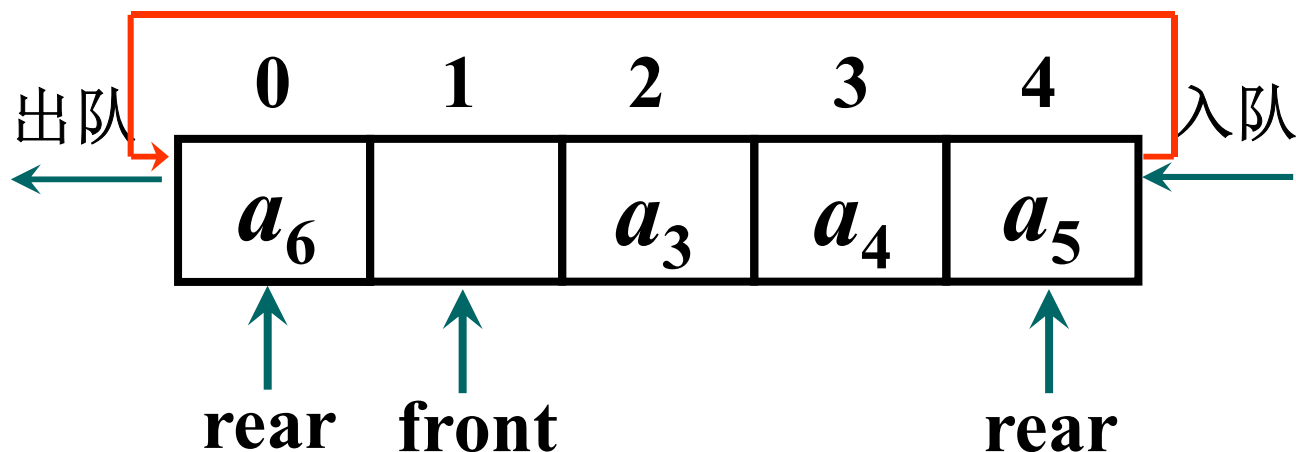




2.4.3 队列的数组实现(Cont.)

队列的顺序存储结构及实现

■ 如何避免假溢出？



- 用循环数组表示队列：将数组最后一个单元的下一个单元看成是0号单元，即把数组头尾相接----按模加1



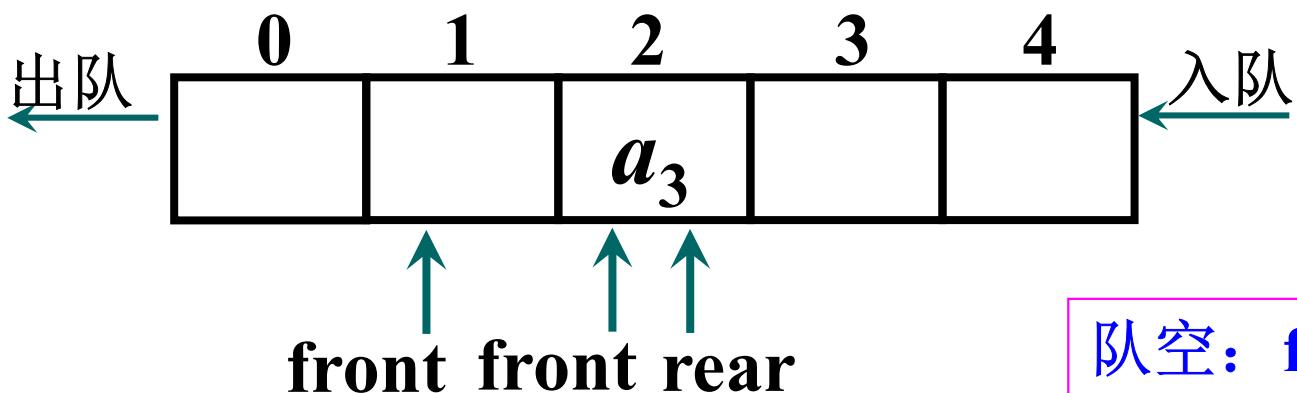
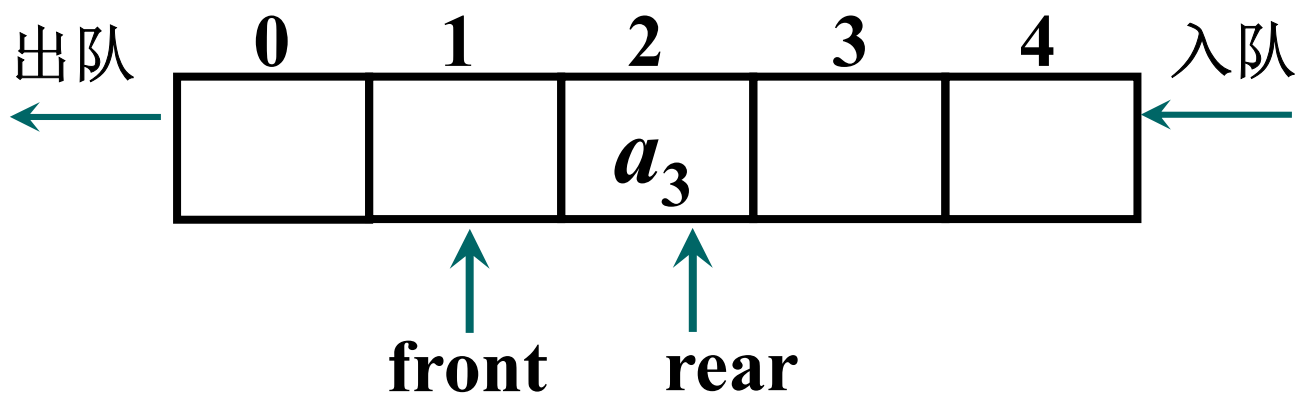


2.4.3 队列的数组实现(Cont.)

队列的顺序存储结构及实现

如何判断循环队列队空和队满?

队空时，front和rear的相对位置



队空: $\text{front} == \text{rear}$



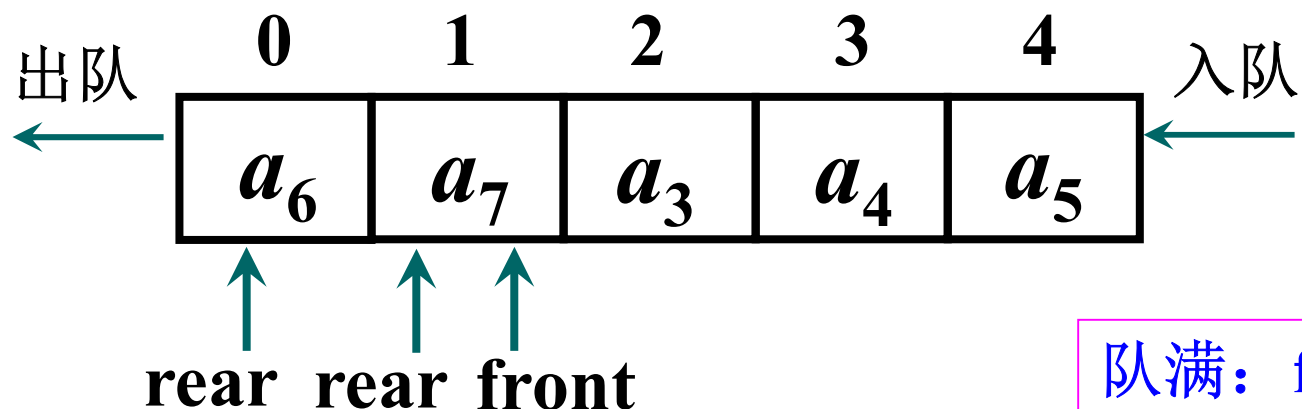
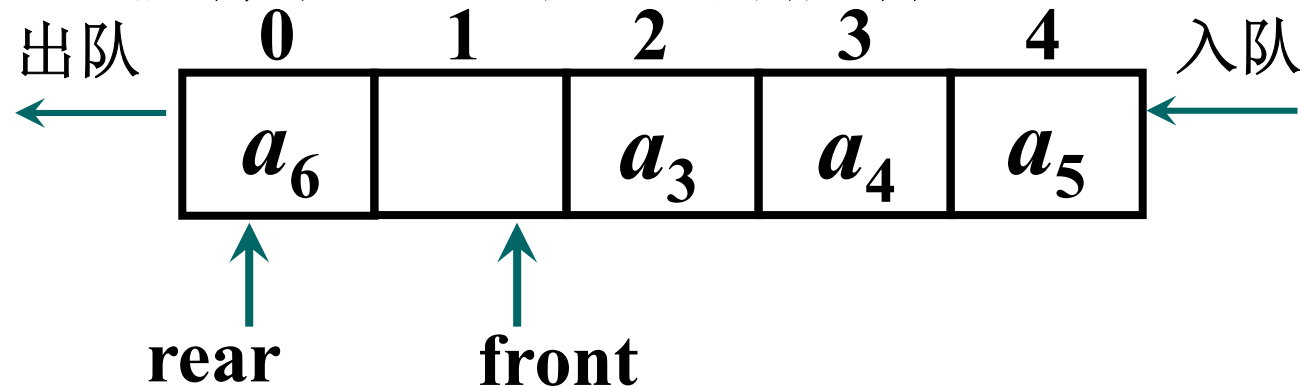


2.4.3 队列的数组实现(Cont.)

队列的顺序存储结构及实现

如何判断循环队列队空和队满?

● 队满时, **front**和**rear**的相对位置



队满: **front==rear**





2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 如何区分队空、队满的判定条件？

● **方法一：** 增设一个存储队列中元素个数的计数器**count**:

- ◆ 当 $\text{front} == \text{rear}$ 且 $\text{count} == 0$ 时，队空；
- ◆ 当 $\text{front} == \text{rear}$ 且 $\text{count} == \text{MaxSize}$ 时，队满；

● **方法二：** 设置标志**flag**,

- ◆ 当 $\text{front} == \text{rear}$ 且 $\text{flag} == 0$ 时为队空；
- ◆ 当 $\text{front} == \text{rear}$ 且 $\text{flag} == 1$ 时为队满。

● **方法三：**

- ◆ 保留队空的判定条件: $\text{front} == \text{rear}$;
- ◆ 队满判定条件修改为: $(\text{rear} + 1) \% \text{MaxSize} == \text{front}$
- ◆ **代价：** 浪费一个元素空间，队满时数组中有一个空闲单元；



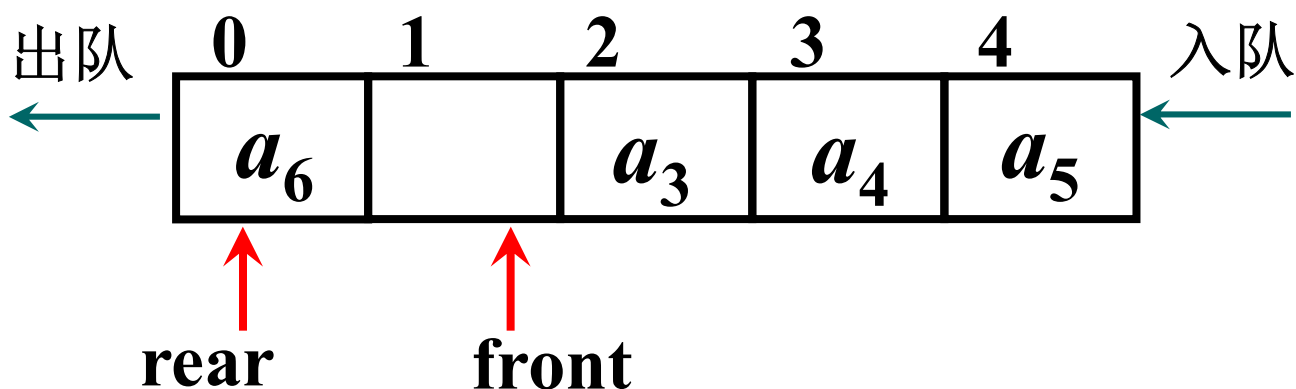


2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 存储结构的定义

```
struct QUEUE {  
    ElemType data [ MaxSize ] ;  
    int front ;  
    int rear ;  
}; //队列的类型
```





2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 操作的实现---- ①队列初始化

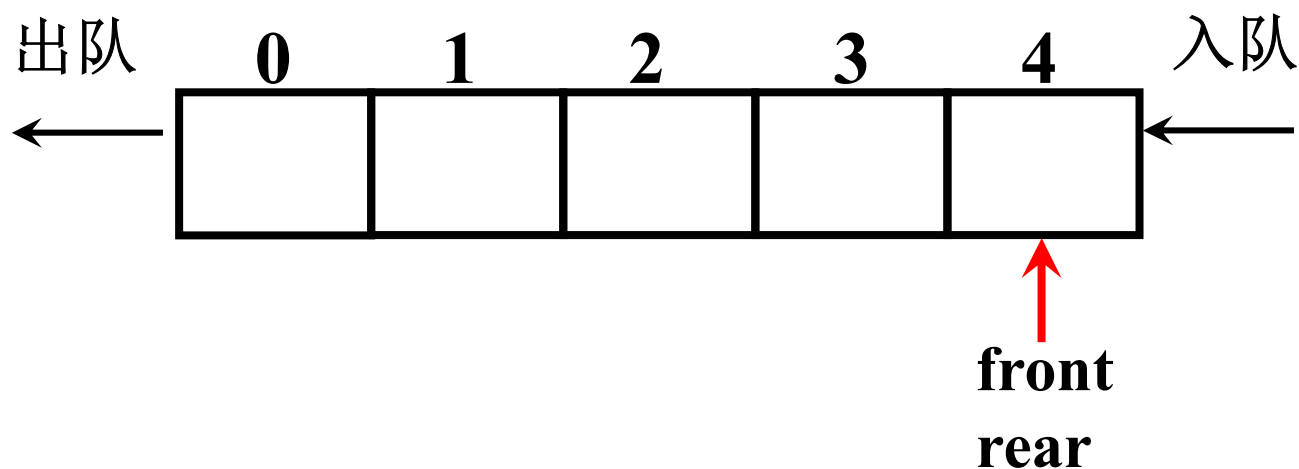
```
void MakeNull ( QUEUE &Q)
```

```
{
```

```
    Q.front = MaxSize-1;
```

```
    Q.rear  = MaxSize-1;
```

```
}
```





2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 操作的实现---- ②队列判空

```
bool Empty( QUEUE Q )
```

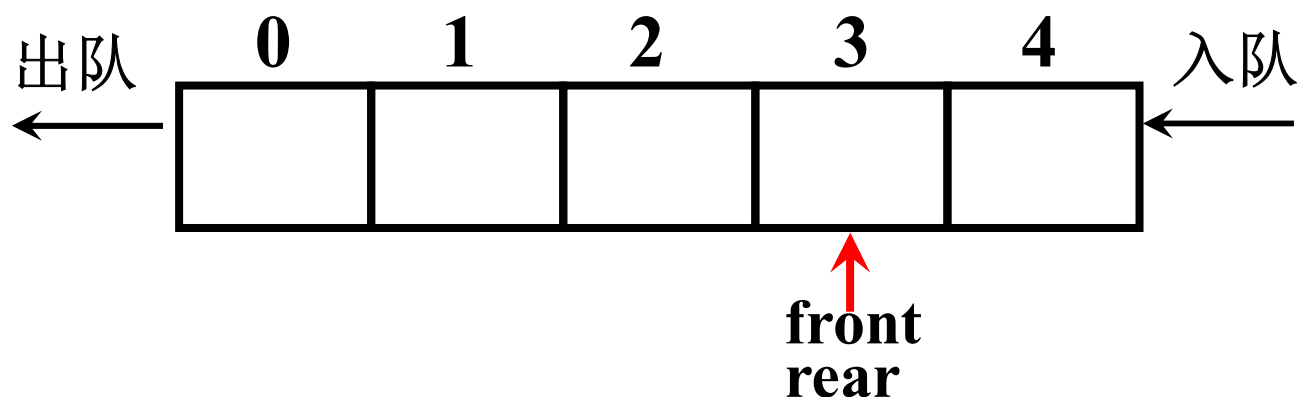
```
{  if ( Q.rear == Q.front )
```

```
    return TRUE ;
```

```
    else
```

```
        return FALSE ;
```

```
}
```





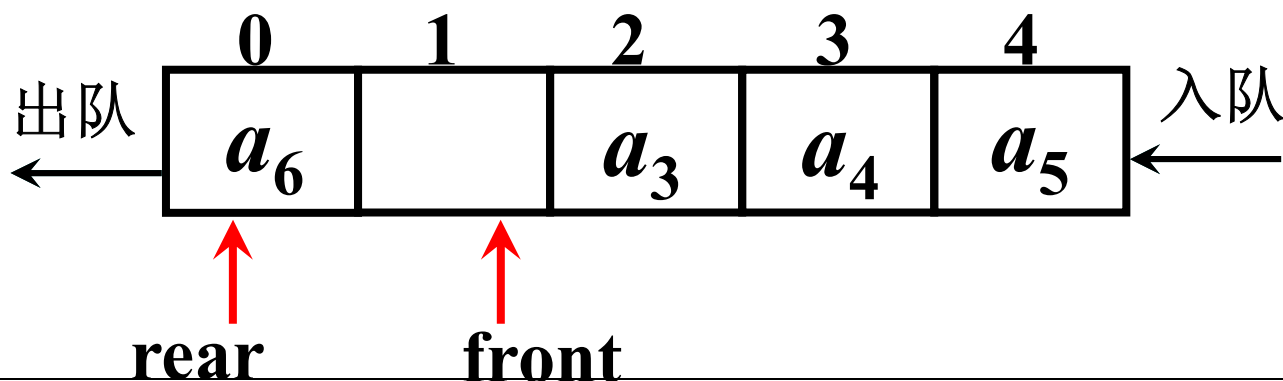
2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 操作的实现---- ③返回队首元素

ElemType Front(QUEUE Q)

```
{ if ( Empty( Q ) ) return NULLESE ;  
  else {   Q.front=(Q.front+1)%MaxSize ;  
          return (Q.data[ Q. front ] );  
  }  
}
```





2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 操作的实现---- ④入队

```
void EnQueue ( ElemType x, QUEUE &Q )
```

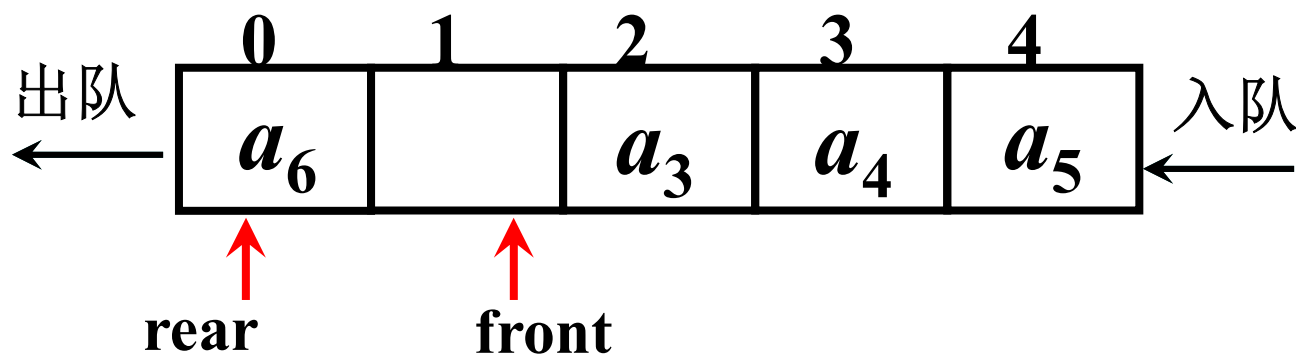
```
{   if ( (Q.rear+1)%MaxSize == Q.front )
```

```
        cout<< “队列满” ;
```

```
    else{   Q.rear=(Q.rear+1)%MaxSize ;
```

```
        Q.data[ Q.rear ] = x ;
```

```
}
```





2.4.3 队列的数组实现(Cont.)

➡ 队列的顺序存储结构及实现

■ 操作的实现---- ⑤出队

```
void DeQueue ( QUEUE Q );
```

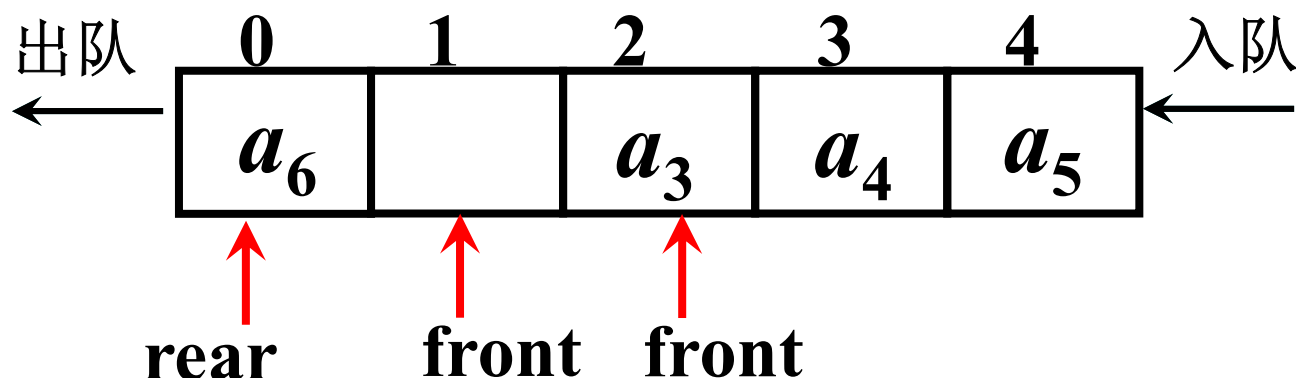
```
{ if ( Empty ( Q ) )
```

```
    cout<< “空队列” <<endl;
```

```
    else
```

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

```
}
```





2.4.4 队列的应用

↓ 队列使用的原则

■ 凡是符合先进先出原则的

● 服务窗口和排号机、打印机的缓冲区、分时系统、树型结构的层次遍历、图的广度优先搜索等等

↓ 举例

■ **约瑟夫出圈问题**： n 个人排成一圈，从第一个开始报数，报到 m 的人出圈，剩下的人继续开始从1报数，直到所有的人都出圈为止。

■ **舞伴问题**：假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。若两队**初始人数不相同**，则较长的那一队中未配对者等待下一轮舞曲。现要求写算法模拟上述**舞伴配对问题**。





2.5 特殊线性表—字符串

2.5.1 串的逻辑结构

- **串**：零个或多个**字符**组成的有限**序列**。
- **串长度**：串中所包含的字符个数。
- **空串**：长度为0的串，记为：“ ”。
- **非空串**通常记为： $S = "s_1 s_2 \dots s_n"$
 - 其中： S 是串名，双引号是**定界符**，双引号引起来的部分是串值， s_i ($1 \leq i \leq n$) 是一个任意字符。
 - 字符集：ASCII码、扩展ASCII码、Unicode字符集
- **子串**：串中任意个连续的字符组成的子序列。
- **主串**：包含子串的串。
- **子串的位置**：子串的第一个字符在主串中的序号。





2.5.1 串的逻辑结构

串的操作

- **String MakeNull() ;**
- **bool IsNull (S) ;**
- **void In(S, a) ;**
- **int Len(S) ;**
- **void Concat(S1, S2) ;**
- **String Substr(S, m, n) ;**
- **bool Index(S, T) ;**
-

与其他线性结构相比，串的操作对象有什么特点？

- 串的操作通常以**串的整体**作为操作对象。





2.5.2 串的存储结构

顺序串:

- 用数组来存储串中的字符序列。

非压缩形式



.....	<i>C</i>	<i>h</i>	<i>i</i>	<i>n</i>	<i>s</i>	<i>e</i>
-------	----------	----------	----------	----------	----------	----------	-------

压缩形式



.....		<i>C</i>	<i>e</i>		
		<i>h</i>	<i>s</i>			
		<i>i</i>	<i>e</i>			
		<i>n</i>				





2.5.2 串的存储结构(Cont.)

如何表示串的长度?

- 方法一：用一个变量来表示串的实际长度，同一般线性表
- 方法二：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。

0	1	2	3	4	5	6	Max-1
<i>C</i>	<i>h</i>	<i>i</i>	<i>n</i>	<i>e</i>	<i>s</i>	<i>e</i>	空 闲		7

0	1	2	3	4	5	6	7	Max-1
<i>C</i>	<i>h</i>	<i>i</i>	<i>n</i>	<i>e</i>	<i>s</i>	<i>e</i>	<i>\0</i>	空 闲		



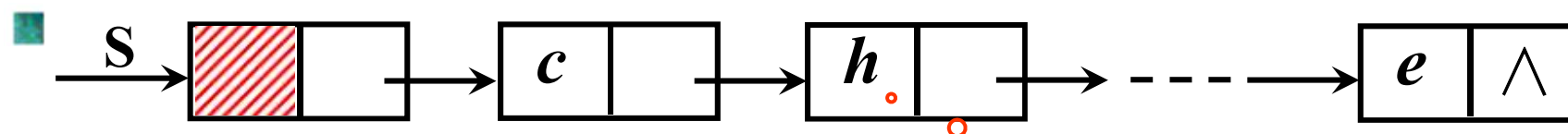


2.5.2 串的存储结构(Cont.)

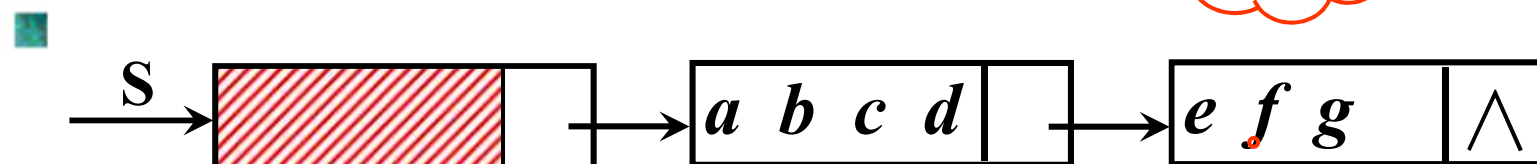
➡ 链接串:

- 用链接存储结构来存储串。

➡ 非压缩形式



➡ 压缩形式



- 假设地址值4四个字节，每个字符一个字节。

1/5

3/8

- 如何实现压缩形式的字符串的插入和删除等操作?





2.5.3 模式匹配

➡ **模式匹配** (字符串匹配是计算机的基本任务之一)

■ 给定 $S = "S_1 S_2 \dots S_n"$ (主串)和 $T = "T_1 T_2 \dots T_m"$ (模式), 在 S 中寻找 T 的过程称为**模式匹配**。如果**匹配成功**, 返回 T 在 S 中的**起始位置**; 如果**匹配失败**, 返回**0**。

➡ **朴素模式匹配算法---Brute-Force算法(枚举法, 回溯算法)**

■ **BF基本思想**

- 从主串 S 的第一个字符开始和模式 T 的第1个字符进行比较, 若相等, 则继续比较两者的后续字符; 否则, 从主串 S 的第2个字符开始和模式 T 的第1个字符进行比较。
- 重复上述过程, 直到 T 中的字符全部比较完毕, 说明本遍匹配成功; 或 S 中字符全部比较完, 则说明匹配失败。





2.5.3 模式匹配(Cont.)

设主串S= "ababcabcacbab", 模式串T= "abcac"

第1遍匹配	主串	ab a bcabcacbab	i=3	
	模式串	ab c	j=3	匹配失败
第2遍匹配	主串	a b abcabcacbab	i=2	
	模式串	a	j=1	匹配失败
第3遍匹配	主串	ababca b acacbab	i=7	
	模式串	abcac c	j=5	匹配失败
第4遍匹配	主串	abab c abcacbab	i=4	
	模式串	a	j=1	匹配失败
第5遍匹配	主串	abab c abcacbab	i=5	
	模式串	a	j=1	匹配失败
第6遍匹配	主串	ababc abc acbab	i=10	//返回(i+1)-lenT
	模式串	abcac	j=5	匹配成功

特点:主串指针需回溯 ($i-j+2$) , 模式串指针需复位 ($j=1$) 。





2.5.3 模式匹配(Cont.)

➡ BF算法实现步骤:

- 1. 在串S和串T中设比较的起始下标i和j;
- 2. 循环直到S或T的所有字符均比较完;
 - 2.1 如果 $S[i]=T[j]$, 继续比较S和T的下一个字符;
 - 2.2 否则, 将i回溯($i=i-j+2$), j复位, 准备下一遍比较;
- 3. 如果T中所有字符均比较完, 则匹配成功, 返回主串起始比较下标; 否则, 匹配失败, 返回0。





2.5.3 模式匹配(Cont.)

```
int Index_BF ( char* S, char* T )
```

```
{ /*S为主串T为模式，长度分别为lenS和lenT；串采用顺序存储结构*/
```

```
    int i = 1 , j = 1;                // 从第1个位置开始比较
```

```
    while (i<=lenS && j<=lenT) {
```

```
        if (S[i] == T[j]) { ++i; ++j; } // 继续比较后继字符
```

```
        else { i = i - j + 2; j = 1; } // 指针后退重新开始匹配
```

```
    }
```

```
    // 返回与模式第一字符相等的字符在主串中的序号
```

```
    if ( j > lenT)
```

```
        return i - lenT; //返回T在S中的起始位置
```

```
    else
```

```
        return 0;                // 匹配不成功
```

```
}
```





2.5.3 模式匹配(Cont.)

➔ **BF算法时间复杂度** $|S|=n, |T|=m$. 可能匹配成功位置(**$1 \sim n-m+1$**).

■ ①第1遍就匹配成功

- 仅需比较T与S的前m个字符，总比较次数为m次
- BF算法时间复杂度为O(**m**)

■ ② 每遍匹配，失配均在T的第1位，最后1遍匹配成功

- 例如， $S = \text{"aaaaaaaaaabc"}$ ， $T = \text{"bc"}$
- 失配比较次数： **$n-m$** ；最后1遍匹配成功的比较次数**m**
- BF算法时间复杂度为O(**n**)

■ ③每遍匹配，失配均在T的最后1位，最后1遍匹配成功

- 例如， $S = \text{"aaaaaaaaaaab"}$ ， $T = \text{"aaab"}$
- 每遍失配和最后1遍匹配成功的比较次数**m**，失配遍数 **$n-m$**
- 总的比较次数 **$m \times (n-m + 1)$**
- BF算法时间复杂度为O(**$m \times n$**)

■ ④匹配失败，即T不在S中，BF算法时间复杂度为同③





2.5.3 模式匹配(Cont.)

改进的模式匹配算法---KMP算法(无回溯算法)

为什么BF算法时间性能低？

- 在每遍匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。

如何在匹配不成功时主串不回溯？

- 主串不回溯，模式就需要向右移动一段距离。

如何确定模式的右移距离？

- 利用已经得到的“部分匹配”的结果
- 将模式向右移动尽可能远的一段距离后，继续进行匹配
 - 将T失配位置前面下标尽可能小的字符与主串的失配位置对齐，继续进行下一遍模式匹配

字符串的(最长、公共)真前缀和后缀：ababcabcacbabab





2.5.3 模式匹配(Cont.)

➔ 设主串ababcabcacbab, 模式abcac, KMP算法匹配过程示例:

第1遍匹配

↓ i=1----3失配
a b a b c a b c a c b a b
a b c
↑ j=3失配, 下1遍: i不变, j=next[j]=1

第2遍匹配

↓ i=3----7失配
a b a b c a b c a c b a b
a b c a c
↑ j=5失配, 下1遍: i不变, j=next[j]=2

第3遍匹配

↓ i=7----10 返回
a b a b c a b c a c b a b
a b c a c
↑ j=5 匹配成功





2.5.3 模式匹配(Cont.)

思考的开始:

- 假定主串为 $S_1S_2\dots S_n$, 模式串为 $T_1T_2\dots T_m$
- 无回溯模式匹配问题变为: 当主串中的第 i 个字符 S_i 和模式中的第 j 个字符 T_j 不匹配(即 $S_i \neq T_j$), 主串中的 S_i (i 不回溯)应该与模式中的哪个字符对齐再进行比较呢?

进一步思考

- 假设主串中的 S_i 与模式中的 T_k ($k < j$) 继续比较, 则应有

$$T_1T_2\dots T_{k-1} = S_{i-(k-1)}S_{i-(k-2)}\dots S_{i-1}$$

- 可能有多多个 k , 取哪一个?

- 而根据已有的匹配, 有

$$T_{j-(k-1)}T_{j-(k-2)}\dots T_{j-1} = S_{i-(k-1)}S_{i-(k-2)}\dots S_{i-1}$$

- 因此

$$T_1T_2\dots T_{k-1} = T_{j-(k-1)}T_{j-(k-2)}\dots T_{j-1}$$

i:	1	2	3	4	5	6	7	8	9	.	.	.
S:	a	b	a	b	a	b	a	a	b	a	b	a
T:	a	b	a	b	a	c	b					
j:	1	2	3	4	5	6	7					





2.5.3 模式匹配(Cont.)

➔ 模式的特征函数next的定义

$$\text{next}(j) = \begin{cases} 0, & \text{当 } j = 1 \text{ 时} \\ \text{Max}\{k | 1 < k < j \text{ 且 } "T_1T_2 \cdots T_{k-1}" = "T_{j-(k-1)}T_{j-(k-2)} \cdots T_{j-1}"\}, & \text{当此集合非空时} \\ 1, & \text{其他情况} \end{cases}$$

■ $k = \text{next}(j)$ 实质是找 $T_1T_2 \cdots T_{j-1}$ 中的最长的公共前缀 $T_1T_2 \cdots T_{k-1}$ 和后缀 $T_{j-(k-1)} T_{j-(k-2)} \cdots T_{j-1}$

■ 当 $j=1$ 时, $\text{next}(j)=0$

● 表示从模式首字符与主串的下一个字符对齐进行比较

■ 当 $j>1$ 时, $\text{next}(j)$ 的值为: 模式的 $T_1 \cdots T_{k-1}$ 中最长首尾相同的子串的长度+1

● 表示从模式 $\text{next}(j)$ 位与主串失配位置对齐进行比较

■ 其他情况: 即模式 $T_1 \cdots T_{k-1}$ 无首尾相同子串, $\text{next}(j)=1$

● 表示从模式首字符与主串失配位置开始进行比较





2.5.3 模式匹配(Cont.)

➔ 设主串ababcabcacbab, 模式abcac, KMP算法匹配过程示例

■ 对模式串T预处理 next[]

模式串T	a	b	c	a	c
可能失配位置j	1	2	3	4	5
k=next[j]	0	1	1	1	2

第1遍匹配

↓ i=1----3失配
a b a b c a b c a c b a b
a b c
↑ j=3失配, 下1遍: i不变, j=next[j]=1

第2遍匹配

↓ i=3----7失配
a b a b c a b c a c b a b
a b c a c
↑ j=5失配, 下1遍: i不变, j=next[j]=2

第3遍匹配

↓ i=7----10 返回
a b a b c a b c a c b a b
a b c a c
↑ j=5 匹配成功





2.5.3 模式匹配(Cont.)

➔ KMP算法实现步骤:

- 1. 在串S和串T中分别设比较的起始下标i和j;
- 2. 循环直到S中所剩字符长度小于T的长度或T中所有字符均比较完毕
 - 2.1 如果 $S[i]=T[j]$, 继续比较S和T的下一个字符; 否则
 - 2.2 将j向右滑动到 $next[j]$ 位置, 即 $j=next[j]$;
 - 2.3 如果 $j=1$, 则将i和j分别加1, 准备下一遍比较;
- 3. 如果T中所有字符均比较完毕, 则返回匹配的起始下标; 否则返回0。





2.5.3 模式匹配(Cont.)

➔ KMP算法的实现

```
int Index_KMP(char* S, char* T)
```

```
{ /*S为主串T为模式，长度分别为lenS和lenT；串采用顺序存储结构*/
```

```
    int i = 1, j = 1;                                // 从第1个位置开始比较
```

```
    while (i<=lenS&& j<=lenT) {
```

```
        if (S[i] == T[j]) { ++i; ++j; }              //继续比较后继字符
```

```
        else j = Next[j];                            // 模式串向右移
```

```
    } //将i回溯(i=i-j+2), j复位（对照朴素算法）
```

```
    if (j > lenT) return i-lenT; // 返回与模式第1字符相等的字符在主串中的序号
```

```
    else return 0;    // 匹配不成功
```

```
}//时间复杂度：O（n+m）
```







2.5.3 模式匹配(Cont.)

➡ 模式的特征函数next的计算方法

next(j)的性质

■ $k = \text{next}(j) = \text{Max} \{k \mid 1 < k < j \text{ 且 } T_1 T_2 \dots T_{k-1} = T_{j-(k-1)} T_{j-(k-2)} \dots T_{j-1}\}$

■ $k < j$ ，因为当失配时必然使模式T向右移，从而导致 $k < j$ 。

移的幅度越小， k 与 j 相差越小

■ k 应取所有可能值中的最大值，因为取最大值就意味着移的幅度最小，也就避免错过成功匹配的机会

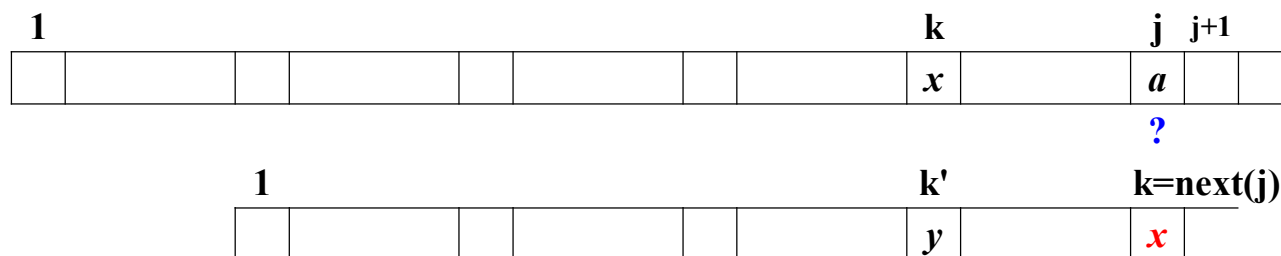
■ k 与 j 具有函数关系，由当前失配位置 j ，可以计算出滑动位置 k （即比较的新起点）

■ 滑动位置 k 仅与模式自身 T 有关，而与主串无关



模式的特征函数next的计算方法（归纳法，动态规划）

- **归纳基础**: 若 $j=1$, 则 $k=\text{next}(j)=0$, 即 $\text{next}(1)=0$
- **归纳假设**: 假设对于 $1 \leq j \leq |T|$ (模式长度), 所有 $\text{next}(1)$ 至 $\text{next}(j)$ 的值均已经求得, 现在来计算 $\text{next}(j+1)$
- **归纳计算**: 设 $\text{nex}(j)=k$, 即当在模式的第 j 个字符 $T[j]$ 处失配时, 应将 $T[j]$ 与 $T[k]$ 对齐作为下一遍匹配起点。此时在模式 $T[1...j-1]$ 中已经有长度为 $k-1$ 的公共前缀和后缀, 即 $T[1...k-1]=T[j-k+1...j-1]$ 。考虑以下2种情况:





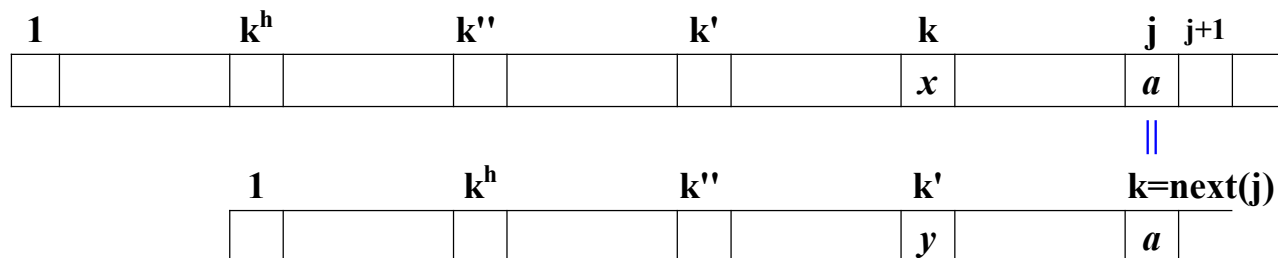
2.5.3 模式匹配(Cont.)

➡ 模式的特征函数next的计算方法（归纳法，动态规划）

■ 归纳计算：考虑以下2种情况：

(1) 若 $T[k] = T[j]$ ，则在模式中有 $T[1...k] = T[j-k+1...j]$ ，即找到了长度为 k 的公共前缀和后缀，
因此，

$$\text{next}(j+1) = k+1 = \text{next}(j) + 1$$





2.5.3 模式匹配(Cont.)

(2)若 $T[k] \neq T[j]$,即 $T[1...k] \neq T[j-k+1...j]$, 则可将求 $\text{next}(j+1)$ 视为一个模式自匹配问题: 模式 T 既是主串也是模式, 已有的匹配结果是 $T[1...k-1]=T[j-k+1...j-1]$, 失配位置是 $T[k](\neq T[j])$ 。

为此, 应将模式右移, 使模式的第 $k'=\text{next}(k)$ 个字符 $T[k']$ 与主串的 $T[j]$ 对齐进行比较, 考虑以下2种情况:

1		k^h		k''		k'		k		j	$j+1$
								x		a	
										+	
		1		k^h		k''		k'		$k=\text{next}(j)$	
								y		x	
										?	
				1		k^h		k''		$k'=\text{next}(k)$	
								z		y	





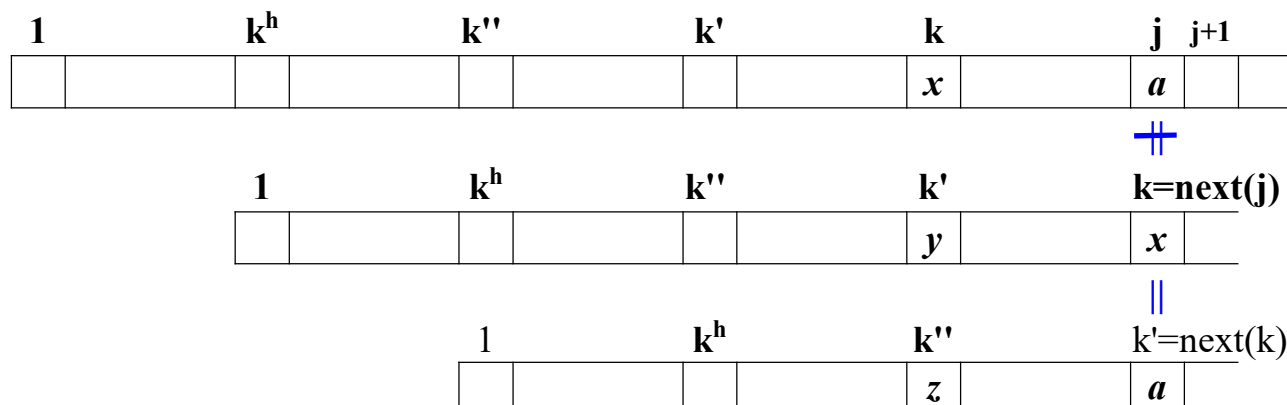
2.5.3 模式匹配(Cont.)

(2.1)若 $T[k'] = T[j]$ ，则在模式中有 $T[1...k'] = T[j-k' + 1...j]$ ，
其中，

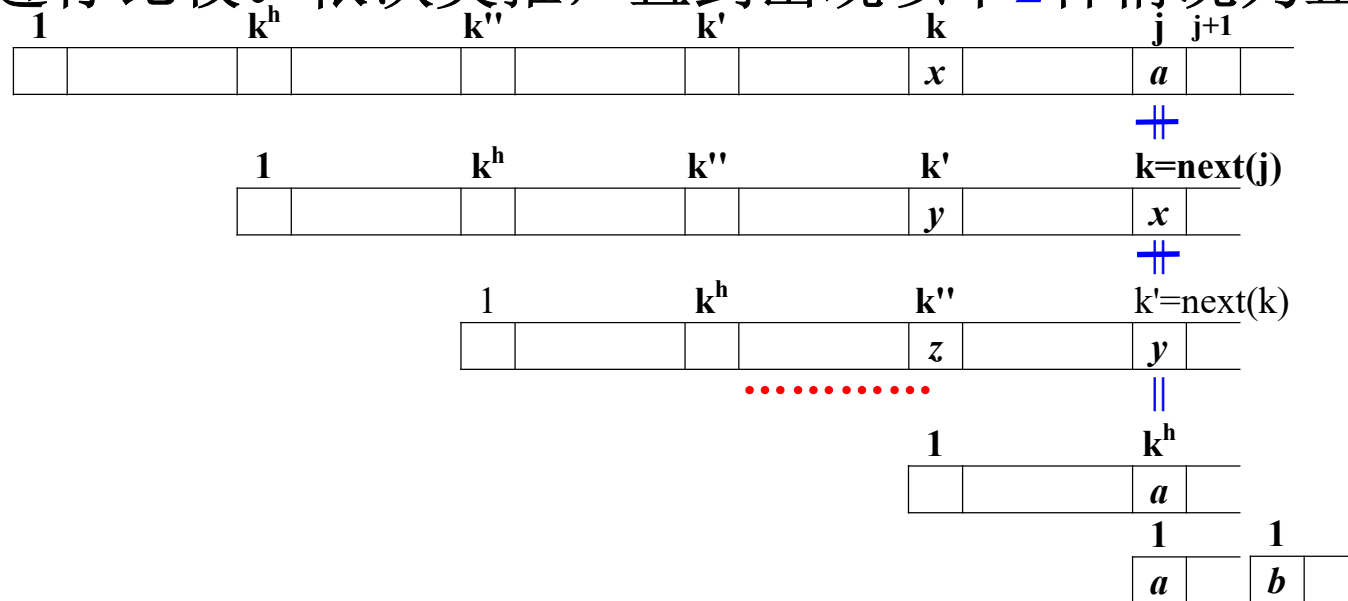
$$1 \leq k' < k < j$$

即找到了长度为 k' 的公共前缀和后缀，因此，

$$\text{next}(j+1) = k' + 1 = \text{next}(k) + 1 = \text{next}(\text{next}(j)) + 1$$



(2.2)若 $T[k'] \neq T[j]$,即 $T[1...k'] \neq T[j-k'+1...j]$,则将模式继续右移,使模式的第 $k''=\text{next}(k')$ 个字符 $T[k'']$ 与主串的 $T[j]$ 对齐进行比较。依次类推,直到出现以下2种情况为止:



(2.2.1)存在 $T[k^h]$ 与 $T[j]$ 匹配成功, 即 $T[k^h]=T[j]$, 其中 $1 \leq k^h < \dots < k'' < k'' \leq j$ 。此时, $next(j+1) = k^h + 1$ 。

(2.2.2)不存在任何 k^h 满足 $1 \leq k^h < \dots < k'' < k' < j$, 即 $k^h = 1$ 但 $T[k^h] = T[1] \neq T[j]$, 则 $\text{next}(j+1) = 0$





2.5.3 模式匹配(Cont.)

➔ next(j)的实现—KMP算法的预处理

```
void GetNext(char* T, int next[])
```

```
{ //求模式T的特征函数值并存入next数组中
```

```
//仍是模式匹配过程,只是主串和模式串是同一个串T
```

```
int i = 1, j = 0, next[1] = 0; //与图对比: i是j, j是k
```

```
while (j < lenT)
```

```
if (j == 0 || T[i] == T[j]) { //旨在找寻T中重复的字符
```

```
    i++; j++; next[i] = j; // next[++i] = ++j;
```

```
}
```

```
else j = next[j];
```

```
}
```

```
//时间复杂度: O(m)
```

1.首先由定义得, next[1]=0;

2.假设已知next[j]=k, 若T[j] = T[k], 则显然有next[j+1]=k+1;

3.若T[j] != T[k], 则令k=next[k], 直至T[j]等于T[k]为止。





2.5.3 模式匹配(Cont.)

↓ 计算模式串的next[j]数组示例:

■ j=1时, next[j]=0;

■ j=2时, next[j]=1; //真子串

■ j=3时, $T_1 \neq T_2$, 即无相同的前/后缀, 因此, next[j]=1

■ j=4时, $T_1 = T_3$, 即相同的前/后缀长度为1, 因此, next[j]=2

■ j=5时, $T_1 = T_4$, 即相同的前/后缀长度为1, 因此, next[j]=2

■ j=6时, $T_1T_2 = T_4T_5$, 即相同的前/后缀长度为2, 因此, next[j]=3

■ j=7时, $T_1 \neq T_6$, 即无相同的前/后缀, 因此, next[j]=1

■ j=8时, $T_1 = T_7$, 即相同的前/后缀长度为1, 因此, next[j]=2

j	1	2	3	4	5	6	7	8
T	a	b	a	a	b	c	a	c
next(j)	0	1	1	2	2	3	1	2





2.6 (多维)数组

➡ 数组:

- 是由下标 (**index**) 和值 (**value**) 组成的序对 (**index, value**) 的序列。
- 也可以定义为是由**相同类型**的数据元素组成有限序列。
- 每个元素受 $n(n \geq 1)$ 个**线性关系**的约束, 每个元素在 n 个线性关系中的序号 i_1 、 i_2 、...、 i_n 称为该元素的下标, 并称该数组为 **n 维数组**。

➡ 数组的特点:

- 元素本身可以具有某种结构, 属于同一数据类型;
- 数组是一个具有固定格式和数量的数据集合。

➡ 示例:





2.6 (多维)数组(Cont.)

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \mathbf{a_{22}} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$



$$A = (A_1, A_2, \dots, A_n)$$

其中:

$$A_i = (a_{1i}, a_{2i}, \dots, a_{mi}) \\ (1 \leq i \leq n)$$

- 元素 a_{22} 受两个线性关系的约束，在行上有一个行前驱 a_{21} 和一个行后继 a_{23} ，在列上有一个列前驱 a_{12} 和一个列后继 a_{32} 。
- 二维数组是数据元素为线性表的线性表。





2.6 (多维)数组(Cont.)

➤ 数组的基本操作

■ 初始化: **Create ()**

- 建立一个空数组;

- **int A[][]**

■ 存取: **Retrieve (array, index)**

- 给定一组下标, 读出对应的数组元素;

- **A[i][j]**

■ 修改: **Store (array, index, value) :**

- 给定一组下标, 存储或修改与其相对应的数组元素。

- **A[i][j]=8**

■ 无需插入和删除操作





2.6 (多维)数组(Cont.)

➤ 数组的存储结构

- 数组没有插入和删除操作，所以，不用预留空间，适合采用**顺序存储**。

➤ 数组的顺序存储

- 用一组连续的存储单元来实现（多维）数组的存储。
- 高维数组可以看成是由多个低维数组组成的。

➤ 二维数组的存储与寻址

- 常用的映射方法有两种：

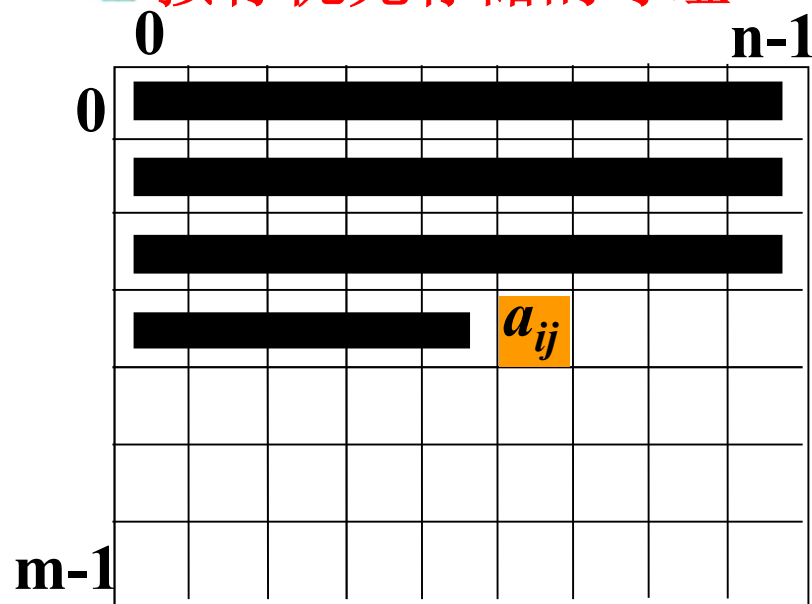
- 按**行**优先：**先行后列**，先存储行号较小的元素，行号相同者先存储列号较小的元素。
- 按**列**优先：**先列后行**，先存储列号较小的元素，列号相同者先存储行号较小的元素。





2.6 (多维)数组(Cont.)

按行优先存储的寻址——二维数组



每行元素个数



本行中 a_{ij} 前面的元素个数

a_{ij} 前面的元素个数 k

= 整行数 \times 每行元素个数 + 本行中

a_{ij} 前面的元素个数 = $i \times n + j$

$$\text{Loc}(a_{ij}) = \text{Loc}(a_{00}) + (i \times n + j) \times c$$

Sa	a_{00}	a_{01}	a_{02}			a_{ij}			$a_{n-1,n-1}$
k=	0	1	2	...		$i \times n + j$			$n^2 - 1$

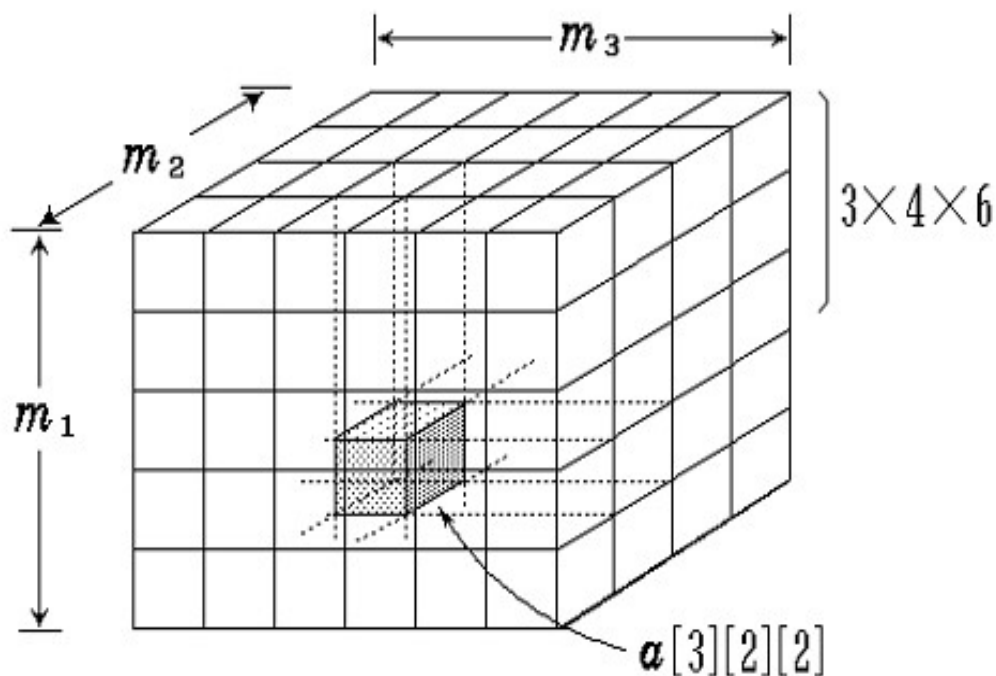




2.6 (多维)数组(Cont.)

■ 按行优先存储的寻址—多维数组

n ($n > 2$) 维数组一般也采用按行优先和按列优先两种存储方法。



$$\text{Loc}(a_{ijk}) = \text{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

更高维的数组呢？





2.6 (多维)数组(Cont.)

➡ 特殊矩阵的压缩存储

■ **特殊矩阵**：矩阵中很多**值相同**的元素并且它们的**分布有**一定的**规律**。

● 如对称矩阵、上/下三角矩阵、带状(对角)矩阵等

■ **稀疏矩阵**：矩阵中有很多特定值的（如零）元素。

● 分布没有规律

● 在 $m*n$ 的矩阵中，有 t 个元素不为零。令 $\alpha=t/m*n$ ，称 α 为矩阵的**稀疏因子**。

● 通常认为 $\alpha \leq 0.05$ 时称为稀疏矩阵

➡ 压缩存储的基本思想是：

■ 为多个**值相同**的元素只分配**一个**存储空间；

■ 对**特定值（如零）**的元素**不分配**存储空间。

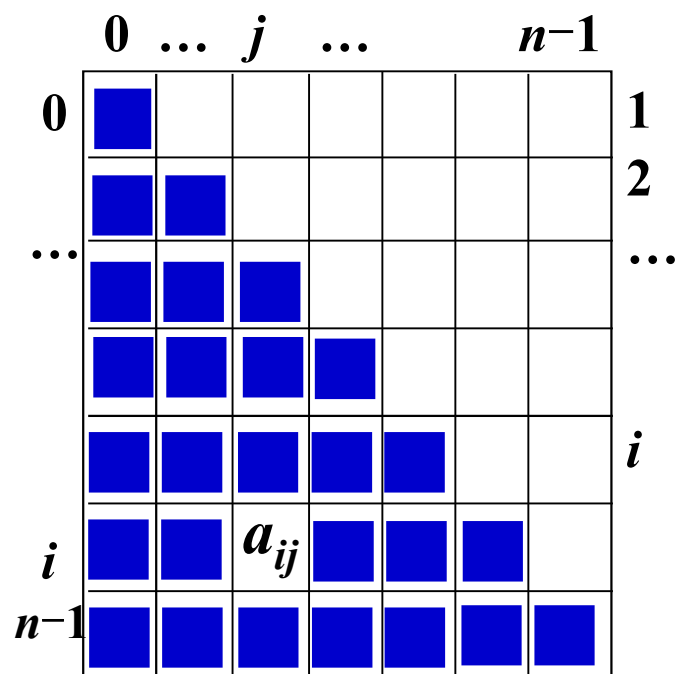




2.6 (多维)数组(Cont.)

➡ 三角矩阵的压缩存储——上三角矩阵

- 只存储下三角部分的元素。
- 对角线上方的常数不存或只存一个



$$A = \begin{pmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$

矩阵中任意一个元素 a_{ij} 在一维数组中的下标 k 与 i 、 j 的对应关系:

$$k = i \times (i+1)/2 + j \quad (i \geq j)$$

$$k = n \times (n+1)/2 \quad \text{存常数 } c$$

Sa	a_{00}	a_{10}	a_{11}	a_{20}	...	$a_{i,j}$...	$a_{n-1,0}$...	$a_{n-1,n-1}$	c
k=	0	1	2	3		$i(i+1)/2+j$		$n(n-1)/2$		$n(n+1)/2-1$	*



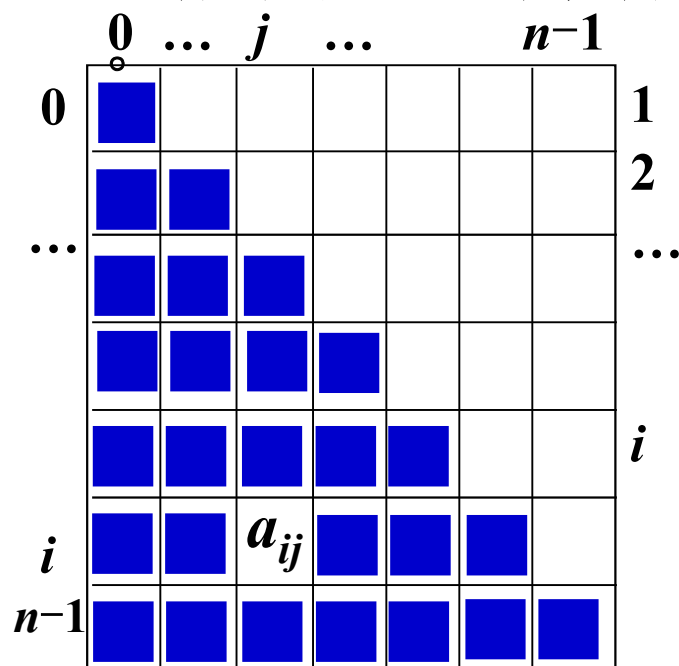


2.6 (多维)数组(Cont.)

对称矩阵的压缩存储

对称矩阵特点: $a_{ij}=a_{ji}$

只存储下/上三角部分的元素



$$A = \begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$

a_{ij} 在一维数组中的序号

$$= i \times (i+1)/2 + j + 1$$

\because 一维数组下标从0开始

$\therefore a_{ij}$ 在一维数组中的下标

$$k = i \times (i+1)/2 + j \quad (i \geq j)$$

$$k = j \times (j+1)/2 + i \quad (i < j)$$

Sa	a_{00}	a_{10}	a_{11}	a_{20}	...	$a_{i,j}$...	$a_{n-1,0}$...	$a_{n-1,n-1}$
k=	0	1	2	3		$i(i+1)/2+j$		$n(n-1)/2$		$n(n+1)/2-1$





2.6 (多维)数组(Cont.)

稀疏矩阵的压缩存储 --三元组顺序表

■ 如何只存储非零元素？

● 稀疏矩阵中的非零元素的分布没有规律。

■ 将稀疏矩阵中的每个非零元素表示为：

● (行号，列号，非零元素值)——三元组表

```
typedef struct {  
    int i, j ;  
    ElemType v ;  
} Triple ;  
typedef struct {  
    Triple data[MaxSize+1];  
    int mu, nu, tu;  
} TSMatrix;
```

$$M = \begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6 \times 7}$$





2.6 (多维)数组(Cont.)

稀疏矩阵的压缩存储 -- 三元组顺序表

■ 如何存储三元组表?

● 按行优先的顺序存到一个三元组数组。

$$A = \begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6 \times 7}$$

	i	j	v
0	0	1	12
1	0	2	9
2	2	1	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	7
	.	.	.
M-1	.	.	.

data

mu: 矩阵行数 6

nu: 矩阵列数 7

tu: 非零元数 8





2.6 (多维)数组(Cont.)

稀疏矩阵的转置算法

$$A = \begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6 \times 7}$$



0	0	1	12
1	0	2	9
2	2	0	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	-7
	.	.	.
M-1	.	.	.



$$A^T = \begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{7 \times 6}$$



0	0	2	-3
1	0	5	15
2	1	0	12
3	1	4	18
4	2	0	9
5	2	3	24
6	3	5	-7
7	5	2	14
	.	.	.
M-1	.	.	.





2.6 (多维)数组(Cont.)

稀疏矩阵的转置算法

```
void TransposMatrix(TSMatrix M, TSMatrix &T)
{
    T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
    if ( T.tu ) {
        int q = 0 ; // T的下标
        for ( int col = 0; col < M.nu; ++col )//遍历nu遍
            for ( int p = 0; p < M.tu; ++p )//每遍检查所有元素 M-1
                if ( M.data[p].j == col ) {
                    T.data[q].i = M.data[p].j ;
                    T.data[q].j = M.data[p].i ;
                    T.data[q].e = M.data[p].e ;
                    ++ q ;
                }
    }
    //采用三元组顺序表存储表示，求稀疏矩阵M的转置矩阵T
    //时间复杂度：  $T(n) = O(nu \cdot tu)$ 
```

0	0	1	12
1	0	2	9
2	2	0	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	-7
	.	.	.
M-1	.	.	.

0	0	2	-3
1	0	5	15
2	1	0	12
3	1	4	18
4	2	0	9
5	2	3	24
6	3	5	-7
7	5	2	14
	.	.	.
M-1	.	.	.



2.6 (多维)数组(Cont.)

改进的稀疏矩阵转置算法—以空间换时间

改进思路：若能预先确定矩阵M中每一列(即T中每一行)的第一个非零元在T.data中恰当位置。那么在对M.data中的三元组一次做转置时，便可直接放到T.data中恰当的位置上去。

■ 设两个向量：num和cpot

■ num[col]表示矩阵M中第col列中的非零元素个数。

■ cpot[col]M中第col列的第1个非零元在T.data中的恰当位置

■ cpot计算公式：

$$\begin{cases} cpot[0] = 0 \\ cpot[col] = cpot[col-1] + num[col-1], 1 \leq col < M.nu \end{cases}$$

col	0	1	2	3	4	5	6
num[col]	2	2	2	1	0	1	0
cpot[col]	0	2	4	6	7	7	8

0	0	1	12
1	0	2	9
2	2	0	-3
3	2	5	14
4	3	2	24
5	4	1	18
6	5	0	15
7	5	3	-7
	.	.	.
M-1	.	.	.



2.6 (多维)数组(Cont.)

改进的稀疏矩阵转置算法

```
void FastTransposMatrix(TSMatrix M, TSMatrix &T)
{
    T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
    if ( T.tu ) {
        for ( col = 0; col < M.nu; ++col ) num[ col ] = 0 ;
        for ( t = 0; t < M.tu; ++t )
            ++num[ M.data[ t ].j ] ;
        cpot [ 0 ] = 0 ;
        for ( col=1;col<M.nu;col++)
            cpot [ col ] = cpot[ col - 1 ] + num[ col - 1 ] ;
        for ( p = 0; p < M.tu; ++p ) {
            col = M.data[ p ].j ; q = cpot[ col ] ;
            T.data[ q ].i = M.data[ p ].j ;
            T.data[ q ].j = M.data[ p ].i ;
            T.data[ q ].e = M.data[ p ].e ;
            ++cpot[ col ] ;
        }
    }
} //转置的改进算法:  $T(n) = O(nu + tu)$ 
```





2.6 (多维)数组(Cont.)

稀疏矩阵的压缩存储 —— 十字链表

- 采用链接存储结构存储三元组表，每个非零元素对应的三元组存储为一个链表结点，结构为：

left		up
row	col	val

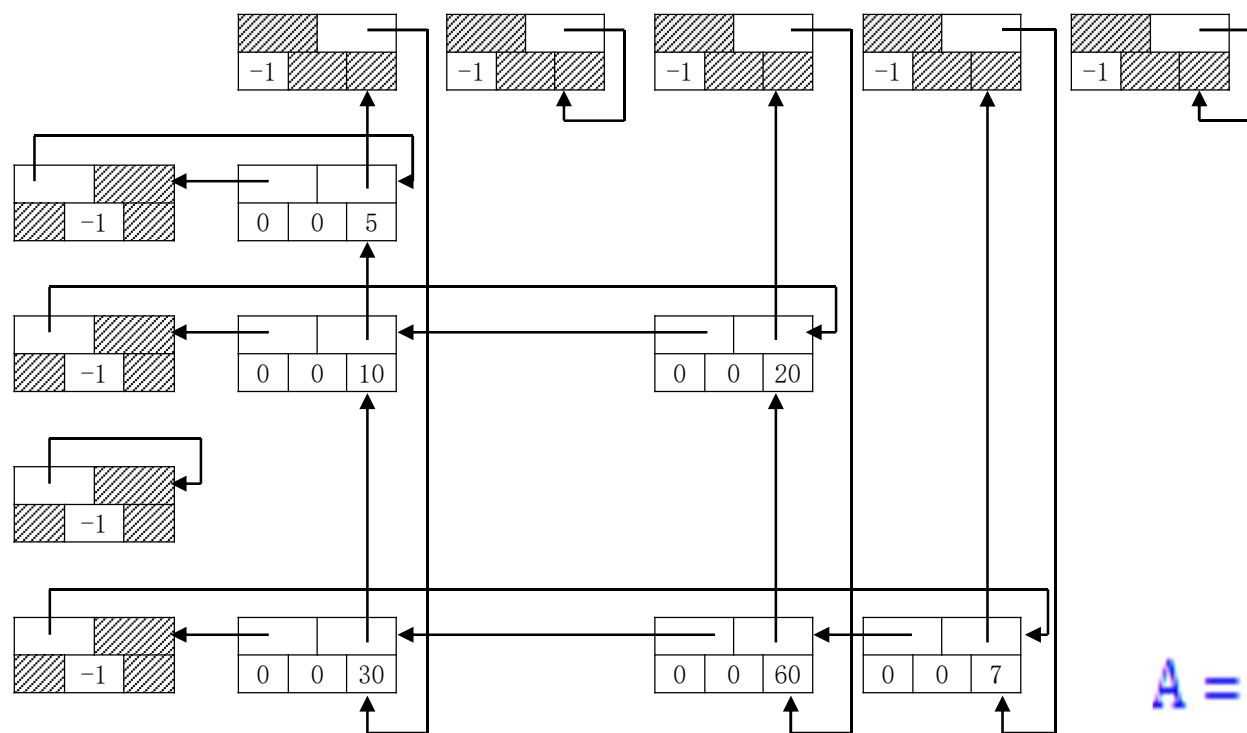
- **left**: 指针域，指向同一行中的前一个三元组
- **up**: 指针域，指向同一列中的上一个三元组
- **row**: 存储非零元素的行号
- **col**: 存储非零元素的列号
- **val**: 存储非零元素的值





2.6 (多维)数组(Cont.)

稀疏矩阵的压缩存储 -- 十字链表



left		up	
row	col	val	

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 30 & 0 & 60 & 7 & 0 \end{bmatrix}_{4 \times 5}$$





2.7 广义表

➤ **广义表**： n ($n \geq 0$) 个数据元素的有限序列，记作：

$$LS = (a_1, a_2, \dots, a_n)$$

其中： LS 是广义表的**名称**， a_i ($1 \leq i \leq n$) 可以是**单个的数据元素**，也可以是一个**广义表**，分别称为 LS 的**单个元素**（或**原子**）和**子表**。

- **长度**：广义表 LS 中的直接元素的个数；
- **深度**：广义表 LS 中括号的最大嵌套层数。
- **表头**：广义表 LS 非空时，称第一个元素为 LS 的表头；
- **表尾**：广义表 LS 中除表头外其余元素组成的广义表。





2.7 广义表(Cont.)

广义表示例:

- $A = (a, (b, a, b), (), c, ((2)))$;
- $B = ()$;
- $C = (e)$;
- $D = (A, B, C)$;
- $E = (a, E)$;

广义表性质:

- 广义表的元素可以是子表，子表的元素还可以是子表，
… …，广义表是一个多层次的结构（层次性）；
- 一个广义表可以被其他广义表所共享（共享性）。
- 广义表可以是其本身的子表（递归性）。





2.7 广义表(Cont.)

➤ 广义表基本操作:

- ①Car (L) :返回广义表 L 的第一个元素
- ②Cdr (L) :返回广义表 L 除第一个元素以外的所有元素
- ③Append (L, M) :返回广义表 L + M
- ④Equal (L, M) :判 广义表 L 和 M 是否相等
- ⑤Length (L) :求广义表 L 的长度

➤ 广义表存储结构





2.7 广义表(Cont.)

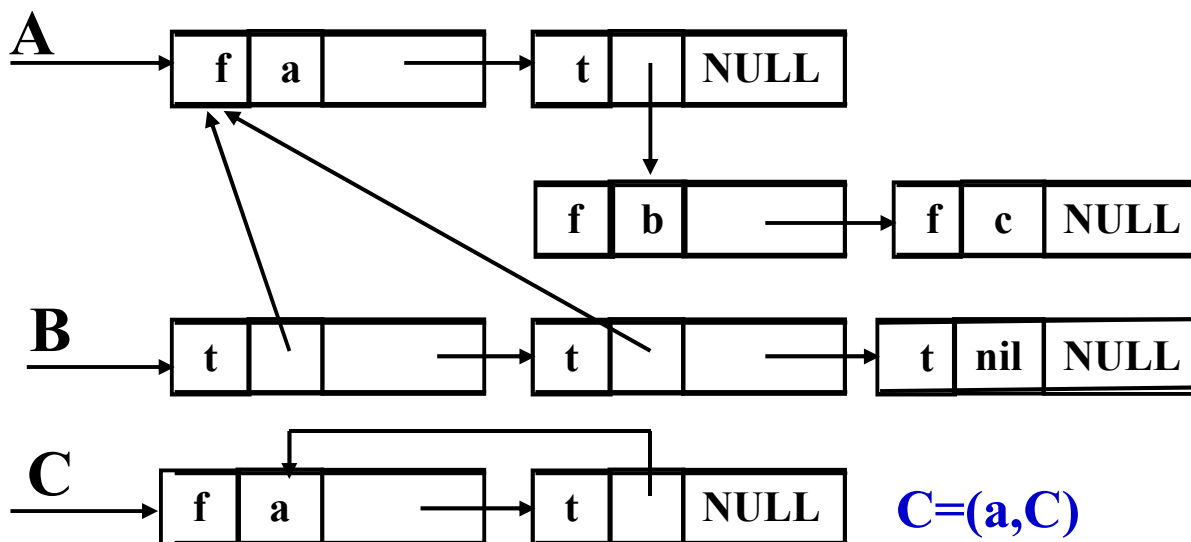
广义表存储结构

$A=(a,(b,c))$

$B=(A,A,())$

```
struct listnode {  
    listnode *link ;  
    boolean tag ;  
    union {  
        char data ;  
        listnode *dlink ;  
    } ;  
} ;
```

```
typedef listnode *listpointer ;
```





2.7 广义表(Cont.)

广义表操作的实现

```
bool Equal( listpointer S, listpointer T )
{
    boolean x, y ;
    y = FALSE ;
    if ( ( S == NULL ) && ( T == NULL ) )
        y = TRUE ;
    else if ( ( S != NULL ) && ( T != NULL ) )
        if ( S→tag == T→tag )
        {
            if ( S→tag == FALSE
                { if ( S→element.data == T→element.data )
                    x = TRUE ;
                else
                    x = FALSE ;
            }
            else
                x = Equal( S→element.data, T→element.data );
            if ( x == TRUE )
                y = Equal( S→link, T→link ) ;
        }
    return y ;
} //S和T均为非递归的广义表
```



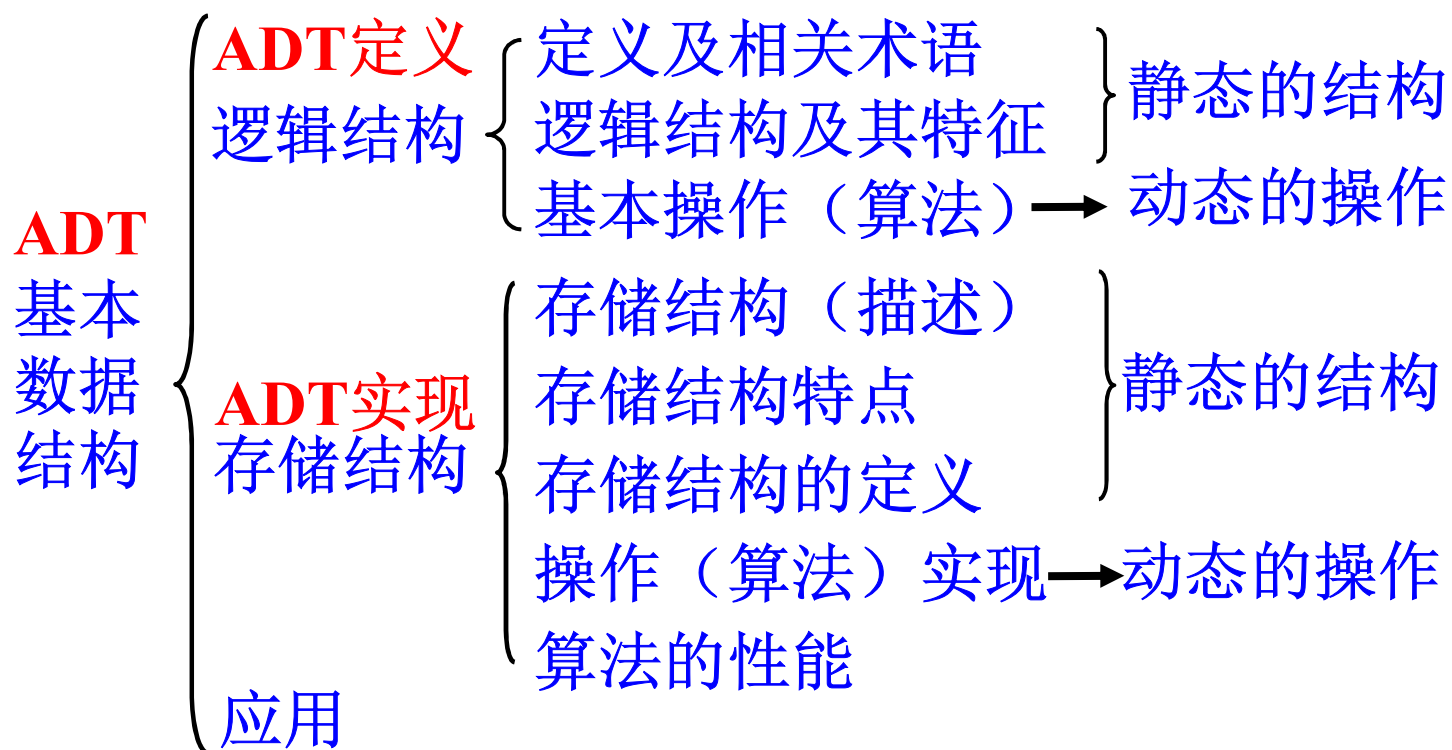


本章小结

知识点:

■ 线性表、栈、队列、串、（多维）数组、广义表

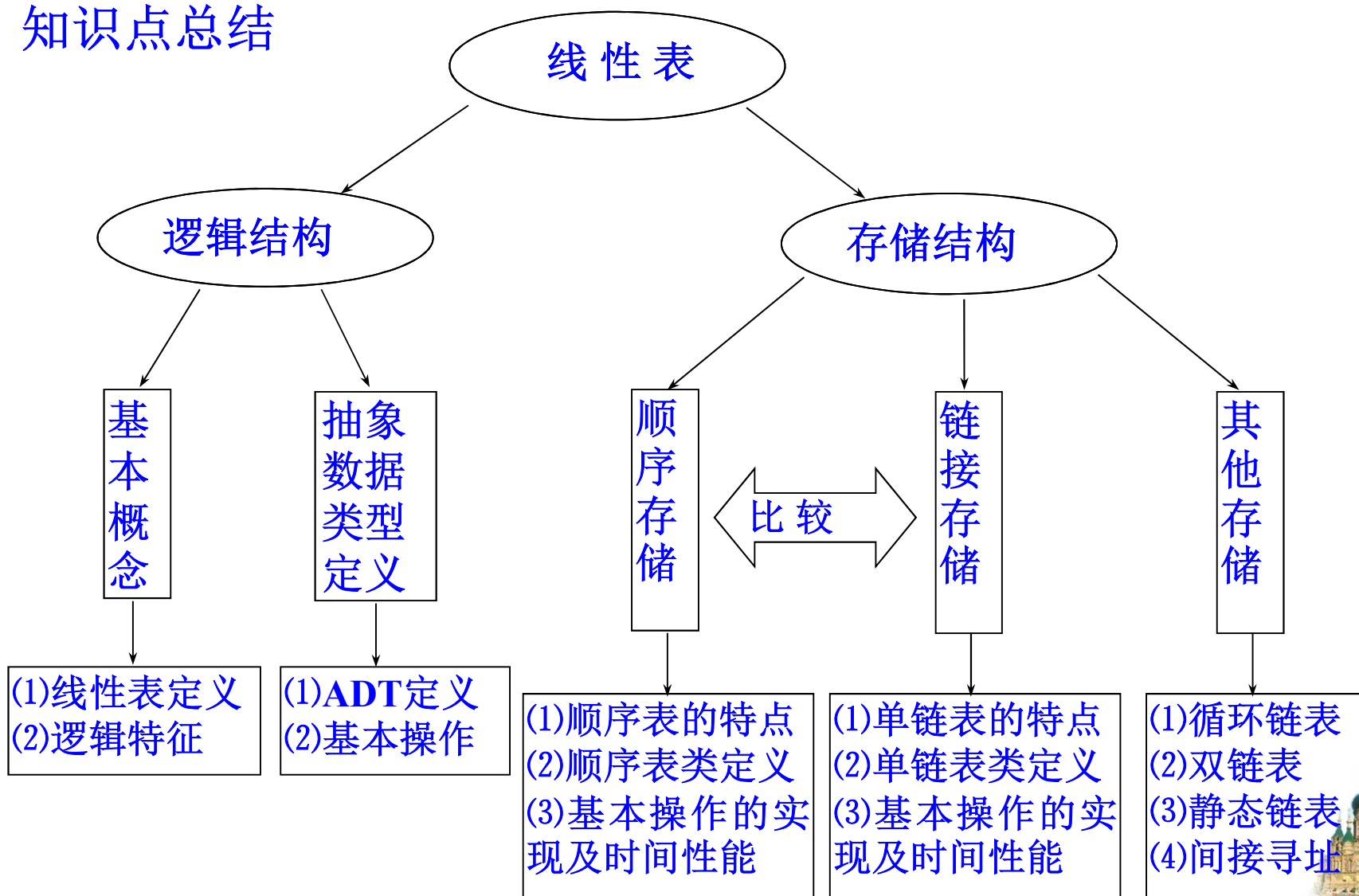
知识点体系结构





本章小结

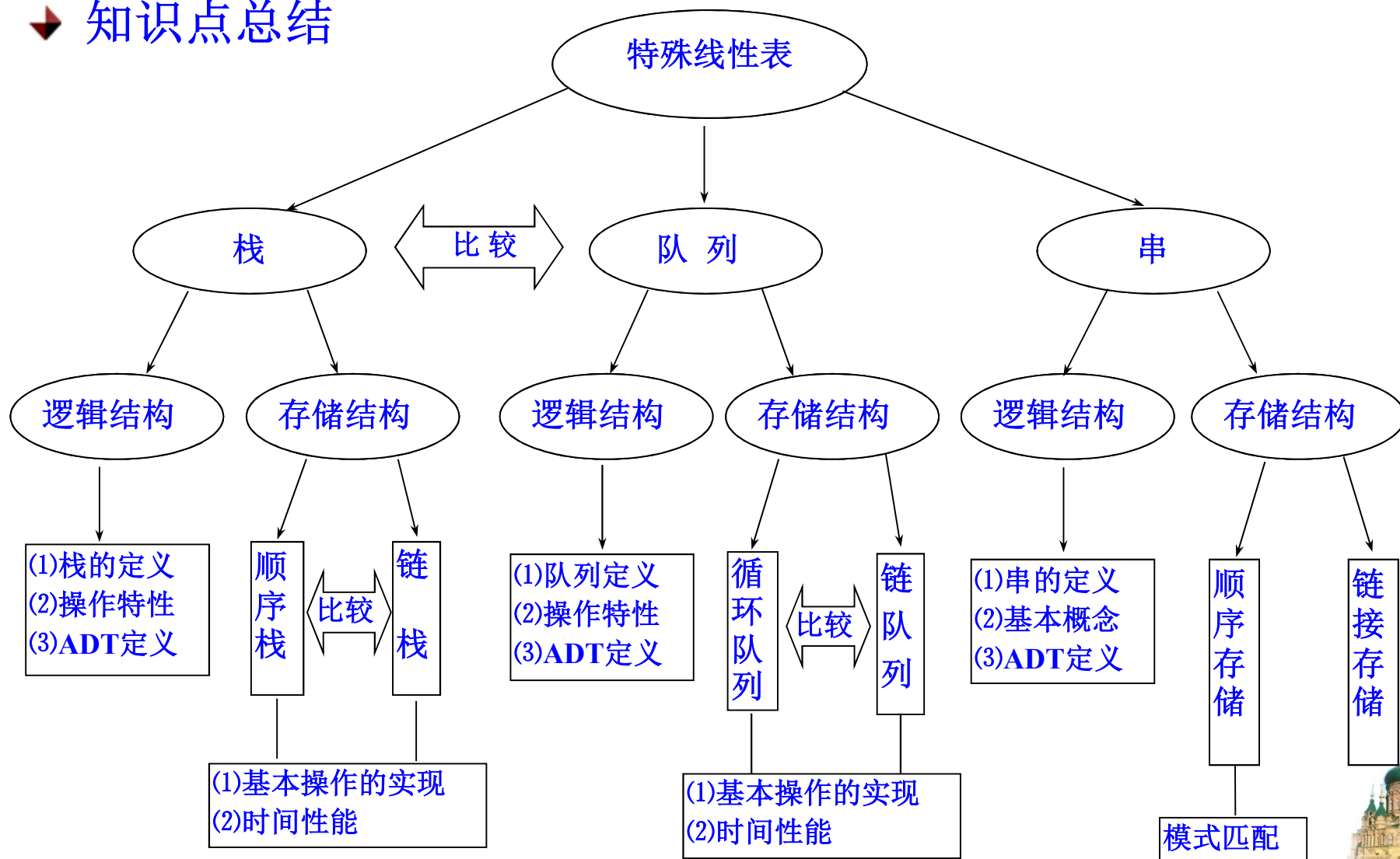
知识点总结





本章小结

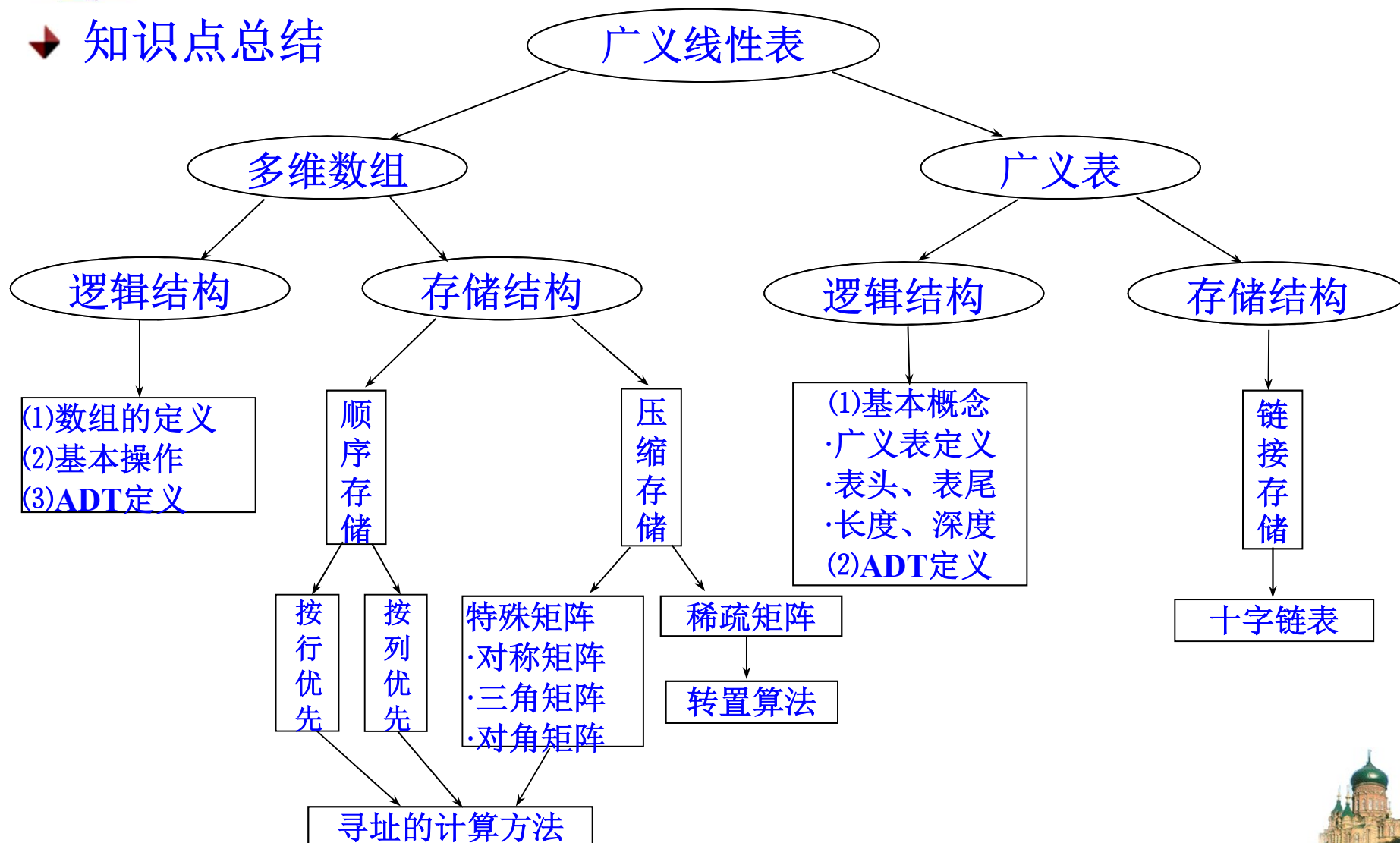
知识点总结





本章小结

知识点总结





本章小结

冬夜读书示子聿（其三）

陆游

古人学问无遗力，
少壮工夫老始成。
纸上得来**终觉浅**，
绝知此事**要躬行**。

- 做学问就要竭尽全力。只有少年时加倍努力，将来才能成就一番事业。
- 做学问应当持之以恒。趁着年少精力旺盛，抓住美好时光奋力拼搏，莫让青春年华付诸东流。
- 从书本得来的知识比较浅薄，只有经过亲身实践，才能变成自己的东西。
- 学习过程中要“躬行”，获取知识后还要“躬行”。通过亲身实践化为己有，转为己用。在实践中使书本知识夯实并进一步获得升华。

