数据结构与算法
线性表

臧天仪 教授

tianyi.zang@hit.edu.cn

哈尔滨工业大学计算学学部

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性以及相关的查找、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应用问题。
- 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 /ADT

2.2 线性表的存储结构

2.3 栈（Stack）

2.4 队列（Queue）

2.5 串（String）

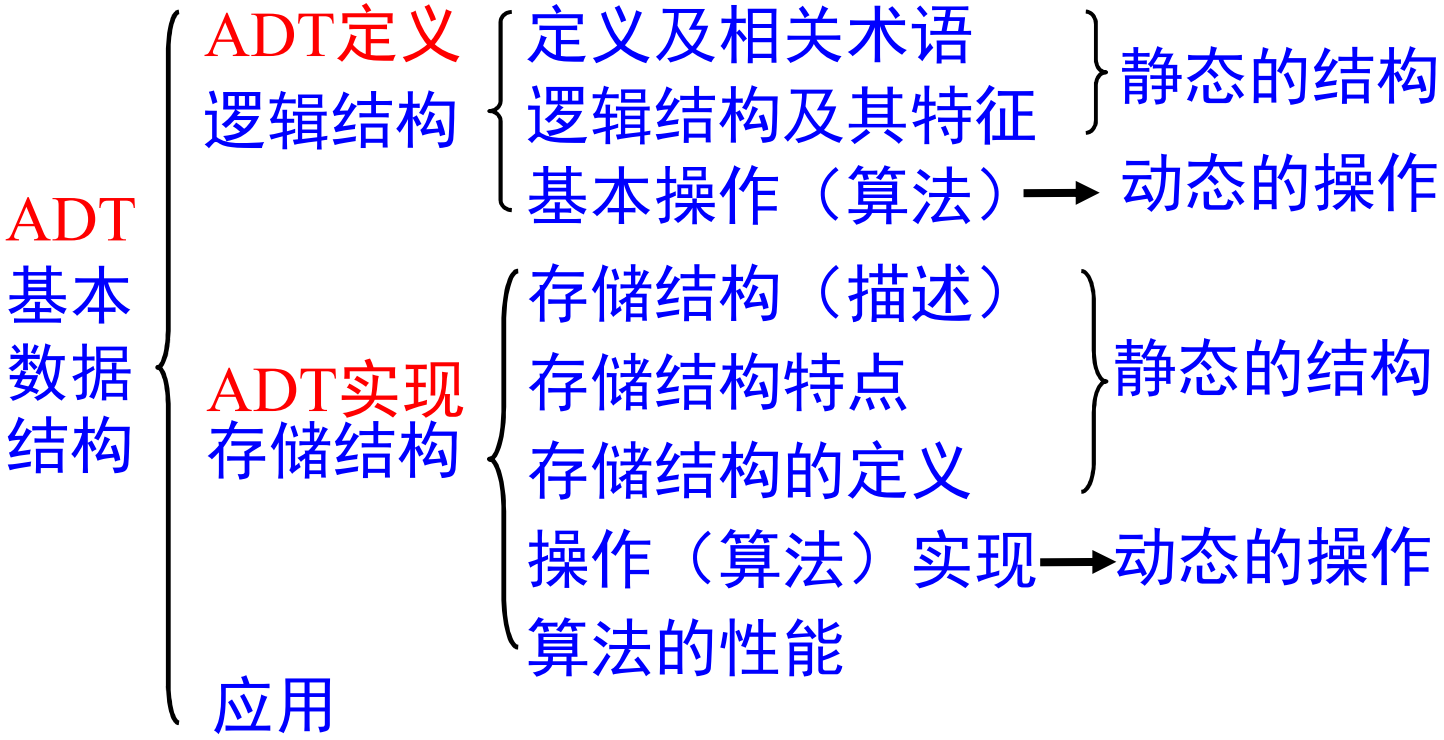
2.6 数组（Array）

2.7 广义表（Generalized List）

本章小结

本章的知识点结构

- 知识点：
  - 线性表、栈、队列、串、（多维）数组、广义表
- 知识点体系结构



2.1 线性表的逻辑结构

- 线性表定义：
  - 由n（n≥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，……a_{i-1}，a_i，a_{i+1}，……，a_n）
    - a_i（$1 \le i \le n$）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=（ ）。
  - 图形表示：
    - 线性表L=（a_1，a_2，…a_i，……，a_n）的图形表示如下：

2.1 线性表的逻辑结构(Cont.)

- 逻辑特征：

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

  – 有限性:
    - 线性表中数据元素的个数是有穷的。
  – 相同性:
    - a_i线性表中的元素类型相同
  – 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…a_{i-1},a_i,a_{i+1}…(1<i<n)，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 是型为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.)

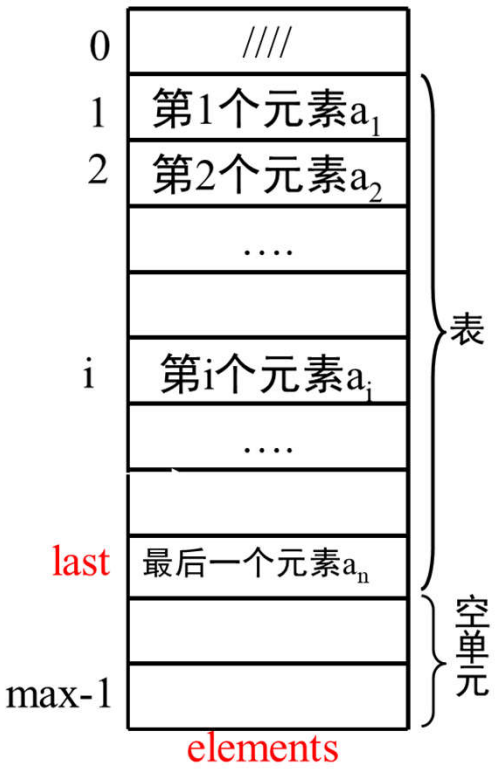
- ADT应用举例：
  - 设计函数 DeleteVal（LIST &L, ElemType d)，其功能为删除 L 中所有值为 d 的元素。

```
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 ) ;
        }
    }
```

2.2.1 顺序表

- 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）
  - 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。



2.2.1 顺序表(Cont.)

存储结构定义：

- 类型定义：

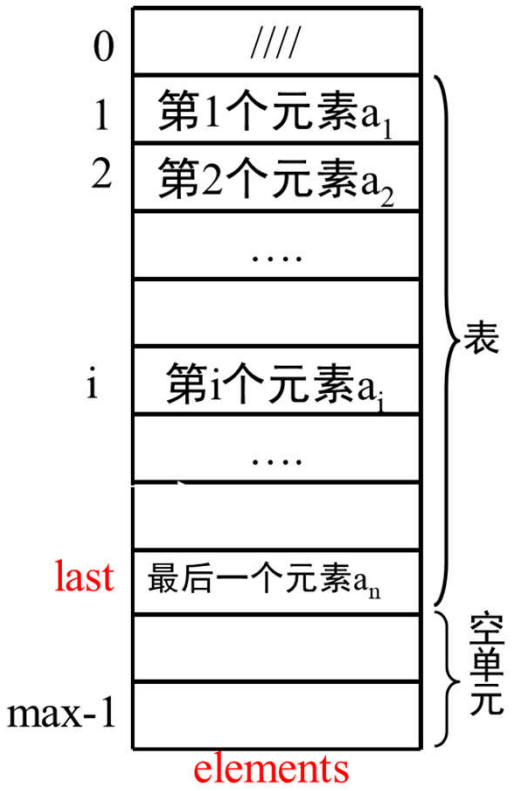
```
#define  max  100
struct LIST{
    ElemType elements[max];
    int last;
};
```

- 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
```



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

a_{p-1}和a_p之间的逻辑关系发生了变化

↓

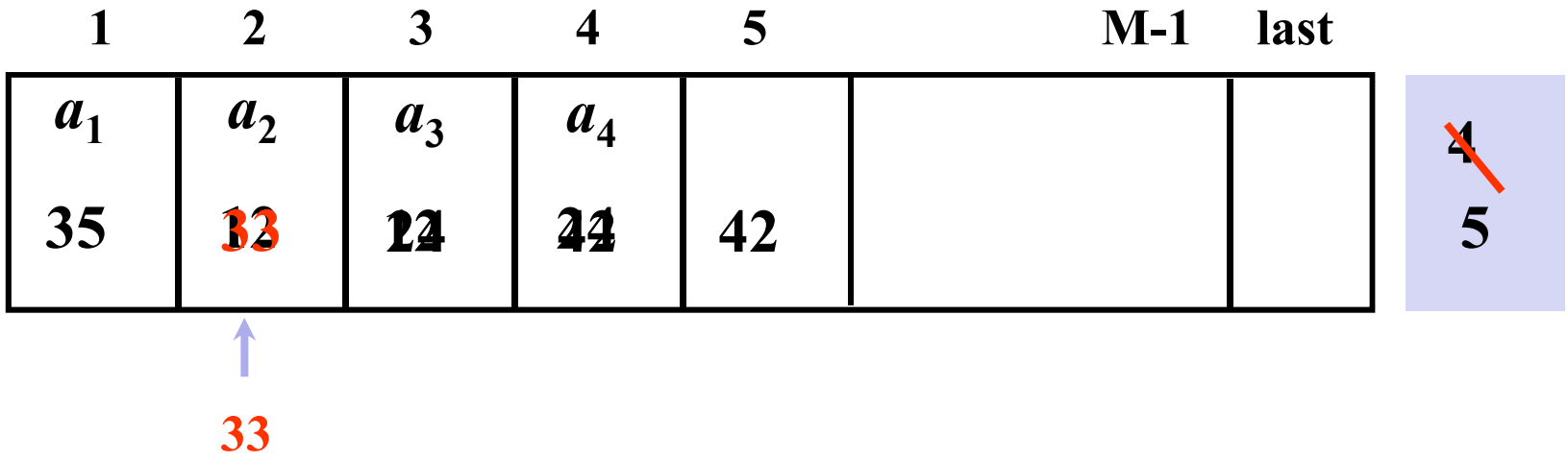
顺序存储要求存储位置反映逻辑关系

↓

存储位置要反映发生的变化

2.2.1 顺序表(Cont.)

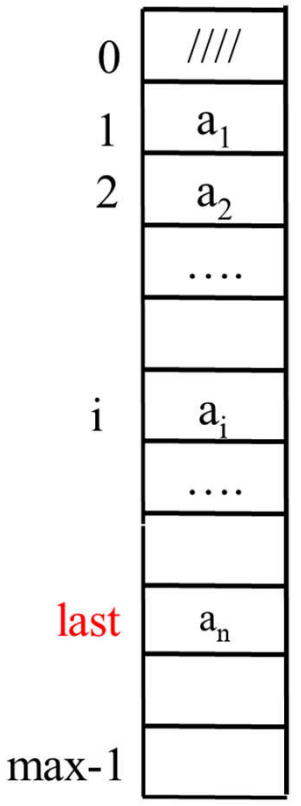
- 例:（35，12，24，42），在p=2的位置上插入33。



- 什么时候不能插入？
  - 表满时：L.last >=Max　//边界条件
  - 合理的插入位置：$1 \le p \le L.last+1$

2.2.1 顺序表(Cont.)

```
// 插入操作①
void  Insert ( ElemType  x,  position  p, LIST  &L)  // 操作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 ;
     }
}
```

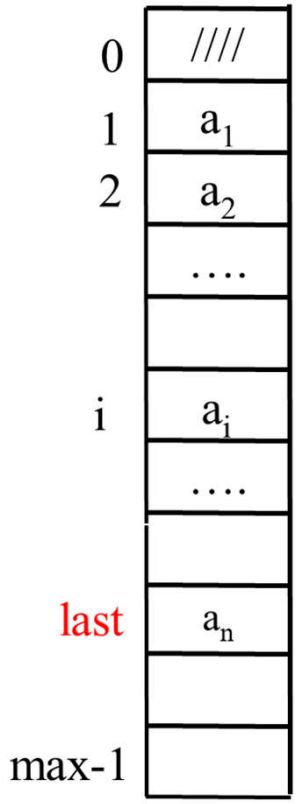


2.2.1 顺序表(Cont.)

- 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（$1 \le i \le n+1$）：

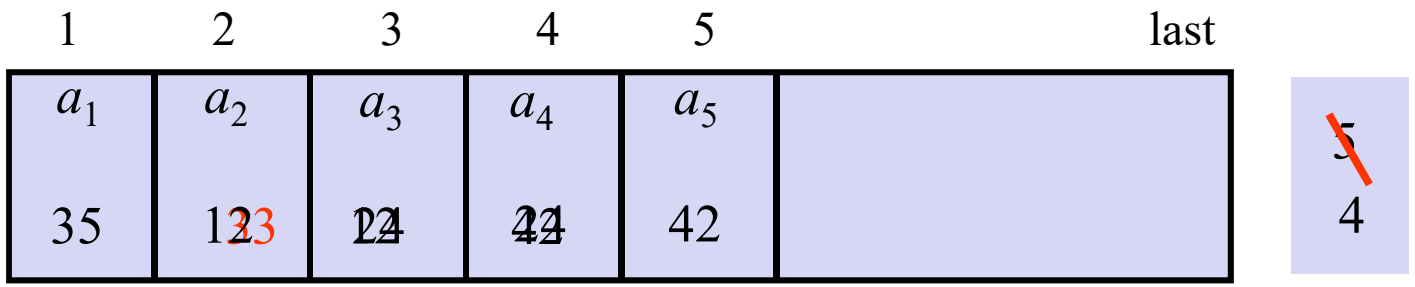
$$\sum_{i=1}^{n+1} p_i (n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2} = O(n)$$

  - 时间复杂度为$O(n)$。



2.2.1 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：$(a_1, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_n)$
  - 删除后：$(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n)$
- 例：（35, 33, 12, 24, 42），删除p=2的数据元素。



- 分析边界条件？
- 操作算法的实现
- 时间性能分析

2.2.1 顺序表(Cont.)

```
//操作②
    void Delete( position p, LIST  &L)
    {   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)

2.2.1 顺序表(Cont.)

- 其他操作的实现

```
position  Locate ( ElemType x , LIST  L ) //操作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 ) //操作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)
```

2.2.1 顺序表(Cont.)

- 其他操作的实现

```
position  Previous( position  p , LIST  L ) //操作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 ) //操作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)
```

2.2.1 顺序表(Cont.)

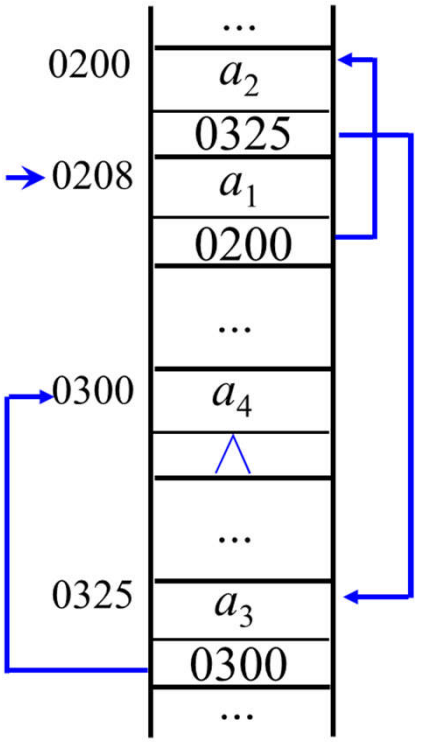
- 其他操作的实现

```
position   MakeNull( LIST &L )  //操作⑦
  {    L.last = 0 ;
       return ( L.last +1 );
  } //时间复杂性：O(1)

position  First( LIST  L )          //操作⑧
   {   if ( L.last>0 )    return ( 1 );
       else cout<<“表为空”;//return 0;
    } //复杂性：O(1)

position  End( LIST  L )            //操作⑨
   {  return( L.last + 1 );
    } // O(1)
```

2.2.2 链接表

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；
  - 形成一个单向链接式存储结构，简称单向链表或单链表。
  - 例： $(a1, a2, a3, a4)$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）



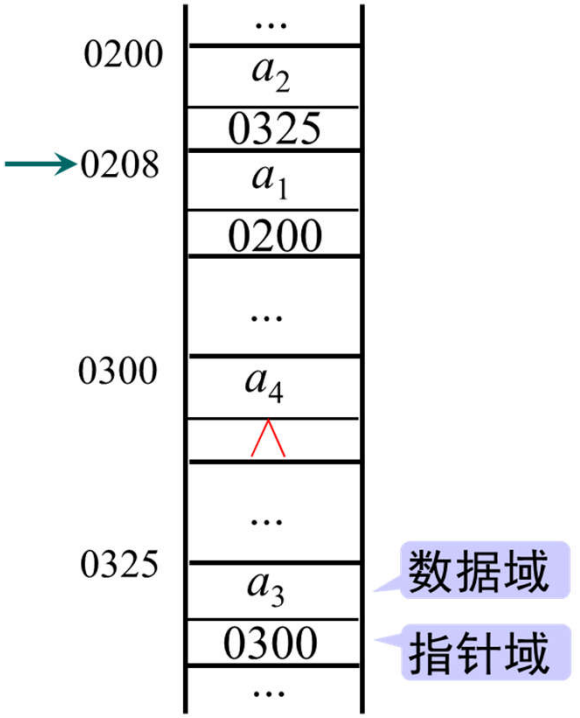
2.2.2 链接表(Cont.)

- 存储结构定义:
  - 结点结构:

数据域	指针域
data	next

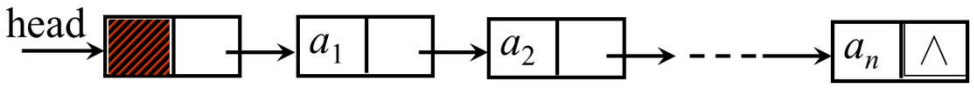
  - 存储结构类型定义:

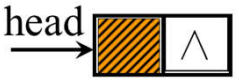
```
struct   celltype  {
      ElemType  data ;
      celltype  *next ;
} ; //结点型

typedef  celltype *LIST; //线性表的型
typedef  celltype  *position; //位置型
```



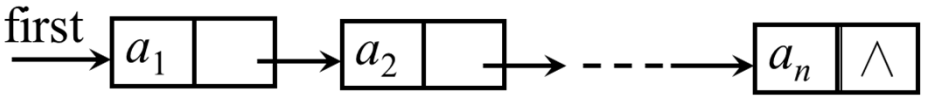
2.2.2 链接表(Cont.)

- 存储结构定义:
  - 单链表：
    - 带表头结点的单链表





    - 不表带头结点的单链表



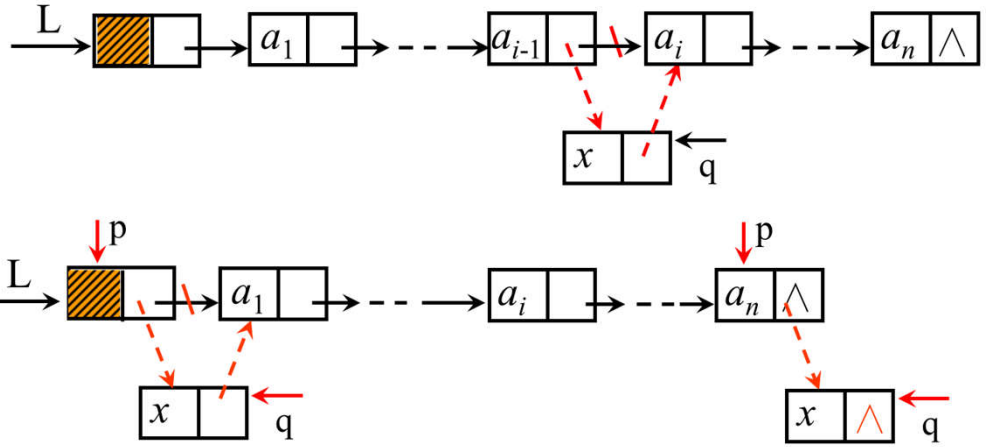
first==NULL；

  - 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
    - 是否带表头结点取决于是否方便实现操作

2.2.2 链接表(Cont.)

- 操作实现：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)
```

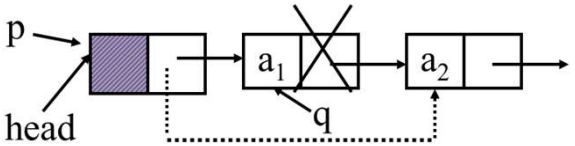


2.2.2 链接表(Cont.)

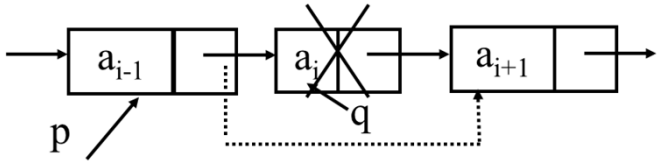
- 操作实现---- ②删除操作

```
q = p→next ;
p→next = q→next ;
delete  q ;
```

```
void Delete ( position  p, LIST  &L )
{   position  q ;
    if  ( p→next != NULL ) {
        q = p →next ;
        p →next = q →next ;
        delete  q ;
    }
```

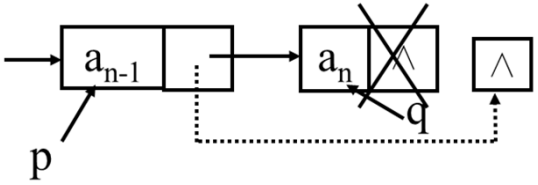


(a) 删除第一个元素



(b) 删除中间元素

- 时间复杂性：O（1）



(c) 删除表尾元素

2.2.2 链接表(Cont.)

- 操作实现：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)
```

- 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)
```

2.2.2 链接表(Cont.)

- 操作实现：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)
```

2.2.2 链接表(Cont.)

- 操作实现： 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)
```

2.2.2 链接表(Cont.)

- 操作实现： 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)
```

2.2.2 链接表(Cont.)

- 操作实现： 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)
```

2.2.2 链接表(Cont.)

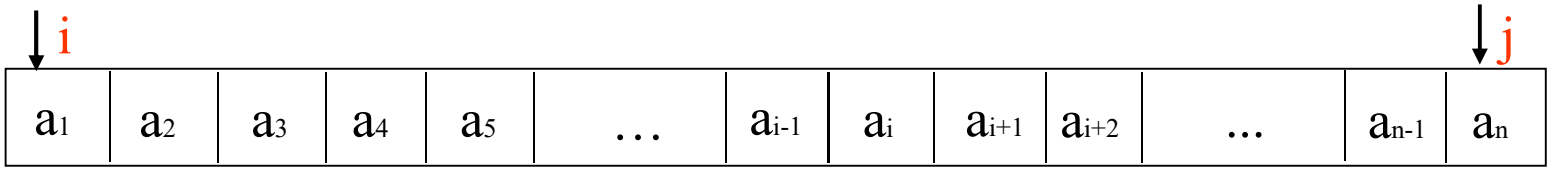
例：设计一个算法，遍历线性表，即按照线性表中元素的顺序，依次访问表中的每一个元素，每个元素只能被访问一次。

```
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 ;
     }
}
```

2.2.2 链接表(Cont.)

- 例：设计一个将线性表的元素逆置的算法
  - 顺序表逆置




```
void reverse(List &L)
{
   for ( i = 1, j = L.last; i < j; i ++, j -- )
        L.elements[i]<->L.elements[j];
}
```

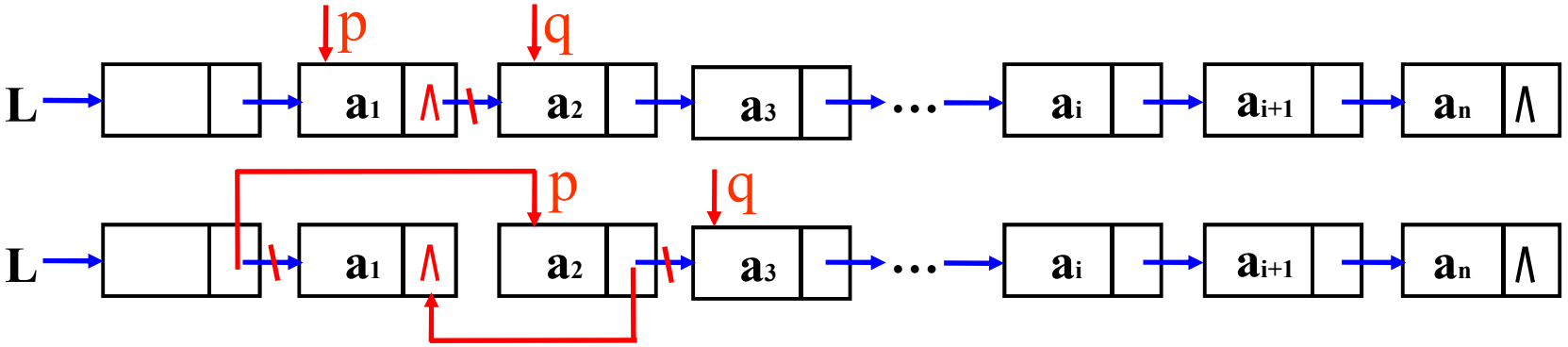
- 例：将数组元素循环左移/右移k位，如k=3

11	12	13	14	15	16	17	18	19
13	12	11	19	18	17	16	15	14
14	15	16	17	18	19	11	12	13

2.2.2 链接表(Cont.)

- 例：设计一个将线性表的元素逆置的算法
  - 链表逆置

```
void reverse(List &L)
{   p=L->next;
    if ( p ) {
        q=p->next; p->next=NULL;
        while (q!=NULL) {
            p=q; q=q->next;
            p->next=L->next;
            L->next=p; }}
}
```



2.2.1链接表(Cont.)

- 顺序表与 链表的比较

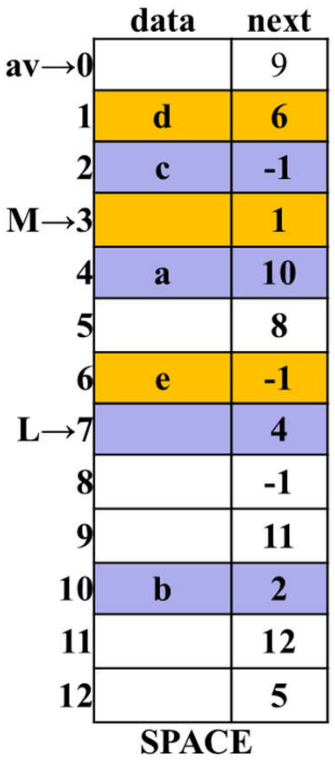
顺序表 单链表的比较

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

2.2.3 静态链表

静态链表

- **静态链表：**链表的数组+游标表示
  - 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）。
  - 举例：
    - 线性表：L=（a，b，c）
    - 线性表：M=（d，e）
    - 线性表：av=0--空闲表(可用空间表)



2.2.3 静态链表(Cont.)

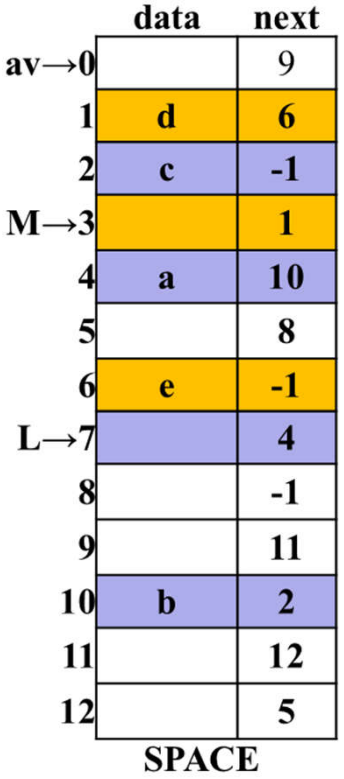
- 结点形式：

数据域	游标域
data	next

- 存储结构定义：

类型定义：

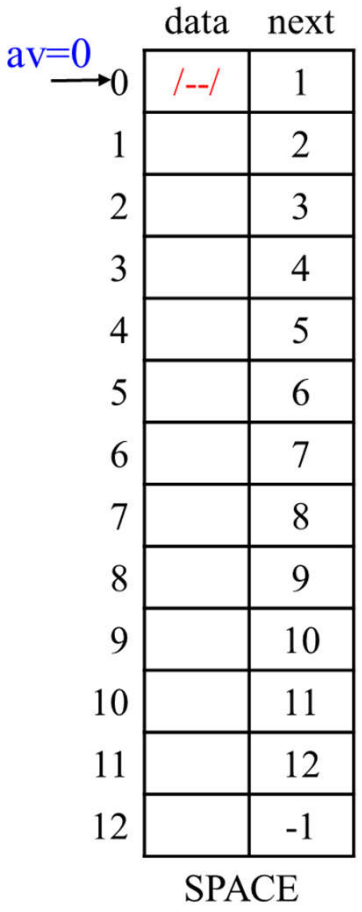
```
typedef struct {
    ElemType  data ;
    int    next ;
} spacestr;  /*结点类型*/
spacestr  SPACE[ maxsize ] ;/*存储池*/
typedef  int  position，Cursor ；
Cursor   L，M，av;
```



2.2.3 静态链表(Cont.)

- 操作实现: ①可用空间初始化

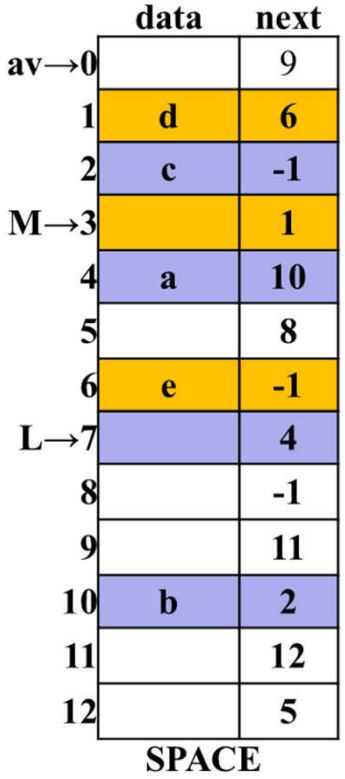
```
void  Initialize()
{
   int j;
   /* 依次链接池中结点 */
   for (j=0; j<maxsize-1; j++ )
        SPACE[j].next=j+1;
   /* 最后一个结点指针域为空 */
   SPACE[j].next = -1;
   /* 标识线性表 */
   av=0;
}
```



2.2.3 静态链表(Cont.)

- 操作实现：②可用空间的分配操作

```
Cursor GetNode()   //模仿q=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*/
```



SPACE

- 操作实现：③可用空间的回收操作

```
void FreeNode(Cursor q)  //模仿delete q;
{
        SPACE [ q ].next = SPACE[av].next ;
        SPACE[av].next = q ;
} //放回存储池SPACE中
```

2.2.3 静态链表(Cont.)

- 操作实现：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 ;
```

- 操作实现：⑤删除操作

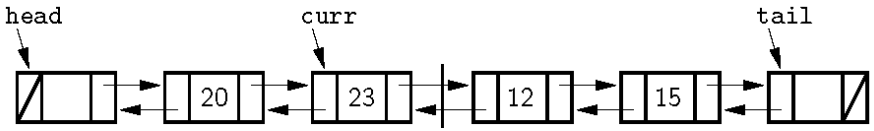
```
void Delete(position p, spacestr  *SPACE)
{    position  q ;
     if  ( SPACE[ p ].next != -1 ) {
          q = SPACE[ p ].next ;
          SPACE[ p ].next = SPACE[ q ].next ;
          FreeNode( q ) ;
     }
}
```

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

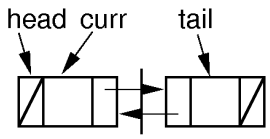
2.2.4 双向链表

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：

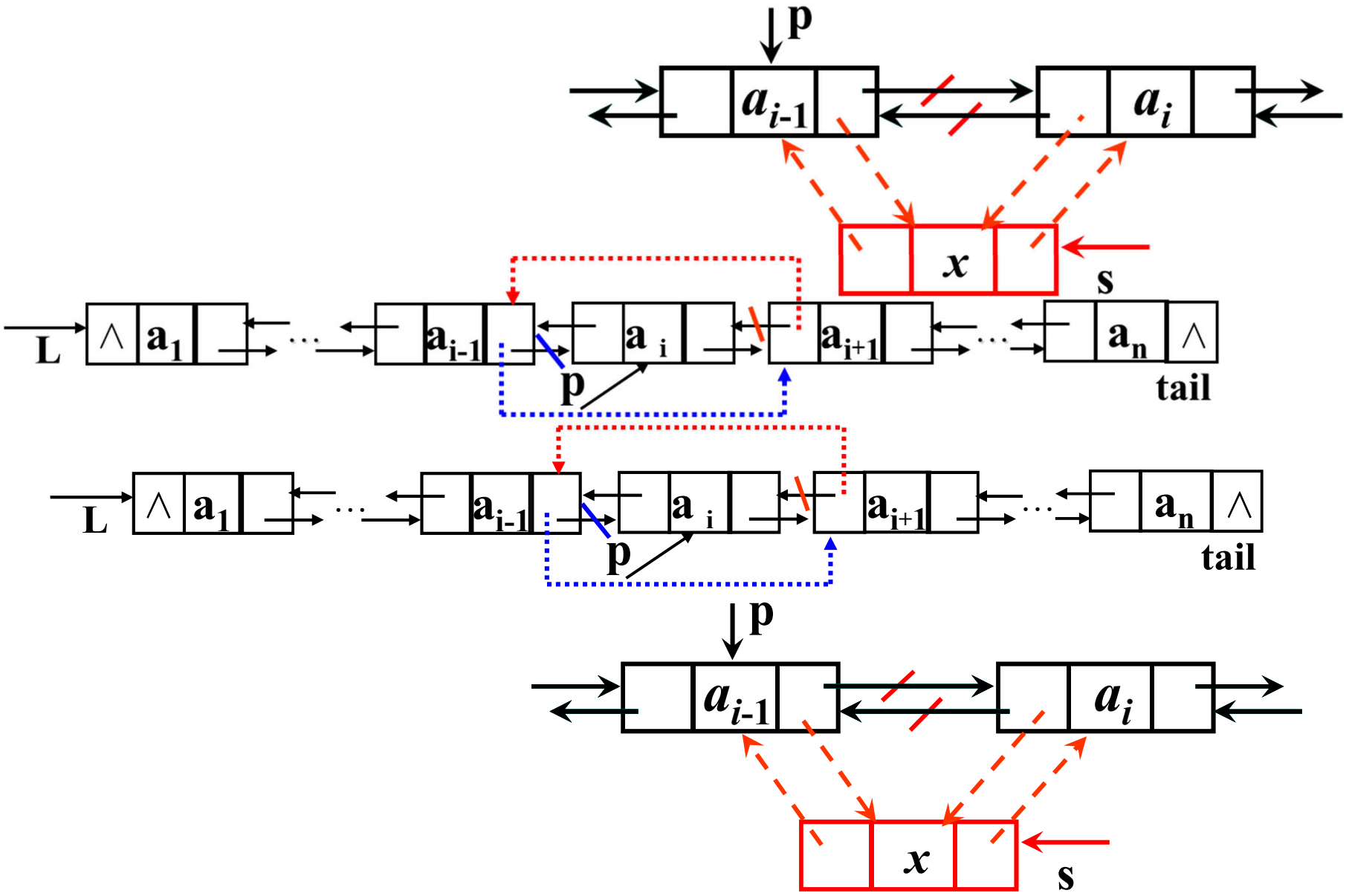


- 优点：
  - 实现双向查找
  - 表中的位置i 可以用指示含有第i 个结点的指针表示。
- 缺点：空间开销大

- 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 ;
```

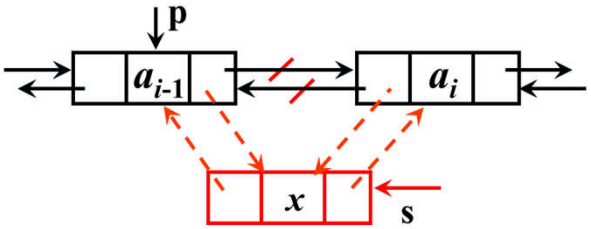
2.2.4 双向链表(Cont.)



2.2.4 双向链表(Cont.)

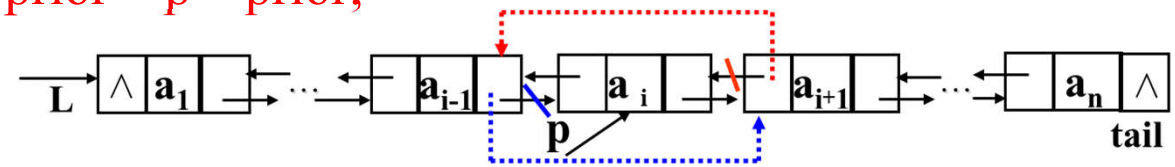
- 插入操作：在带头结点的表中, 位置p插入元素x。

```
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;
}
```



- 删除操作：在不带头结点的表中，删除位置p的元素。

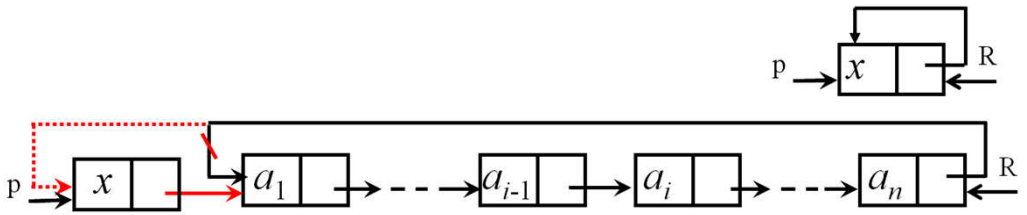
```
void Delete( position  p, DLIST &L)
{      if (p->prior!=NULL)
           p→prior→next = p→next;
       if (p→next!=NULL)
           p→next →prior = p→prior;
       delete p;
}
```



2.2.5 单向环形链表 (Cont.)

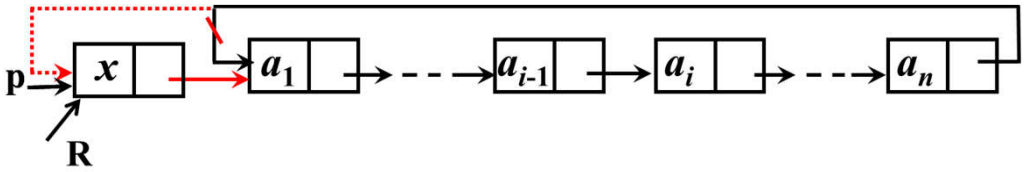
- 在表左端插入结点LInsert(x,R) → Insert(x,First(R),R)

```
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 ;  }
}
```



- 在表右端插入结点RInsert(x,R))→ Insert(x,End(R), R)

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

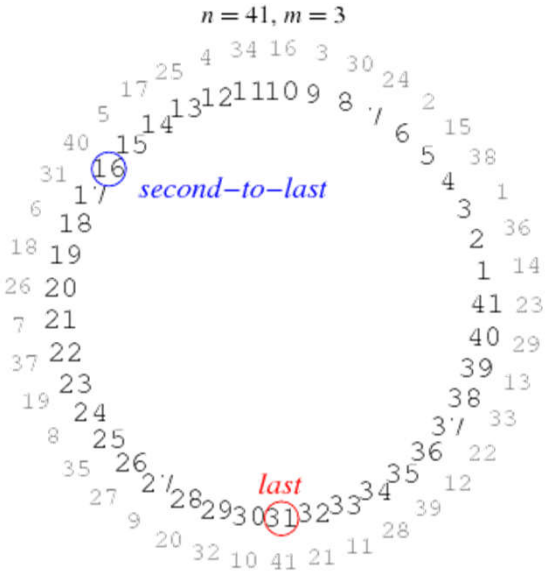


2.2.5 单向环形链表 (Cont.)

- 用循环链表求解约瑟夫问题
  - 约瑟夫问题:

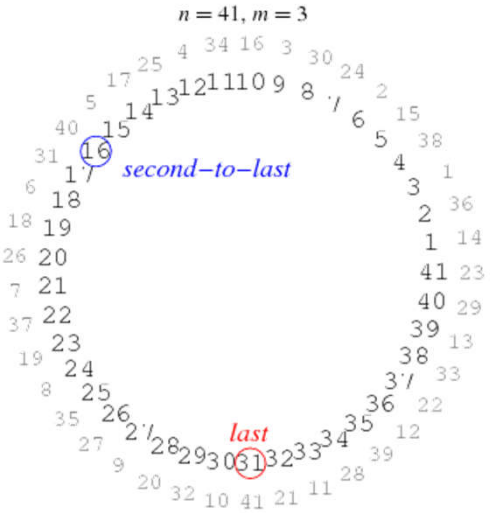
    n个人围成一个圆圈，首先，第1个人从1开始，顺时针报数, 报到第m个人，令其出列。然后再从下一个人开始，从1顺时针报数，报到第m个人，再令其出列，…，如此下去, 直到圆圈中只剩一个人为止。此人即为优胜者。

  - 用什么数据结构?
  - 算法基本思路?



2.2.5 单向环形链表 (Cont.)

- 用循环链表求解约瑟夫问题
  - 约瑟夫问题求解算法:




```
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;
     }
}
```

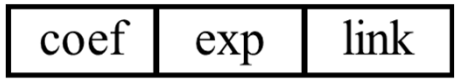
2.2.6 一元多项式运算

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
```

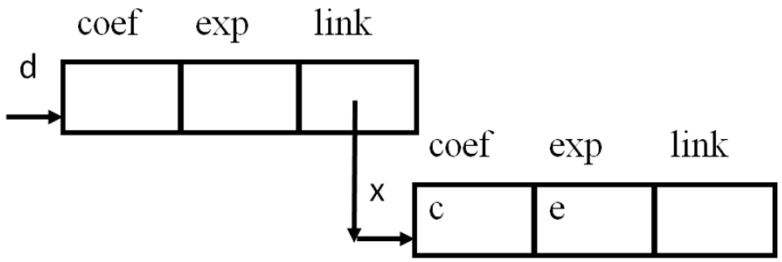


2.2.6 一元多项式运算

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。

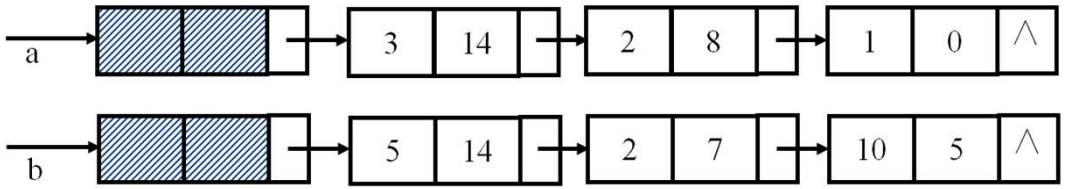
```
polypointer  Attch ( int  c , int e , polypointer  d )
{     polypointer   x  ;
      x = new   polynode ;
      x→coef = c ;
      x→exp = e ;
      d→link = x ;
      return  x ;
}
```



2.2.6 一元多项式运算

- 多项式加法：

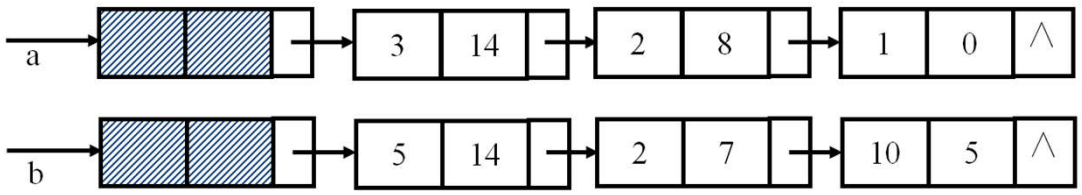
```
polypointer PolyAdd ( polypointer a , polypointer b )
{    polypointer p, q, d, c;
     int y ;
     p = a→link;  q = b→link;
     c = new polynode; d = c ;
     while ( (p != NULL) && (q != NULL) )
         switch ( Compare ( p→exp, q→exp ) )
         {  case '=' :
                y = p→coef + q→coef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



2.2.6 一元多项式运算

- 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;
}
```

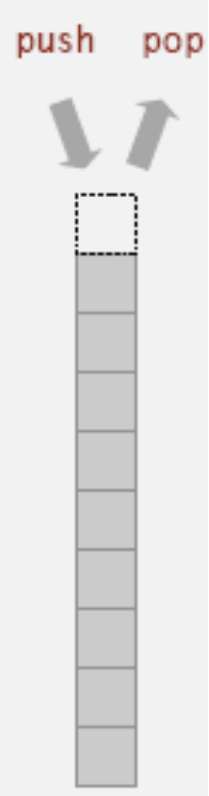


2.2.6 一元多项式运算

- 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

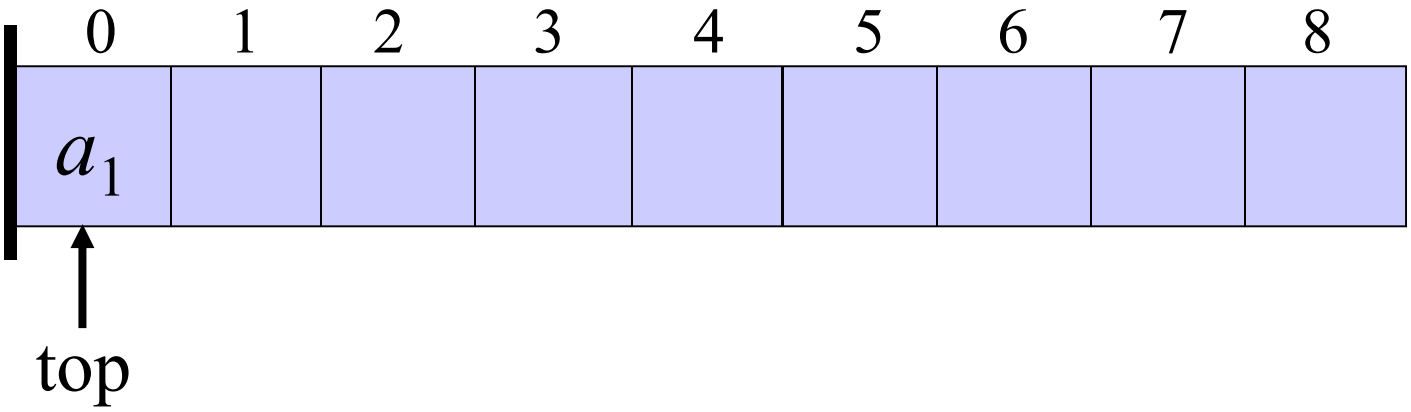
2.3 特殊的线性表:栈

- 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

- 栈的顺序存储结构与实现
- 如何改造数组实现栈的顺序存储？

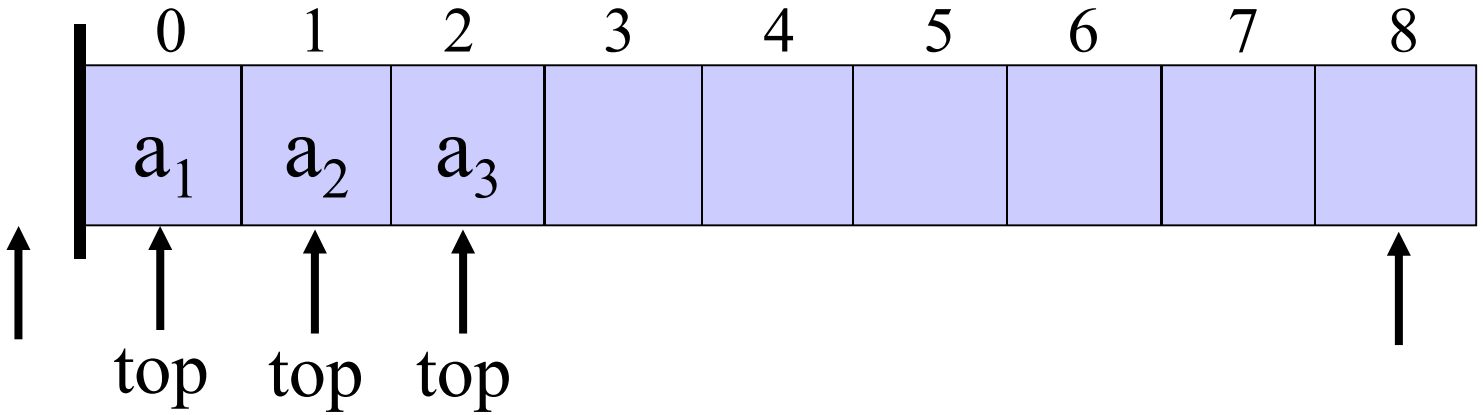


确定用数组的哪一端表示栈底。

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

出栈：top减1

栈空：top= -1

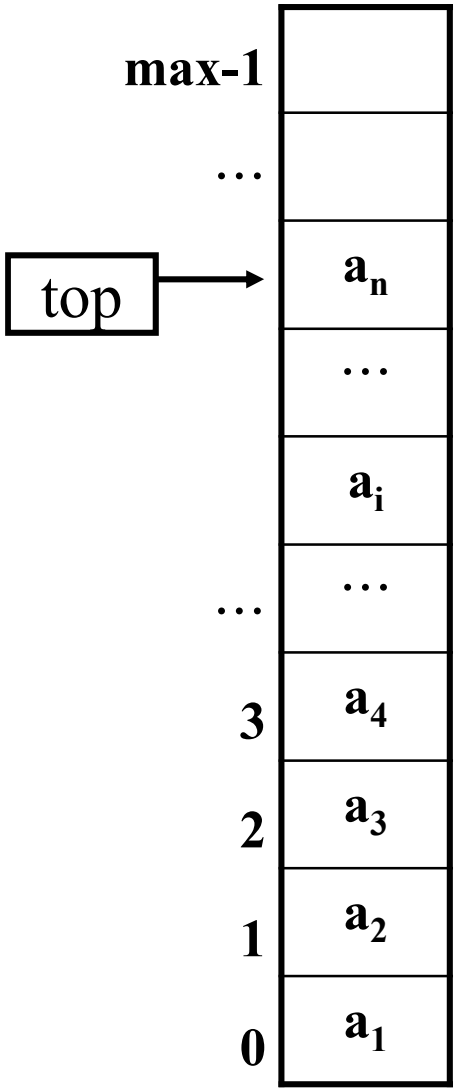
栈满：top= max-1

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 ;



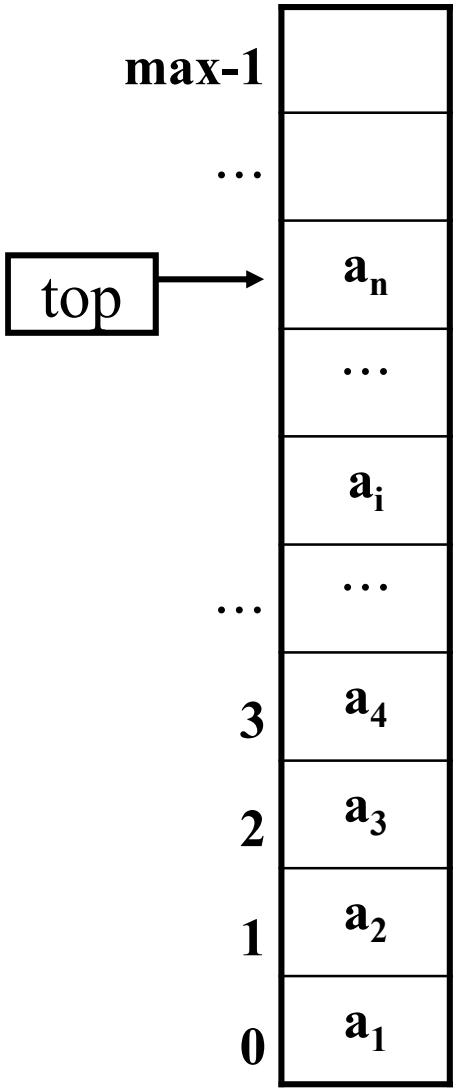
顺序栈示意图

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 ;
}
```

2.3.1 栈的数组实现：顺序栈(Cont.)

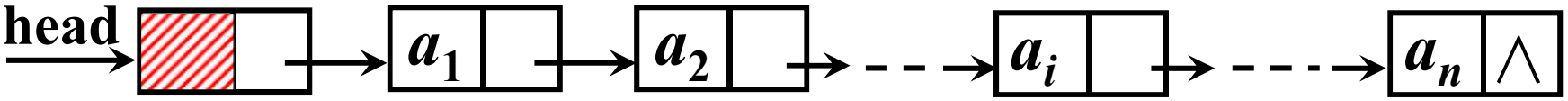
- 栈操作实现： ⑤ Push()

```
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 ;
          }
      }
```

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

- 如何改造链表实现栈的链接存储？

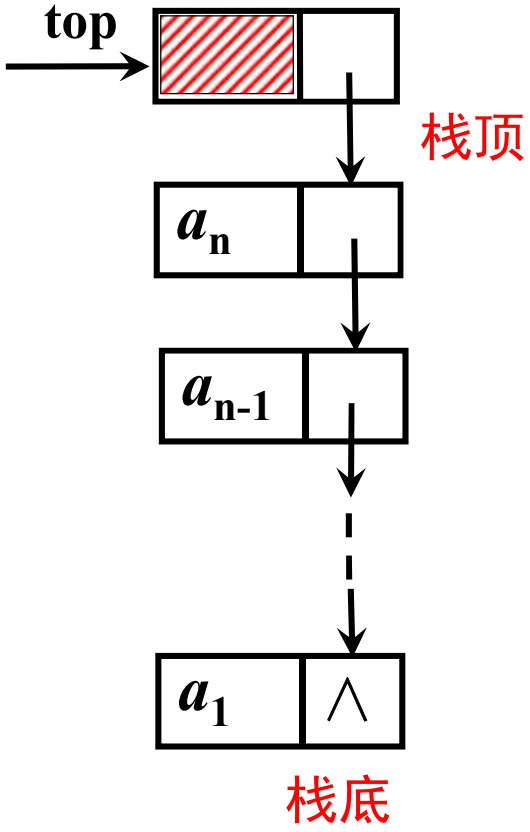


- 将哪一端作为栈顶？
  - 将链表首端作为栈顶，方便操作。
- 链栈需要加头结点吗？
  - 表头结点的作用与链表相同

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

①STACK MakeNull()

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

②boolean Empty(STACK stk)

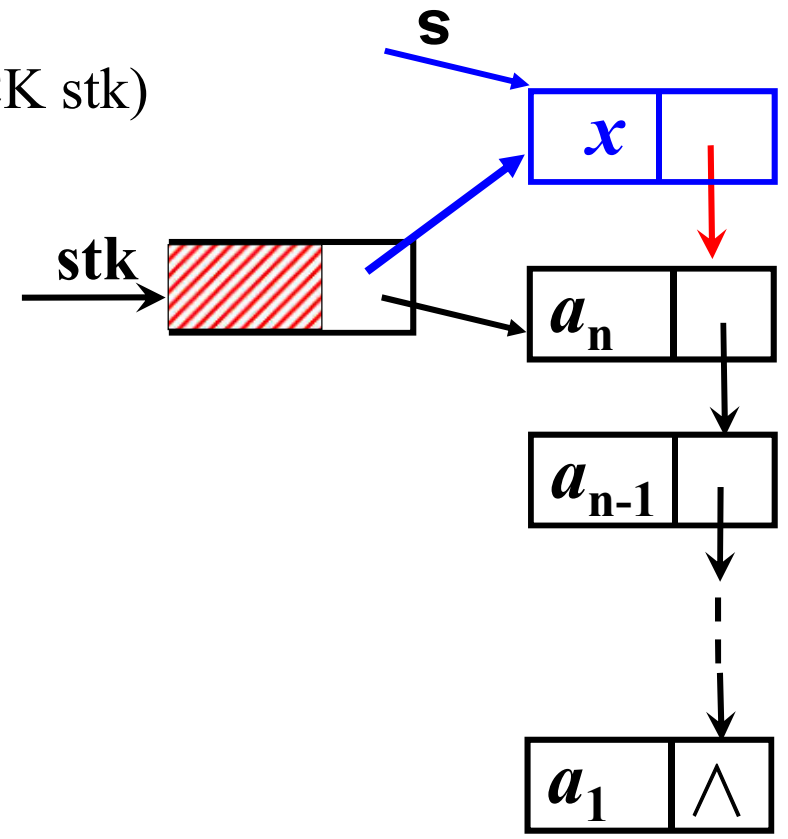
```
{       if (stk->next)
            return FALSE;
        else
            return TRUE;
}
```

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)

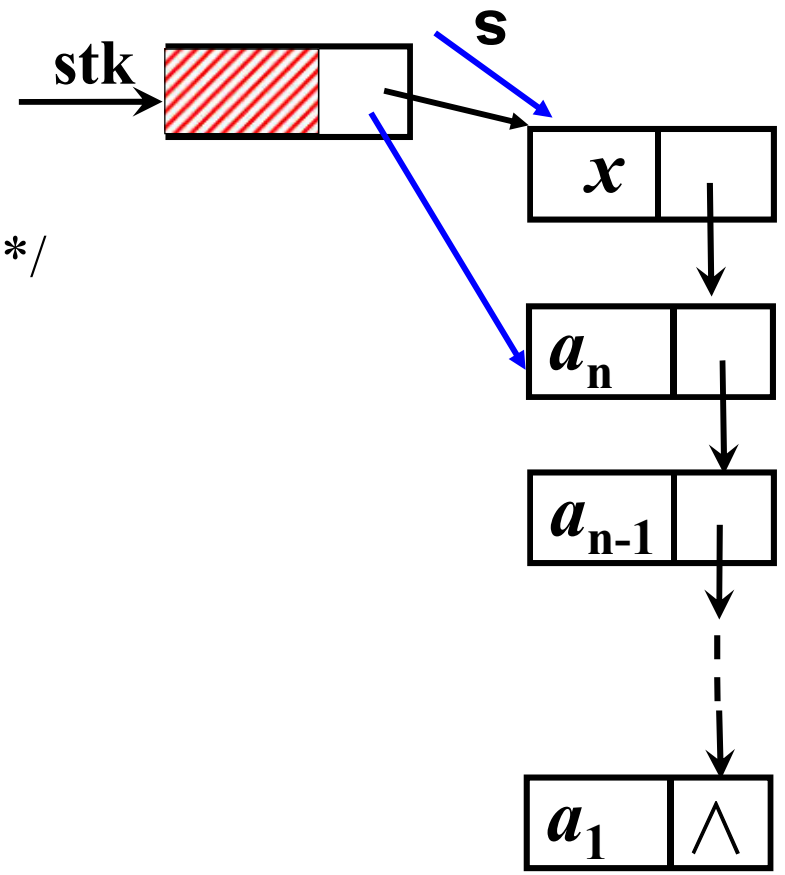
```
{
        STACK s;
        s=new node;
        s->data=elm;
        s->next=stk->next;
        stk->next=s;
}
```



2.3.2 栈的指针实现：链栈(Cont.)

- 栈的操作的实现

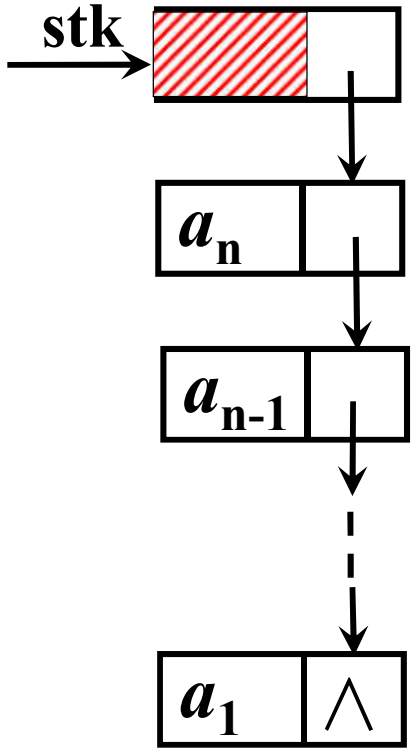
④ void Pop(STACK stk)
{ STACK s;
if (stk->next){/*stk->next!=NULL*/
s=stk->next;
stk->next=s->next;
delete s; /* free(s) */
}
}



2.3.2 栈的指针实现：链栈(Cont.)

- 栈的操作的实现

```
⑤ElemType Top( STACK stk)
{    if (stk->next)
          return (stk->next->data);
     else
          return NULLES;
}
```



2.3.3 栈与递归调用

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同类小问题，再把这些小问题进一步分解成更小的同类小问题，直至每个最小问题可直接求解。
- 递归的要素
  - 递归终止条件：确定递归终止的边界条件，也称为递归出口；
  - 递归模式：将大问题分解为小问题，也称为递归体

递归应用示例：求解阶乘

- 阶乘数学计算公式:

$$n! = n \times (n-1) \times \cdots \times 1.$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

终止条件

将大问题分解为小问题

递归过程

每步递归由两部分组成：

1. 若是最小问题(一个或多个），可直接求解;

2. 否则，递归模式

   - 将当前问题分解成一个或多个同类小问题，逐渐接近最小问题。
   - 直到分解为可求解的最小问题

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

应用示例：求解阶乘算法

- 递归函数
  - 程序中函数调用本身.

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

递归算法通常由if条件语句构成：

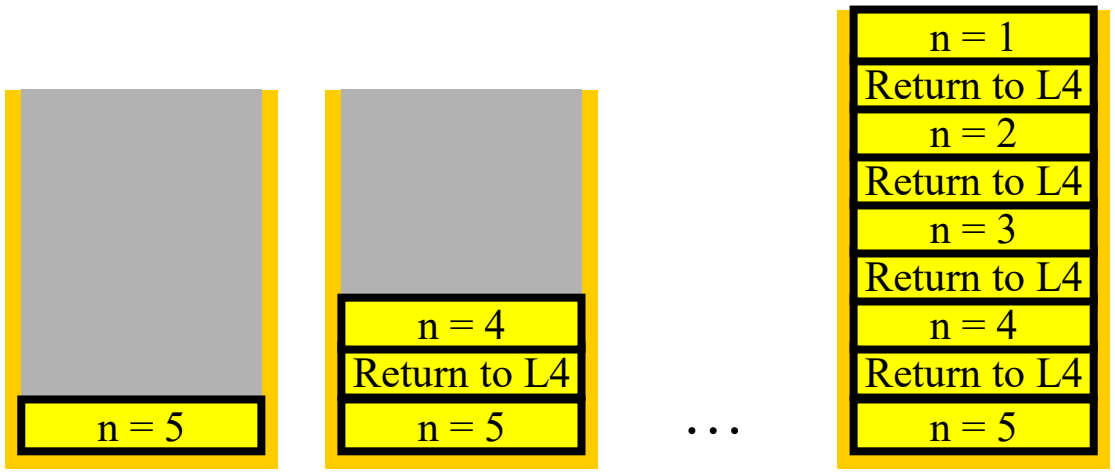
+: 简练

-: 需要跟踪记录中间过程与结果

递归调用过程

```
long factorial(int n){
L1      if (n==0)
L2        return(1);
L3      else
L4        return(n * factorial(n-1));
}
```

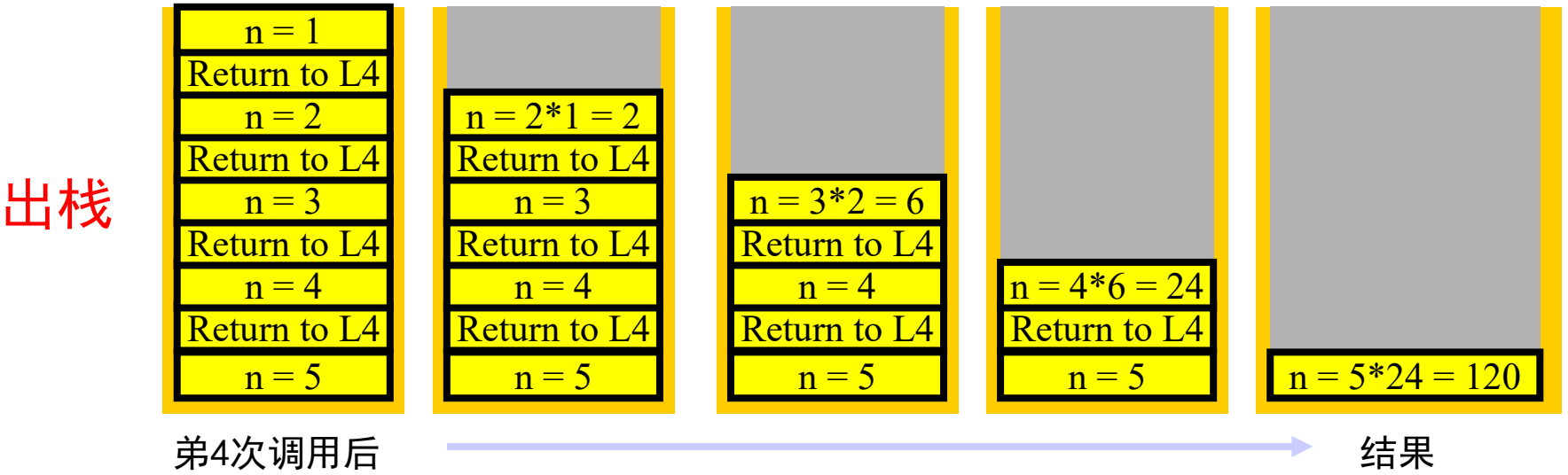
压栈



- 每次调用创建一个或多个局部变量
- 压栈：调用后的返回地址、变量当前值。

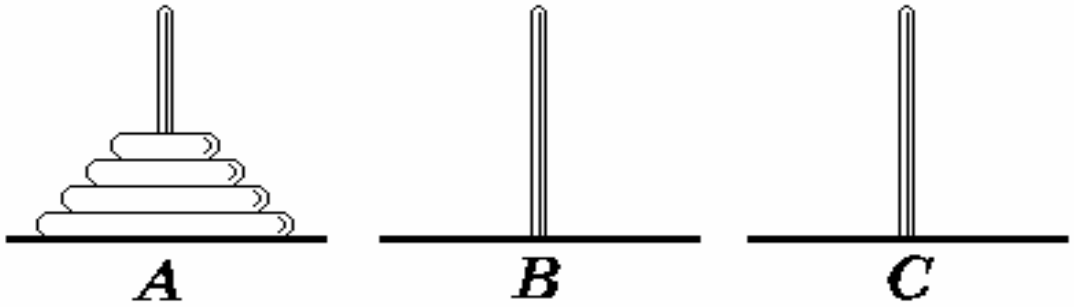
回归求值过程

```
long factorial(int n){
L1      if (n==0)
L2        return(1);
L3      else
L4        return(n * factorial(n-1));
}
```



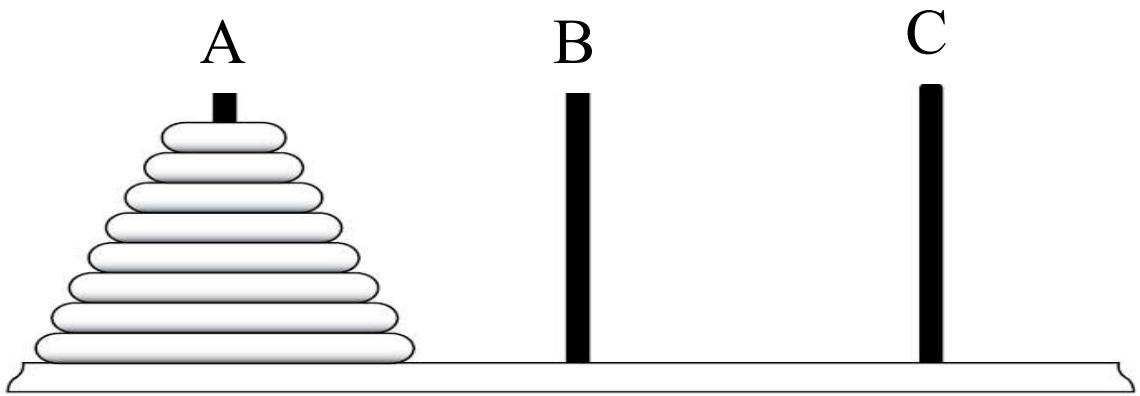
递归示例：汉诺塔问题

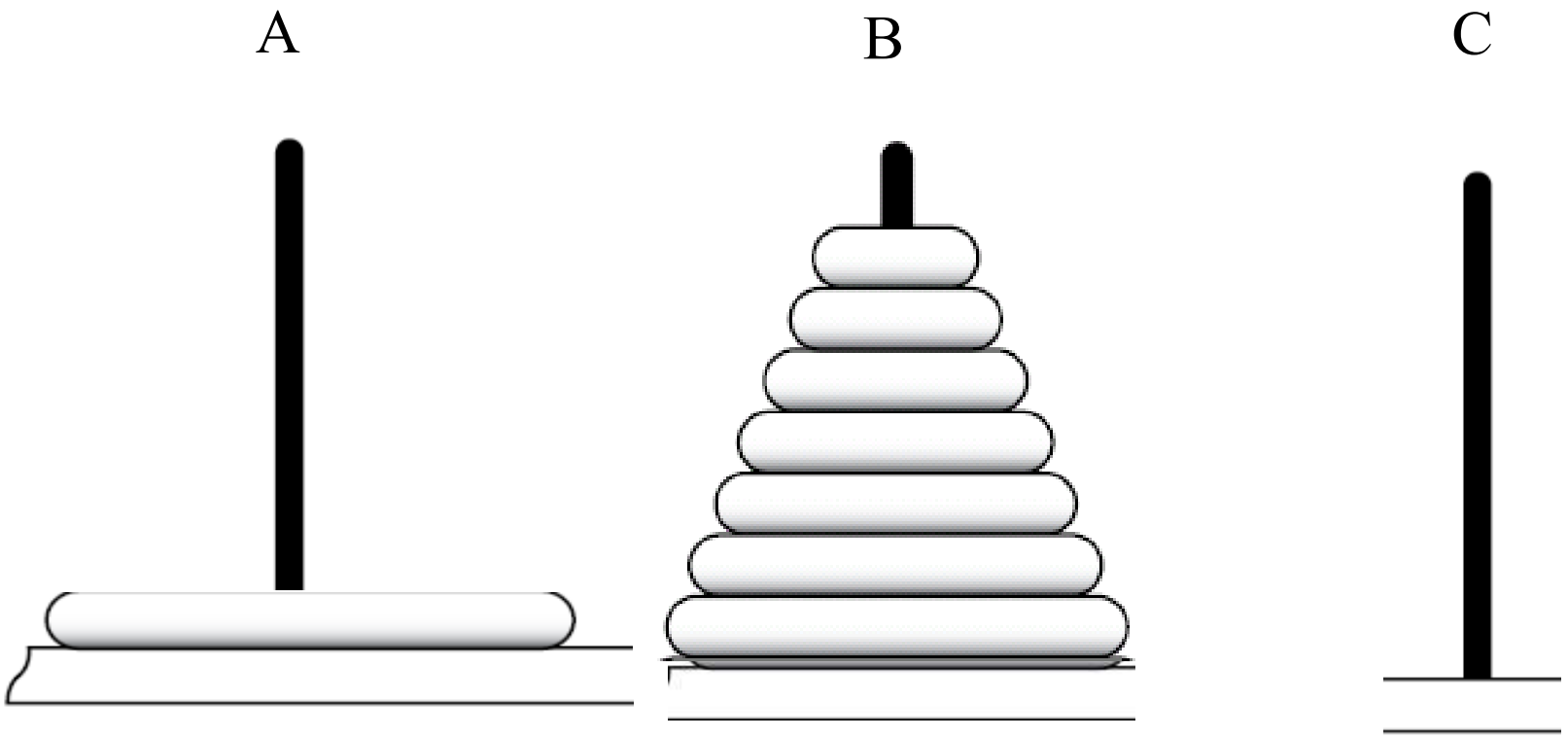
- 汉诺塔问题（Tower of Hanoi）：递归的经典问题
  - 印度古老传说：在世界刚被创建的时候有一座钻石宝塔（塔A），其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。
  - 把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。



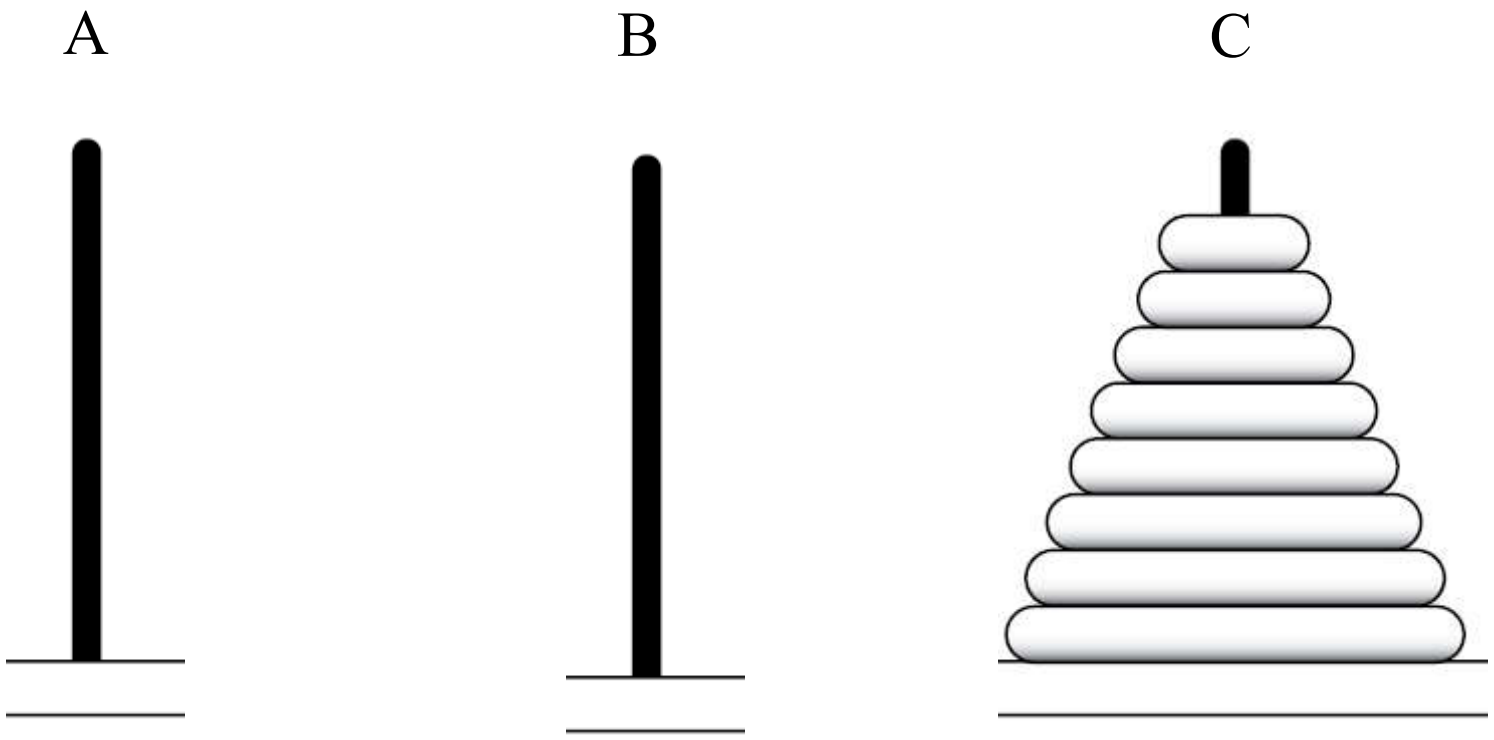
递归示例：汉诺塔问题

- 任务:借助于塔B，把塔A上的碟子移动到塔C上
- 规则:
  1. 每次只能移动一个碟子；
  2. 任何时候都不能把一个碟子放在比它小的碟子上面





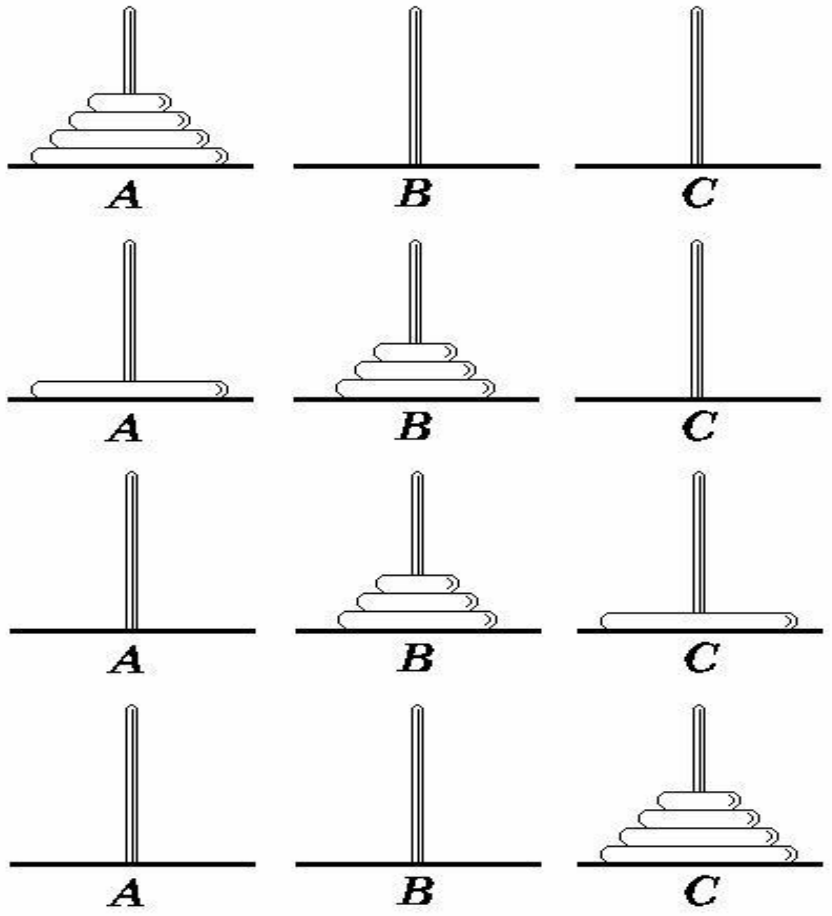
需要到这种状态，才能移动最大/最底盘子



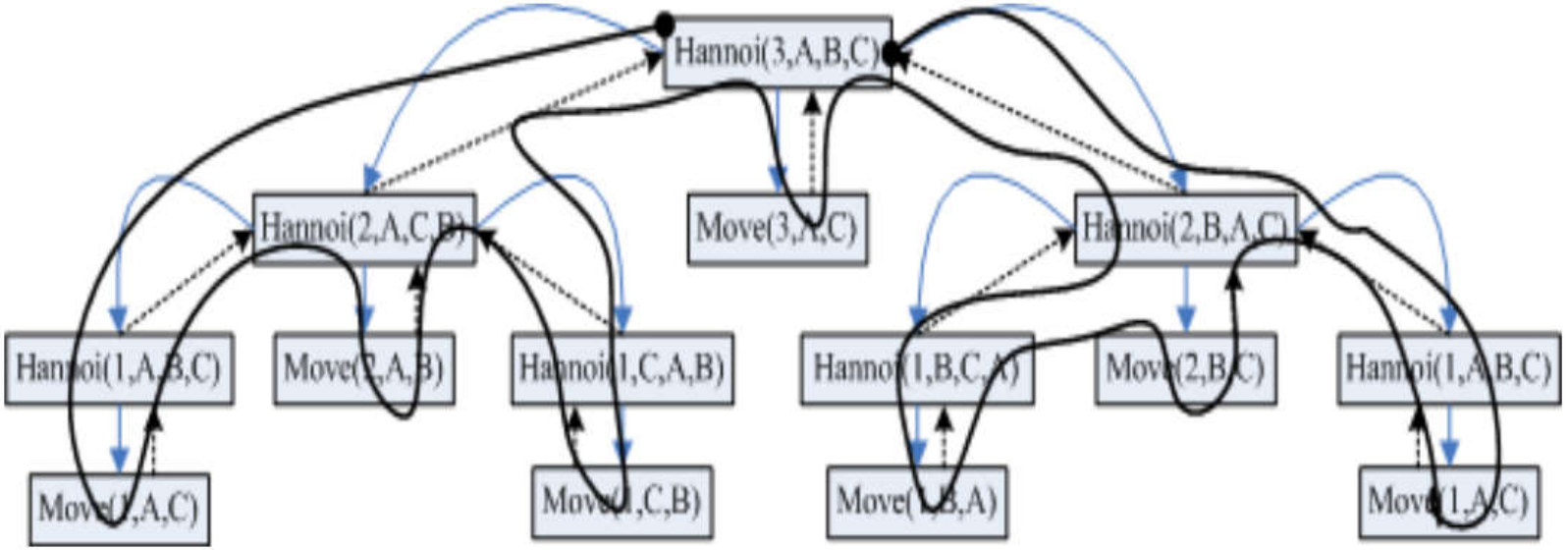
利用递归方法求解

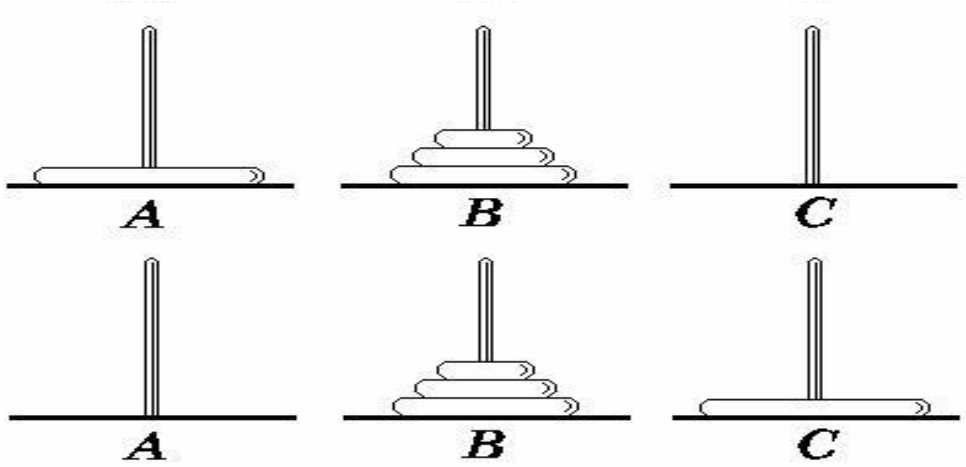
递归函数求解

```
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);
     }
}
```



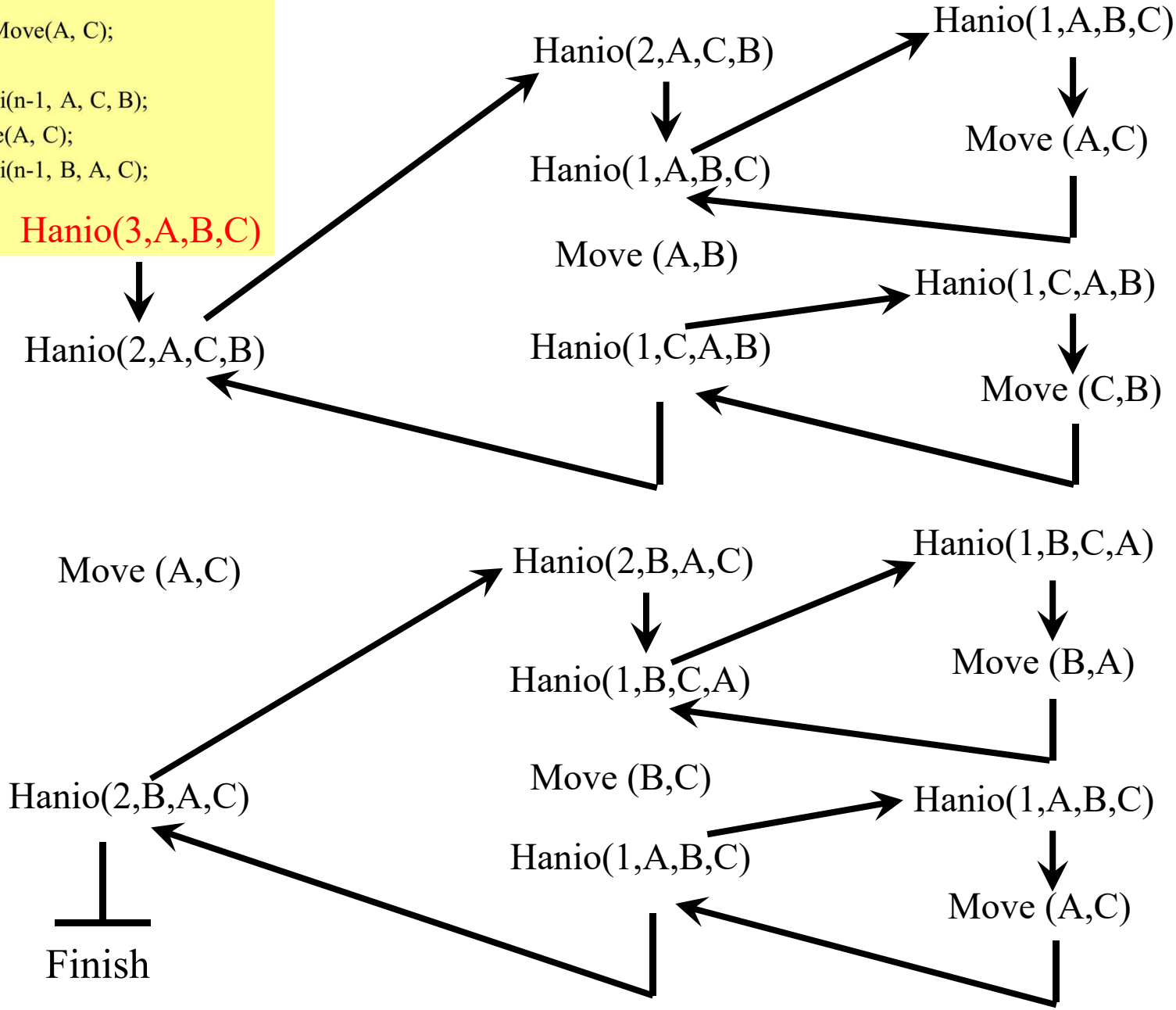
递归求解过程


Hannoi(3,A,B,C)
Hannoi(2,A,C,B)
Move(3,A,C)
Hannoi(2,B,A,C)
Hannoi(1,A,B,C)
Move(2,A,B)
Hannoi(1,C,A,B)
Hannoi(1,B,C,A)
Move(2,B,C)
Hannoi(1,A,B,C)
Move(1,A,C)
Move(1,C,B)
Move(1,B,A)
Move(1,A,C)


A
B
C
A
B
C

```
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);
    }
}
```

Hanio(3,A,B,C)



递归求解汉诺塔问题

时间复杂度：

- 多少步骤？

  $F(n) = 2F(n-1)+1 = 2^n-1$

- 若n=64,则F(64)约为$1.8*10^{19}$
- 若每秒移动1个碟子，则移动64个需要约580亿年

```
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);
    }
}
```

2.3.4 栈的应用

栈的应用：数制转换

- 数制转换：计算机实现计算的基本问题。
- 除留余数法：
  - N=(N/ d) × d+ N% d
- 示例：
- 十进制转二进制：(100)10 = (1100100)2
  - 用2除以十进制数
  - 直到商为0
  - 输出：从下至上输出余数，产生商的相反顺序。
- 如何实现？
  - 数据结构：栈

2	100	
2	50	0
2	25	0
2	12	1
2	6	0
2	3	0
2	1	1
	0	1

栈的应用：数制转换

- 除留余数法的实现：
  - 对输入的任意非负十进制整数,打印输出与其等值的**d**进制数

```
void main()
{    STACK s=NEWSTACK();
            cin>>n;
     while(n){
                  Push(n%d,s);
                  n/=d;
              }
              while(! Empty(s)) {
              cout<<Top(s);
                   POP(s) ;
     }
}//时间复杂度?
```

栈的应用：表达式求值

- 表达式求值：编译器和计算器设计中一个最基本问题
- 表达式的三种形式

$$\text{表达式}:\begin{cases}\text{前缀表达式(波兰式)}\\ \text{中缀表达式}\\ \text{后缀表达式(逆波兰式)}\end{cases}\quad \text{例如，}(a+b)*(a-b)=\begin{cases}* + a\ b - a\ b\\ (a+b)*(a-b)\\ a\ b + a\ b - *\end{cases}$$

  - 高级语言采用类似自然语言的中缀表达式，但计算机对后缀或前缀表达式的处理则简单容易。
  - 后缀表达式特点：
    - 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，运算操作符的位置确定运算顺序。

栈的应用:表达式求值

表达式求值方法：算符优先法

第一步：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：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。

栈的应用:表达式求值

第二步：由后缀表达式计算表达式值（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，与第一步相反，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。

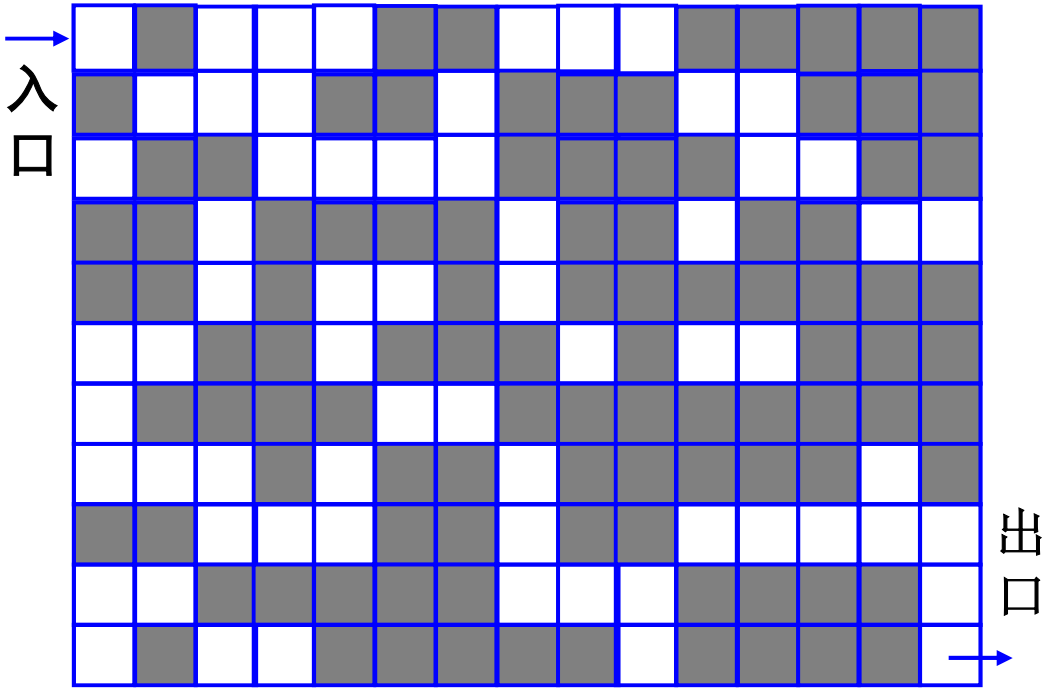
思考：如何处理中缀表达式?

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

栈的应用：迷宫求解

迷宫求解

- 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

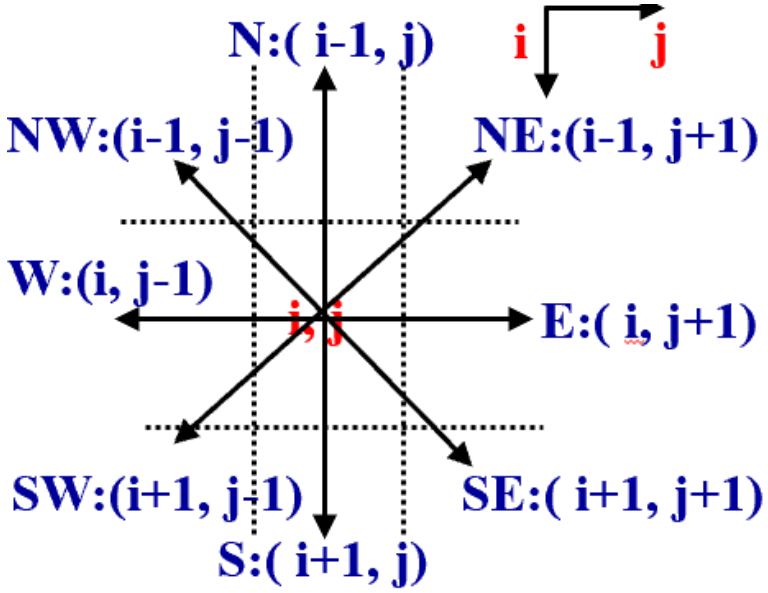
栈的应用：迷宫求解

- 迷宫求解
  - 分析：
    - 迷宫表示：用二维数组maze[i , j] ($1 \le i \le m, 1 \le j \le n$) 表示，入口maze[1, 1] = 0 ；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。
    - 移动模式：方向 v 按从正东开始且顺时针分别记为1,2,…,8，v=1,…,8;
    - 利用二维数组 move 记下八个方位的增量。

栈的应用：迷宫求解

迷宫求解

– 移动一步



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

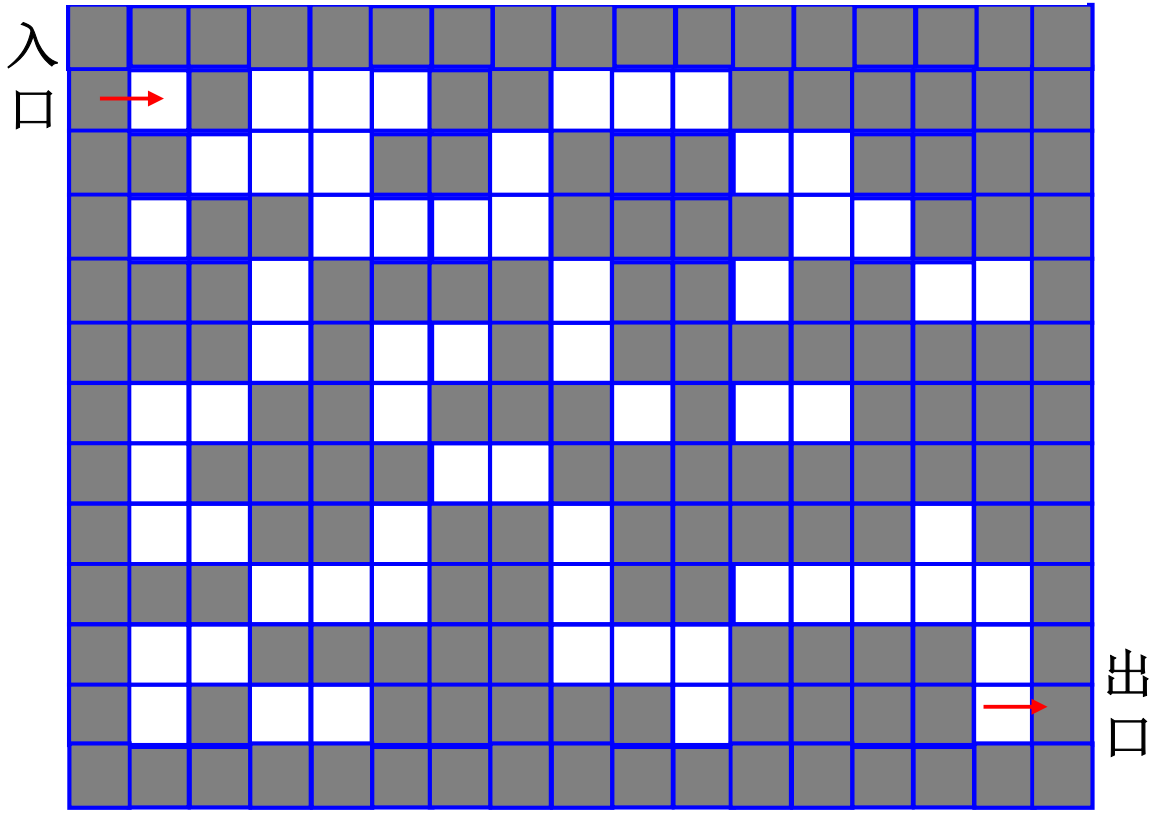
move

– 如从(i, j) 到 (g, h) 且 v = 2 (东南），则有：

- g = i + move[2, 1] = i + 1 ;
- h = j + move[2, 2] = j + 1 ;

栈的应用：迷宫求解

- 为避免每步监测边界状态，可把二维数组maze[1:m, 1:n]扩大为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

栈的应用：迷宫求解

- 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。

思考：能够回退到哪里？

S
S→P
S →P →Q
S →P→Q →R
S →P→Q →R →T
S →P→Q →R
S →P→Q
S →P
S

- 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;

栈的应用：迷宫求解

算法思想

1. 老鼠在(1, 1)进入迷宫，并向正东（v=1）方向试探；
2. 检测下一位置(g, h)：若(g, h)=(m, n)且maze[m, n]=0，则老鼠到达出口，输出走过的路径，算法结束；
3. 若(g, h) ≠(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)位置换一个方向试探。
4. 若(i,j)位置周围8个方位阻塞或已经过，则需退一步，并换一个位置试探。若(i, j)=(1, 1)则到达入口，说明迷宫走不通。

栈的应用：迷宫求解

算法实现

```
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!” ;
}
```

2.4 特殊的线性表: 队列

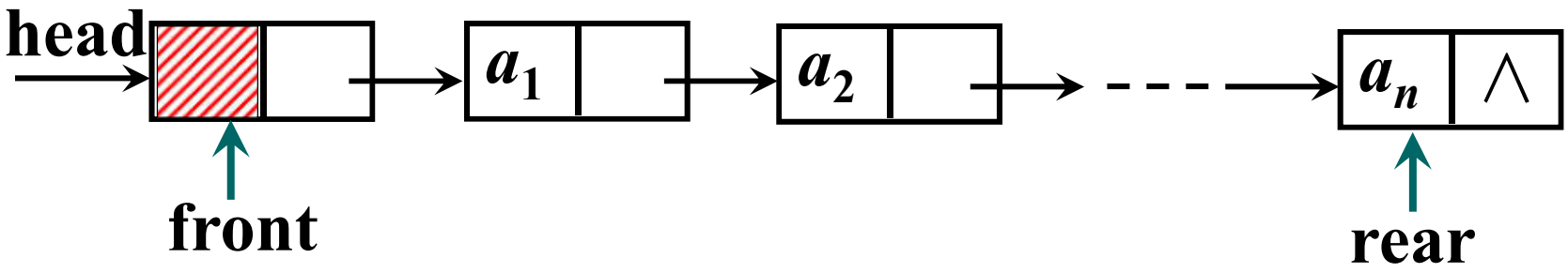
2.4 队列

- 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。
- 队尾：允许插入（也称入队、进队）的一端
- 队首：允许删除（也称出队）的一端称
- 空队列：不含任何数据元素的队列

- 队列操作特性：先进先出 FIFO
- 队列的操作（ADT）：
  - MakeNull(Q)、Front(Q)、Empty(Q)
  - EnQueue(x, Q)、DeQueue(Q)

2.4.1 队列的指针实现

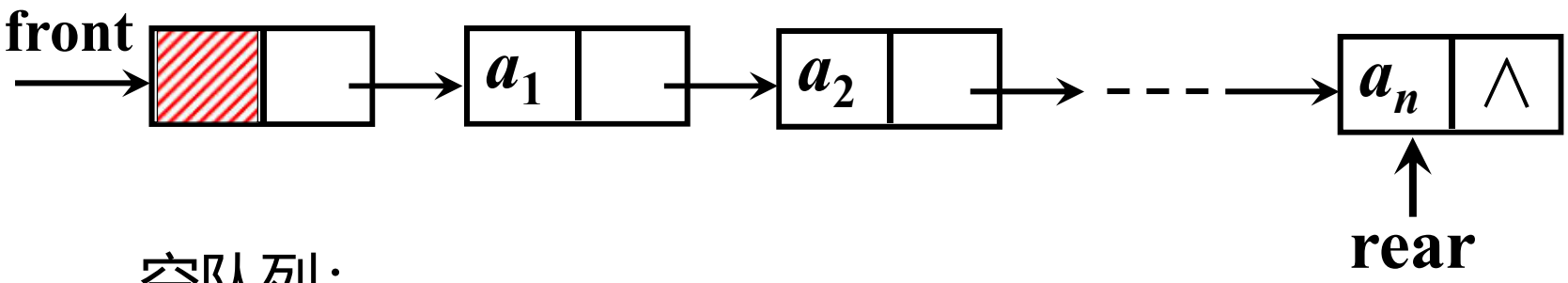
- 队列的链接存储结构及实现
  - 链队列：队列的链接存储结构
  - 如何改造单链表实现队列的链接存储？



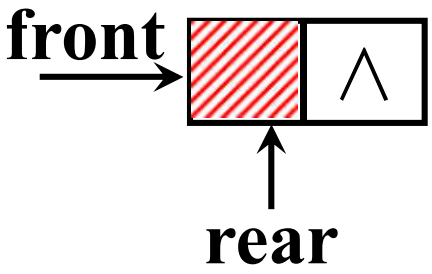
  - 队首指针：即为链表的头结点指针
  - 队尾指针：新增一个指向队尾结点的指针

2.4.1 队列的指针实现

- 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

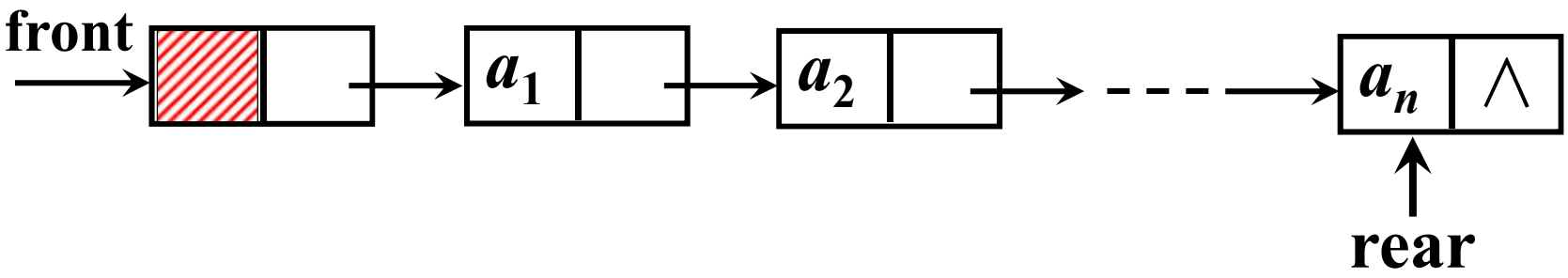
- 队列的链接存储结构及实现
  - 存储结构定义

结点的类型：

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

队列的 类型：

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

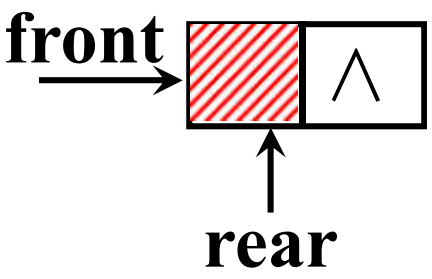


2.4.1 队列的指针实现

- 队列的链接存储结构及实现
  - 操作实现：初始化和判空

① void MakeNull(QUEUE &Q)

```
{
   Q.front = new celltype ;
   Q.front→next = NULL ;
   Q.rear = Q.front ;
}
```

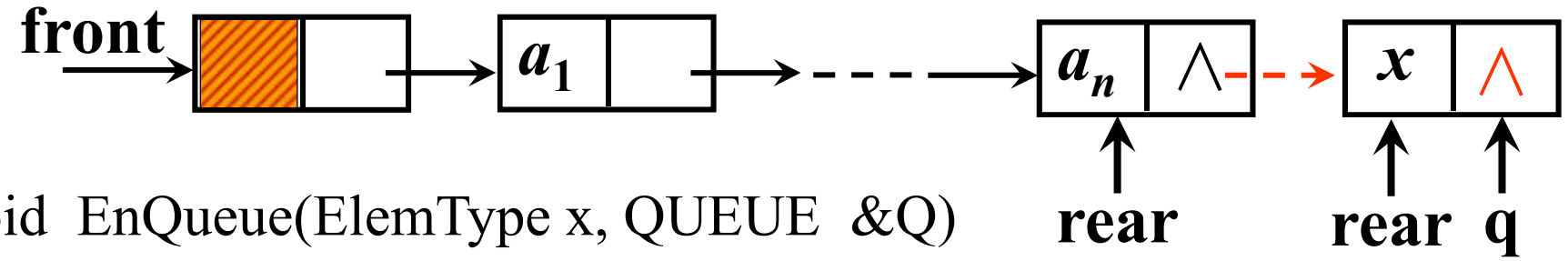


② Boolean Empty(QUEUE &Q)

```
{  if  ( Q.front == Q.rear )
        return  TRUE ;
   else
        return  FALSE ;
}
```

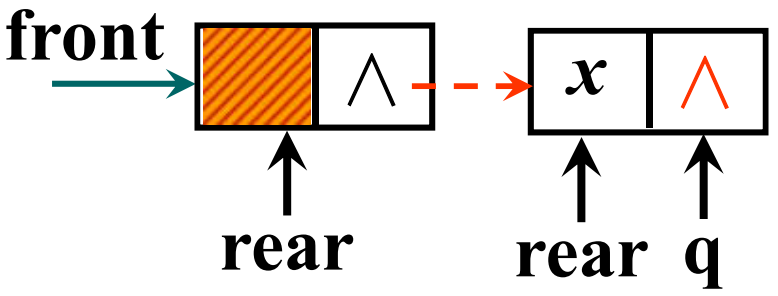
2.4.1 队列的指针实现

- 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)

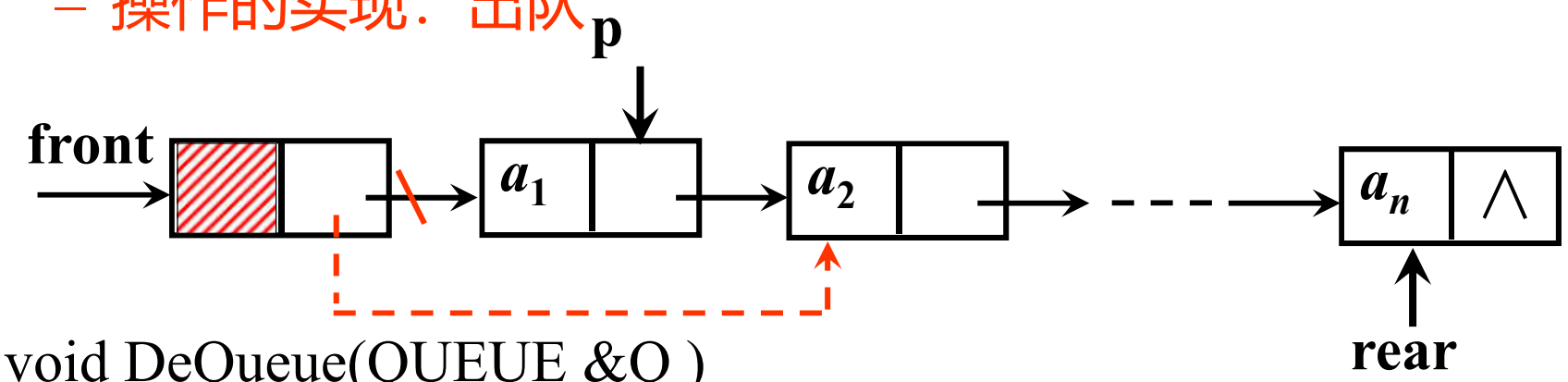
```
{    q=new cwlltype;
     q->data=x ;
     q->next=NULL;
     //q->next=Q.rear->next;
     Q.rear->next=q;
     Q.rear=q;
}
```



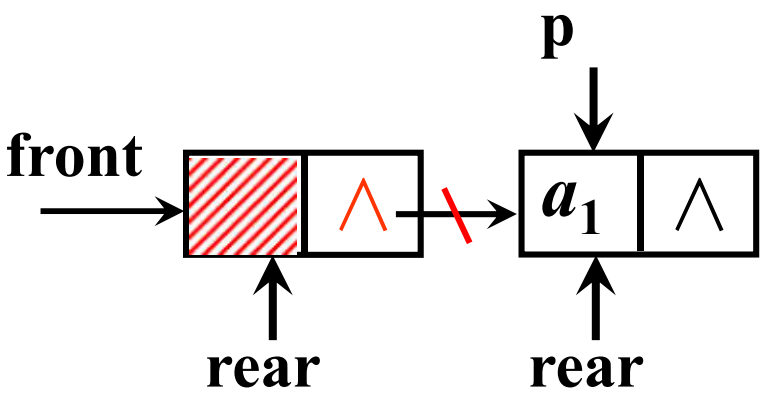
思考：如果没有头结点会怎样？

2.4.1 队列的指针实现

- 队列的链接存储结构及实现
  - 操作的实现：出队



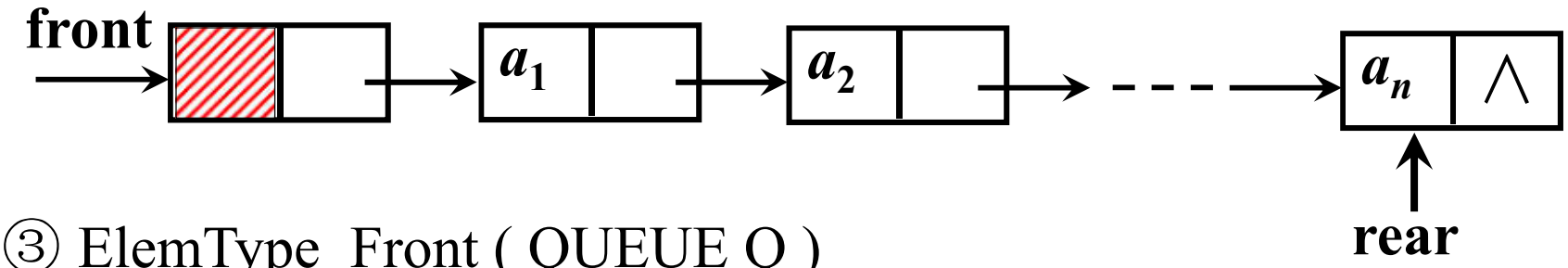

```
void DeQueue(QUEUE &Q )
{    if (Q.rear==Q.front) cout<<“队空";
     p=Q.front->next;
     Q.front->next=p->next;
    if (p->next==NULL) Q.rear=Q.front;
    delete p;
}
```



考虑边界情况：队列中只有一个元素？

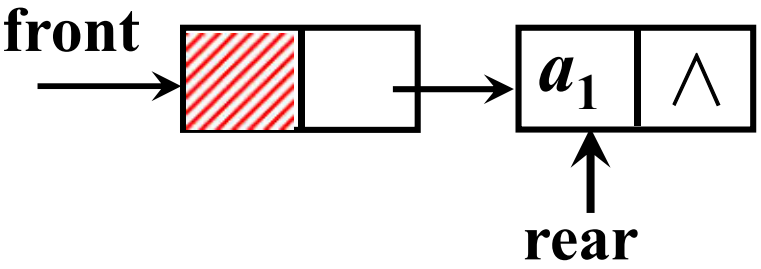
2.4.1 队列的指针实现

- 队列的链接存储结构及实现
  - 操作的实现：返回队首元素



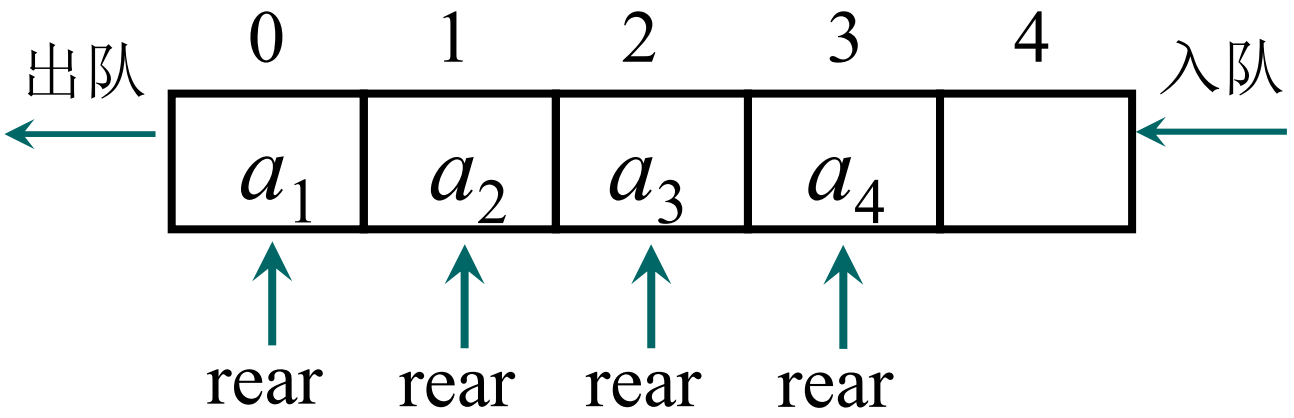
③ ElemType Front (QUEUE Q)

```
{   if ( Q.front→next )
        return  Q.front→next→ data ;
}
```



2.4.2 队列的数组实现

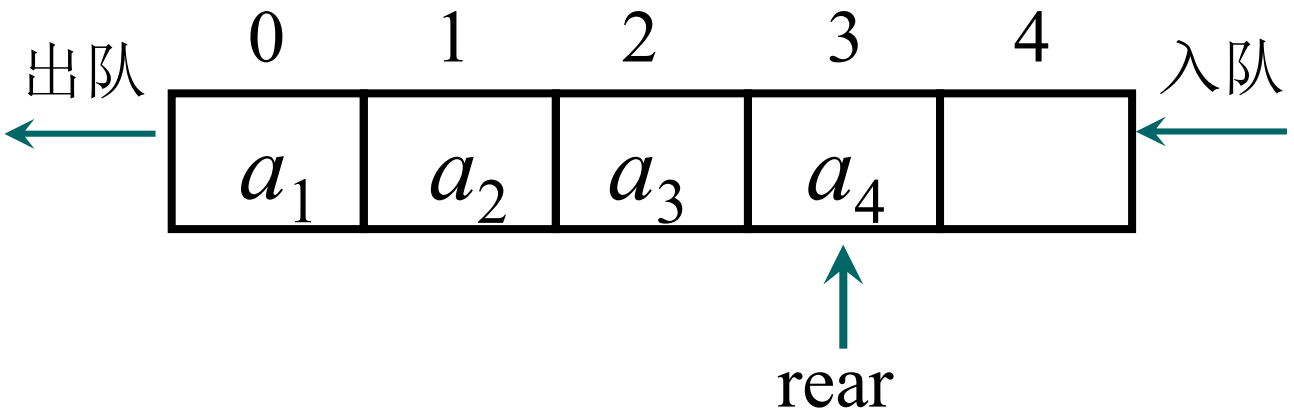
- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储?
    - $a_1a_2a_3a_4$依次入队



入队操作时间性能为O(*1*)

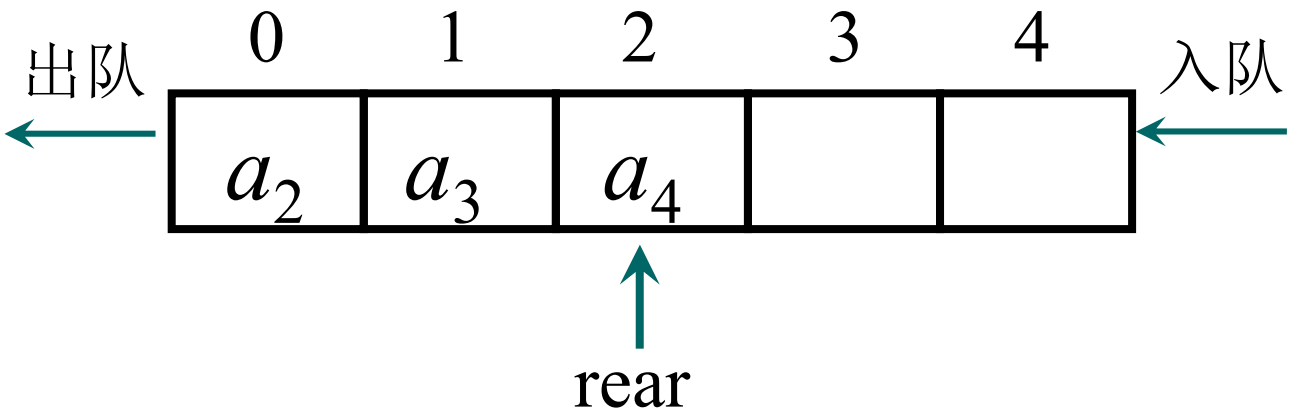
2.4.2 队列的数组实现

- 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_2依次出队



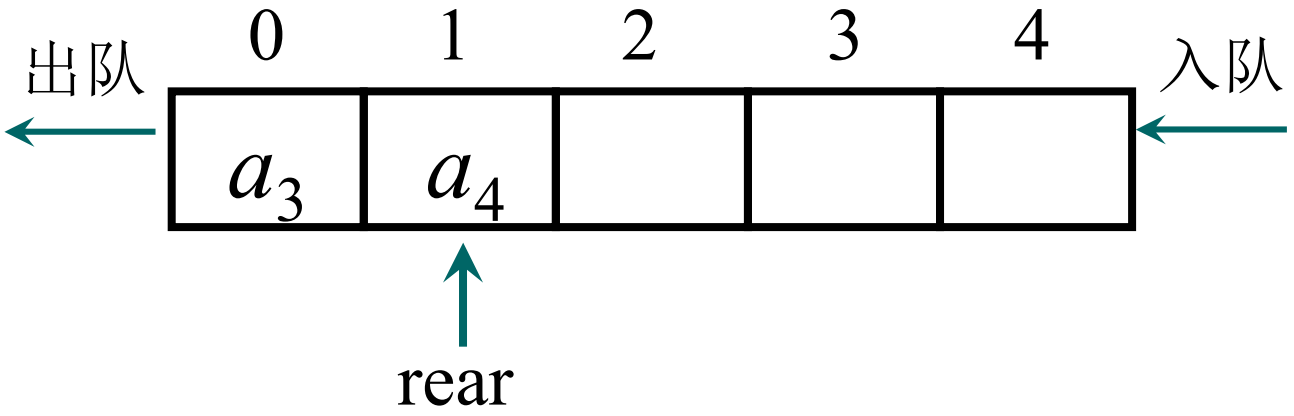
2.4.2 队列的数组实现

- 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_2依次出队



2.4.2 队列的数组实现

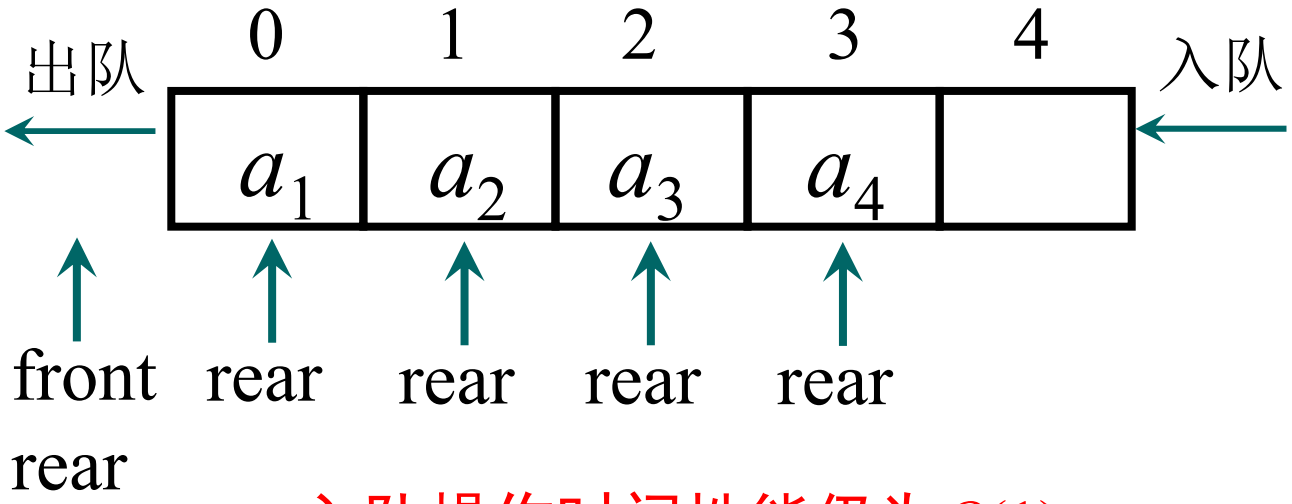
- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储?
    - a_1a_2依次出队



出队操作时间性能为0(*n*)

2.4.2 队列的数组实现

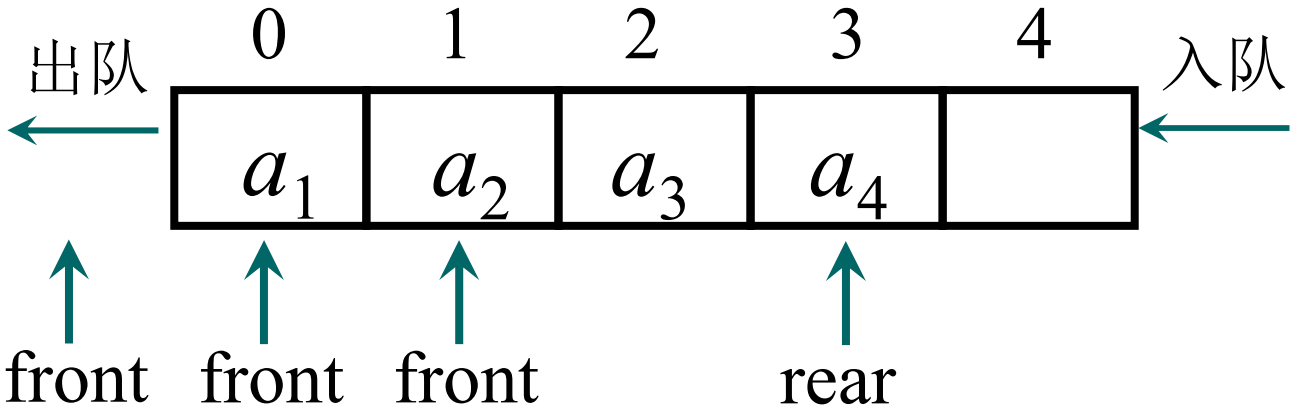
- 队列的顺序存储结构及实现
  - 如何改进出队的时间性能？
    - 所有元素不必存储在数组的前n个单元;
    - 只要求队列的元素存储在数组中连续单元;
    - 设置队头、队尾两个指针
    - 例： $a_1a_2a_3a_4$依次入队



入队操作时间性能仍为$O(1)$

2.4.2 队列的数组实现

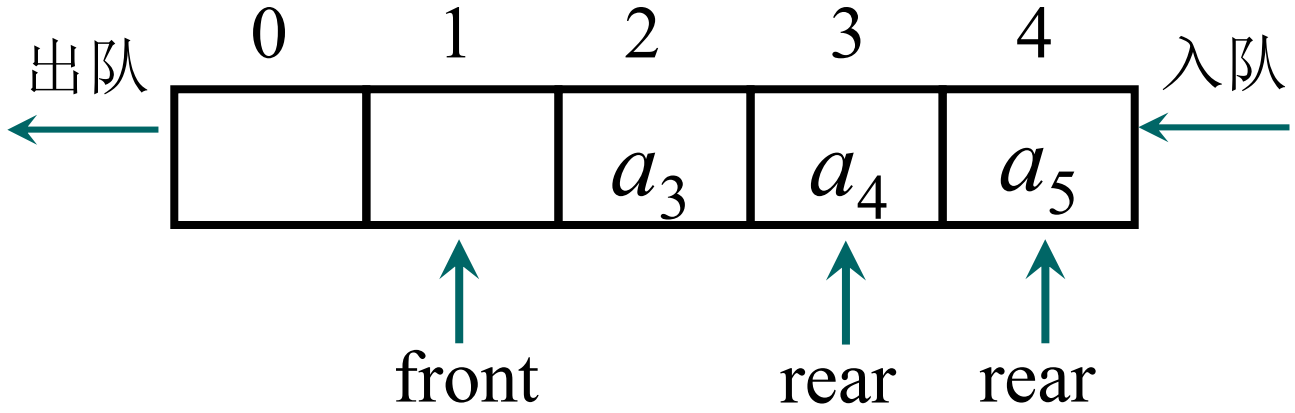
- 队列的顺序存储结构及实现
  - 如何改进出队的时间性能?
    - a_1a_2依次出队



出队操作时间性能提高为$O(1)$

2.4.2 队列的数组实现

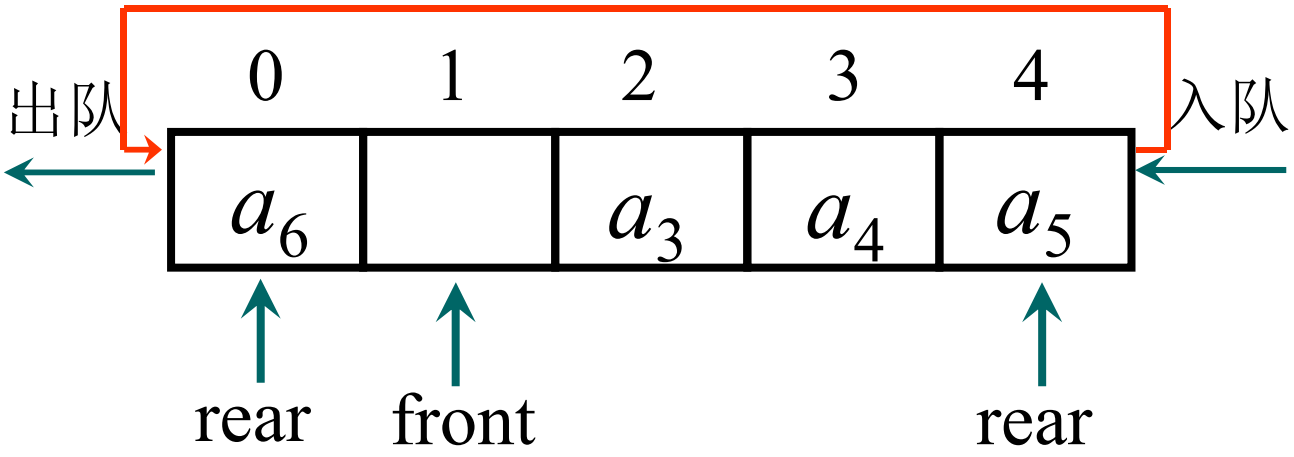
- 队列的顺序存储结构及实现
  - 队列的移动有什么特点？



  - 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。

2.4.2 队列的数组实现

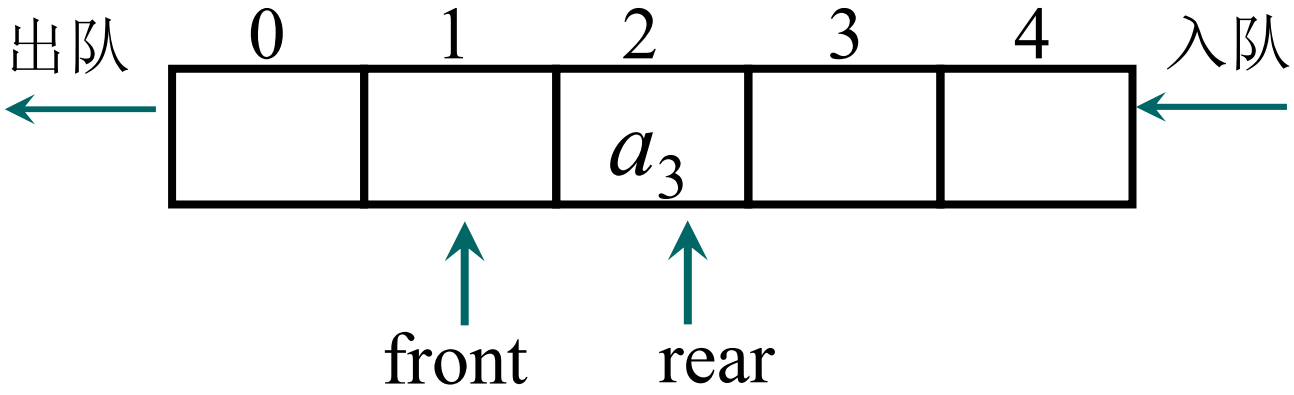
- 队列的顺序存储结构及实现
  - 如何避免假溢出？

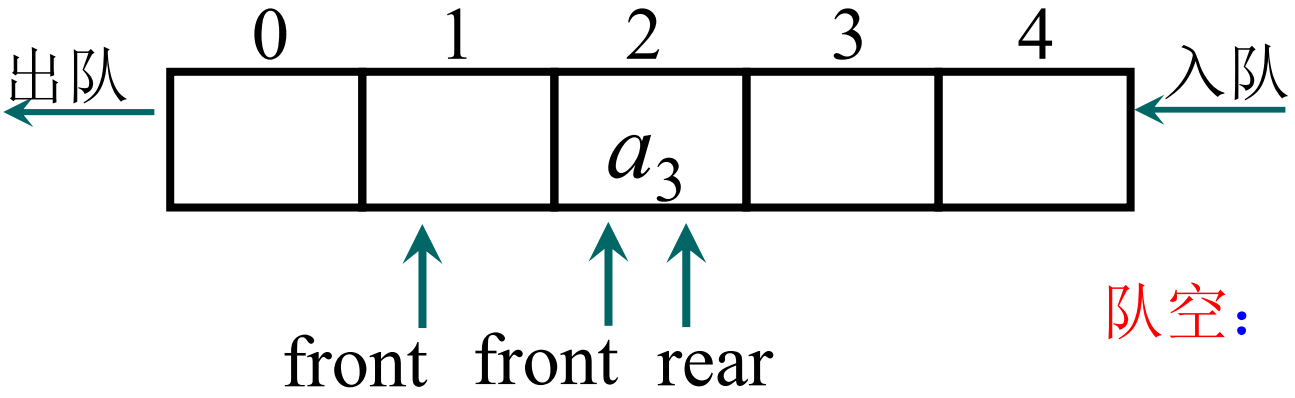


  - 用循环数组表示队列：将数组最后一个单元的下一个单元看成是0号单元，即把数组头尾相接：模运算

2.4.2 队列的数组实现

- 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

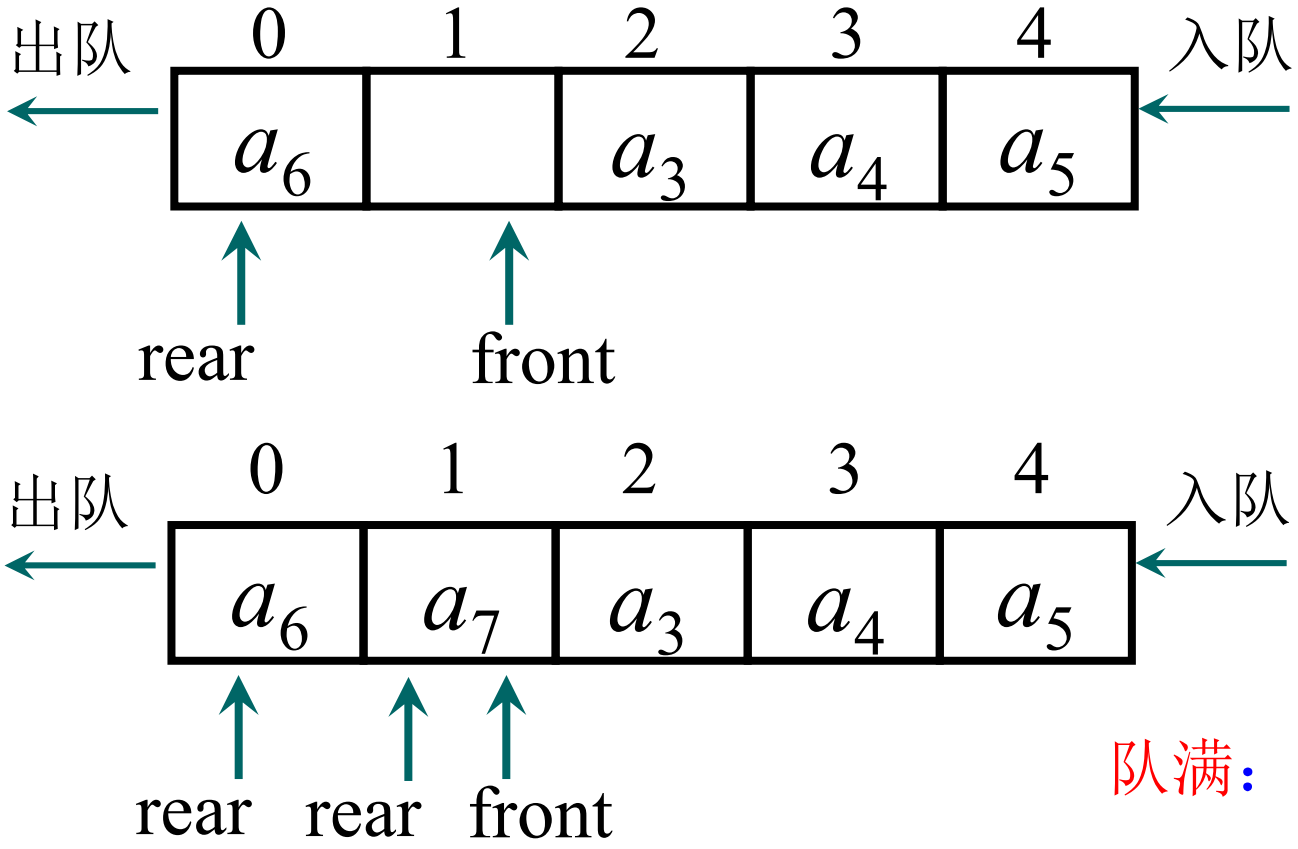




队空：front==rear

2.4.2 队列的数组实现

- 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

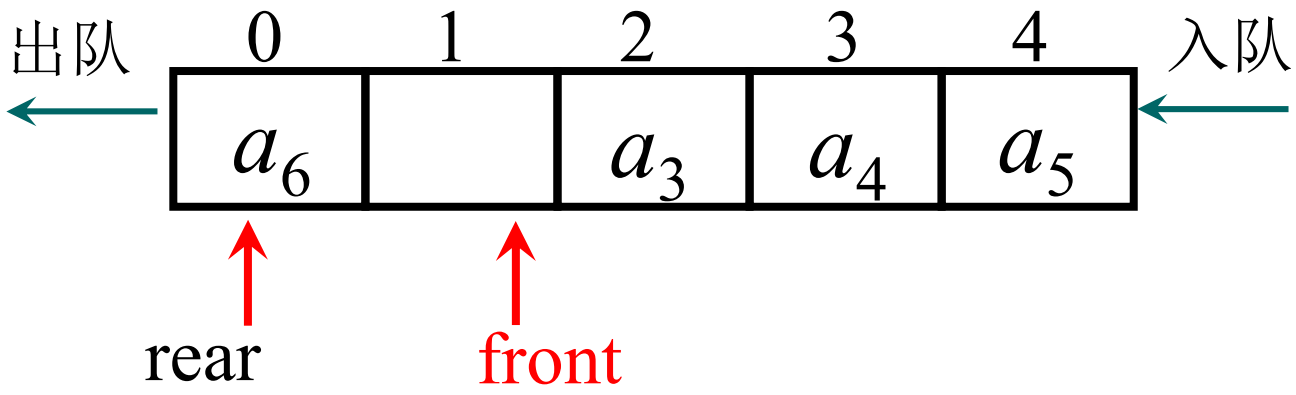
2.4.2 队列的数组实现

- 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front==rear 且count==0时，队空；
      - 当front==rear 且count==MaxSize时，队满；
    - 方法二：设置标志flag，
      - 当front==rear且flag==0时为队空；
      - 当front==rear且flag==1时为队满。
    - 方法三：
      - 保留队空的判定条件：front==rear；
      - 队满时数组中有一个空闲单元
      - 队满判定条件修改为：((rear+1)%MaxSize==front)
      - 代价：浪费一个元素空间。

2.4.2 队列的数组实现

- 队列的顺序存储结构及实现
  - 存储结构的定义

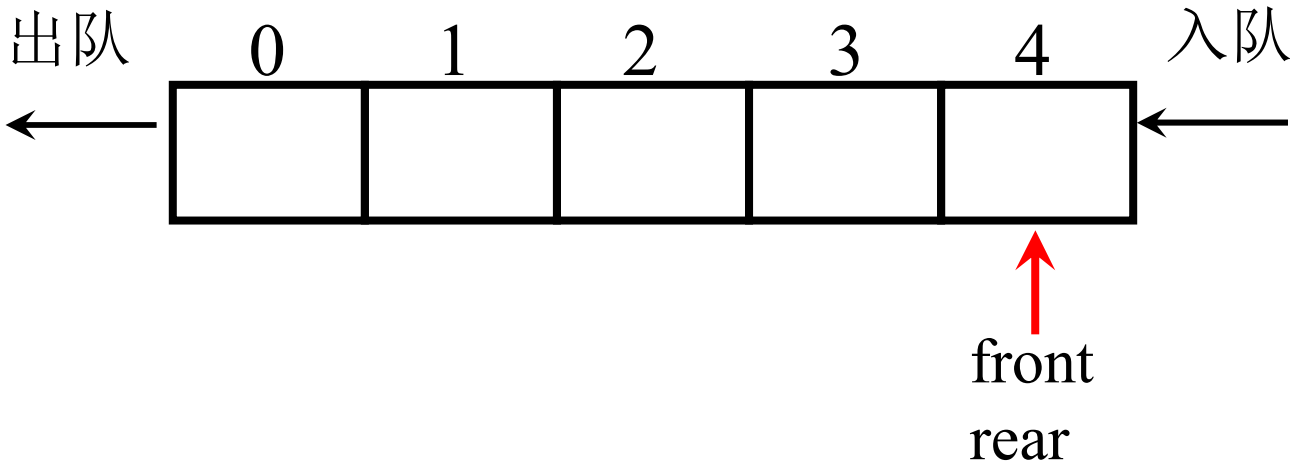
```
struct  QUEUE {
    ElemType  data [ MaxSize ] ;
    int   front ;
    int   rear ;
};   //队列的类型
```



2.4.2 队列的数组实现

- 队列的顺序存储结构及实现
  - 操作的实现： ①队列初始化

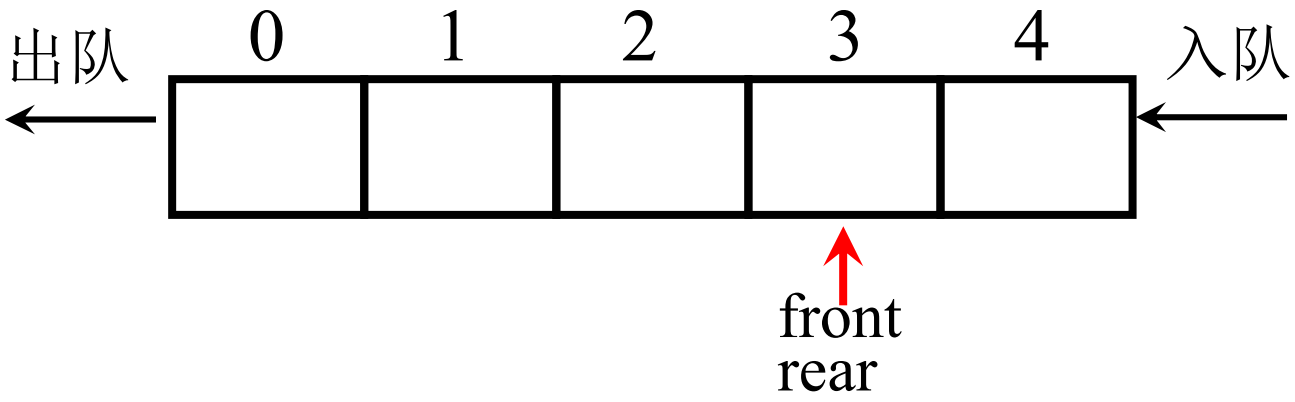
```
void  MakeNull ( QUEUE  &Q)
{
      Q.front = MaxSize-1;
      Q.rear   = MaxSize-1;
}
```



2.4.2 队列的数组实现

- 队列的顺序存储结构及实现
  - 操作的实现：②队列判空

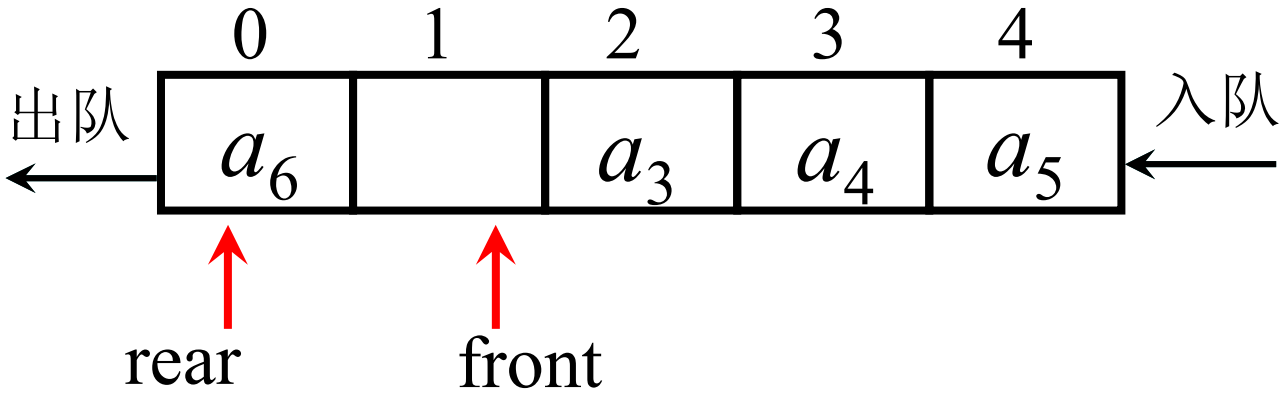
```
bool Empty( QUEUE  Q )
{    if ( Q.rear ==  Q.front  )
            return  TRUE ;
     else
            return  FALSE ;
}
```



2.4.2 队列的数组实现

- 队列的顺序存储结构及实现
  - 操作的实现: ③返回队首元素

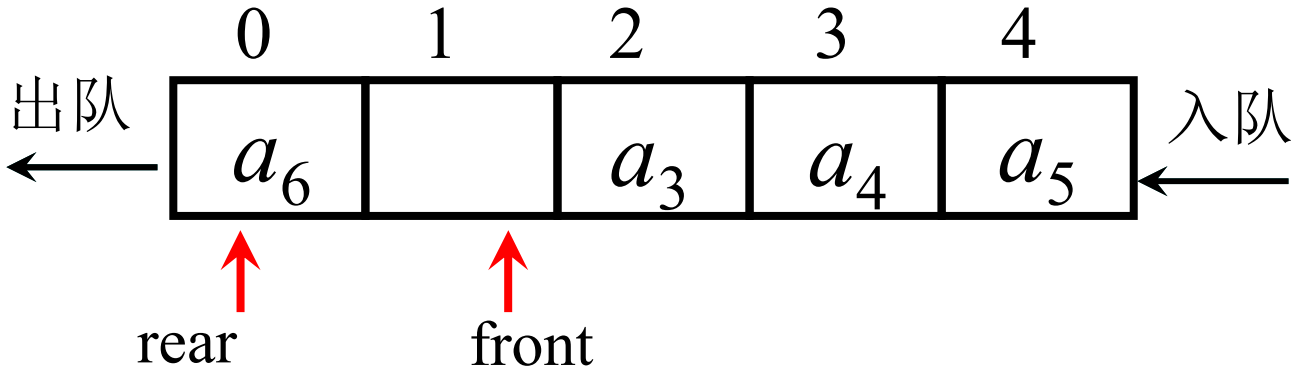
```
ElemType  Front( QUEUE  Q )
{    if  ( Empty( Q ) )    return  NULL;
     else {return  (Q.data[Q.front+1)%MaxSize ] ;
      }
}
```



2.4.2 队列的数组实现

- 队列的顺序存储结构及实现
  - 操作的实现： ④入队

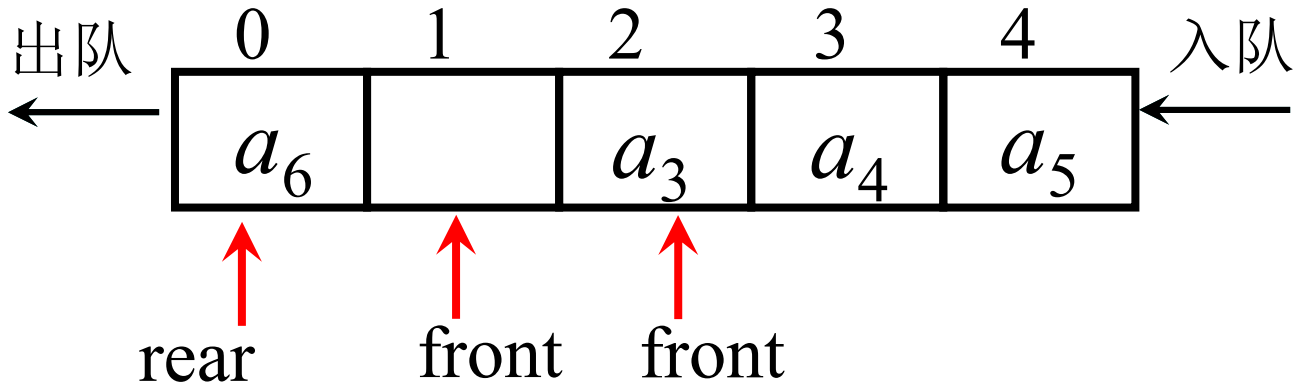
```
void  EnQueue ( ElemType x, QUEUE &Q )
{      if ( (Q.rear+1)%MaxSize ==Q.front )
                  cout<< “队列满”;
       else{      Q.rear=(Q.rear+1)%MaxSize ;
                  Q.data[ Q.rear ] = x ;
}
```



2.4.2 队列的数组实现

- 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 ;
}
```



两种实现方式比较

- 时间复杂度：所有函数两种实现的时间复杂度是常量 O(1)
- 空间复杂度：链式实现空间的额外开销
- 应用：先进先出（FIFO） 特征是应用队列的关键

2.4.3 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

- 实例
  - 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。
  - 排队等候

2.5 特殊线性表: 字符串

2.5.1 串逻辑结构

逻辑结构: 数据对象

- 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=" $s_1\ s_2\ \ldots\ldots\ s_n$ "
  - 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（$1 \le i \le n$）是一个任意字符。
  - 字符集：ASII码、扩展ASII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) ;
  - ……
- 相比其他线性结构，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

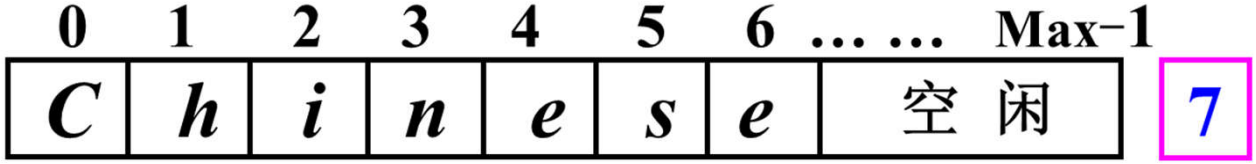
……	*C*	*h*	*i*	*n*	*s*	*e*	……

- 压缩形式

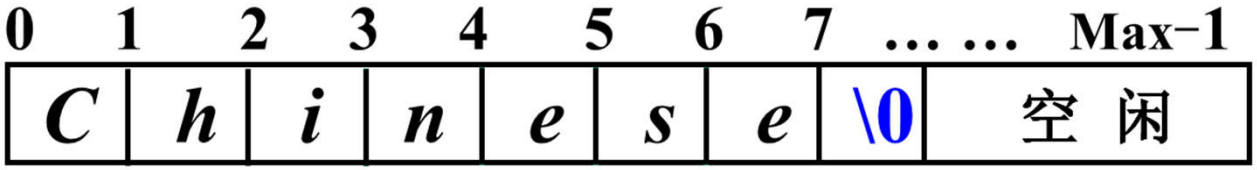
……		*C* *h* *i* *n*	*e* *s* *e*			……

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

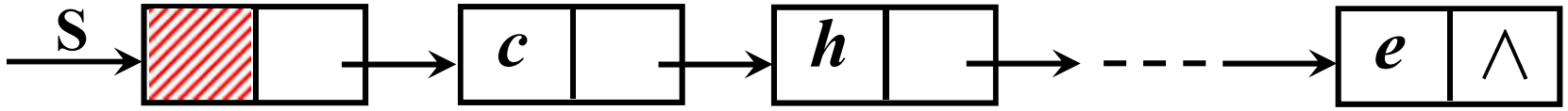


  - 方法二：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。

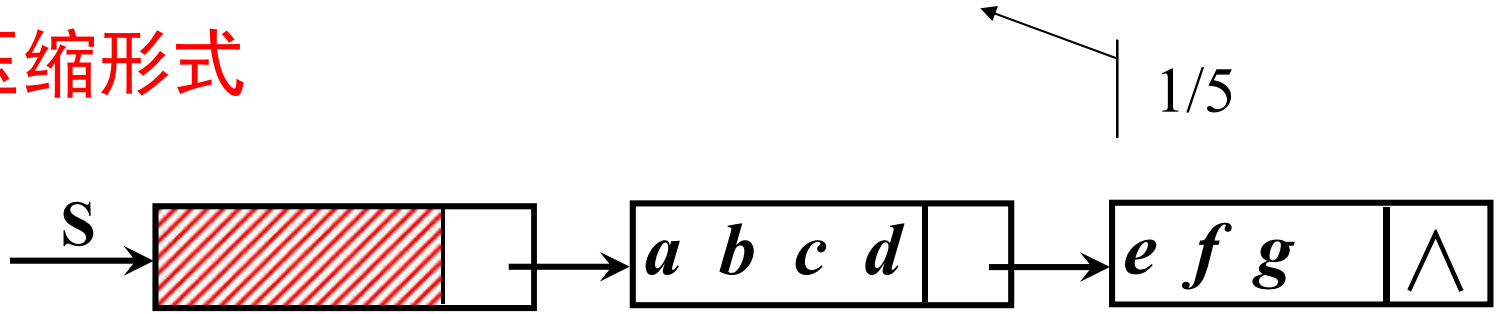


2.5.2 串的存储结构(Cont.)

- 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。
  - 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="$S_1\ S_2\ \ldots\ S_n$"(主串)和T= "$T_1\ T_2 \ldots T_m$"(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(BF)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第2个字符开始和模式T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	ababcabcacbab	i=3	
	模式串	abc	j=3	匹配失败
第2遍匹配	主串	ababcabcacbab	i=2	
	模式串	a	j=1	匹配失败
第3遍匹配	主串	ababcabcacbab	i=7	
	模式串	abcac	j=5	匹配失败
第4遍匹配	主串	ababcabcacbab	i=4	
	模式串	a	j=1	匹配失败
第5遍匹配	主串	ababcabcacbab	i=5	
	模式串	a	j=1	匹配失败
第6遍匹配	主串	ababcabcacbab	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算法主要步骤：

1. 在串S和串T中设比较的起始下标分别为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，长度分别为了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
    }
     if ( j > lenT) // 返回与模式第一字符相同字符在主串的序号
        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~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="aaaaaaaaaabc"，T="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="aaaaaaaaaaab"，T="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× (n-m +1)
- BF算法时间复杂度为O(m ×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，产生大量回溯，没有利用已经产生的部分匹配结果。导致大量重复工作。
  - 如何在匹配不成功时，主串不回溯？
    - 主串可不回溯，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，然后继续进行匹配

2.5.3 模式匹配(Cont.)

KMP算法匹配过程示例:

设主串ababcabcacbab，模式abcac

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：无回溯模式匹配问题
  - 假定主串为 $S_1S_2\ldots S_n$，模式串为 $T_1T_2\ldots T_m$
  - 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？
- 分析：
  - 假设主串中的S_i与模式中的T_k ($k<j$)继续比较，则

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

    可能有多个k，取哪一个？

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

  - 根据已有的匹配，有

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

    $T_1T_2\ldots T_{k-1} = T_{j-(k-1)}T_{j-(k-2)}\ldots T_{j-1}$ //与主串无关

KMP 失败函数

示例：给定一个模式字符串P：abaaba

- $F(j)$数组$P[0..j]$ 前缀与数组$P[1..j]$后缀最大公共串长度

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$P[1..j]$		b	ba	baa	baab	baaba
$P[0..j]$	a	ab	aba	abaa	abaab	abaaba
$F(j)$ 重叠数	0	0	1	1	2	3
Next(j) 下次比对下标 参考F (j-1)		0	0	1	1	2

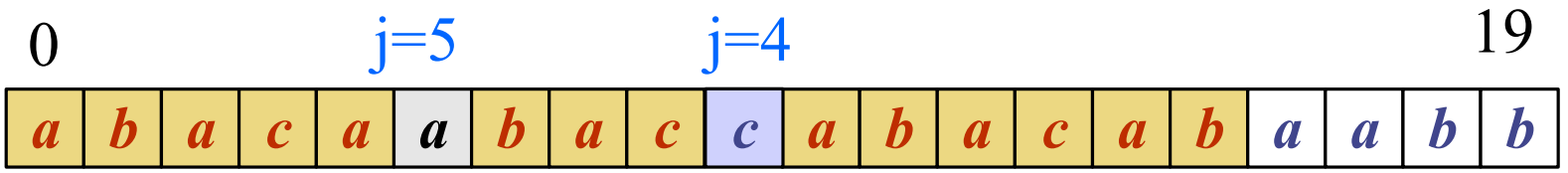
- P数组第一个元素下标为0;
- 若$P[j]$失配，下一步从模式索引 $j \leftarrow F(j-1)$位与主串失配位置对齐比较

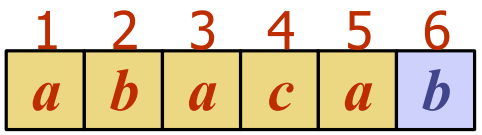
```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1    //P第二个字符下标
    j ← 0    //P第一个字符下标
    while i < m
        if P[i] ==P[j]
            // 已经匹配 j + 1 个字符
            F[i] ← j + 1
            i ← i + 1;
            j ← j + 1;
        else if j > 0 then
            //use failure function to shift P
            j ← F[j - 1];
        else
            F[i] ← 0;   // no match
            i ← i + 1;
```

时间复杂度$O(m)$

2.5.3 模式匹配(Cont.)

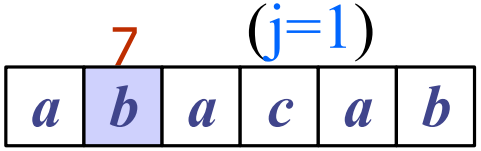
利用失败函数匹配过程示例:



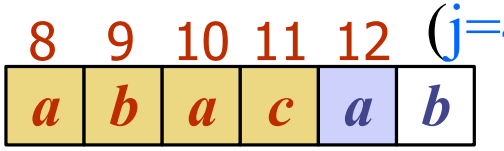


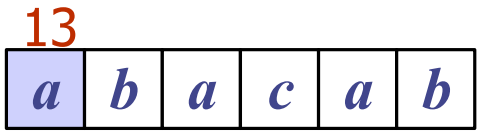
(j=5)

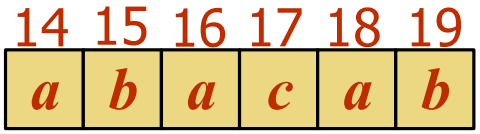
(j=1)



(j=4)







j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
P[1..*j*]		b	ba	bac	baca	bacab
P[0..*j*]	a	ab	aba	abac	abaca	abacab
F(*j*)	0	0	1	0	1	2
F(*j*-1)	0	0	0	1	0	1

下次比对字符(index=*F(j-1)*)，参考重叠数量 *F(j-1)*

示例: 计算串p的失败函数:

p	a	b	a	b	a	c	a

Initially: m = length[p] = 7

F[1] = 0

Step 1: q = 2

F [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0					

Step 2: q = 3

F[3] = 1

p[2..j]： ba

p[1..j]： aba

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1				

Step 3: q = 4

F [4] = 2

p[2..j]： bab

p[1..j]： abba

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2			

计算***P***[1..***j***]前缀与 ***P***[2..***j***]后缀最大公共串(重叠字符数）

Step 4: q = 5

F [5] = 3

p[2..j]： baba

p[1..j]： ababa

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5: q = 6

F [6] = 0

p[2..j]： babac

p[1..j]： ababac

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Step 6: q = 7

F [7] = 1

p[2..j]： babaca

p[1..j]： ababaca

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

计算***P***[1..***j***]前缀与 ***P***[2..***j***]后缀最大公共串长度(重叠字符数）

2.5.3 模式匹配(Cont.)

KMP算法匹配过程示例:

设主串ababcabcacbab，模式abcac

j	1	2	3	4	5
模式串T	a	b	c	a	c
T[2..j]		b	bc	bca	bcac
T[1..j]	a	ab	abc	abca	abcac
F(j) 重叠数量	0	0	0	1	0
k=next[j] 下次比对位置 参考F(j-1)	1	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]；
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算法实现

```
int Index_KMP(char* S, char* T)
{ /*S为主串T为模式，长度分别为了lenS和lenT；串采用顺序存储结构*/
    int i = 1 , j = 1;                          // 从串第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）
```

2.6 (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$d(d\geq 1)$个线性关系的约束，每个元素在n个线性关系中的序号$i_1, i_2, \ldots, i_d$称为该元素的下标，并称该数组为 d 维数组。
- 数组特点:
  - 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。

2.6 (多维)数组(Cont.)

示例:

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

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

其中:

$A_i=(a_{1i}, a_{2i}, \ldots\ldots, a_{mi})$

$(1\le i\le n)$

- 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.)

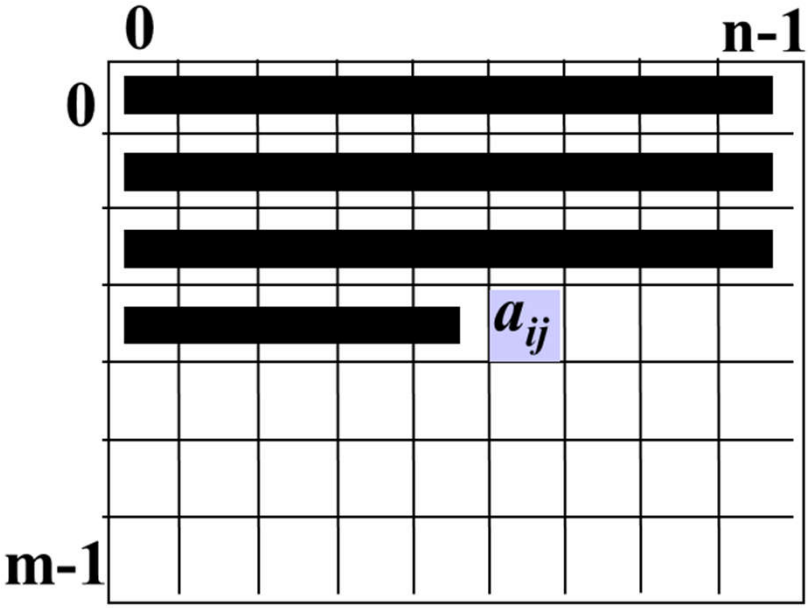
- 数组的基本操作
  - 初始化：Create（ ）
    - 建立一个空数组；
    - int A[ ][ ]
  - 存取： Retrieve(array, index)
    - 给定一组下标，读出对应的数组元素；
    - A[i][j]
  - 修改： Store(array，index，value)：
    - 给定一组下标，存储或修改与其相对应的数组元素。
    - 例如：A[i][j]=8
  - 无需插入和删除操作

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



a_{ij}前面的元素个数k
=整行数×每行元素个数+本行中
a_{ij}前面的元素个数$= i \times n + j$

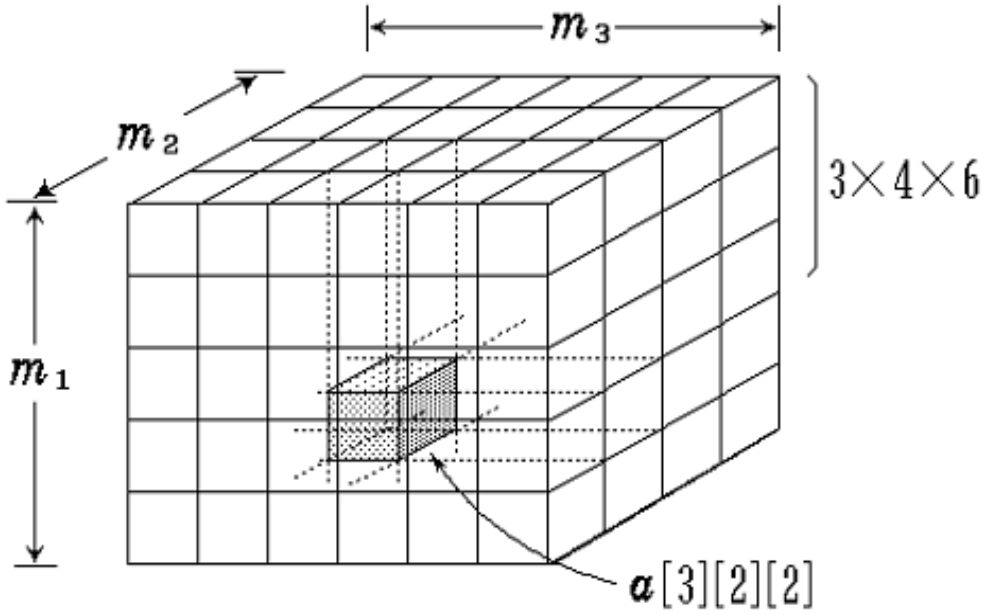
$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{00}) + (i \times n + j) \times c$$

Sa	a_{00}	a_{01}	a_{02}			a_{ij}			$a_{n-1,n-1}$
k=	0	1	2	…		i×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。



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

更高维的数组呢？

2.6 (多维)数组(Cont.)

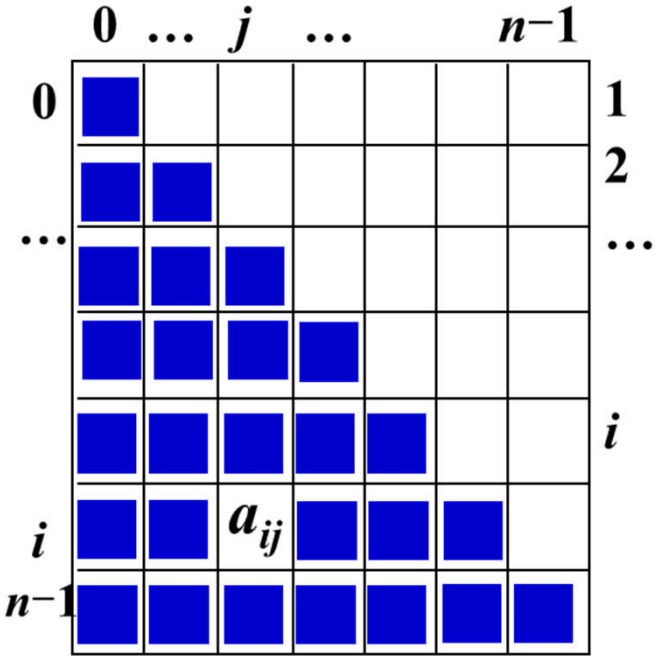
- 特殊矩阵的压缩存储
  - 特殊矩阵：矩阵中很多值相同的元素并且它们的分布有一定的规律。
    - 如对称矩阵、上/下三角矩阵、带状(对角)矩阵等
  - 稀疏矩阵：矩阵中有很多特定值的（如零）元素。
    - 分布没有规律
    - 在m*n的矩阵中，有t个元素不为零。令α=t/m*n，称α为矩阵的稀疏因子。
    - 通常认为α<=0.05时称为稀疏矩阵
- 压缩存储的基本思想是：
  - 为多个值相同的元素只分配一个存储空间；
  - 对特定值（如零）的元素不分配存储空间。

2.6 (多维)数组(Cont.)

三角矩阵的压缩存储：下\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：

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

$k= n\times(n+1)/2$ 存常数c

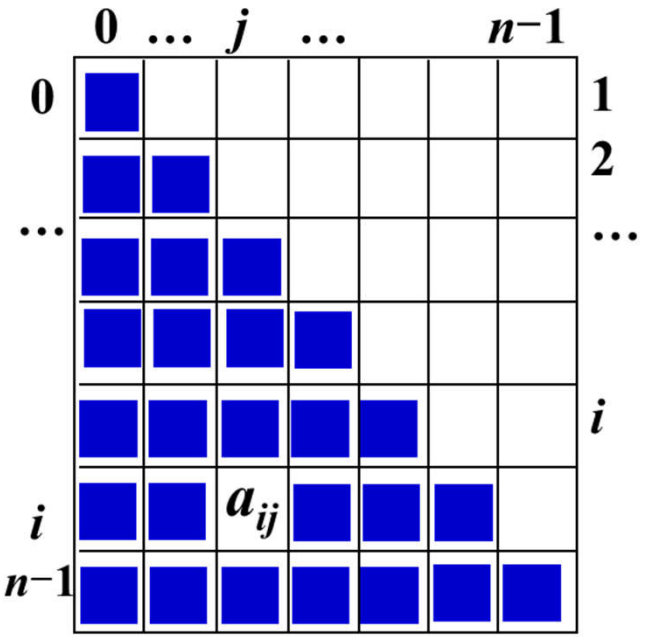
Sa	$a_{0\,0}$	$a_{1\,0}$	$a_{1\,1}$	$a_{2\,0}$	…	$a_{i,j}$	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$	c
k=	0	1	2	3		i(i+1)/2+j		n(n-1)/2		n(n+1)/2-1	*

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

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

∵一维数组下标从0开始

∴a_{ij}在一维数组中的下标

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

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

Sa	$a_{0\,0}$	$a_{1\,0}$	$a_{1\,1}$	$a_{2\,0}$	…	$a_{i,j}$	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$
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

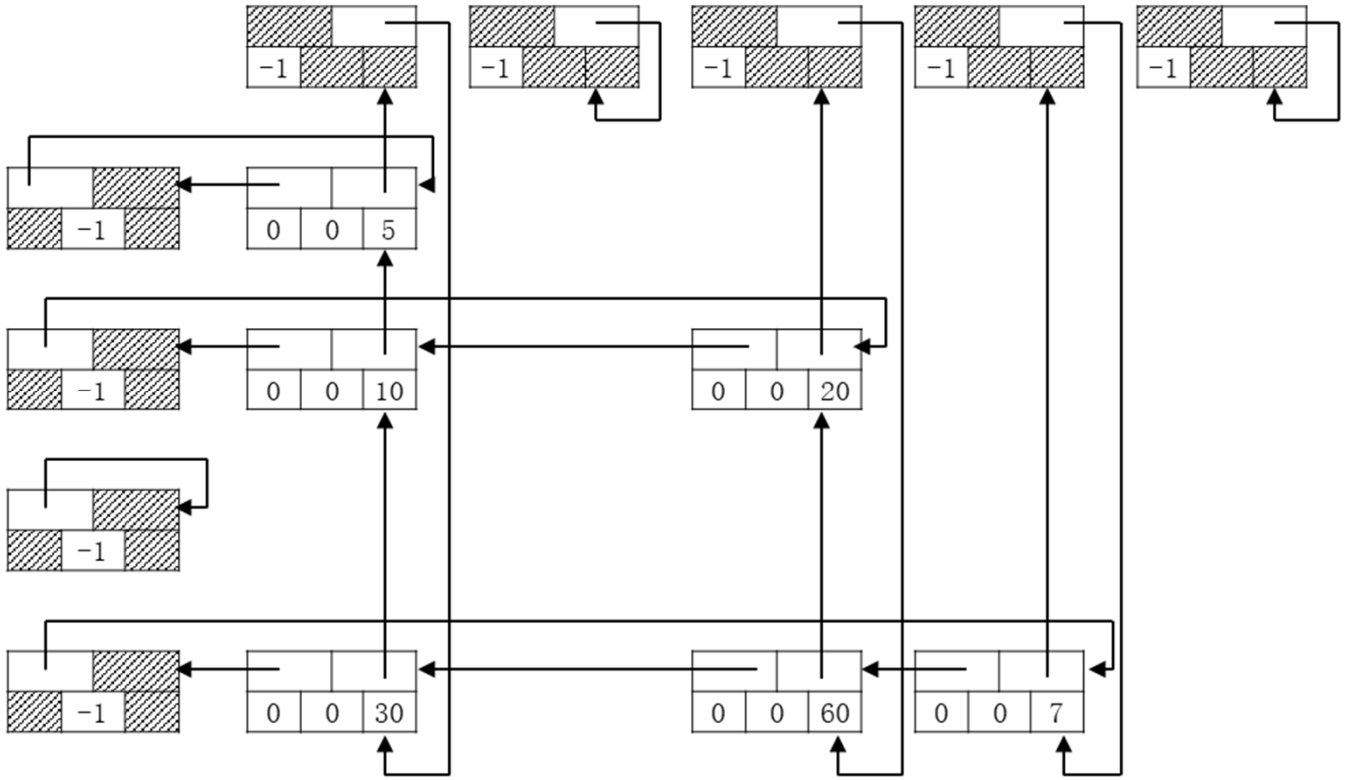
  - left：指针域，指向同一行中的前一个三元组
  - up：指针域，指向同一列中的上一个三元组
  - row：存储非零元素的行号
  - col：存储非零元素的列号
  - val：存储非零元素的值

2.6 (多维)数组(Cont.)

- 稀疏矩阵的压缩存储：十字链表

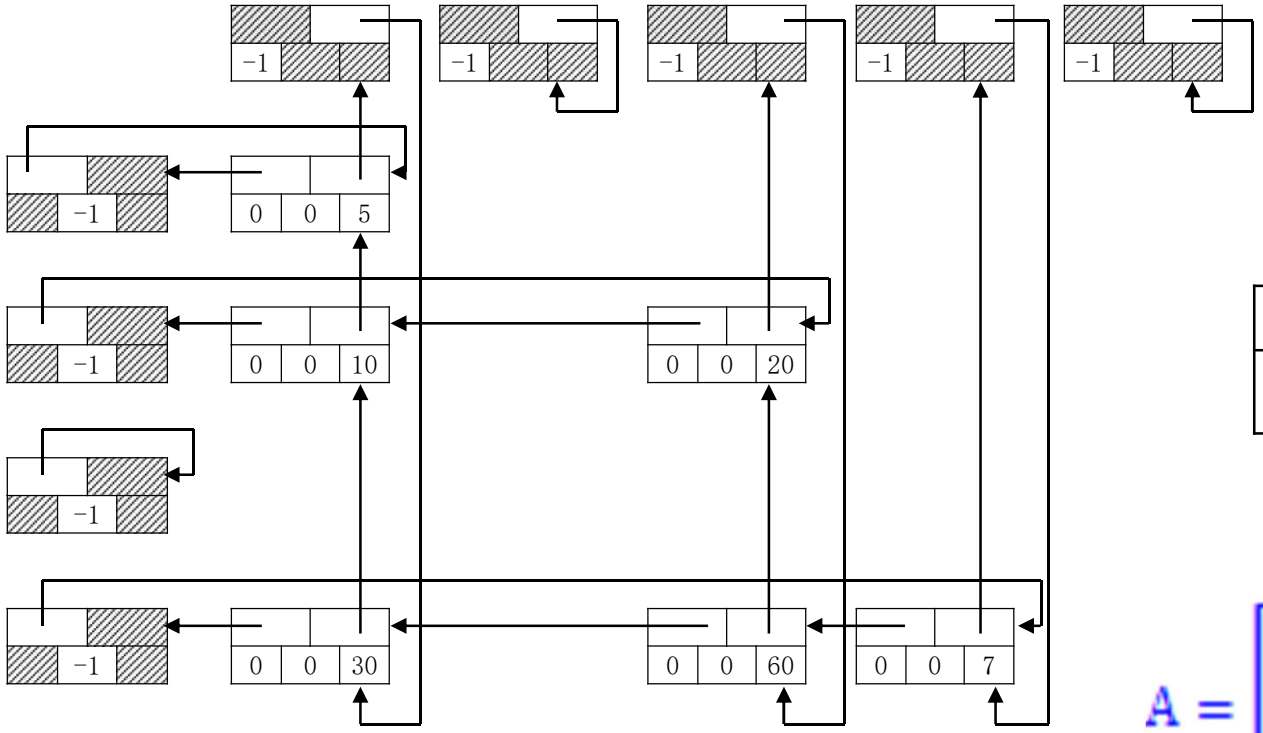
left		up
row	col	val

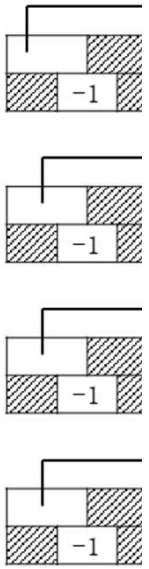
$$A=\begin{bmatrix}5 & 0 & 0 & 0 & 0\\ 10 & 0 & 20 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 30 & 0 & 60 & 7 & 0\end{bmatrix}_{4\times5}$$



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

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 30 & 0 & 60 & 7 & 0 \end{bmatrix}_{4\times5}$$

2.7 广义表

- 广义表：n（$n\geq 0$）个数据元素的有限序列，记作：

$$LS=(a_1，a_2，\ldots\ldots，a_n)$$

其中：LS是广义表的名称，a_i（$1\leq i\leq n$）可以是单个数据元素，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：

①Cal（ 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

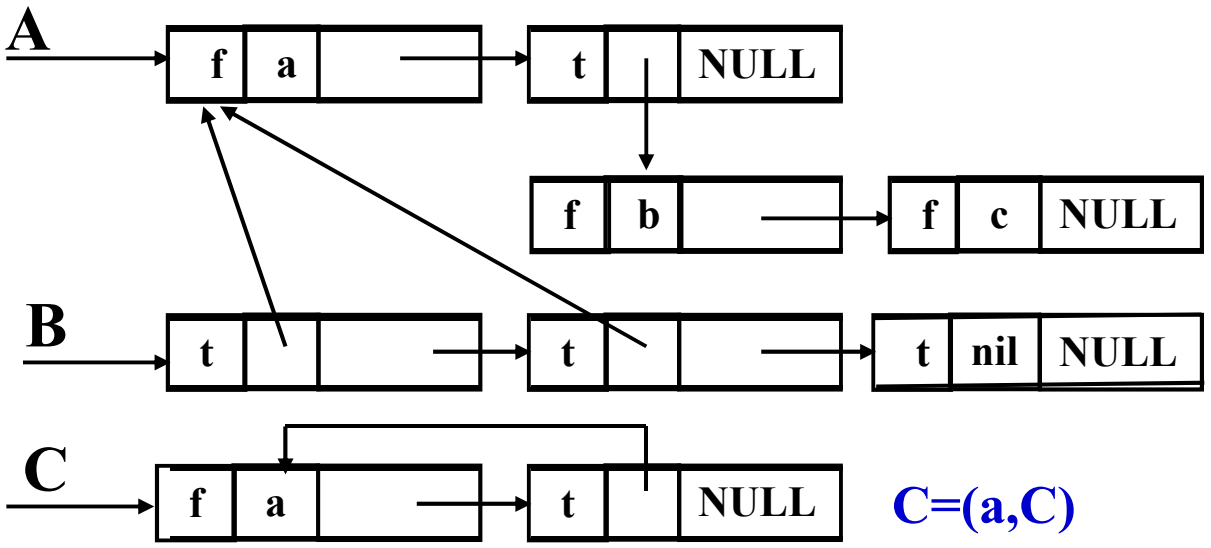
⑤Length（ L ）：求广义表 L 的长度

2.7 广义表(Cont.)

- 广义表存储结构

A=(a,(b,c))

B=(A,A,( ))

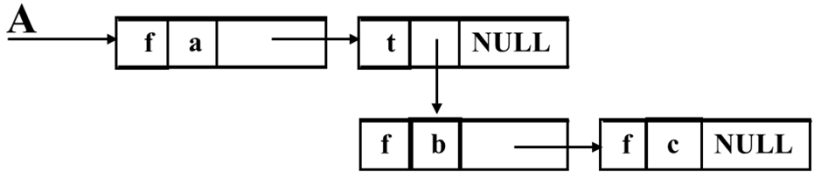



```
struct  listnode {
      listnode  *link ;
      boolean  tag ;
      union  {
            char  data ;
            listnode  *dlink ;
      } ;
 } ;
typedef   listnode  *listpointer ;
```

2.7 广义表(Cont.)

广义表操作的实现

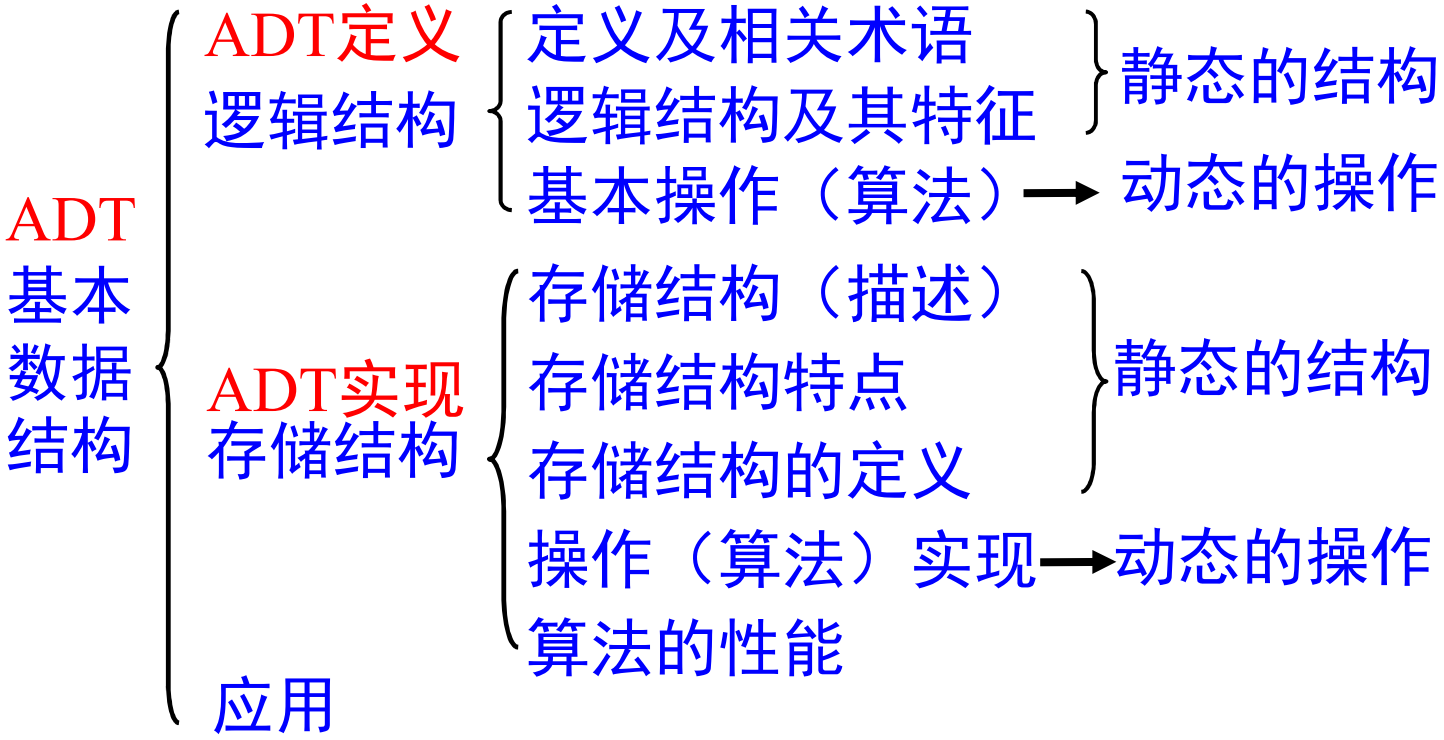
A=(a,(b,c))




```
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 ;       //data域相等
                   else
                      x = FALSE ;
               else
                    x = Equal( S→element.data,T→element.data ); // tag==TRUE
               if ( x== TRUE )
                    y = Equal( S→link, T→link ) ;      //判断link域是否相等
             }
     return y ;
}  //S和T均为非递归的广义表
```

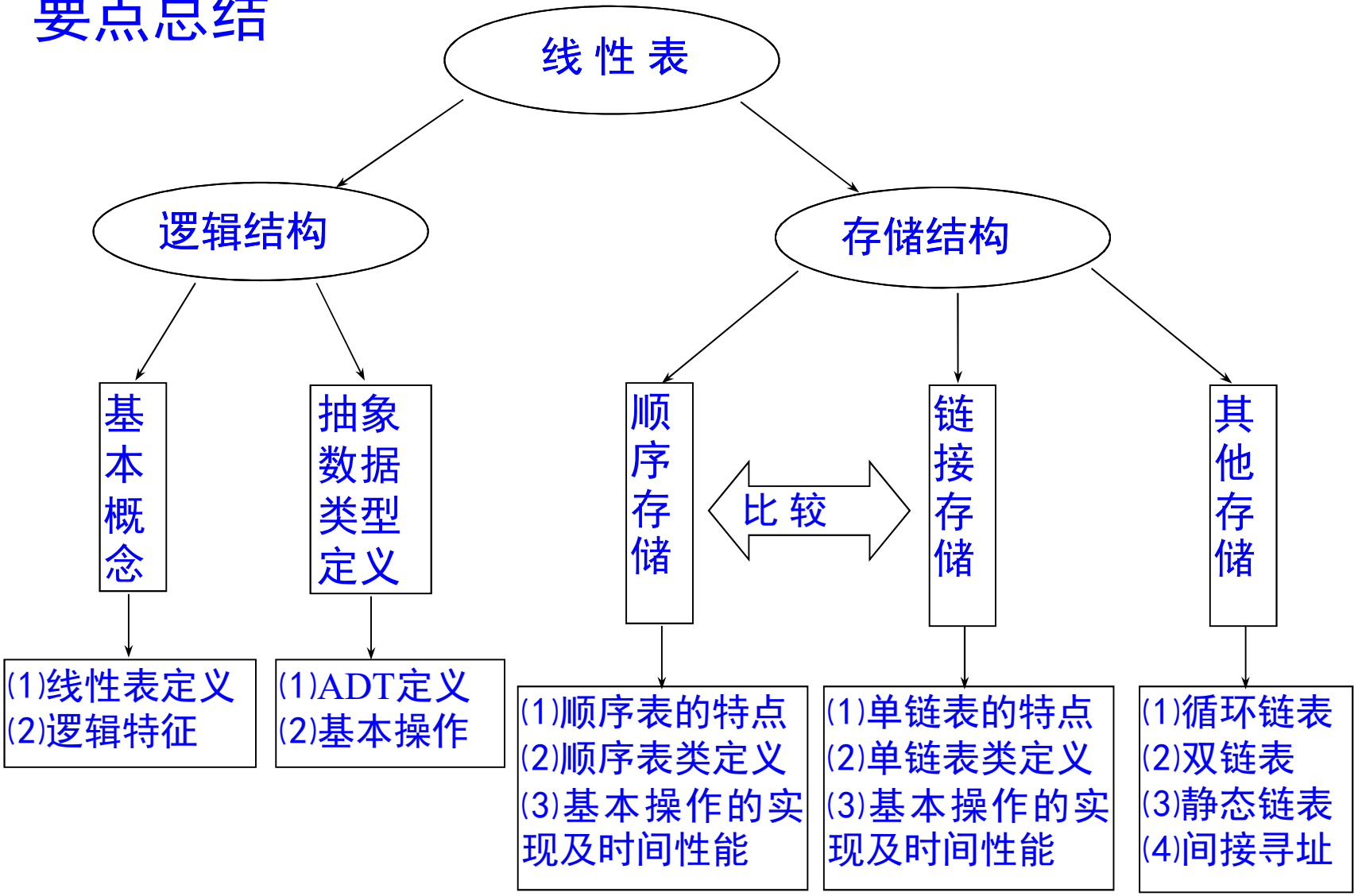
本章小结

- 知识点：
  - 线性表、栈、队列、串、（多维）数组、广义表
- 知识点体系结构



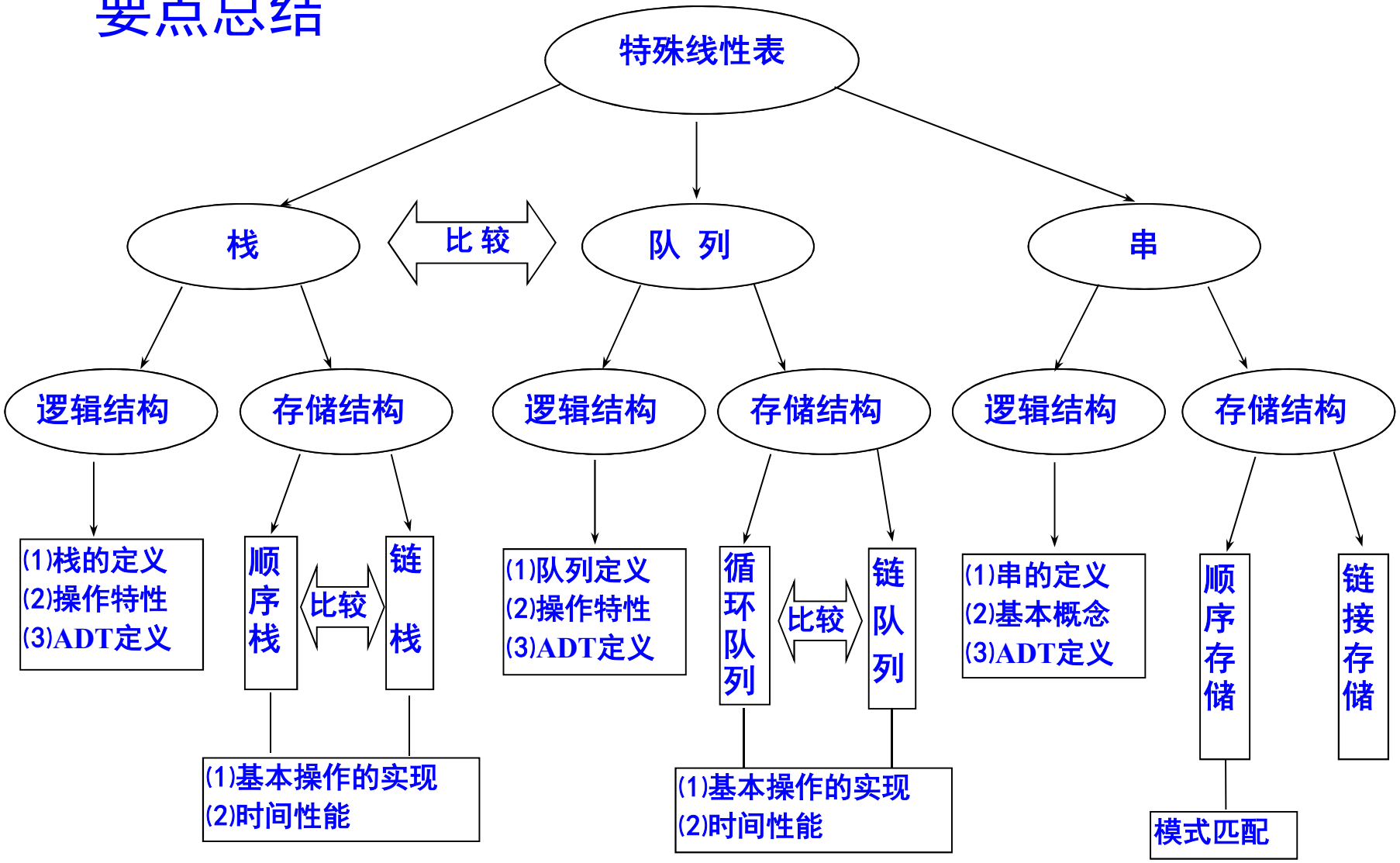
本章小结

要点总结


线性表
逻辑结构
存储结构
基本概念
抽象数据类型定义
顺序存储
比较
链接存储
其他存储
(1)线性表定义
(2)逻辑特征
(1)ADT定义
(2)基本操作
(1)顺序表的特点
(2)顺序表类定义
(3)基本操作的实现及时间性能
(1)单链表的特点
(2)单链表类定义
(3)基本操作的实现及时间性能
(1)循环链表
(2)双链表
(3)静态链表
(4)间接寻址

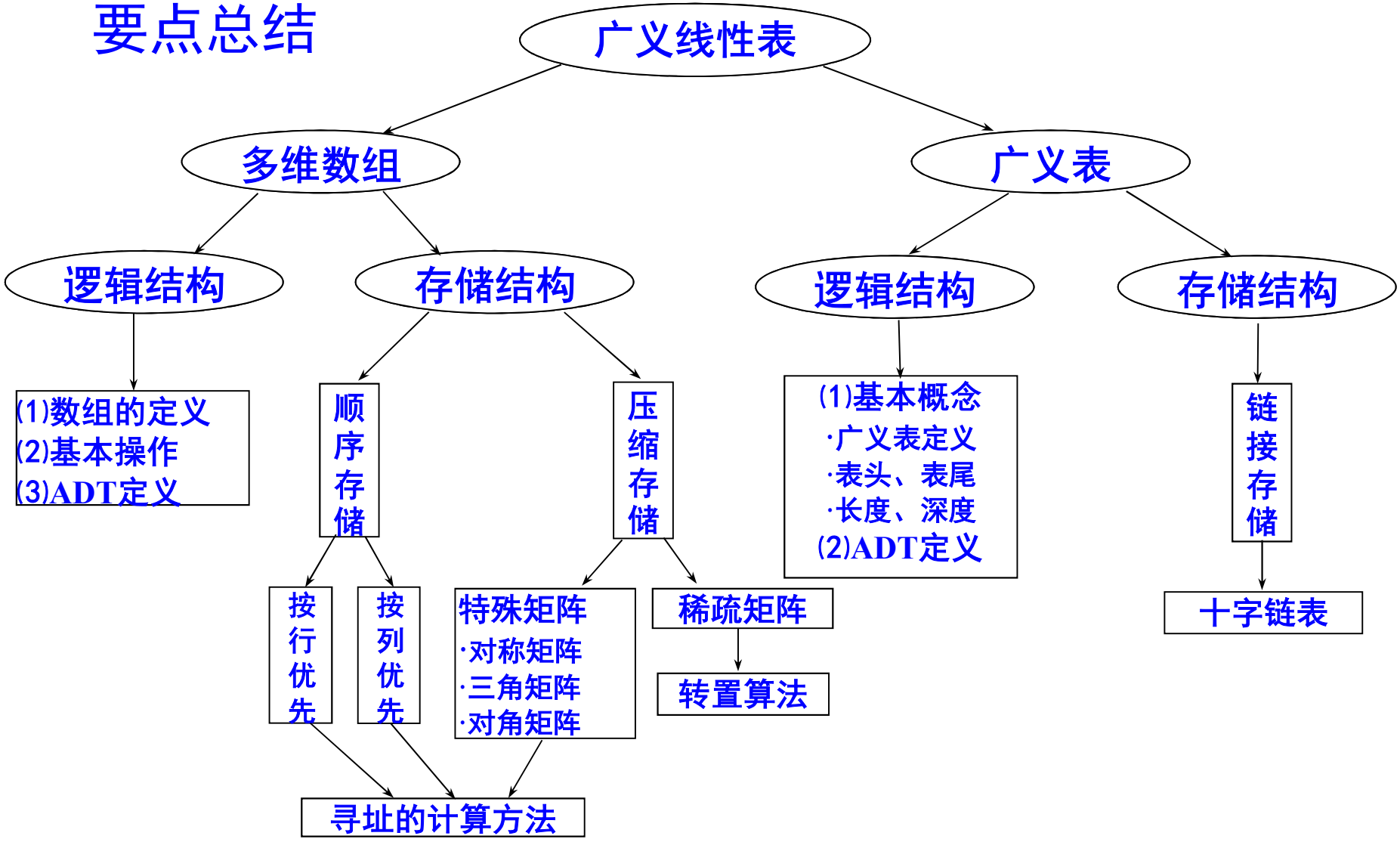
本章小结

要点总结


特殊线性表
栈
比 较
队 列
串
逻辑结构
存储结构
逻辑结构
存储结构
逻辑结构
存储结构
(1)栈的定义
(2)操作特性
(3)ADT定义
顺序栈
比较
链栈
(1)基本操作的实现
(2)时间性能
(1)队列定义
(2)操作特性
(3)ADT定义
循环队列
比较
链队列
(1)基本操作的实现
(2)时间性能
(1)串的定义
(2)基本概念
(3)ADT定义
顺序存储
链接存储
模式匹配

本章小结

要点总结


广义线性表
多维数组
广义表
逻辑结构
存储结构
逻辑结构
存储结构
(1)数组的定义
(2)基本操作
(3)ADT定义
顺序存储
压缩存储
(1)基本概念
·广义表定义
·表头、表尾
·长度、深度
(2)ADT定义
链接存储
按行优先
按列优先
特殊矩阵
·对称矩阵
·三角矩阵
·对角矩阵
稀疏矩阵
十字链表
转置算法
寻址的计算方法