

```
#include<iostream.h>
#include<string.h>
void main()
{ char flower[100],*r,*b,*w, temp;
int count;
cin>>flower;
count=strlen(flower);
r=flower;
b=w=&flower[count-1];
while(r-w<0)
{ switch (*w)
{ case 'r' : {temp=*r; *r=*w; *w=temp; r++; break;}
  case 'w' : {w--;break;}
  case 'b' : { temp=*w; *w=*b; *b=temp; w--; b--;break;}
}
cout<<flower;}
```

第2章 线性表（Liner List）

2.1 线性表的抽象数据型

2.2 线性表的实现

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）

知识点：

- 线性表的逻辑结构和各种存储表示方法
- 定义在逻辑结构上的各种基本运算（操作）
- 在各种存储结构上如何实现这些基本运算
- 各种基本运算的时间复杂性
- 特殊线性表

重点：

- 熟练掌握顺序表和单链表上实现的各种算法及相关的时间复杂性分析

难点：

- 使用本章所学的基本知识设计有效算法解决与线性表相关的应用问题

2.1 线性表的抽象数据型

[定义] 线性表是由 $n(n \geq 0)$ 个相同类型的元素组成的序列集合。

记为：

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

其中：① 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（）。

② a_i 为线性表中的元素，类型定义为elementtype

③ a_1 为表中第1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。（位置概念！）

④ 线性表是有限的，也是有序的。

数学模型

线性表 $\text{LIST} = (\text{D}, \text{R})$

$\text{D} = \{ a_i \mid a_i \in \text{Elementset}, i = 1, 2, \dots, n, n \geq 0 \}$

$\text{R} = \{ H \}$

$H = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 2, \dots, n \}$

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

- ① **Insert(x, p, L)**
- ② **Locate(x, L)**
- ③ **Retrieve(p, L)**
- ④ **Delete(p, L)**
- ⑤ **Previous(p, L), Next(p, L)**
- ⑥ **Makenull(L)**
- ⑦ **First(L)**
- ⑧ **End(L)**

例：设计函数 **Deleval** (LIST &L, elementtype d) , 其功能为删除 L 中所有值为 d 的元素。

```
void Deleval( LIST &L, elementtype 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 线性表的实现

问题：确定数据结构（存储结构）实现型LIST，并在此基础上 实现各个基本操作。

存储结构的三种方式：

- ① 连续的存储空间（数组） → 静态存储
- ② 非连续存储空间——指针（链表） → 动态存储
- ③ 游标（连续存储空间+动态管理思想） → 静态链表

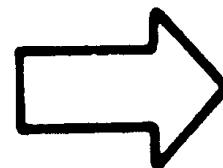
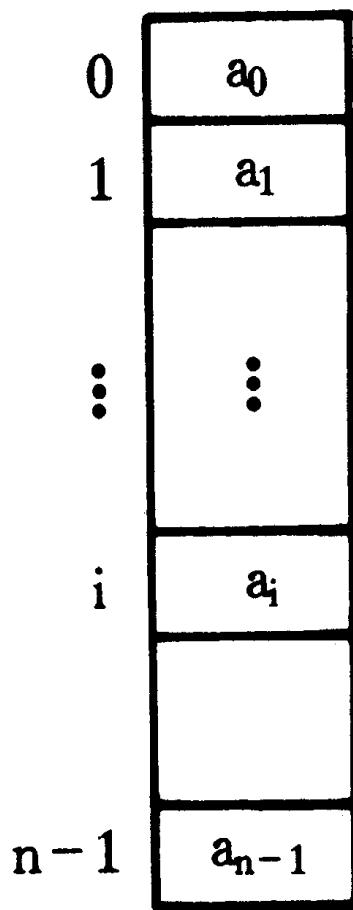
顺序表

是指用一组地址连续的存储单元依次存储线性表的数据元素。

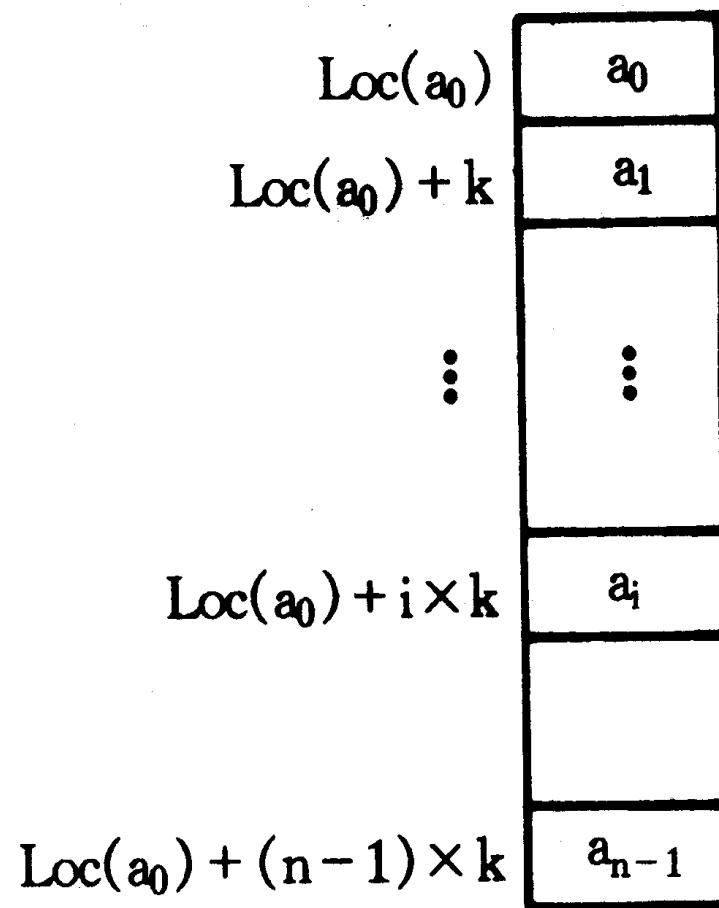
特点

- 元素之间的逻辑关系（相继/相邻关系）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）；
- 是一种随机存储结构，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。由于高级语言中数组类型具有这种特性，通常都用数组描述。

逻辑地址 数据元素



存储地址 数据元素



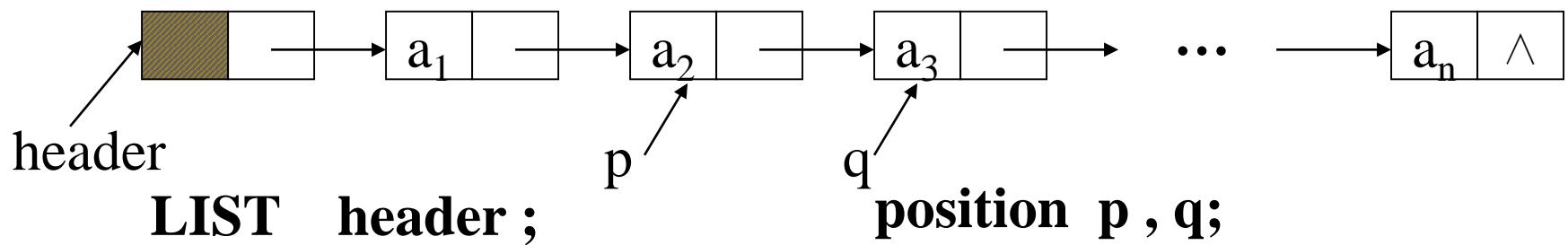
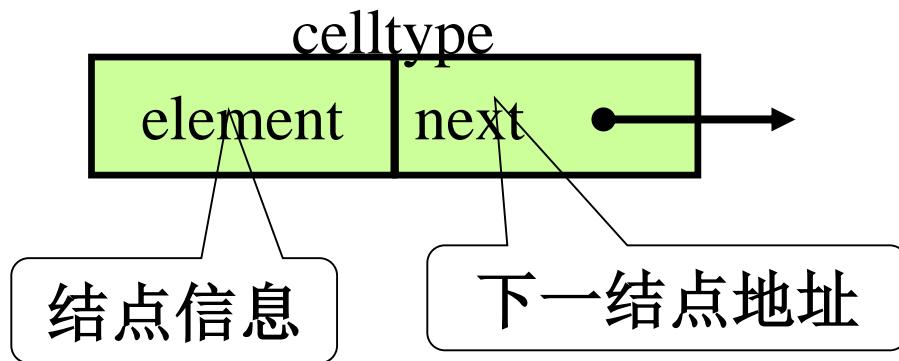
2.2.2 线性表的指针实现

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，简称单向链表或单向线性链表。

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；
- 非随机存储结构

结点形式



类型定义：

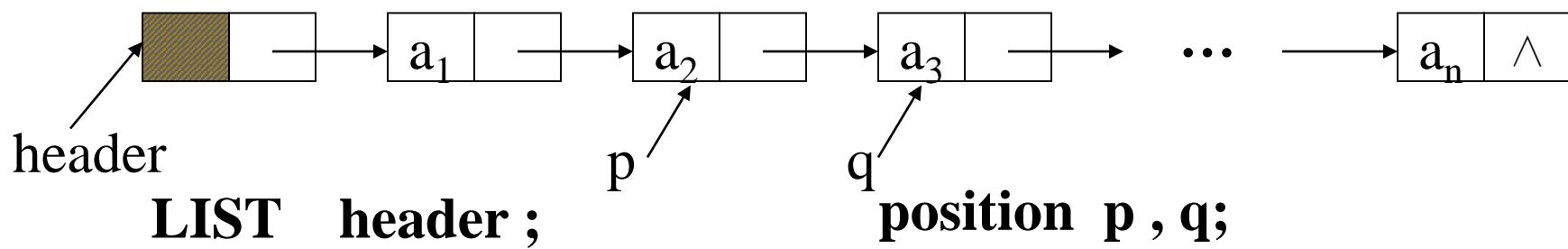
```
struct celltype {  
    elementtype element ;  
    celltype *next ;  
} ; //结点型
```

线性表的型

```
typedef celltype *LIST;
```

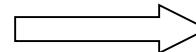
位置型

```
typedef celltype *position;
```



记法：

a₂: (*p).element ;
q: (*p).next ;



a₂: p→element ;
q: p→next ;

- 指针的使用使得不同区域的代码可以轻易的共享内存数据。
- 指针使得一些复杂的链接性的数据结构的构建成为可能，比如链表，链式二叉树等等
- 有些操作必须使用指针。如操作申请的堆内存。
- C语言中的一切函数调用中，值传递都是“按值传递”的。如果要在函数中修改被传递过来的对象，就必须通过这个对象的指针来完成。
- 计算机访问某个数据的时候，首先要通过地址总线传送数据存储或者读取的位置，然后在通过数据总线传送需要存储或者读取的数据。一般地，int整型的位数等于数据总线的宽度，指针的位数等于地址总线的宽度。

1. 设字符集为字符和数字的集合，字符的顺序为A, B, C, …, Z, 0, 1, 2, …, 9, 请将下列字符串按字典顺序排列、存储：PAB, 5C, PABC, CXY, CRSI, 7, B899, B9, 并分析可以采取的存储方案。

本题可以采用顺序、链式及索引等方式存储

1. 采用二维数组存储**String[8][5]**, 优点：紧凑，简单；缺点：空间浪费。

B	8	9	9	\0
B	9	\0		
C	R	S	I	\0
C	X	Y	\0	
P	A	B	\0	
P	A	B	C	\0
5	C	\0		
7	\0			

2. 采用链式存储。优点：增删快，空间可以不连续；缺点：代码复杂，排序从头开始，费时。

3. 采用索引存储，大数组存数据，字符指针数组存每个字符串索引。
优点：排序时交换地址，数据本身不动，高效；
缺点：代码不会写

例：遍历线性链表，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。

```
void Travel( LIST L )
{ position p ;
  p = L->next ;
  while ( p != NULL)
    { cout << p->element ;
      p = p->next ;
    }
}
```

讨论：

1. 判断给定的链表是否有环？
2. 如何找到链表的中间结点？
3. 如何从表尾输出链表？
4. 如何找到链表的倒数第K个结点？

作业（选作）：给定的K（K>0），逆置链表中包含K个结点的块。

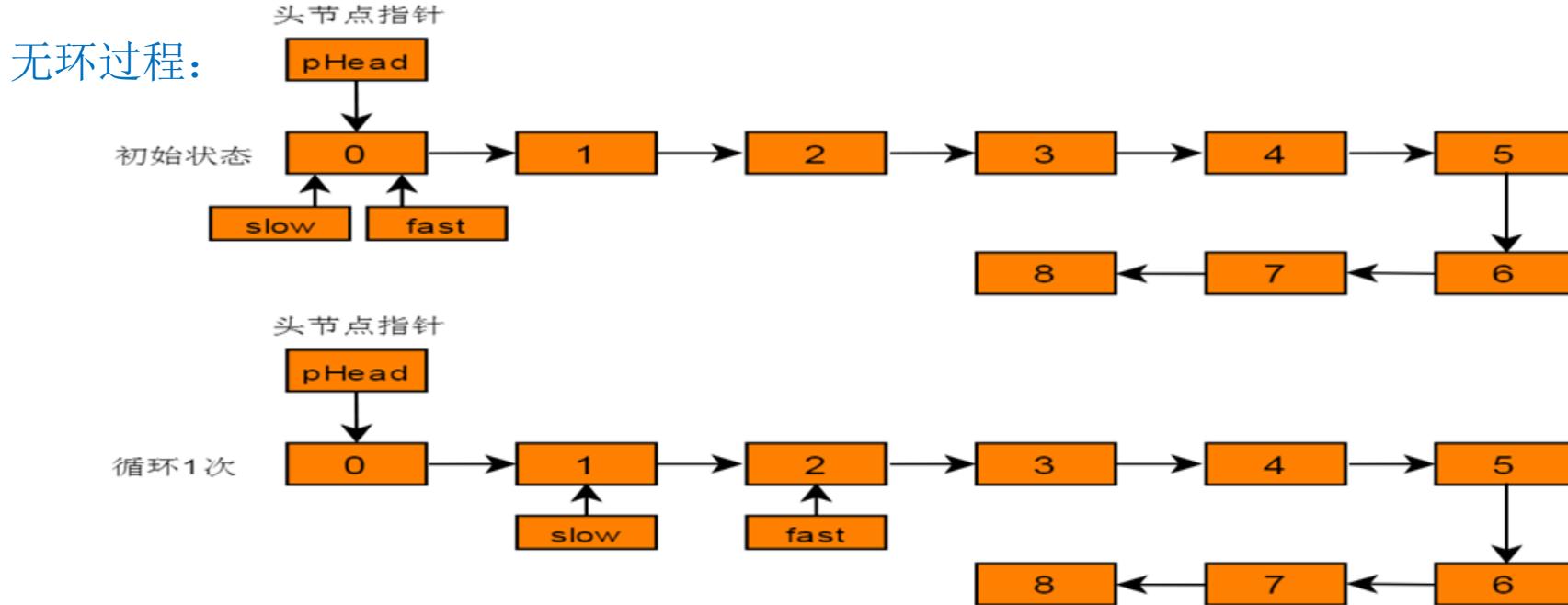
例：输入 1 2 3 4 5 6 7 8 9 10 对于不同的K值，输出为：

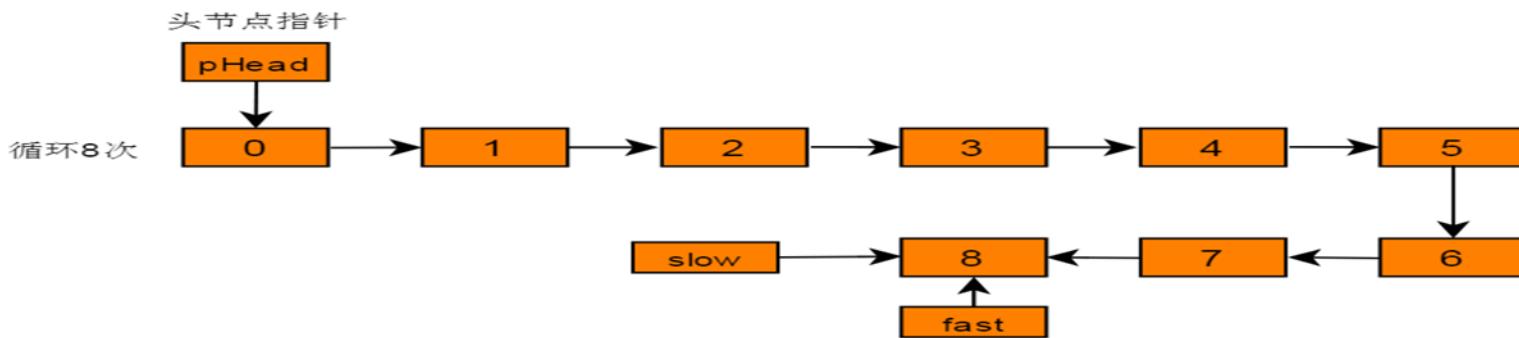
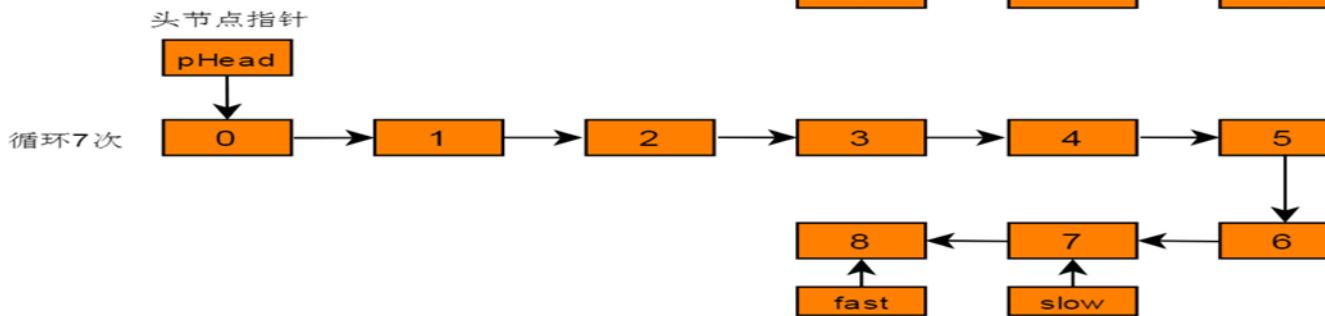
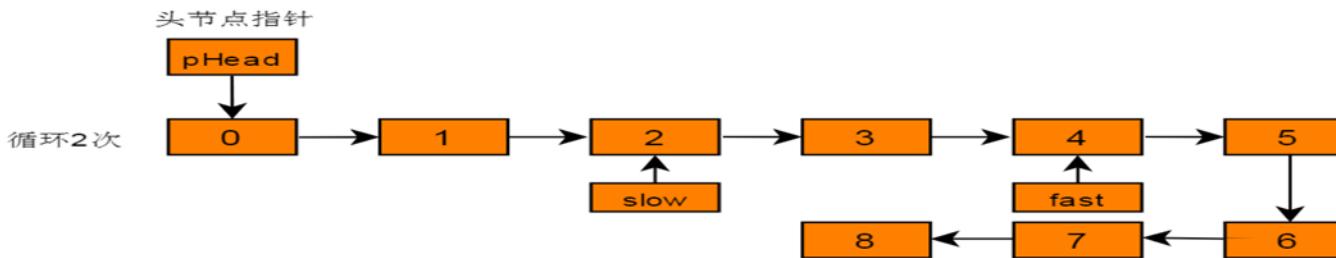
K=2:2 1 4 3 6 5 8 7 10 9 ; K=3:3 2 1 6 5 4 9 8 7 10 ; K=4:4 3 2 1 9 7 6 5 9 10

1. 判断给定的链表是否有环？

方法一：快慢指针法

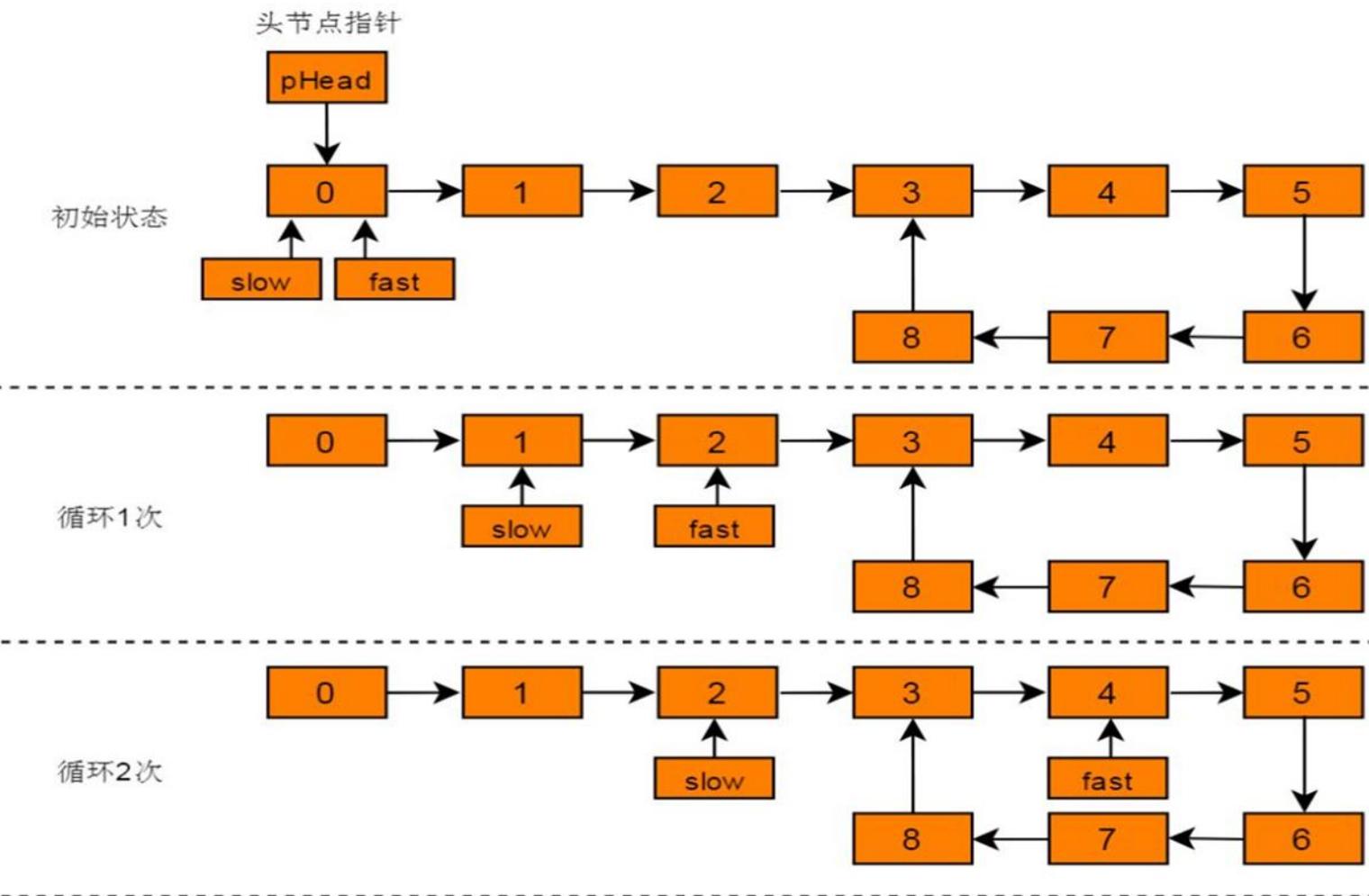
- (1) 定义两个指针分别为 `slow`, `fast`, 并且将指针均指向链表头节点。
- (2) 规定, `slow` 指针每次前进 1 个节点, `fast` 指针每次前进两个节点。
- (3) 当 `slow` 与 `fast` 相等, 且二者均不为空, 则链表存在环。

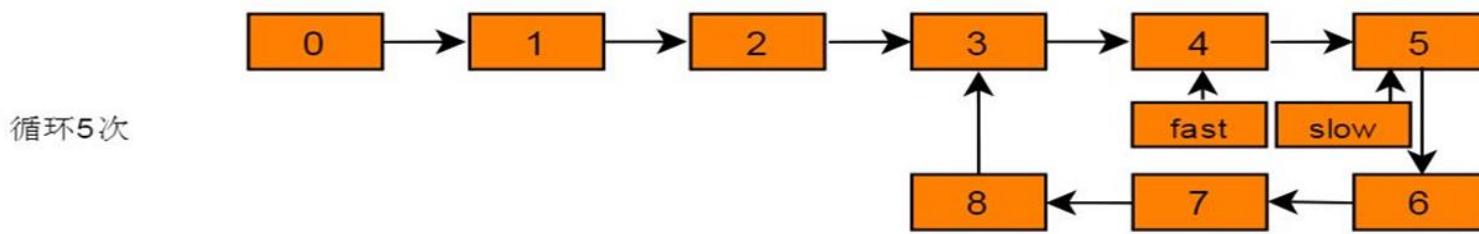
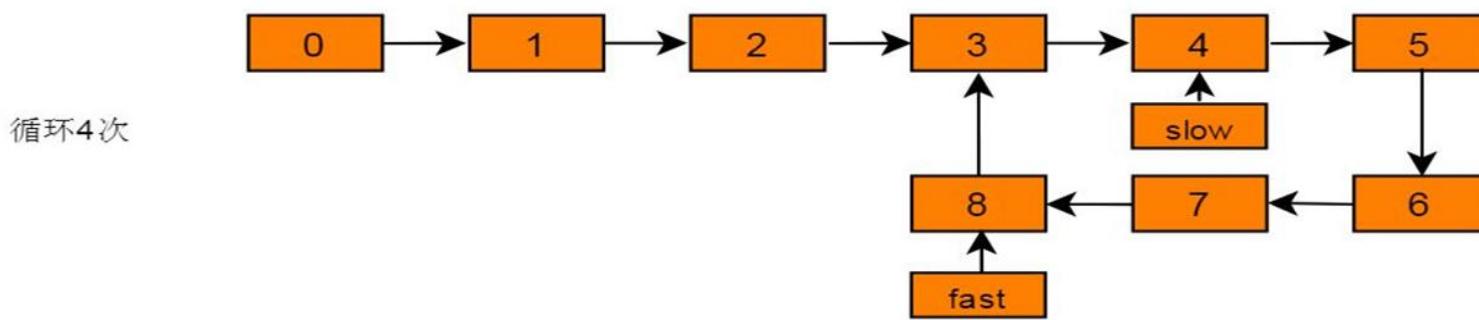
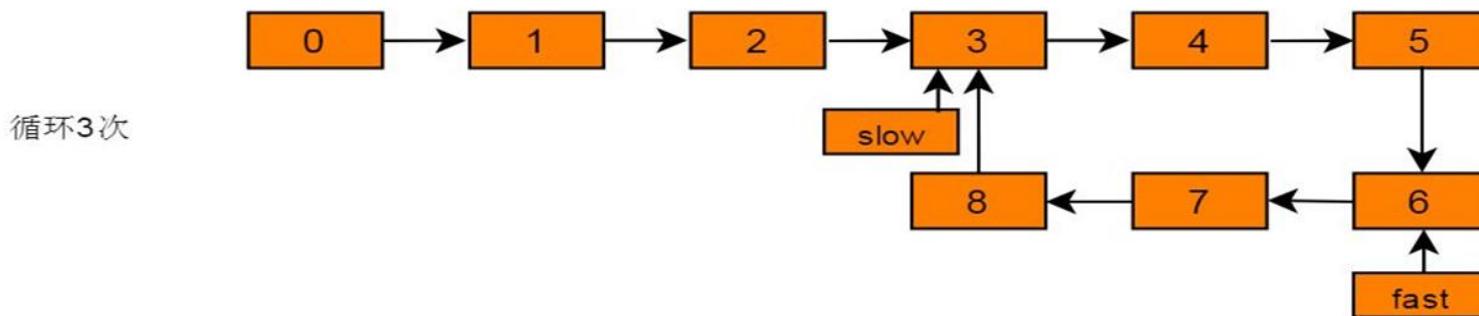




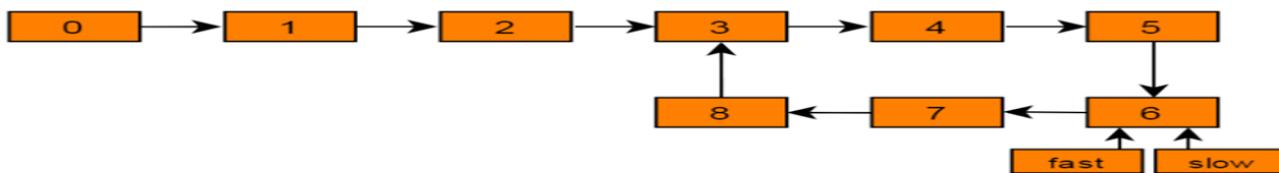
通过图解过程可以看出，若表中不存在环形，**fast** 与 **slow** 指针只能在链表末尾相遇。

有环过程：





循环6次



若链表中存在环，则快慢指针必然能在环中相遇。这就好比在环形跑道中进行龟兔赛跑。由于兔子速度大于乌龟速度，则必然会出现兔子与乌龟再次相遇情况。因此，当出现快慢指针相等时，且二者不为NULL，则表明链表存在环。

```
bool isExistLoop(ListNode* pHead) {  
    ListNode* fast; // 快指针，每次前进2个节点  
    ListNode* slow; // 慢指针，每次前进一个节点  
    slow = fast = pHead; // 两个指针均指向链表头节点  
  
    // 当没有到达链表结尾，则继续前进  
    while (slow != NULL && fast->next != NULL) {  
        slow = slow->next; // 慢指针前进一个节点  
        fast = fast->next->next; // 快指针前进两个节点  
        if (slow == fast) // 若两个指针相遇，且均不为NULL则存在环  
            return true;  
    }  
    // 到达末尾仍然没有相遇，则不存在环  
    return false;  
}
```

方法二：哈希缓存法

既然在穷举遍历时，元素比较过程花费大量时间，那么有什么办法可以提高比较速度呢？

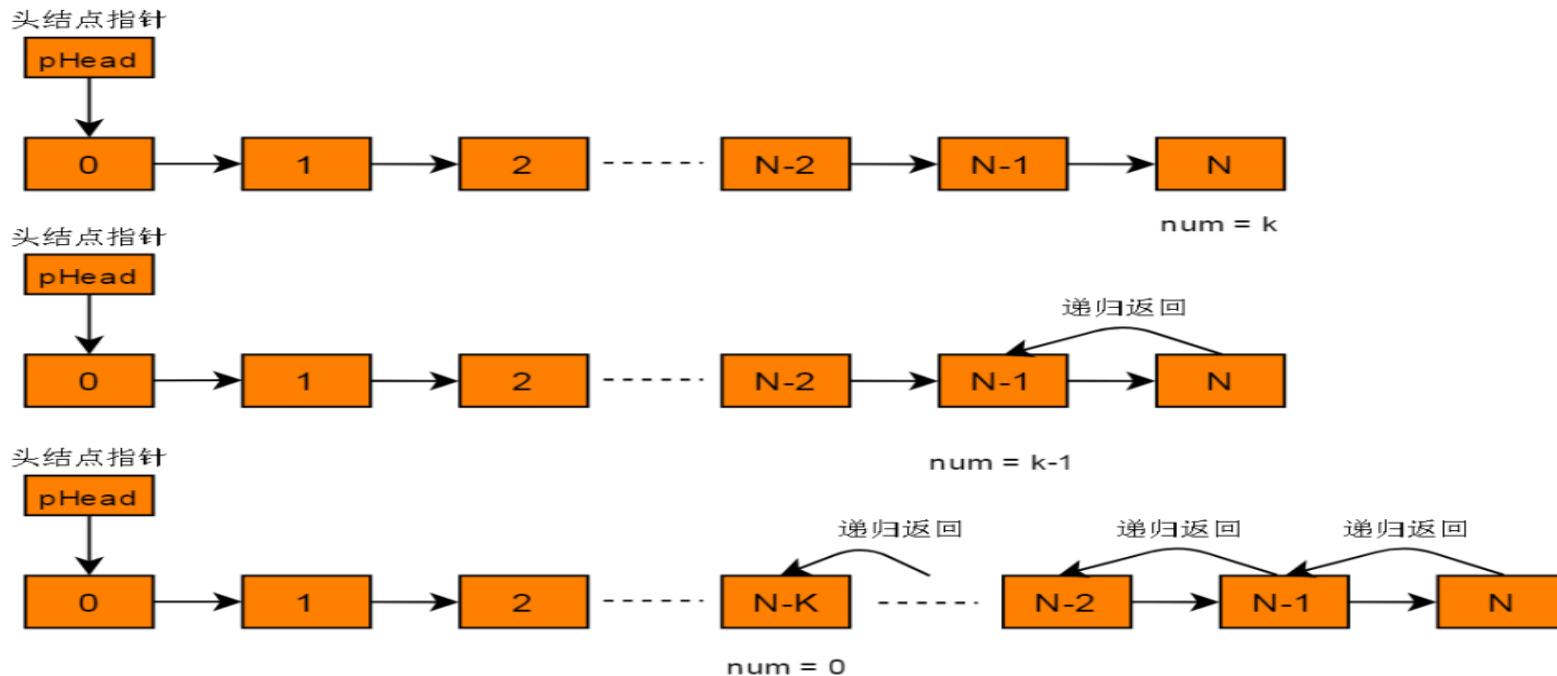
- (1) 首先创建一个以结点ID为键的 HashSet集合，用来存储曾经遍历过的结点。
- (2) 从头结点开始，依次遍历单链表的每一个结点。
- (3) 每遍历到一个新结点，就用新结点和HashSet集合当中存储的结点作比较，如果发现 HashSet当中存在相同节点ID，则说明链表有环，如果HashSet当中不存在相同的结点ID，就把这个新结点ID存入 HashSet，之后进入下一结点，继续重复刚才的操作。

假设从链表头结点到入环点的距离是 a ，链表的环长是 r 。而每一次 HashSet查找元素的时间复杂度是 $O(1)$ ，所以总体的时间复杂度是 $1 * (a + r) = a + r$ ，可以简单理解为 $O(n)$ 。而算法的空间复杂度还是 $a + r - 1$ ，可以简单地理解成 $O(n)$ 。

问题4：如何找到链表的倒数第K个结点？

方法一：递归法

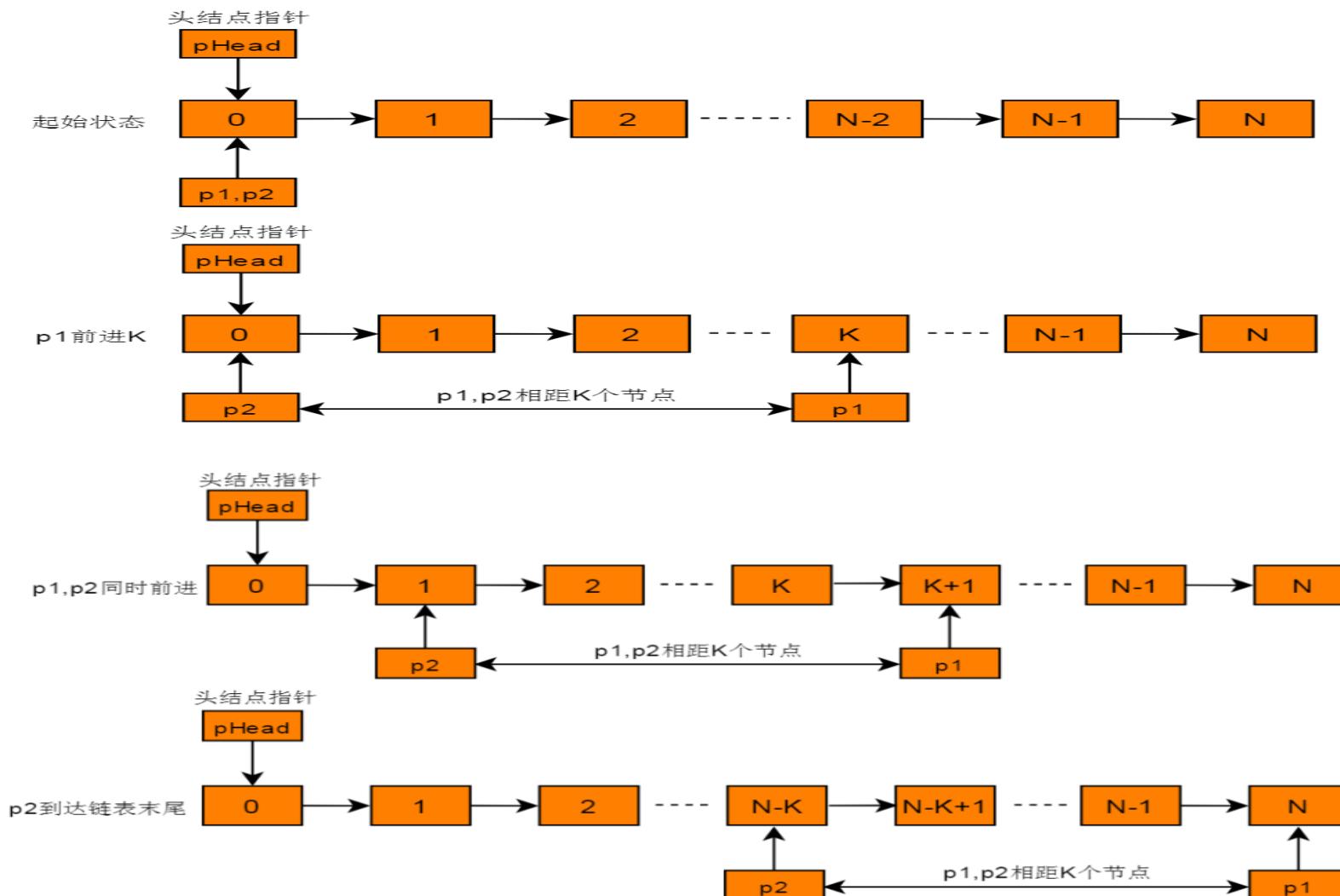
- (1) 定义 $num = k$
- (2) 使用递归方式遍历至链表末尾。
- (3) 由末尾开始返回，每返回一次 num 减 1
- (4) 当 num 为 0 时，即可找到目标节点



```
int num;//定义num值
ListNode* findKthTail(ListNode* pHead, int k) {
    num = k;
    if(pHead == NULL)
        return NULL;
    //递归调用
    ListNode* pCur = findKthTail(pHead->next, k);
    if(pCur != NULL)
        return pCur;
    else{
        num--;// 递归返回一次， num值减1
        if(num == 0)
            return pHead;//返回倒数第K个节点
        return NULL;
    }
}
```

方法二：双指针法

- (1) 定义两个指针 p1 和 p2 分别指向链表头节点。
- (2) p1 前进 K 个节点，则 p1 与 p2 相距 K 个节点。
- (3) p1, p2 同时前进，每次前进 1 个节点。
- (4) 当 p1 指向到达链表末尾，由于 p1 与 p2 相距 K 个节点，则 p2 指向目标节点。



```
ListNode* findKthTail(ListNode *pHead, int K){  
    if (NULL == pHead || K == 0)  
        return NULL;  
    ListNode *p1 = pHead; //p1, p2均指向头节点  
    ListNode *p2 = pHead; //p1先出发, 前进K个节点  
    for (int i = 0; i < K; i++) {  
        if (p1)//防止k大于链表节点的个数  
            p1 = p1->_next;  
        else  
            return NULL;  
    }  
    while (p1)//如果p1没有到达链表结尾, 则p1, p2继续遍历  
    {  
        p1 = p1->_next;  
        p2 = p2->_next;  
    }  
    return p2;//当p1到达末尾时, p2正好指向倒数第K个节点  
}
```

静态存储 与 动态链表的 比较

顺序存储

固定, 不易扩充

随机存取

插入删除费时间

估算表长度, 浪费空间

比较参数

←表的容量→

←存取操作→

←时间→

←空间→

链式存储

灵活, 易扩充

顺序存取

访问元素费时间

实际长度, 节省空间

作业（选作）：链表的维护与文件形式的保存

- 要求

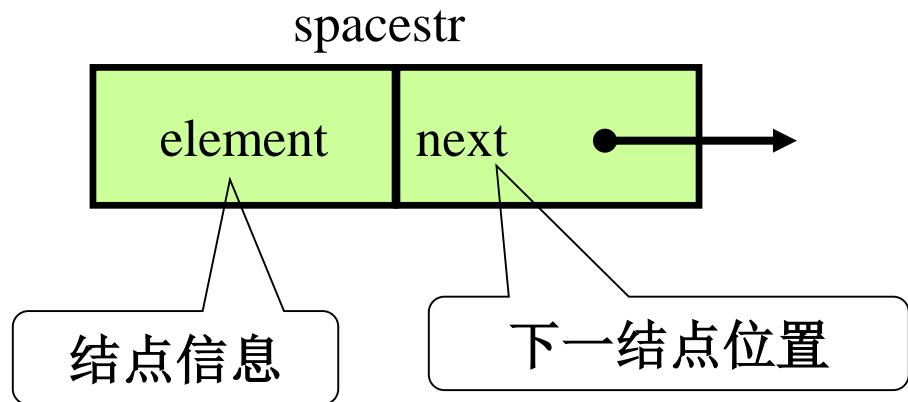
- 1) 用链表结构的有序表表示某商场家电的库存模型；
- 2) 当有提货或进货时需要对该链表进行维护；
- 3) 每个工作日结束之后，将该链表中的数据以文件形式保存，每日开始营业之前，需将以文件形式保存的数据恢复成链表结构的有序表。
- 4) 链表结点的数据域包括家电名称、品牌、单价和数量，以单价的升序体现链表的有序性。
- 5) 程序功能包括：创建表、营业开始（读入文件恢复链表数据）、进货（插入）、提货（更新或删除）、查询信息、更新信息、营业结束（链表数据存入文件）等。

2.2.3 线性表的游标实现

静态链表：

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）。

结点形式



讨论与单链表的区别?

0	d	6
1		5
2	c	-1
3	--	0
4	a	10
5		8
6	e	-1
7	--	4
8		-1
9	--	11
10	b	2
11		12
12		1

多个线性表共用一个存储池

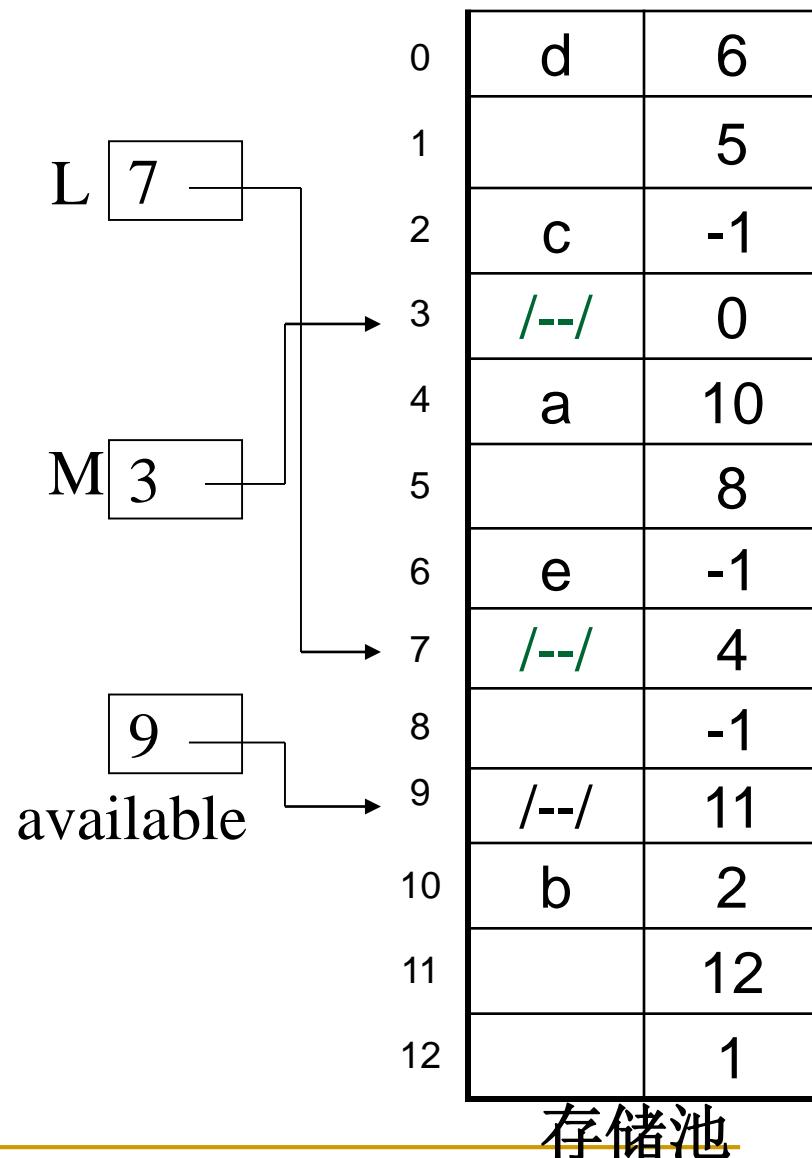
类型定义：

```
typedef struct {  
    elementtype element;  
    int next;
```

```
} spacestr; //结点类型
```

```
spacestr SPACE[ maxsize ]; //存储池  
typedef int position, cursor;
```

```
cursor av; //游标变量，标识线性表
```



线性表:

$$L = (a, b, c) \quad L = 7$$

$$M = (d, e) \quad M = 3$$

空闲表:

available = 9

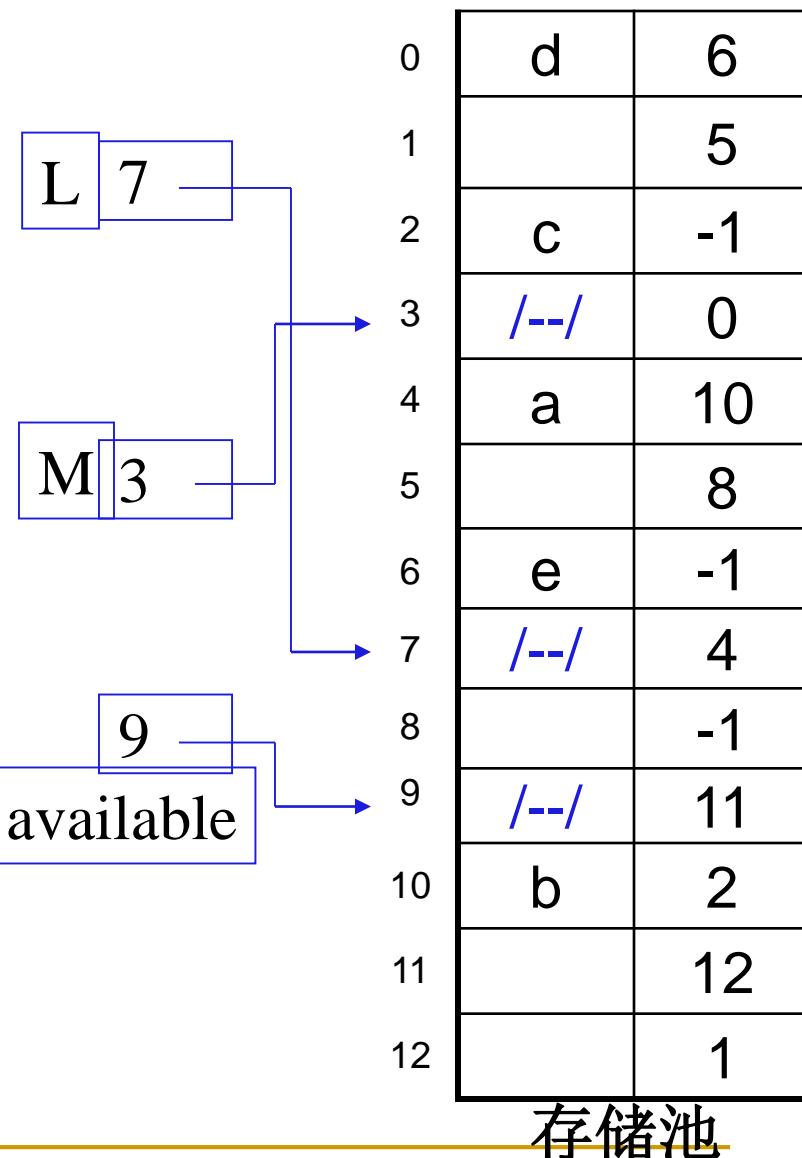
元素:

SPACE[i].element “型” 为
elementtype(基本、复合)

下一元素位置:

SPACE[i].next “型” 为 int

类似指针链表，对游标实现的线性表仍设表头结点。



存储池的管理

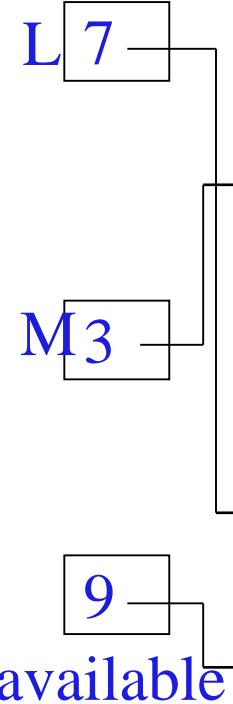
SPACE	
0	0 → 0
1	---
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	11
12	12 → -1

①可用空间的初始化

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

SPACE

0	d	6
1		5
2	c	-1
3	--	0
4	a	10
5		8
6	e	-1
7	--	4
8		-1
9	--	11
10	b	2
11		12
12		1



②可用空间的分配操作

```

cursor Malloc_SL(spacestr &SPACE, int *av)
// q=new spacestr ;
{ cursor q;
if (SPACE[*av].next == -1)
    q=-1;
else
{   q= SPACE[*av].next ; //q=11
    SPACE[*av].next =SPACE[ q ].next ;
    //av=12
    return q; }/* 从池中删除*/
}

```

③回收操作

```

void Free_SL(cursor q, int *av)
//delete q;
{   SPACE [ q ].next =SPACE[*av].next ;
    SPACE[*av].next = q ;
}/* 放回池中*/

```

```
④ void Insert( elementtype x, position p, spacestr &SPACE )
{ position q ;
  q = Malloc_SL() ;
  SPACE[ q ].element = x ;
  SPACE[ q ].next = SPACE[ p ].next ;
  SPACE[ p ].next = q ;
}

⑤ void Delete( position p, spacestr &SPACE )
{ position q ;
  if ( SPACE[ p ].next != -1 )
  {
    q = SPACE[ p ].next ;
    SPACE[ p ].next = SPACE[ q ].next ;
    Free_SL( q ) ;
  }
}
```

q = new celltype ;
q→element = x ;
q→next = p→next ;
p→next = q ;

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

作业（选作）：利用静态链表实现集合运算(**A-B**)U(**B-A**):从键盘输入集合元素。

单链表逆置问题非递归算法：

方法一：

设表头为L,算法如下：

p=L->next->next;

q=p->next;

L->next->next=NULL;

while(p!=null)

{p->next=L->next;

L->next=p;

p=q;

q=q->next;}

方法二：

线性表由q来表示

p=null;

w=q;

while(w!=null)

{

w=w->next;

q->next=p;

p=q;

q=w;}

while(p->next!=null)

{p->next=L->next;

L->next=p;

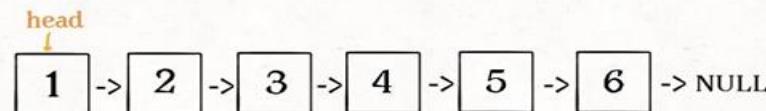
p=q;

q=q->next;}

p->next=L->next;

L->next=p;

单链表逆置问题递归算法：



将「以 **head** 为起点」的链表反转，并返回反转之后的头结点

```
ListNode reverse(ListNode head) {
```

```
    if (head->next == null) return head; // 只有一个节点的时候反转也是它自己，直接返回即可。
```

```
    ListNode last = reverse(head->next);
```

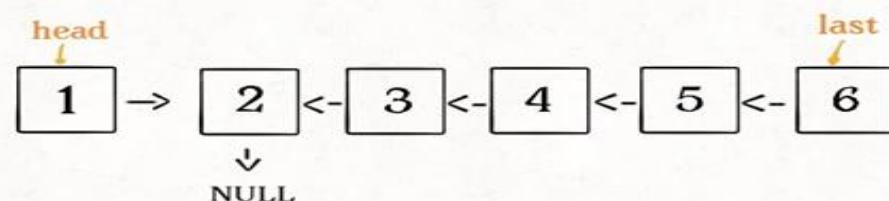
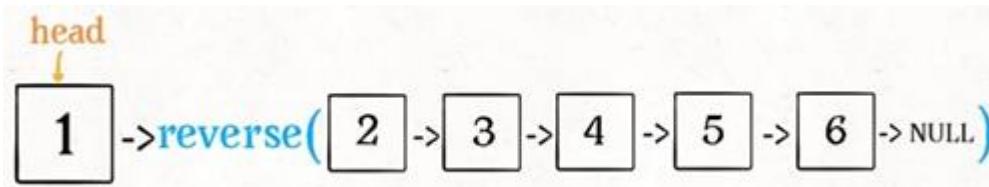
```
    head->next->next = head;
```

```
    head->next = null; // 最后一个节点，别忘了链表的末尾要指向null
```

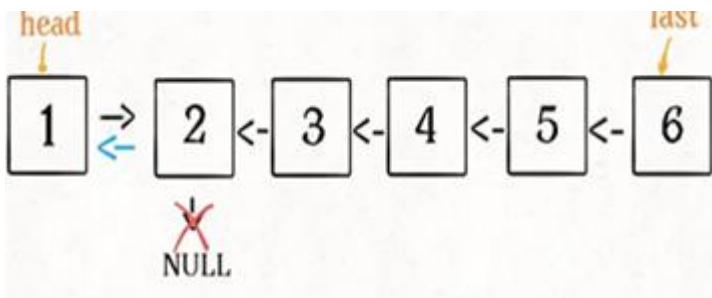
```
    return last;
```

```
}
```

```
ListNode last = reverse(head->next);
```



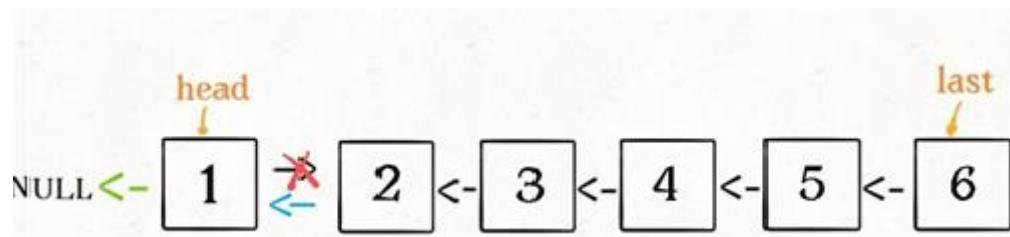
Head->next->next = head;



Head->next = null;
return last;

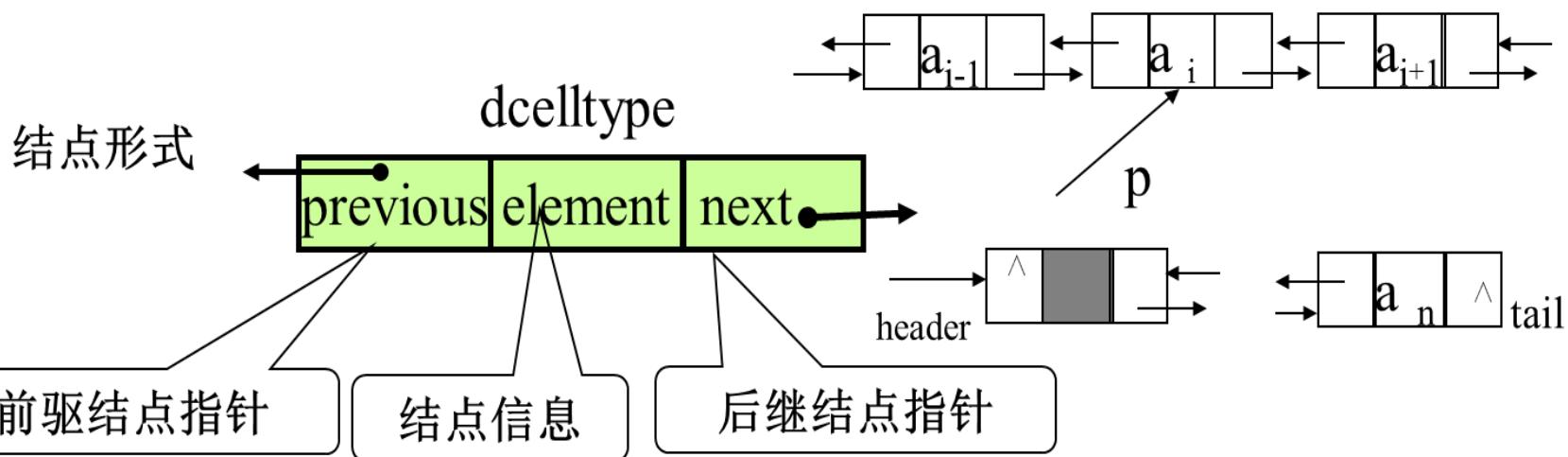
ListNode reverse(ListNode head)

```
{  
    if (head->next == null) return head;  
    ListNode last = reverse(head->next);  
    head->next->next = head;  
    head->next = null;  
    return last;  
}
```



2.2.4 双向链表

双向连表：在单向链表中，对每个结点增加一个域，用一指向该结点的前驱结点。线性表的这种存储结构称为双向链表。



```
/* 结点类型 */
struct dcelltype {
    elementtype element;
    dcelltype *next, *previous;
};

/* 表和位置的类型 */
typedef dcelltype *DLIST;
typedef dcelltype *position;
```

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。

 操作：

```
void Insert( elementtype x, position p)
```

{

```
position q ;
```

```
q = new dcelltype ;
```

```
q->element = x ;
```

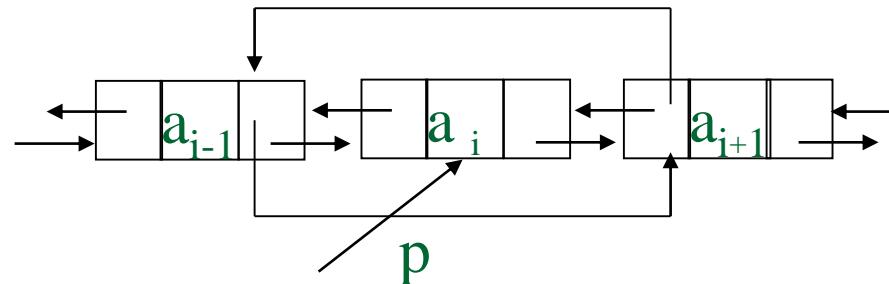
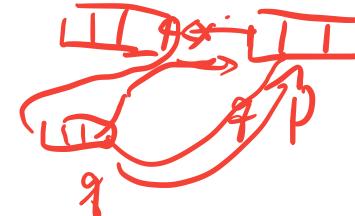
```
p->previous->next = q ;
```

```
q->previous = p->previous ;
```

```
q->next = p ;
```

```
p->previous = q ;
```

}



删除位置p的元素：

```
void Delete( position p)
```

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

```
    p->previous->next = p->next;
```

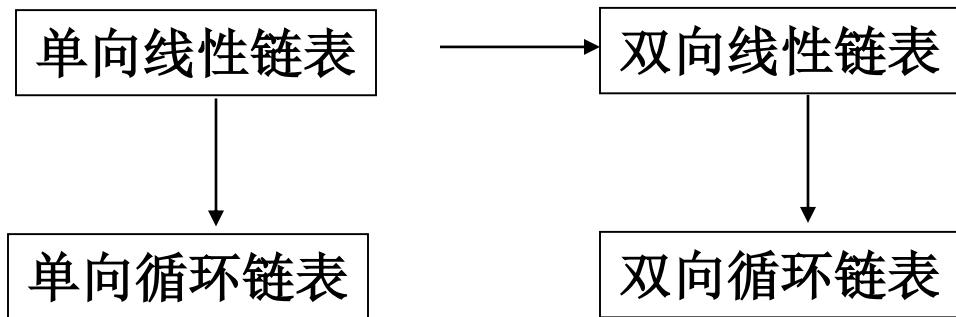
```
if (p->next!=NULL)
```

```
    p->next->previous = p->previous ;
```

```
delete p;
```

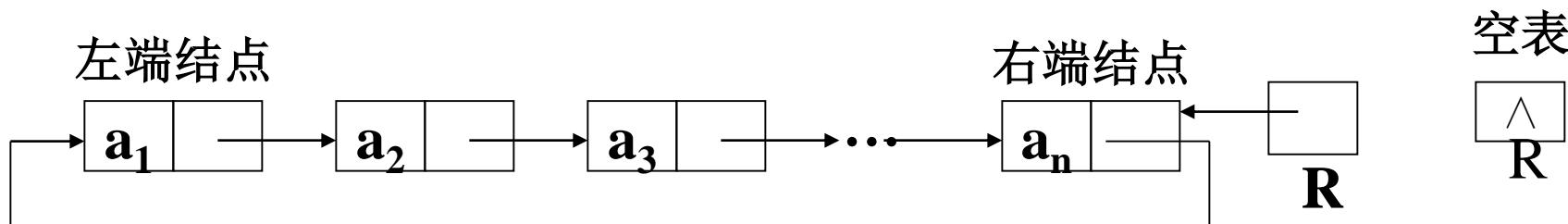
}

2.2.5 环形链表



对线性链表的改进，解决“单向操作”的问题；改进后的链表，能够从任意位置元素开始，访问表中的每一个元素。

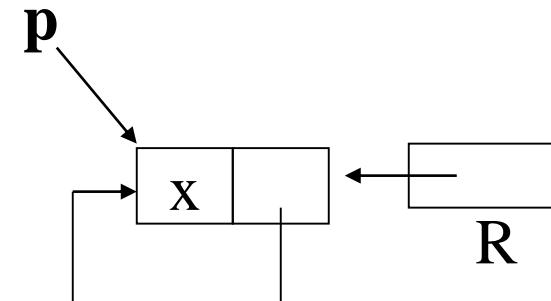
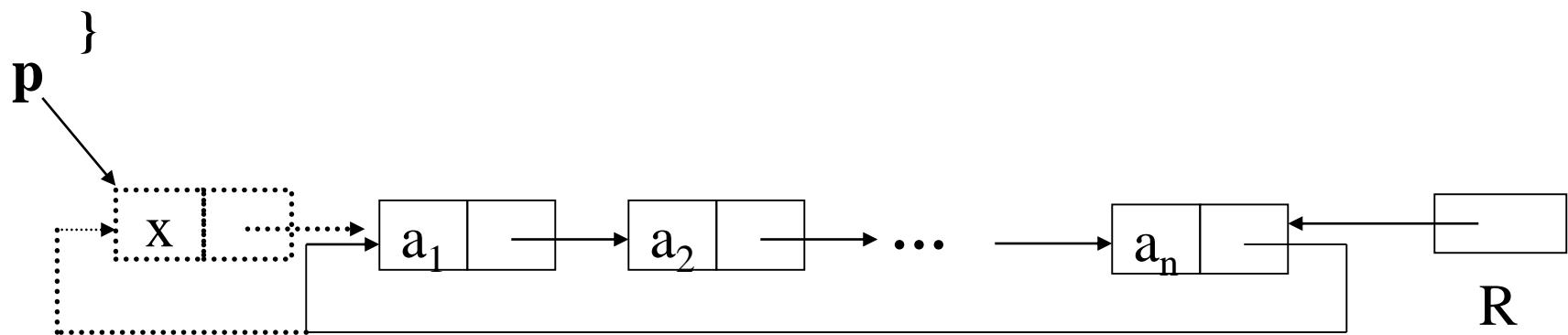
单向环形链表：在(不带表头结点)的单向链表中，使末尾结点的指针域指向头结点，得到一个环形结构；用指向末尾结点的指针标识这个表。



存储结构:与单向链表相同(略)

操作: ①在表左端插入结点Insert(x,First(R),R); →Linsert(x,R)

```
void Linsert( elementtype x , LIST &R )
{  celltype *p ;
   p = new celltype ;
   p->element = x ;
   if ( R == NULL )
      {  p->next = p ;  R = p ; }
   else
      {  p->next = R->next ;  R->next = p ; }
```



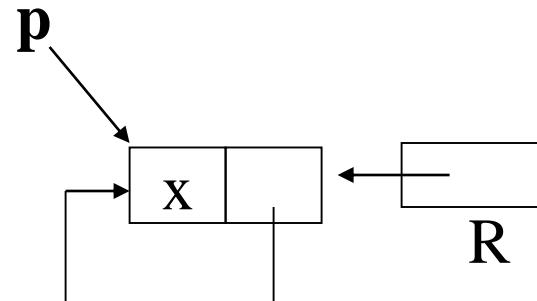
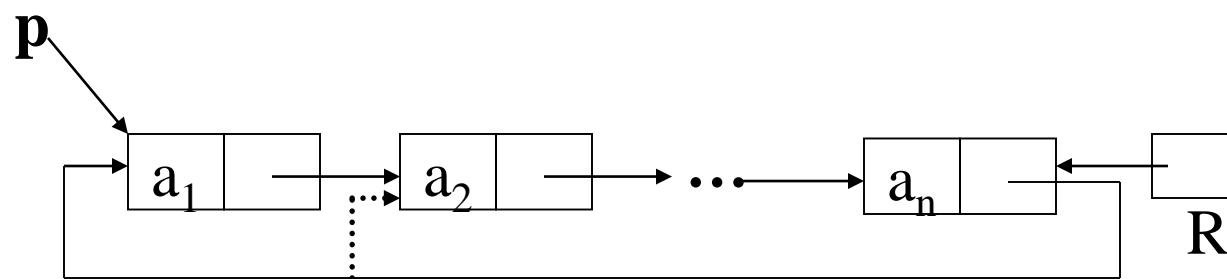
②在表右端插入结点Insert(x,END(R),R); →Rinsert(x,R)

```
void Rinsert( elementtype x , LIST &R )
{  Linsert (x,R) ;  R = R->next ; }
```

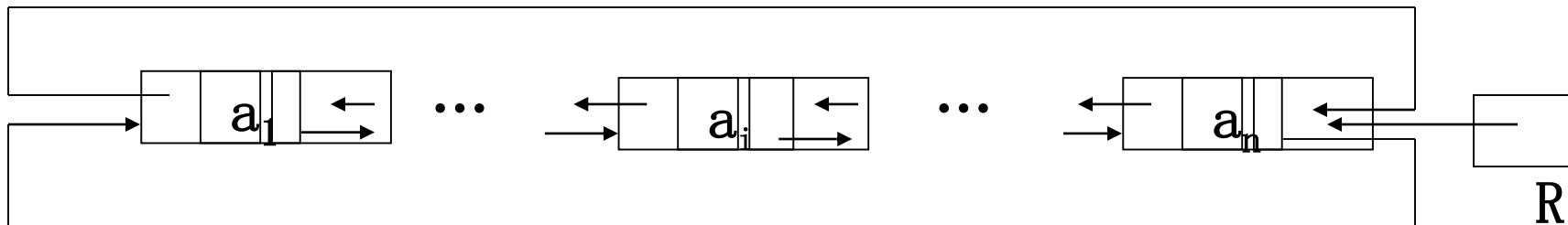
操作：③从表左端删除结点Delete(First(R),R); →Ldelete(R)

void Ldelete(LIST &R)

```
{  celltype *p ;  
  if ( R == NULL )  
    error ( “空表” );  
  else  
  {  p = R->next ;  
    R->next = p->next ;  
    if ( p == R )  
      R=NULL;  
    delete p ;  
  }  
}
```

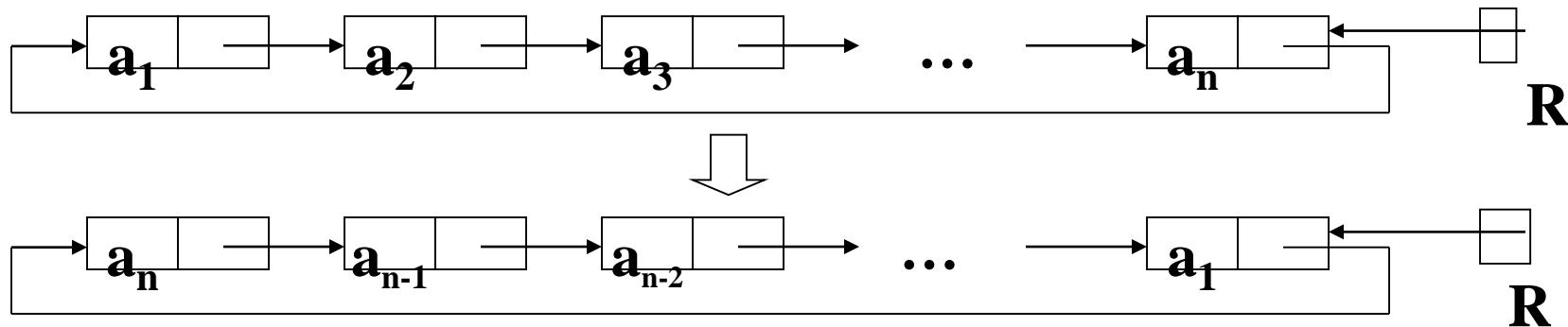


双向环形链表：



双向环形链表的结构与双向连表结构相同，只是将表头元素的空
previous域指向表尾，同时将表尾的空next域指向表头结点，从而形成
向前和向后的两个环形链表，对链表的操作变得更加灵活。

例：设计算法，将一个单向环形链表反向。头元素变成尾元素，尾元素变成新的头元素，依次类推。



算法如下：

```
void Revers( LIST &R )
{ position p, q, s;
  q = R;
  p= R->next ;
  R= R->next ;
```

```
while ( p!= R )
  { s = q ;
    q = p ;
    p = p->next ;
    q->next =s ;
  }
  p->next=q;
}
```

问题：

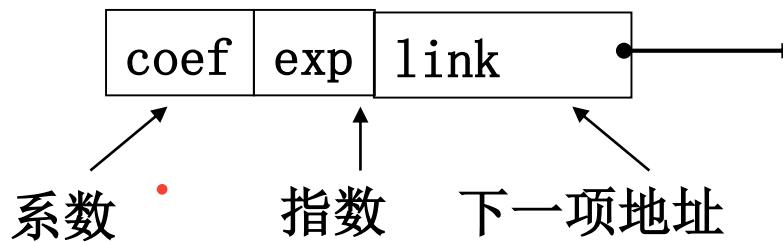
多项式 $p(x) = 3x^{14} + 2x^8 + 1$

$$q(x) = 5x^3 - x^2 + 8$$

$$p+q=?$$

2.2.6 多项式的代数运算

结点结构

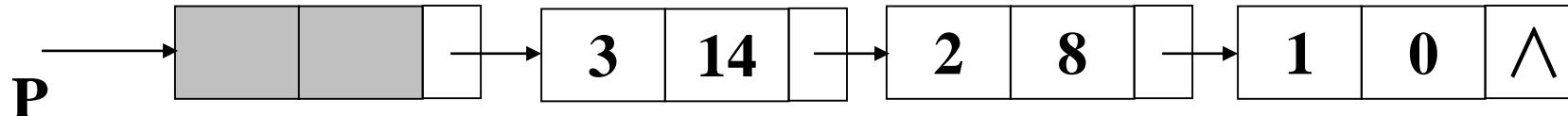


结点类型:

```
struct polynode {  
    int coef ;  
    int exp ;  
    polylink *link ;  
};  
typedef polynode *polypointer ;
```

例如：

多项式 $p(x) = 3x^{14} + 2x^8 + 1$ 采用链表表示如下：



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 ;
};
```

算法 `padd` 实现两个多项式
a, b 相加；
 $c(x) = a(x) + b(x)$

```
polypointer Padd (polypointer a , polypointer b )
{
    polypointer p, q, d, c ;
    int x ;
    p = a->link ; q = b->link ;
    c = new polynode ; d = c ;
    while ( (p != NULL) && (q != NULL) )
        switch ( compare ( p->exp, q->exp ) )
        {
            case '=' :
                x = p->coef + q->coef ;
                if ( x ) d = attch( x, p->exp, d ) ;
                p = p->link ; q = q->link ;
                break ;
            case '>' :
                d = Attch( p->coef, p->exp, d );
        }
}
```

```
p = p->link ;
break ;

case '<':
    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 ;
}
```

$$\begin{array}{r}
 \frac{x}{4} + \frac{7}{8} \text{ 商} \\
 \hline
 \text{除数 } 4x^2 + 2x - 1 \Big) \overline{x^3 + 4x^2 + 3x + 2} \text{ 被除数} \\
 x^3 + \frac{x^2}{2} - \frac{x}{4} \\
 \hline
 \frac{7x^2}{2} + \frac{13x}{4} + 2 \\
 \frac{7x^2}{2} + \frac{7x}{4} - \frac{7}{8} \\
 \hline
 \frac{6x}{4} + \frac{23}{8} \text{ 余数}
 \end{array}$$

$$\frac{x^3 + 4x^2 + 3x + 2}{4x^2 + 2x - 1} = \frac{x}{4} + \frac{7}{8} + \frac{\frac{3}{2}x + \frac{23}{8}}{4x^2 + 2x - 1}$$

假分式

多项式

真分式

2.3 栈 (Stack)

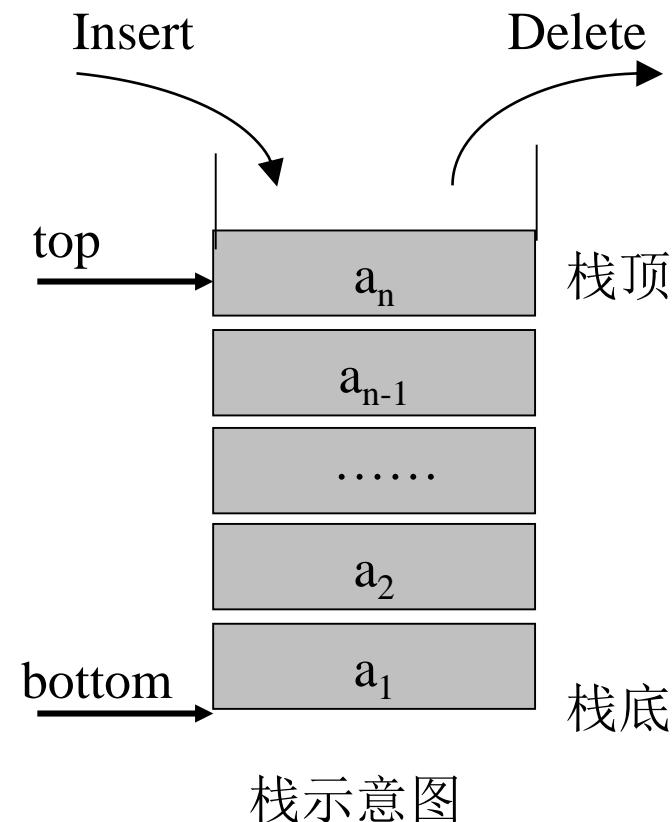
栈是线性表的一种特殊形式，是一种限定性数据结构，也就是在对线性表的操作加以限制后，形成的一种新的数据结构。

定义：是限定只在表尾进行插入和删除操作的线性表。

栈又称为后进先出

(Last In First Out)的线性表。

简称LIFO结构。



2.3 栈

栈的基本操作

- ① **MakeNull (S)**
- ② **Top (S)**
- ③ **Pop (S)**
- ④ **Push (x , S)**
- ⑤ **Empty (S)**

举例：利用栈实现行编辑处理。

设定符号“#”为擦讫符，用以删除“#”前的字符；符号“@”为删行符，用以删除当前编辑行。

原理：

读字符 {
 一般字符进栈；
 擦讫符退栈；
 删行符则清栈

算法：

void LineEdit()

{

Stack S ;
 char c ;
MakeNull (S);
c = getchar () ;
while (c != '\n')

{

if (c == '#')
 Pop (S);
 else if (c == '@')
 MakeNull (S);

else

Push (c , S);
 c = getchar();

}

 逆序输出栈中所有元素；

}

输入：

a b c @ # d : 小猪

a b c @ w # d : d

2.3.1 栈的数组实现

类型定义：

```
enum Boolen {True, False}  
typedef struct {  
    elementtype elements[maxlength];  
    int top ;  
} Stack ;
```

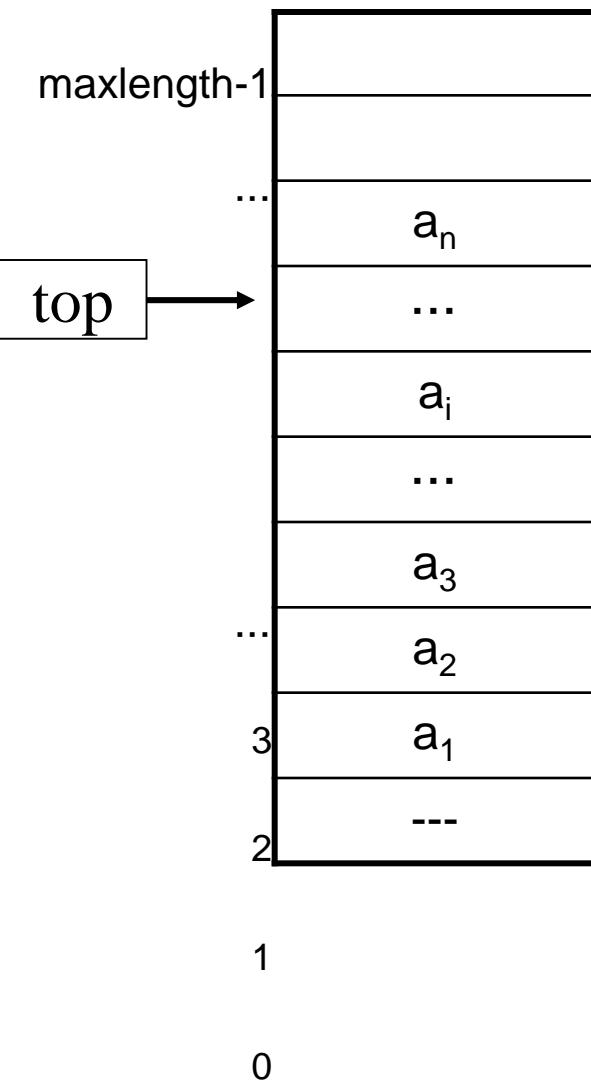
```
Stack S ;
```

栈的容量： $\text{maxlength} - 1$ ；

栈空： $S.\text{top} = 0$ ；

栈满： $S.\text{top} = \text{maxlength} - 1$ ；

栈顶元素： $S.\text{elements}[S.\text{top}]$ ；



```
① void MakeNull( Stack &S )
{ S.top = 0 ; }

② Boolean Empty(Stack S )
{ if ( S.top < 1 )
    return True
  else
    return False ;
}

③ elementtype Top(Stack S )
{ if Empty( S )
    return Null;
  else
    return ( S.elements[ S.top ] );
}
```

```
④ elementtype Pop(Stack &S )
{
  if ( Empty ( S ) )
    error ( “栈空” );
  else
    S.top = S.top - 1 ;
}

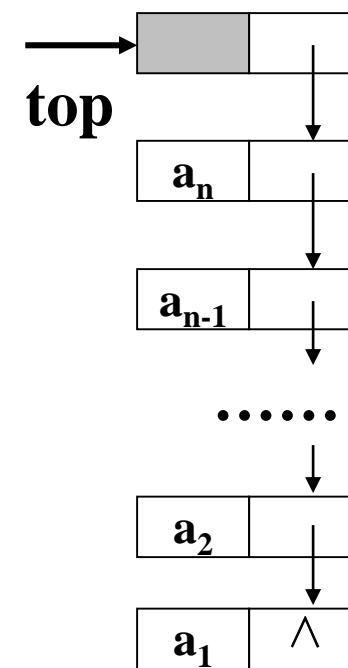
⑤ void Push ( elementtype x, Stack &S )
{
  if ( S.top == maxlen - 1 )
    error ( “栈满” );
  else
    { S.top = S.top + 1 ;
      S.elements[ S.top ] = x ;
    }
}
```

2.3.2 栈的指针实现

采用由指针形成的线性链表来实现栈的存储。

实现的方式如右图所示，其操作与线性链表的表头插入和删除元素相同。

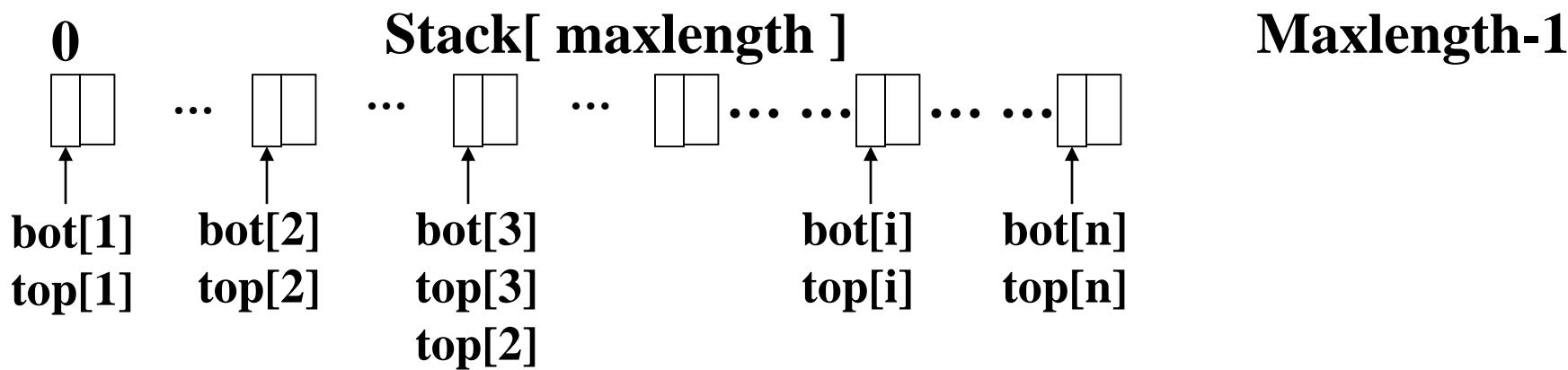
```
struct node {  
    Elementtype val;  
    node * next;  
};  
typedef node * Stack;
```



讨论

1、两个栈共用一个存储空间的

2、多个栈共用一个存储空间的



例1、使用游标形式使得三栈共享

设用一个类型为STATICLIST的一维数组A[m]存储空间建立三个栈. 其中前三个单元的next存放三个栈顶的指针, 第四个单元起共享. 编写一算法从键盘输入n个整数按下列条件进栈:

- (1) $t < 80$ 进1栈
- (2) $80 \leq t \leq 100$ 进2栈
- (3) $t > 100$ 进3栈

	0	1	2		m-1
data					
next					

如果输入数据为:100, 30, 68, 90, 120后结果应如下:

0	1	2	3	4	5	6	7	8	9	10	11	12
/	/	/	100	30	68	90	120	60	156	85		
8	10	9	0	0	4	3	0	5	7	6	12	-1

3

再输入数据为:60, 156, 85后结果变化

如果输入数据为:100, 30, 68, 90, 120后结果应如下:

0	1	2	3	4	5	6	7	8	9	10	11	12
/	/	/	100	30	68	90	120	60	156	85		
8	10	9	0	0	4	3	0	5	7	6	12	-1

再输入数据为:60, 156, 85后结果变化

1栈栈顶连续出栈，情况如下:

0	1	2	3	4	5	6	7	8	9	10	11	12
/	/	/	100	30	68	90	120	60	156	85		
4	10	9	0	0	8	3	0	11	7	6	12	-1

i=A[0].next //存栈顶下标

A[0].next=A[i].next//定义新栈顶下标

A[i].next=top //删除空间链到待用

top=i//空闲起点

```
struct node
{
    int data;
    int next;
};

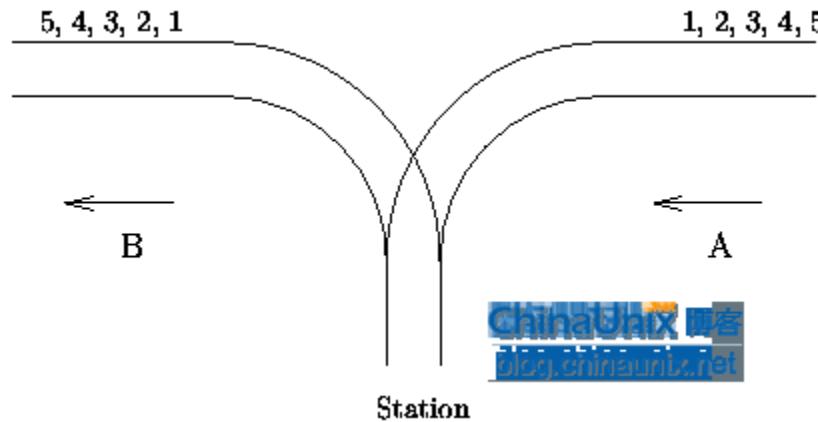
typedef struct node Node;

void Share(Node A[])
{
    int i,top;
    for(i=0;i<3;i++)
        A[i].next=0;
    for(i=3;i<n;i++)
    {
        A[i].next=i+1;
        A[n-1]=0;
    } //初始化
}
```

```
top=3;
for(i=1;i<=n;i++)
{
    j=top;
    cin>>t;
    A[j].data=t;
    top=top++;
    if(t<80)
    {
        A[j].next=A[0].next;
        A[0].next=j;
    }
}
```

```
else if(t>100)
{
    A[j].next=A[2].next;
    A[2].next=j;
}
else
{
    A[j].next=A[1].next;
    A[1].next=j;
}
```

例2 火车有n节车厢，顺序编号为 $1, 2, 3, \dots, n$ ，从A驶入，从B驶出，车站里能停放任意多节车厢。一旦进入车站就不再回到A方向的铁轨上，一旦进入B方向铁轨，不再回车站。判断一个指定车厢顺序能否从B方向驶出。



输入顺序：1 2 3 4 5

输出顺序：5 4 1 2 3

```
#include<stdio.h>
#include <iostream>
#include <stack>
using namespace std;
int isPopSequence (int input[], int target[], int n)
{
    int A=0,B=0;  stack<int>s;  int ok=1;
    while(B<n)
    {
        if(input[A]==target[B])
        {
            A++;      B++;
        }
        //栈不为空，判断栈顶
        else if(!s.empty() && s.top()==target[B])
        {
            s.pop();//已经相当于A++
            B++;//所以B要加1
        }
        else if(A<n)
        {
            s.push(A++);//用input的下一个来比较
        }
        else
        {
            ok=0;      break;
        }
    }
    return ok;
}
```

```
int main()
{
    int a[5]={1,2,3,4,5};

    int b[5]={2,5,4,3,1};

    if(isPopSequence(a,b,5) == 1)

        cout<<"True"<<endl;

    else

        cout<<"False"<<endl;

    return 0;
}
```

• 栈的应用

地图四染色

迷宫问题

函数的嵌套调用

递归的实现

回文游戏

多进制转换

括号匹配的检验

行编辑程序

表达式求值



斐波那契数列

汉诺塔问题

递归函数

2、栈的简单应用

例1 算术表达式中括号作用域合法性的检查

```
int Correct (char ext[], int n) // 数组ext用于存储表达式
{
    Stack S;
    int j = 1;
    MakeNull(S);
    while (表达式没有结束时)
    {
        if (exp[j]是左括号) // 左括号 {, [, (
            Push ( ext[j], S )
        if (ext[j]是右括号) // 右括号 }, ], )
        {
            x = Top(S); // 取栈首元素比较
            if (x与ext[j]匹配) Pop(S);
            else return False;
        }
        j++;
    }
    if ( ! Empty ( S ) ) return False;
    else return True;
}
```

例2 表达式求值

表达式: $\left\{ \begin{array}{l} \text{前缀表达式 (波兰式)} \\ \text{中缀表达式 } 4+2*3-10/5=4+6-10/5=10-10/5=10-2=8 \\ \text{后缀表达式 (逆波兰式)} \end{array} \right.$

例如:

$(a + b) * (a - b)$ $\left\{ \begin{array}{l} *+ab-ab \\ (a + b) * (a - b) \\ ab+ab-* \end{array} \right.$

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：

- ① 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。
- ② 后缀表达式中不需要括弧定义计算顺序，而由运算（操作符）的位置来确定运算顺序。

I. 将中缀表达式转换成后缀表达式

对中缀表达式从左至右依次扫描，由于操作数的顺序保持不变，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。

遇到“（”进栈，当遇到“）”时，退栈输出直到“（”为止。

$c_2 \backslash c_1$	+	-	*	/	()	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	<	<
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

任意两个相继出现的算符C1和C2之间的优先关系至多三种：

C1<C2 C1优先级低于C2

C1=C2 令C1>C2

C1>C2 C1优先级高于C2

例如：

中缀表达式 $a + b * c + (d * e + f) * g$, 其转换成后缀表达式则为

~~a b c * + d e * f + g * +~~

转换过程需要用到栈。

II. 由后缀表达式计算表达式的值

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

假定待求值的后缀表达式为：6 5 2 3 + 8 * + 3 + *，则其求值过程如下：

1) 遍历表达式，遇到的数字首先放入栈中，此时栈如下所示：

TopOfStack	3
	2
	5
	6

2) 接着读到“+”，则弹出3和2，执行 $3+2$ ，计算结果等于5，并将5压入到栈中。

TopOfStack	5
	5
	6

3) 读到8，将其直接放入栈中。

TopOfStack	8
	5
	5
	6

4) 读到“*”，弹出8和5，执行 $8*5$ ，并将结果40压入栈中。而后过程类似，读到“+”，将40和5弹出，将 $40+5$ 的结果45压入栈...以此类推。最后求的值288

3、栈与递归问题

一、递归条件

采用递归方法来解决问题，必须符合以下三个条件：

- 1、可以把要解决的问题转化为一个新问题，这个新问题的解决方法仍与原来的解决方法相同，只是所处理的数据规模更小。
- 2、可以应用这个转化过程使问题得到解决。
- 3、必定要有一个明确的结束递归的条件。

二、递归实例

例1：使用递归的方法求n!

当 $n > 1$ 时，求 $n!$ 的问题可以转化为 $n * (n - 1)!$

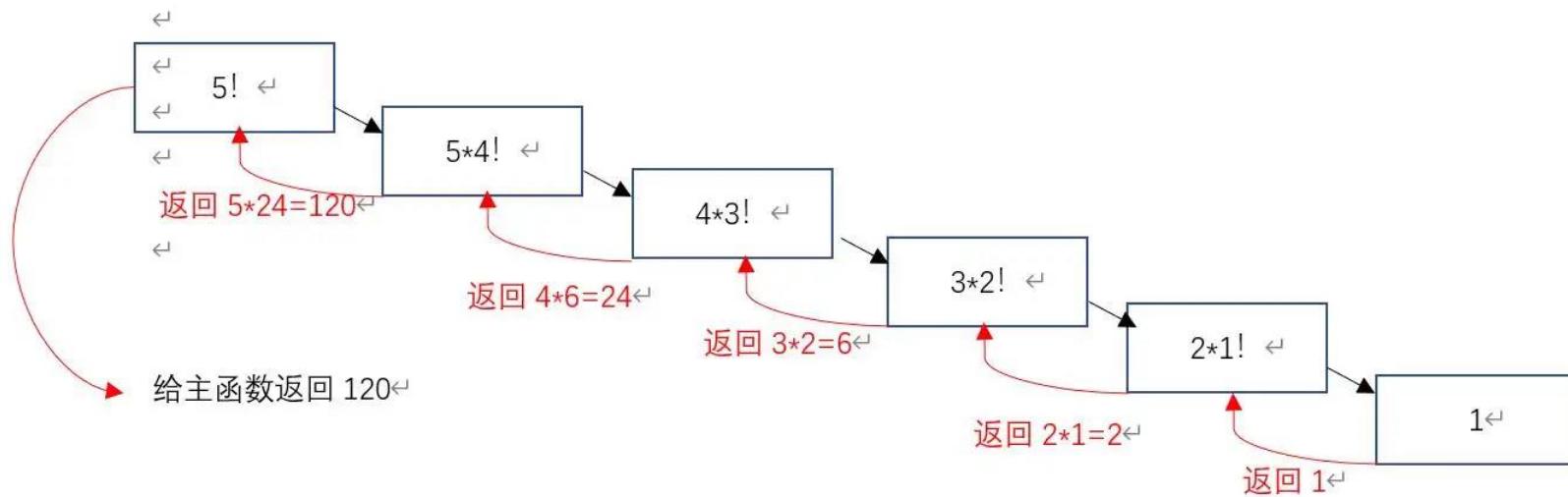


德罗斯特效应—递归的视觉效应

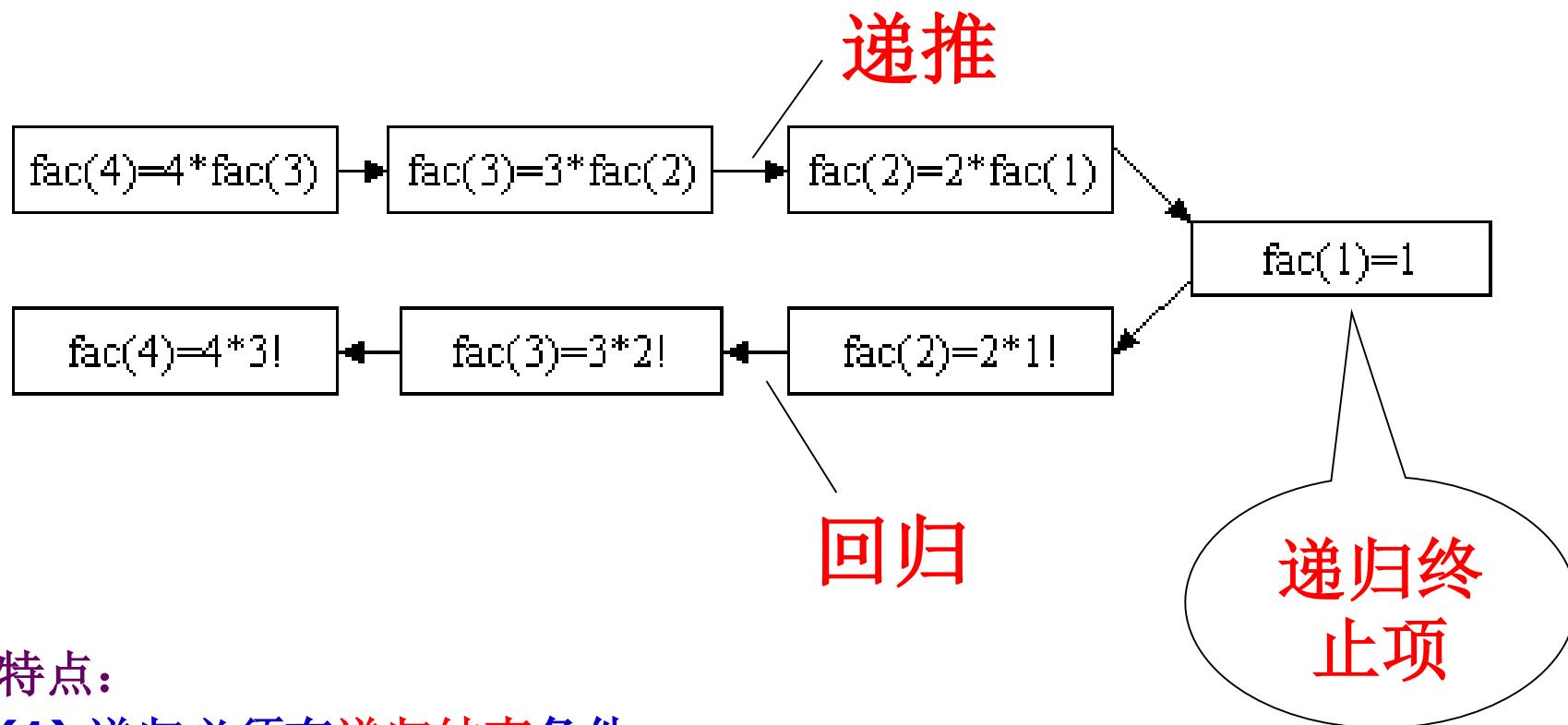
计算5的阶乘 ($5! = 5 * 4 * 3 * 2 * 1$)

int f(int n)

```
{  
    if ( n == 1)  return (1);  
    else  return ( n * f(n-1));  
}
```



递归的执行过程



特点：

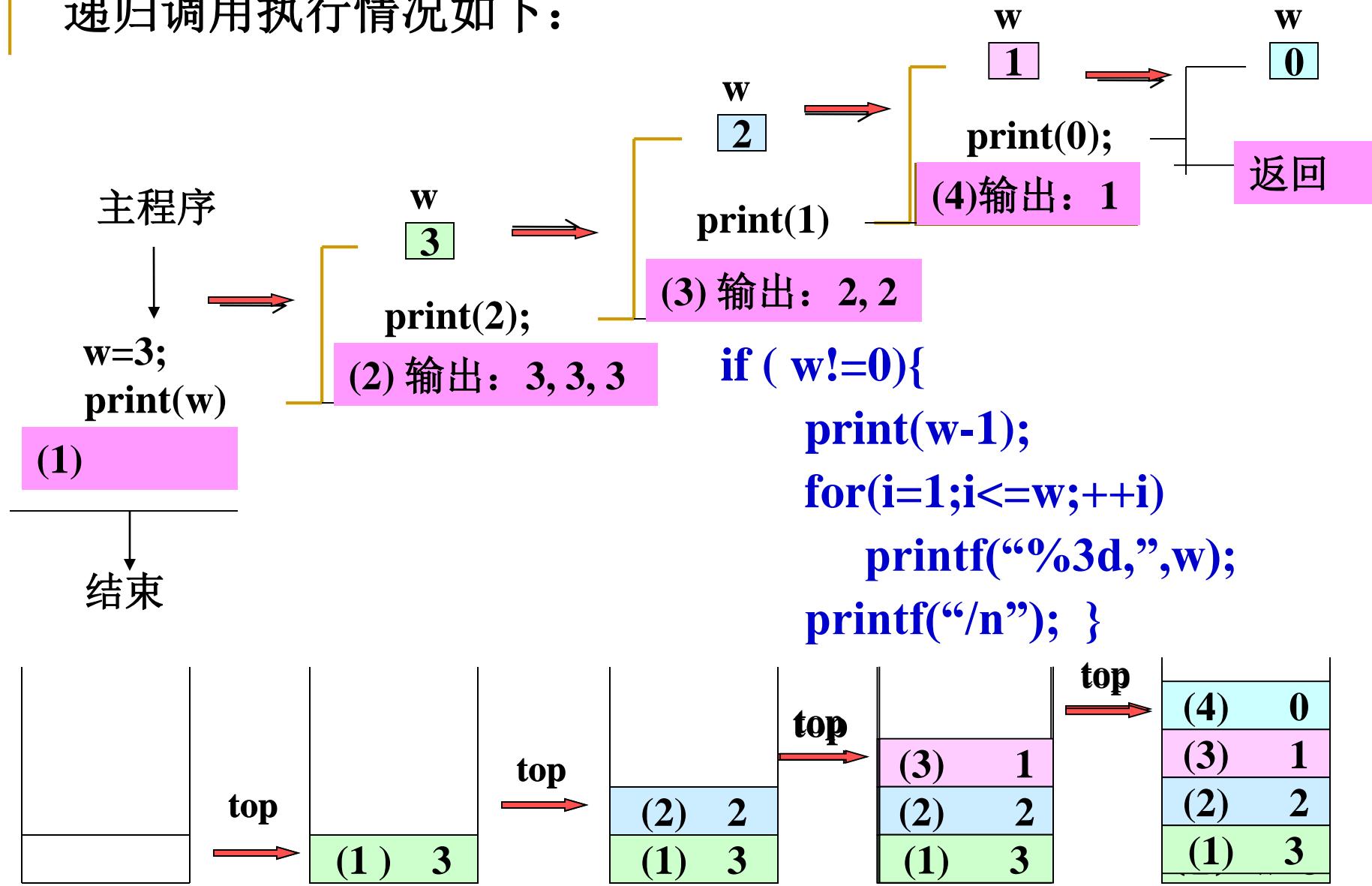
- (1) 递归必须有递归结束条件；
- (2) 递归过程首先是一级一级递推（即当前调用未结束又引出下一层调用），而满足递归结束条件时结束递推，再一级一级的回归（即一级一级再返回上一级的调用继续完成上一级的未完成的操作）。

例2 递归的执行情况分析

```
void print(int w)
{
    int i;
    if ( w!=0)
    {
        print(w-1);
        for(i=1;i<=w;++i)
            printf("%3d,",w);
        printf("/n");
    }
}
```

若w为3时，
运行结果：
1,
2, 2,
3, 3, 3,

递归调用执行情况如下：



例. 设A是一个有n个不同元素的数组，写出求其最大和最小元素的算法，分析其时间复杂性。

```
int max(int n, int * arr)
{
    if (n==1)
        return arr[n-1];
    else
        if (arr[n-1]>max(n-1,arr))
            return arr[n-1];
        else
            return max(n-1,arr);
}
```

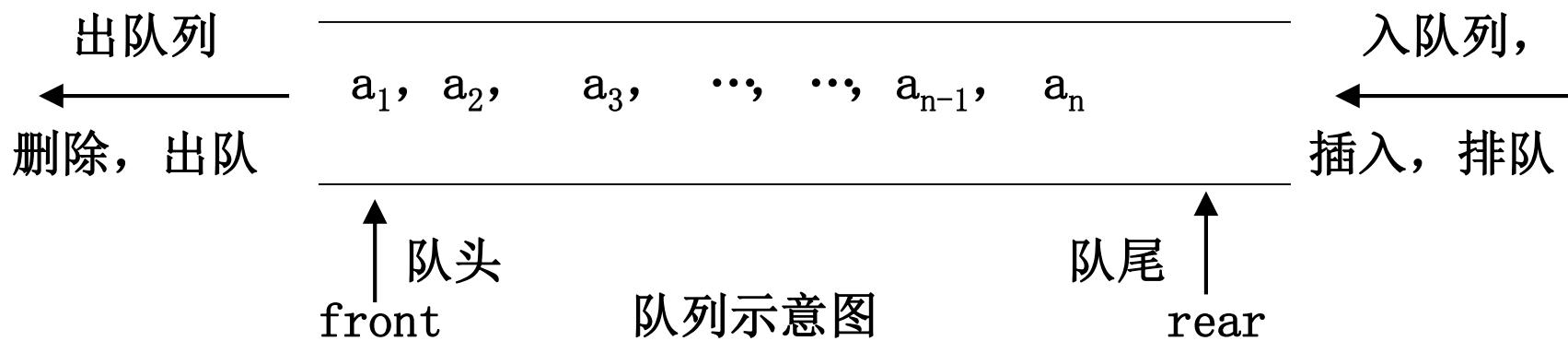
```
int mymax(int from, int to)
{
    int mid,t1,t2;
    if (from==to)
        return a[from];
    else
    {
        mid=(from+to)/2;
        t1=mymax(from,mid);
        t2=mymax(mid+1,to);
        return (t1>t2)?t1:t2;
    }
}
```

```
int findmax(int s[],int l,int r)
{
    int m,x,y;
    if(l==r)  return s[l];
    if(l+1==r)  return s[l]> s[r] ? s[l] : s[r];
    m=(l+r)/2;
    x= findmax (s,l,m);
    y= findmax(s,m+1,r);
    return x> y ? x : y;
}
```

2.4 排队或队列 (Queue)

队列是对线性表的插入和删除操作加以限定的另一种限定性数据结构。

[定义] 将线性表的插入和删除操作分别限制在表的两端进行，和栈相反，队列是一种先进先出（First In First Out，简称 FIFO 结构）的线性表。



操作: MakeNull(Q)、Front(Q)、EnQueue(x, Q)、DeQueue(Q)、Empty(Q)

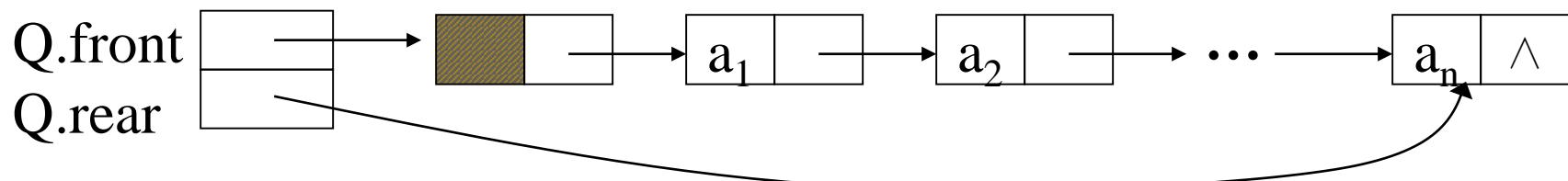
2.4.1 队列的指针实现

元素的“型”

```
struct celltype {  
    elementtype element ;  
    celltype *next ;  
};
```

队列的“型”：

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



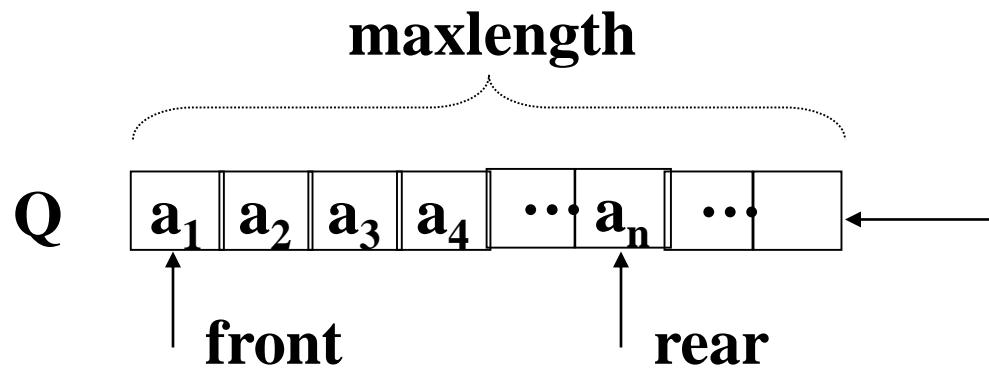
队列的指针实现示意图

2.4.2 队列的数组实现

队列的“型”：

```
struct Queue {  
    elementtype element [ maxlen ];  
    int front ;  
    int rear ;  
};
```

右图：队列Q状态1



下图：队列Q状态2



随着不断有元素出队和进队（插入和删除），队列的状态由1变成2；此时 a_n 占据队列的最后一个位置；第 $n+1$ 个元素无法进队，但实际上，前面部分位置空闲。

解决假溢出的方法有多种；一是通过不断移动元素位置，每当有元素出队列时，后面的元素前移一个位置，使队头元素始终占据队列的第一个位置。

第二种方法是采用**循环队列**。

插入元素：

$Q.\text{rear} = (Q.\text{rear} + 1) \% \text{ maxlen}$

删除元素：

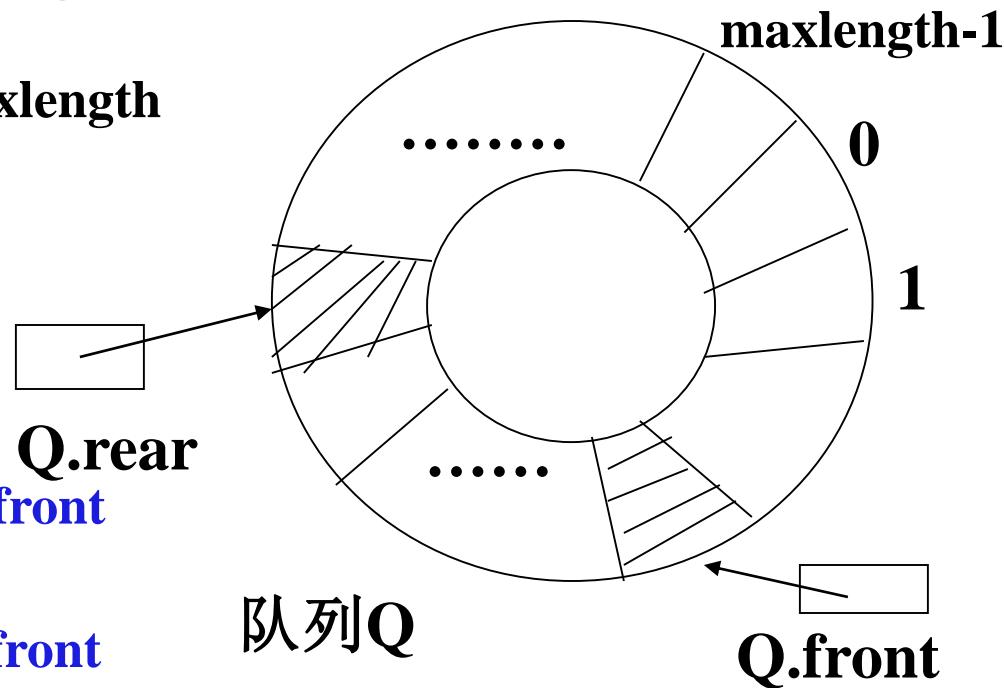
$Q.\text{front} = (Q.\text{front} + 1) \% \text{ maxlen}$

队空：

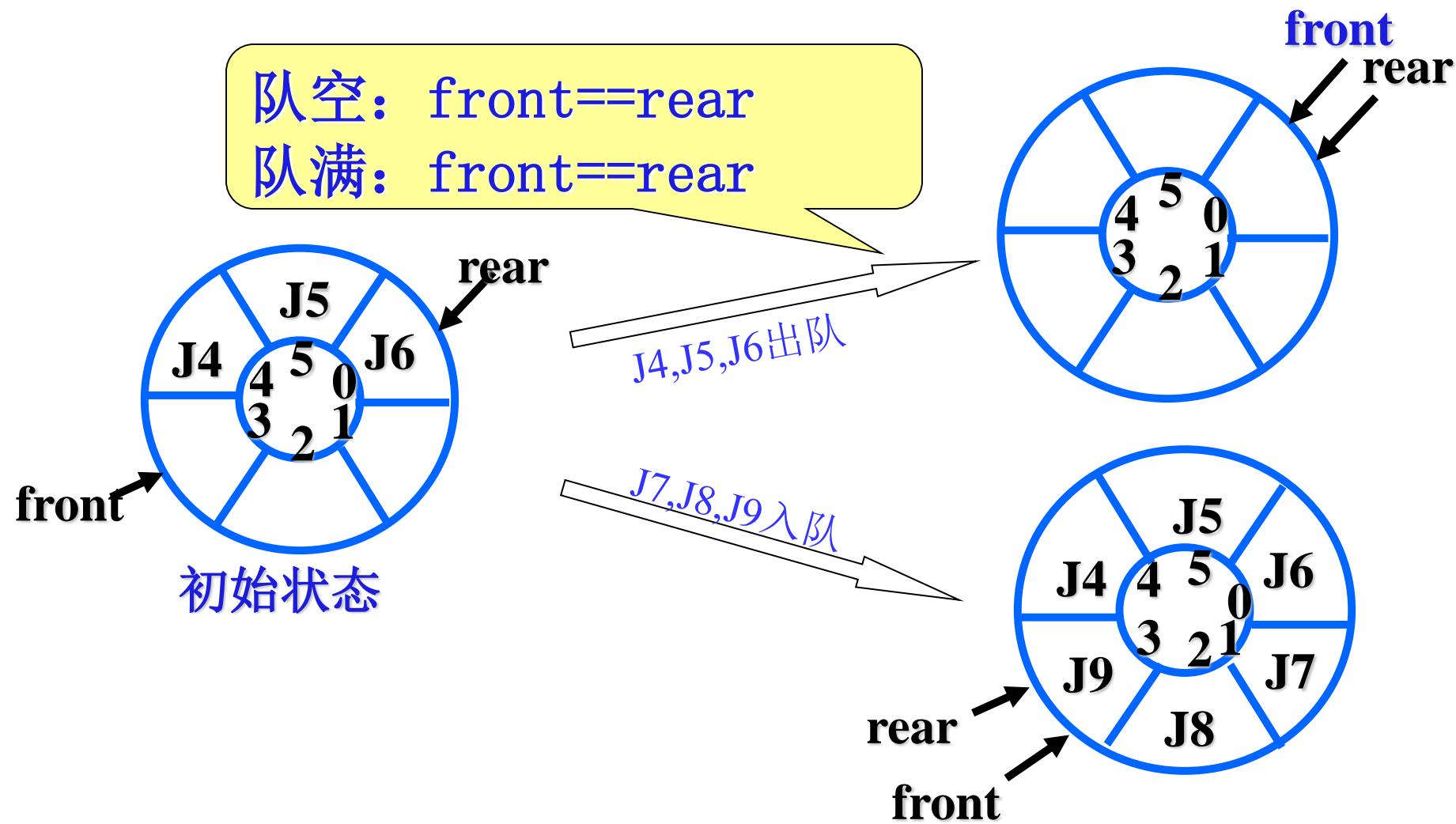
$(Q.\text{rear} + 1) \% \text{ maxlen} == Q.\text{front}$

队满：

$(Q.\text{rear} + 1) \% \text{ maxlen} == Q.\text{front}$



循环队列出入队图示



问题：如何解决循环队列中队空与队满状态相同？

方法一：约定队列头指针在队列尾指针的下一位位置上(即空出一个位置)；
方法二：另设一个标志位用以区别队空与队满两种状态；

结论：两种方法的代价是相同的。

操作：

```
int addone( int i )
{
    return ( ( i + 1 ) % maxlen );
}

① void MakeNull ( Queue &Q )
{
    Q.front = 0 ;
    Q.rear = maxlen - 1;
}
```

```
② boolean Empty(Queue Q )
{
    if ( addone(Q.rear) ==
        Q.front )
        return TRUE ;
    else
        return FALSE ;
}
```

操作: ③ elementtype Front(Queue Q)

```
{  if ( Empty( Q ) )
    return NULL ;
else
    return (Q.elements[ Q.front ] );
}
```

④ void EnQueue (Elementtype x, Queue &Q)

```
{ if ( addone ( addone( Q.rear ) ) ==Q.front )
    error ( “队列满” ) ;
else {
    Q.rear = addone ( Q.rear ) ;
    Q.elements[ Q.rear ] = x ;
}
}
```

⑤ void DeQueue (Queue &Q);

```
{  if ( Empty( Q ) )
    error ( “空队列” ) ;
else
    Q.front = addone ( Q.front ) ;
};
```

■ 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。

2.5 串(String)

2.5.1 抽象数据型串

串是线性表的一种特殊形式，表中每个元素的类型为字符型，是一个有限的字符序列。

串的基本形式可表示成： $S = 'a_1a_2a_3 \dots \dots \dots a_n'$ ；

其中：char a_i ； $0 \leq i \leq n$ ； $n \geq 0$ ；当 $n = 0$ 时，为空串。
 n 为串的长度；

C 语言中串有两种实现方法：

- 1、字符数组，如：char str1[10]；
- 2、字符指针，如：char *str2；

操作：

string MakeNull();
Boolean IsNull(S);
void In(S, a);
int Len(S);
void Concat(S1, S2);

string Substr(S, m, n);
Boolean Index(S, S1);

例1、将串T 插在串S 中第 i 个字符之后INSERT(S, T, i)。

```
void Insert( STRING &S, STRING T, int i )
{
    STRING t1, t2 ;
    if ( ( i < 0 ) || ( i > Len( S ) ) )
        error ‘指定位置不对’ ;
    else
        if ( IsNull( S ) ) S = T ;
        else
            if ( IsNull ( T ) )
                { t1 = Substr( S, 1, i ) ;
                  t2 = Substr( S, i + 1, Len( S ) );
                  S = Concat( t1, Concat( T, t2 ) );
                }
}
```

例2、从串 S 中将子串 T 删除Delete(S, T)。

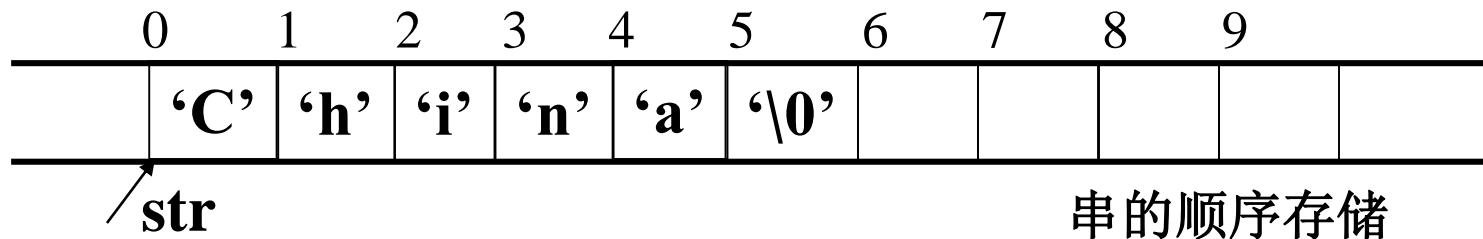
```
void Delete( STRING &S, STRING T )
{   STRING t1, t2 ;
    int m, n ;
    m = Index( S, T ) ;
    if ( m==0 )
        error ‘串S中不包含子串T’ ;
    else
    {
        n = Len( T ) ;
        t1 = Substr( S, 1, m - 1 ) ;
        t2 = Substr( S, m + n, Len( S ) ) ;
        S = Concat( t1, t2 ) ;
    }
}
```

2.5.2 串的实现

1、串的顺序存储

采用连续的存储空间（数组），自第一个元素开始，依次存储字符串中的每一个字符。

```
char str[ 10 ] =“China”;
```



操作： **MakeNull, IsNull, In, Len, Concat, Substr, Index**

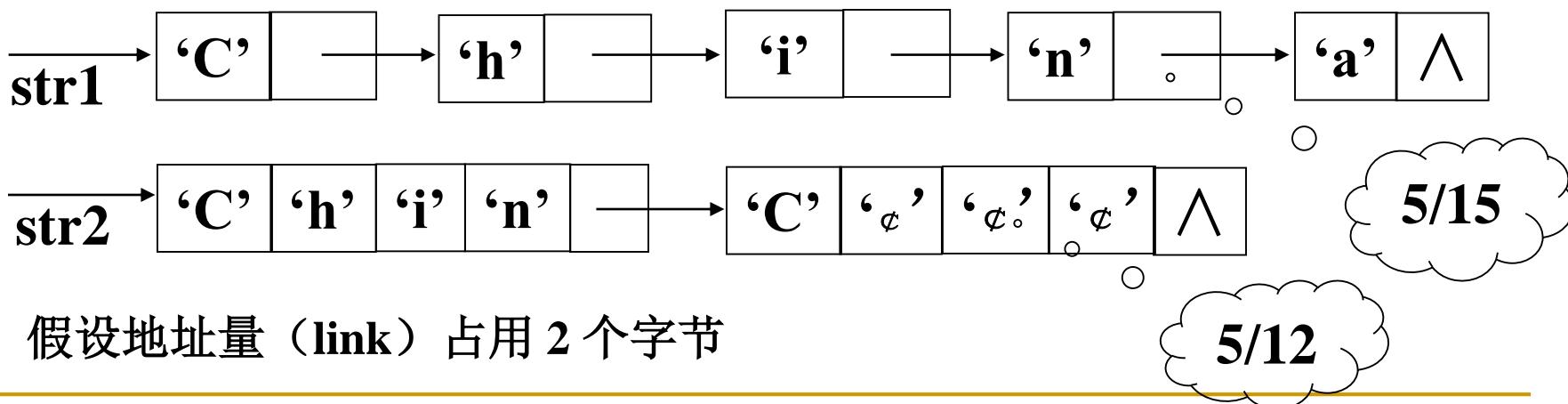
2、串的链式存储——一定长结点

构造线性链表，element类型为char，自第一个元素开始，依次存储字符串中的每一个字符。

2.5.2 串的实现

```
struct node {  
    char data ;  
    node *link ;  
};  
typedef node *STRING1;  
STRING1 str1 ;
```

```
struct node {  
    char data[4] ;  
    node *link ;  
};  
typedef node *STRING2 ;  
STRING2 str2 ;
```



2、串的链式存储

操作：

Index(S,S1)

若S1是S的子串则返回

S1首字符在S中的位置；

否则，返回0；

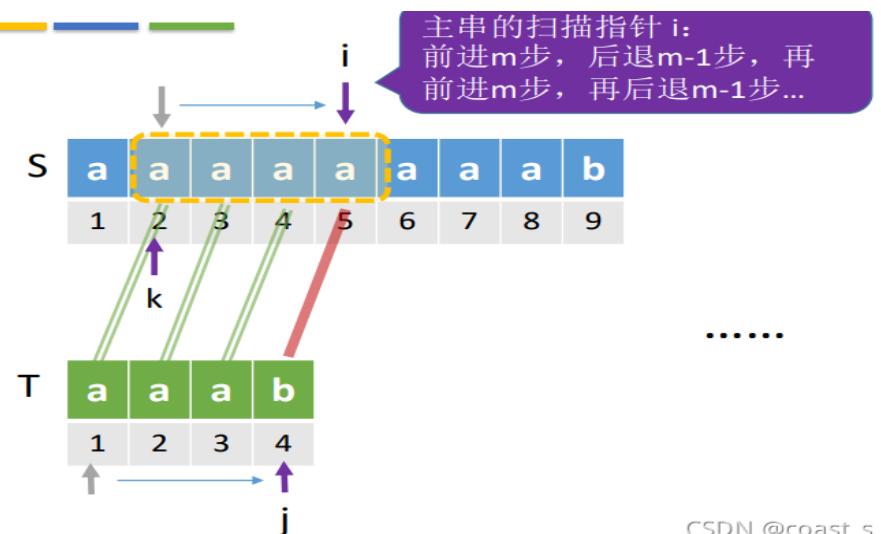
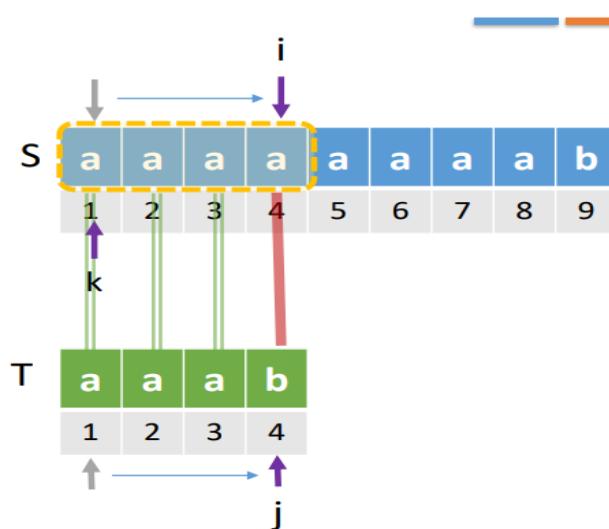
```
int Index ( STRING1 S, S1 )
{
    struct node *p, *q, *i ;
    int t ;
    if ( ( S1 != NULL ) && ( S != NULL ) )
        { t = 1 ; i = S ; q = S1 ; p=S;
        do {
            if ( p->data == q->data )
                { q = q->link ;
                  if ( q == NULL ) return( t ) ;
                  p = p->link ;
                }
            else
                { t = t + 1 ; i = i->link ;
                  p = i ; q = S1 ;
                }
            } while ( p != NULL ) ;
        }
    return 0 ;
}
```

3、模式匹配算法

1) 简单的匹配算法：一旦某个字符匹配失败，从头开始。
(本次匹配起点后一个字符开始)

52个零

模式串 T=00000001，指针要回溯45次。



设主串S=“ababcabcacbab”， 模式串T=“abcac”

第1趟匹配

主串	ab a babcabcacbab	i=3	
模式串	abc	j=3	匹配失败

第2趟匹配

主串	ab a babcabcacbab	i=2	
模式串	abc	j=1	匹配失败

第3趟匹配

主串	ababc a babcacbab	i=7	
模式串	abcac	j=5	匹配失败

第4趟匹配

主串	abab c babcacbab	i=4	
模式串	abc	j=1	匹配失败

第5趟匹配

主串	ababc a babcacbab	i=5	
模式串	abc	j=1	匹配失败

第6趟匹配

主串	ababcabcacbab	i=6	
模式串	abcac	j=6	匹配成功

```
int Index_BF ( char* S, char* T, int pos=1)
```

```
{ /* S为主串，T为模式，串的第0位置存放串长度；串采用顺序  
存储结构 */
```

```
i = pos;    j = 1;           // 从第一个位置开始比较
```

```
while (i<=S[0] && j<=T[0]) {
```

```
    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 > T[0]) return i-T[0];      // 返回与模式第一字符相等
```

```
else                                // 的字符在主串中的序号
```

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

Brute-Force算法的时间复杂性

主串S长n, 模式串T长m。可能匹配成功的位置 (1~n-m+1)。

①最好的情况下，模式串的第一个字符失配

设匹配成功在S的第i个字符，则在前*i-1*趟匹配中共比较了*i-1*次，第*i*趟成功匹配共比较了*m*次，总共比较了(*i-1+m*)次。所有匹配成功的可能共有*n-m+1*种，所以在等概率情况下的平均比较次数：

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

最好情况下算法的平均时间复杂性O(n+m)。

②最坏的情况下，模式串的最后1个字符失配

设匹配成功在S的第*i*个字符，则在前*i*-1趟匹配中共比较了(*i*-1)**m*次，第*i*趟成功匹配共比较了*m*次，总共比较了(*i***m*)次。共需要*n-m+1*趟比较，所以在等概率情况下的平均比较次数：

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

设*n>>m*，最坏情况下的平均时间复杂性为O(*n*m*)。

- ✓ 为什么BF算法时间性能低？

在每趟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。



KMP算法——改进的模式匹配算法

- ✓ 如何在匹配不成功时主串不回溯？

主串不回溯，模式就需要向右滑动一段距离。

- ✓ 如何确定模式的滑动距离？

利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离($\text{next}[j]$)后，继续进行比较

假设主串ababcabcacbab, 模式abcac, 改进算法的匹配过程如下：

第1趟匹配	a	b	a	b	c	a	b	c	a	c	b	a	b
	a	b	c										
第2趟匹配				↑ j=3									
				↓ i=3---7									
第3趟匹配	a	b	a	b	c	a	b	c	a	c	b	a	b
	a	b	c	a	c								
						↑ j=1							
						↓ i=7---11							
	a	b	a	b	c	a	b	c	a	c	b	a	b
	a	b	c	a	c								
						↑ j=2---6							

问题：

假定主串为 $S_1S_2\dots S_n$ ，模式串为 $P_1P_2\dots P_m$

无回溯匹配问题变为：当主串中的第*i*个字符和模式串中第*j*个字符出现不匹配，主串中的第*i*个字符应该与模式串中的哪个字符匹配（无回溯）？

假定主串中第*i*个字符与模式串第*k*个字符相比较，则应有

$$P_1P_2\dots P_{k-1} = S_{i-(k-1)}S_{i-(k-2)}\dots S_{i-1}$$

问题可能有多个*k*，取哪一个？

假设主串为 ‘ $s_1s_2\dots s_n$ ’，模式串为 ‘ $p_1p_2\dots p_m$ ’，当 $s_i \neq p_j$ 时，主串的第 i 个元素到底要与模式串的哪个元素进行比较，设为 p_k ，显然 $k < j$ 。如下：

主串 S

... $s_{i-j}s_{i-j+1}s_{i-j+2}\dots s_{i-k}s_{i-k+1}s_{i-k+2}\dots s_{i-2} s_{i-1} \textcolor{red}{s}_i \dots$

子串 P

..... $p_1 p_2 \dots p_{j-k} p_{j-k+1} p_{j-k+2} \dots p_{j-2} p_{j-1} \textcolor{red}{p}_j \dots$

下一次比较 p_k 的位置

|| || ||
 $p_1 p_2 \dots p_{k-2} p_{k-1} \textcolor{red}{p}_k$

说明： $p_{j-k+1}p_{j-k+2}\dots p_{j-1}=p_1p_2\dots p_{k-2}p_{k-1}$

■ $P_1 P_2 \dots P_{k-1} = P_{j-(k-1)} P_{j-(k-2)} \dots P_{j-1}$ 说明了什么？

- 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 仅与模式串 **自身 T** 有关，而与主串无关。

■ $P_1 P_2 \dots P_{k-1} = P_{j-(k-1)} P_{j-(k-2)} \dots P_{j-1}$ 的物理意义是什么？

从第1位往数右
 $k-1$ 位

从 $j-1$ 位往左
数 $k-1$ 位

■ 模式应该向右滑多远才是最高效率的？

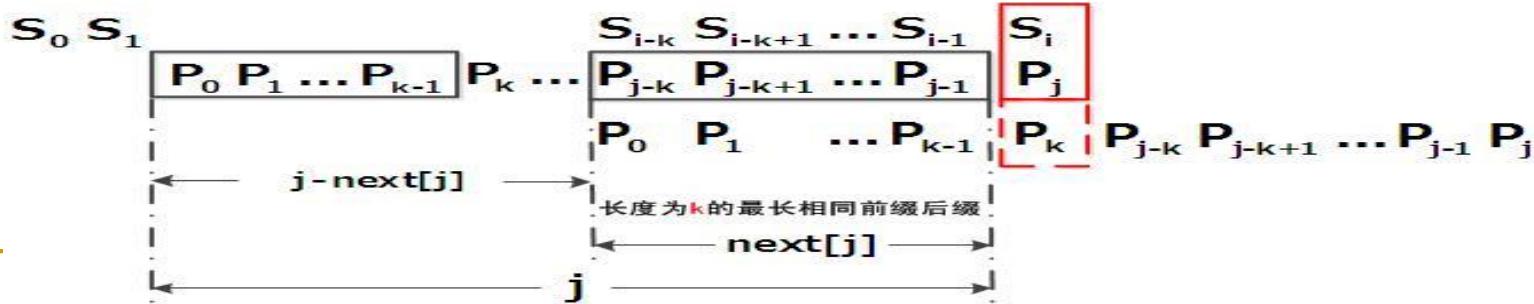
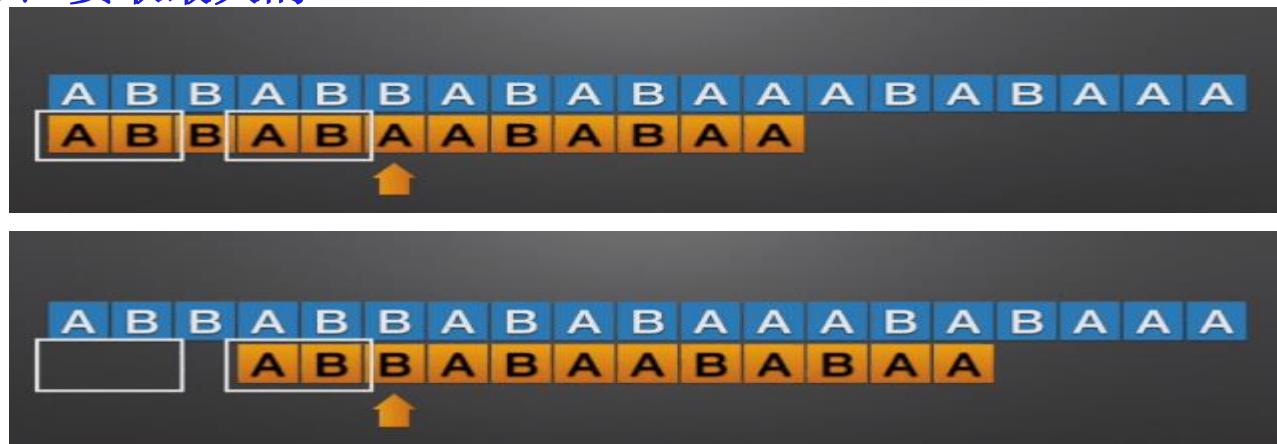
- $k = \max \{ k \mid 1 < k < j \text{ 且 } P_0 P_2 \dots P_{k-1} = P_{j-k} P_{j-(k-1)} \dots P_{j-1} \}$

■ 令 $k = \text{next}[j]$, 则:

$$\text{next}[j] = \begin{cases} -1 & \text{当 } j=0 \text{ 时} \quad // \text{不比较} \\ \max \{ k \mid 1 < k < j \text{ 且 } T_0 \dots T_{k-1} = T_{j-k} \dots T_{j-1} \} \\ 0 & \text{其他情况} \end{cases}$$

如果 $j \neq -1$, 且当前字符匹配失败 (即 $S[i] \neq P[j]$), 则令 i 不变, $j = \text{next}[j]$. 此举意味着失配时, 模式串 P 相对于文本串 S 向右移动了 $j - \text{next}[j]$ 位。

为什么 K 要取最大的?



- $k = \text{next}[j]$ 实质是找 $T_1 T_2 \dots T_{j-1}$ 中的最长相同的前缀 ($T_1 T_2 \dots T_{k-1}$) 和后缀 ($T_{j-(k-1)} T_{j-(k-2)} \dots T_{j-1}$)。
- 模式中相似部分越多，则 $\text{next}[j]$ 函数越大
 - 表示模式 T 字符之间的相关度越高，
 - 模式串向右滑动得越远，
 - 与主串进行比较的次数越少，时间复杂度就越低。
- 仍是一个模式匹配的过程，只是主串和模式串在同一个串 T 中。

寻找最长前缀后缀

"前缀"指除了最后一个字符以外，一个字符串的全部头部组合；

"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

模式串：“ABCDABD”，从左至右遍历整个模式串，子串的前缀后缀如下：

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

next函数的定义

$$\text{next}[j] = \begin{cases} -1 & \text{当 } j=0 \text{ 时} \quad // \text{不比较} \\ \max \{ k \mid 1 < k < j \text{ 且 } T_0 \dots T_{k-1} = T_{j-k} \dots T_{j-1} \} \\ 0 & \text{其他情况} \end{cases}$$

$\text{next}[j]=-1$ 表示根本不进行字符比较

$\text{next}[j]=0$ 表示从模式串头部开始进行字符比较

若模式串P为' abaabc'，由定义可得next函数值

j	0 1 2 3 4 5
模式串	a b a a b c
next[j]	-1 0 0 1 1 2

求模式串的next函数值举例

j	0	1	2	3	4	5	6	7	8
模式串	a	a	a	b	c	a	a	b	a

-1 0 1 2 0 0 1 2 0

j	0	1	2	3	4	5	6	7	8
模式串	a	b	c	a	b	c	a	c	b

-1 0 0 0 1 2 3 4 0

KMP算法手工模拟

j=0	1	2	3	4	5
-1	0	0	1	1	2

主串 $S = 'a\ c\ a\ b\ a\ a\ b\ a\ a\ b\ c\ a\ c\ a\ a\ b\ c'$

模式串 $P = 'a\ b\ a\ a\ b\ c'$

第一趟匹配: 主串 $a\ c\ a\ b\ a\ a\ b\ a\ a\ b\ c\ a\ c\ a\ a\ b\ c$
 模式串 $a\ \textcolor{red}{b}$

$\uparrow j=2 \quad \text{next}[2]=0$
 $\downarrow i=2$

第二趟匹配: 主串 $a\ c\ a\ b\ a\ a\ b\ a\ a\ b\ c\ a\ c\ a\ a\ b\ c$
 模式串 $\textcolor{red}{a}$

$\uparrow j=1 \quad \text{next}[1]=-1$
 $\downarrow i=3 \rightarrow \downarrow i=8$

第三趟匹配: 主串 $a\ c\ a\ b\ a\ a\ b\ a\ a\ b\ c\ a\ c\ a\ a\ b\ c$
 模式串 $a\ b\ a\ a\ b\ \textcolor{red}{c}$

$\uparrow j=1 \rightarrow \uparrow j=6 \quad \text{next}[6]=2$
 $\downarrow i=8 \rightarrow \downarrow i=12$

第四趟匹配: 主串 $a\ c\ a\ b\ a\ a\ b\ a\ a\ b\ c\ a\ c\ a\ a\ b\ c$
 模式串 $(a\ b)\ a\ a\ b\ c$

$\uparrow j=3 \rightarrow \uparrow j=7$

如何求next函数

- 当 $j=0$ 时， $\text{next}[j]=-1$ ；

// $\text{next}[j]=-1$ 表示根本不进行字符比较

- 当 $j>1$ 时， $\text{next}[j]$ 的值为：模式串的位置从1到 $j-1$ 构成的串中所出现的首尾相同的子串的最大长度。
- 当无首尾相同的子串时 $\text{next}[j]$ 的值为0。

// $\text{next}[j]=0$ 表示从模式串头部开始进行字符比较

j	0	1	2	3	4	5	6	7
模式串	b	a	b	b	a	b	a	b

-1 0 0 1 1 2 3 2

求next函数

```
void GetNext(char* p, int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀， p[j]表示后缀
        if (k == -1 || p[j] == p[k])
            {++k; ++j; next[j] = k; }
        else
            { k = next[k]; }
    }
}
```

k 是在模式串当前下标之前已经匹配的前缀后缀的长度。 $\text{Next}[i]$ 表示在下标*i* 之前已经匹配的前缀后缀的长度。

如果 $s[j] == s[k]$, 说明当前前缀和后缀可以匹配到长度为 k 的相同字符;

如果 $s[j] != s[k]$, 就说明了当前前缀后缀无法匹配到长度为 $k+1$ 的相同字符, 需要找有没有更短的可能, 所以就有了 $k = \text{Next}[k]$ 。

A	B	C	D	A	B	C	D	A	B	D
Next:	-1	0	0	0	0	1	2	3	4	5
数组下标:	0	1	2	3	4	5	6	7	8	9

A	B	C	D	A	B	C	D	A	B	D
Next:	-1	0	0	0	0	1	2	3	4	5
数组下标:	0	1	2	3	4	5	6	7	8	9

第六位不能匹配第十位, 那就往回退一下, 看一下能不能找一个短一点的前缀来匹配第十位。

https://blcq.csdn.net/qq_43227036

KMP算法

```
int Kmp (char* s, char* p)
{
    int i = 0; int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        if (j == -1 || s[i] == p[j]) { i++; j++; }
        else
            { j = next[j] }      //next[j]即为j所对应的next值
    }
    if (j == pLen) return i - j;
    else return -1;
}
```

思考：若模式 ‘aaaab’ 在和主串 ‘aaabaaaaab’ 匹配时， next() 函数是否需要修正？

nextval算法的描述：

基于next的算法进行的，弥补next算法的缺陷的。主要解决了模式串中大量连续重复的字符，nextval函数减少了主串的无用比较的次数。

假设主串为：S='aaabaaaaab' 子串为:T='aaaab'

根据next值，next[3]=2，需要将第2位与该位置对齐：

文本串： a a a b a a a a b

模式串： a a a a b next[2]=1

j 0 1 2 3 4

文本串： a a a b a a a a b

模式串： a a a a b, next[1]=0,

模式串 a a a a b

文本串： a a a b a a a a b

next[] -1 0 1 2 3

模式串： a a a a b

next[0]=-1, i+1, j=0

文本串： a a a b a a a a b

模式串： a a a a b

j 0 1 2 3 4

如果位置k的元素与next[k]元素相同时，

nextval[k]=nextval[next[k]]

模式串 a a a a b

如果位置k的元素与next[k]元素不同时，

nextval[k]= next[k]

nextval[] -1 -1 0 1 2

```
int get_nextval(SString T,int &nextval[ ]){  
    //求模式串T的next函数修正值并存入数组nextval。  
    k=-1; nextval[0]=-1; j=0;  
    while(i<T[0]) {  
        if(k== -1||T[k]==T[j]) {  
            ++k; ++j;  
            if (T[k]!=T[j]) nextval[j]=k;  
            else nextval[k]=nextval[j];  
        }  
        else k=nextval[k];  
    }  
    } //get_nextval
```

字符串匹配的算法包括：

1. 朴素算法 (Naive Algorithm)；
2. Knuth–Morris–Pratt 算法 (即 KMP Algorithm)；
3. Rabin–Karp 算法；
4. 有限自动机算法 (Finite Automation)；
5. Boyer–Moore 算法；
6. Simon 算法；
7. Colussi 算法；
8. Galil–Giancarlo 算法；
9. Apostolico–Crochemore 算法、Horspool 算法；
10. Shift–Or 算法；
11. Sunday 算法等。

课外作业：

写一篇综述论文，分别描述上面各算法（1,2除外）的思想，并实现2-3个算法。

2.6 数组(ARRAY)

2.6.1 抽象数据型数组

- 数组是由下标(index) 和值(value) 组成的序对(index, value) 的集合。
- 也可以定义为是由相同类型的数据元素组成有限序列。
- 数组在内存中是采用一组连续的地址空间存储的，正是由于此种原因，才可以实现下标运算。
- 所有数组都是一个一维向量。

数组1: ($a_1, a_2, a_3, \dots; a_i, \dots; a_n$);

数组2: ($a_{11}, \dots; a_{1n}, a_{21}, \dots; a_{2n}, \dots; a_{ij}, \dots; a_{m1}, \dots; a_{mn}$);
 $1 \leq i \leq m, 1 \leq j \leq n$;

数组3: ($a_{111}, \dots; a_{11n}, a_{121}, \dots; a_{12n}, \dots; a_{ijk}, \dots; a_{mn1}, \dots; a_{mnp}$);
 $1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq p$;

2.6.1 数组的抽象数据型

n 维数组中含有 $\prod b_i$ 个数据元素，每个元素都受着 n 个关系的约束。

在每个关系中，元素 $a_{j_1 j_2 \dots j_n}$ ($0 \leq j_i \leq b_i - 2$) 都有一个直接后继元素。

因此，数组仍是一种特殊形式的线性表。对二维数组可以理解成一维数组，其每个元素又是一个一维数组；以此类推，所谓 n 维数组同样如此。

操作： **Create()**；建立一个空数组；**int A[][]**

Retrieve(array, index)；返回第 $index$ 个元素；**A[i][j]**

Store (array, index, value)；

在数组 **array** 中，为第 $index$ 个元素赋值 **value**；**A[i][j]=6**

注：由于高级语言中都提供了数组，本课略去操作。

2.6.2 数组的实现

1、数组的顺序存储

数组的顺序表示，指的是在计算机中，用一组连续的存储单元来实现数组的存储。目前的高级程序设计语言都是这样实现的。

两种存储方式：

- 一是按行存储（C语言、PASCAL等）
- 二是按列存储（FORTRAN等）

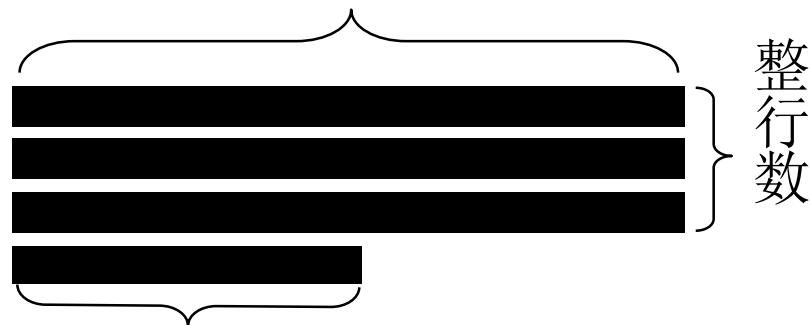
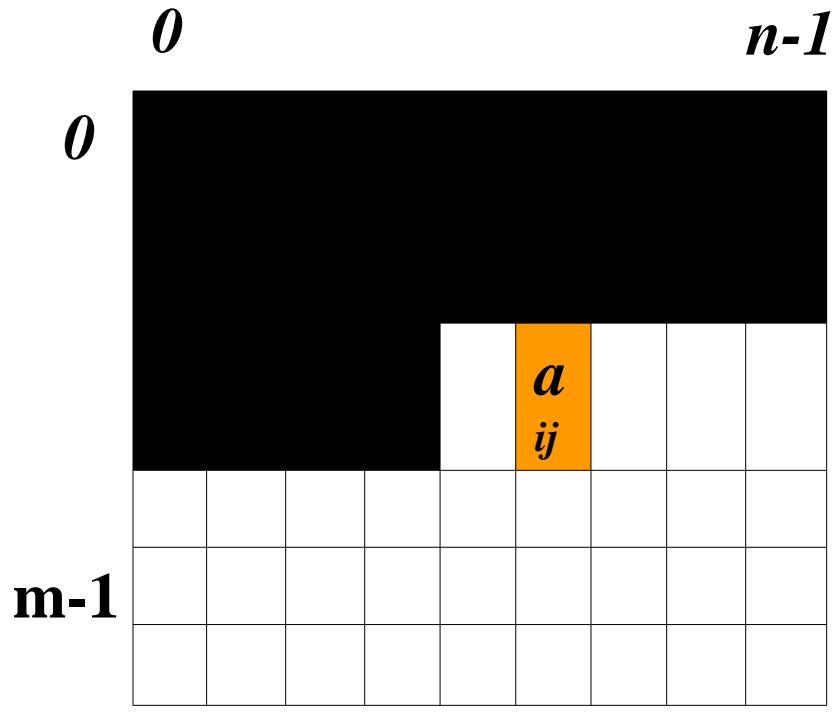
elementtype A[2][3] ;

$$A = \begin{bmatrix} A[0][0] & A[0][1] & A[0][2] \\ A[1][0] & A[1][1] & A[1][2] \end{bmatrix}$$

A[0][0]	A[0][0]	第一列
A[0][1]	A[1][0]	
A[0][2]	A[1][1]	
A[1][0]	A[1][1]	第二列
A[1][1]	A[0][2]	
A[1][2]	A[1][2]	

按行优先存储的寻址----二维数组

每行元素个数

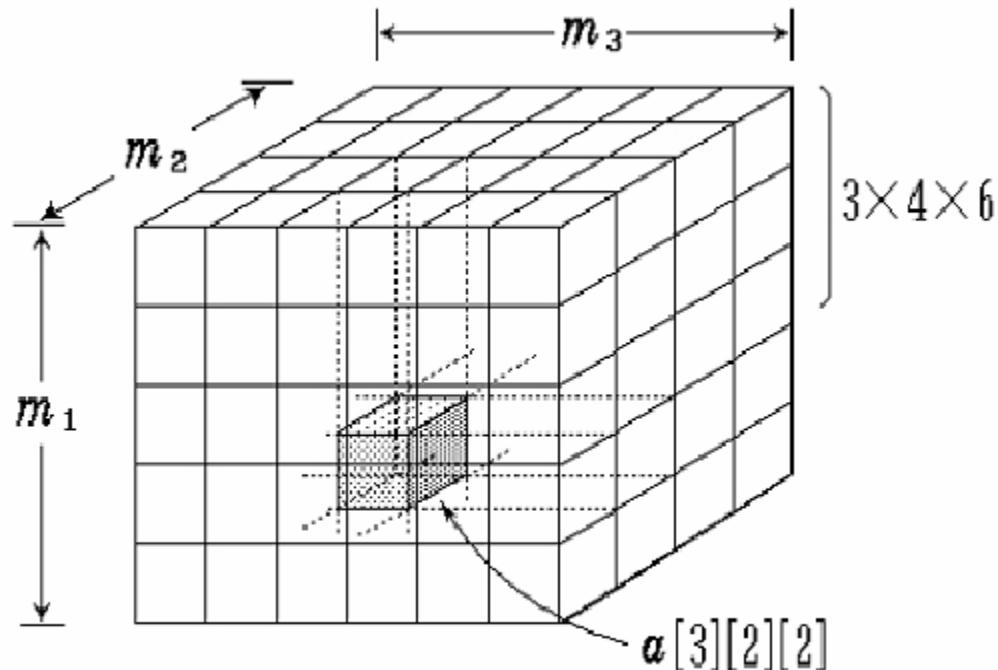
本行中 a_{ij} 前面的元素个数 a_{ij} 前面的元素个数 $=$ 整行数 \times 每行元素个数 + 本行中 a_{ij} 前面的元素个数

$$= i \times n + j$$

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

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。



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

2. 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。

稀疏矩阵：矩阵中有很多零元素。

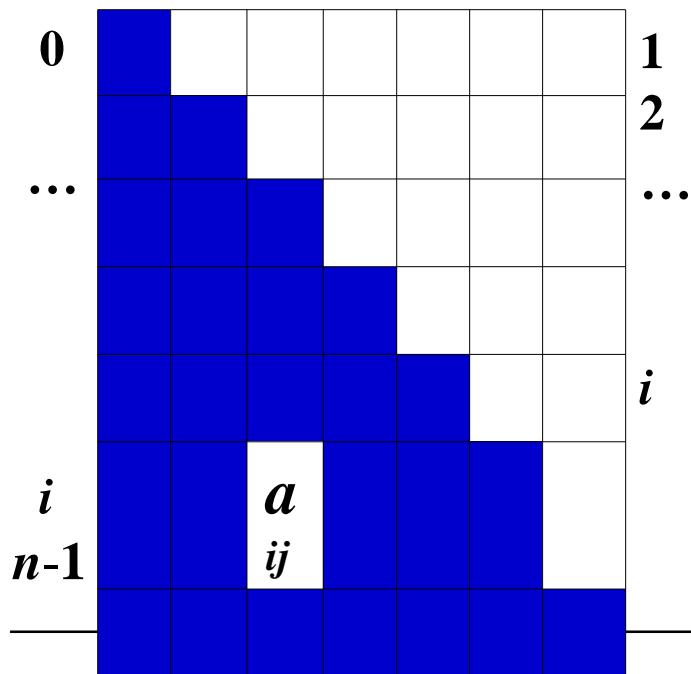
- **压缩存储的基本思想是：**
 - ✓ 为多个值相同的元素只分配一个存储空间；
 - ✓ 对零元素不分配存储空间。

对称矩阵的压缩存储

- 对称矩阵特点: $a_{ij}=a_{ji}$
- 只存储下三角部分的元素。

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

0 ... j ... $n-1$



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

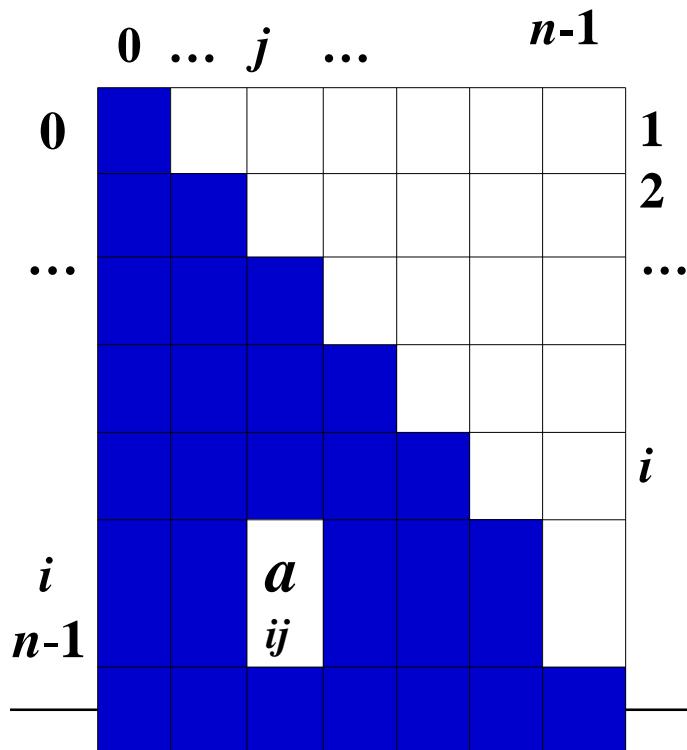
$$k = i \times (i+1)/2 + j \quad (i \geq j)$$

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

► 三角矩阵的压缩存储----下三角矩阵

- 只存储下三角部分的元素。
- 对角线上方的常数只存一个

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



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：

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

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

稀疏矩阵的压缩存储 ----三元组顺序表

稀疏矩阵中的非零元素的分布没有规律

如何只存储非零元素?

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; //总行号, 列号, 非0元素个数  
} TSMatrix;
```

15	0	0	22	0	-15
0	11	3	0	0	0
0	0	0	6	0	0
0	0	0	0	0	0
91	0	0	0	0	0

稀疏矩阵的压缩存储 ----三元组顺序表

□ 如何存储三元组表?

- 按行优先的顺序排列成一个线性表。

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	item
0	0	0	15
1	0	3	22
2	0	5	-15
3	1	1	11
4	1	2	3
5	2	3	6
6	4	0	91
7 (非零元个数)			
5 (矩阵的行数)			
6 (矩阵的列数)			

$$M = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix} \quad T = M' = \begin{pmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

a.data

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

b.data

3、数组的链接式存储

多维数组，特别是稀疏矩阵可以采用链接式存储。

结点形式：

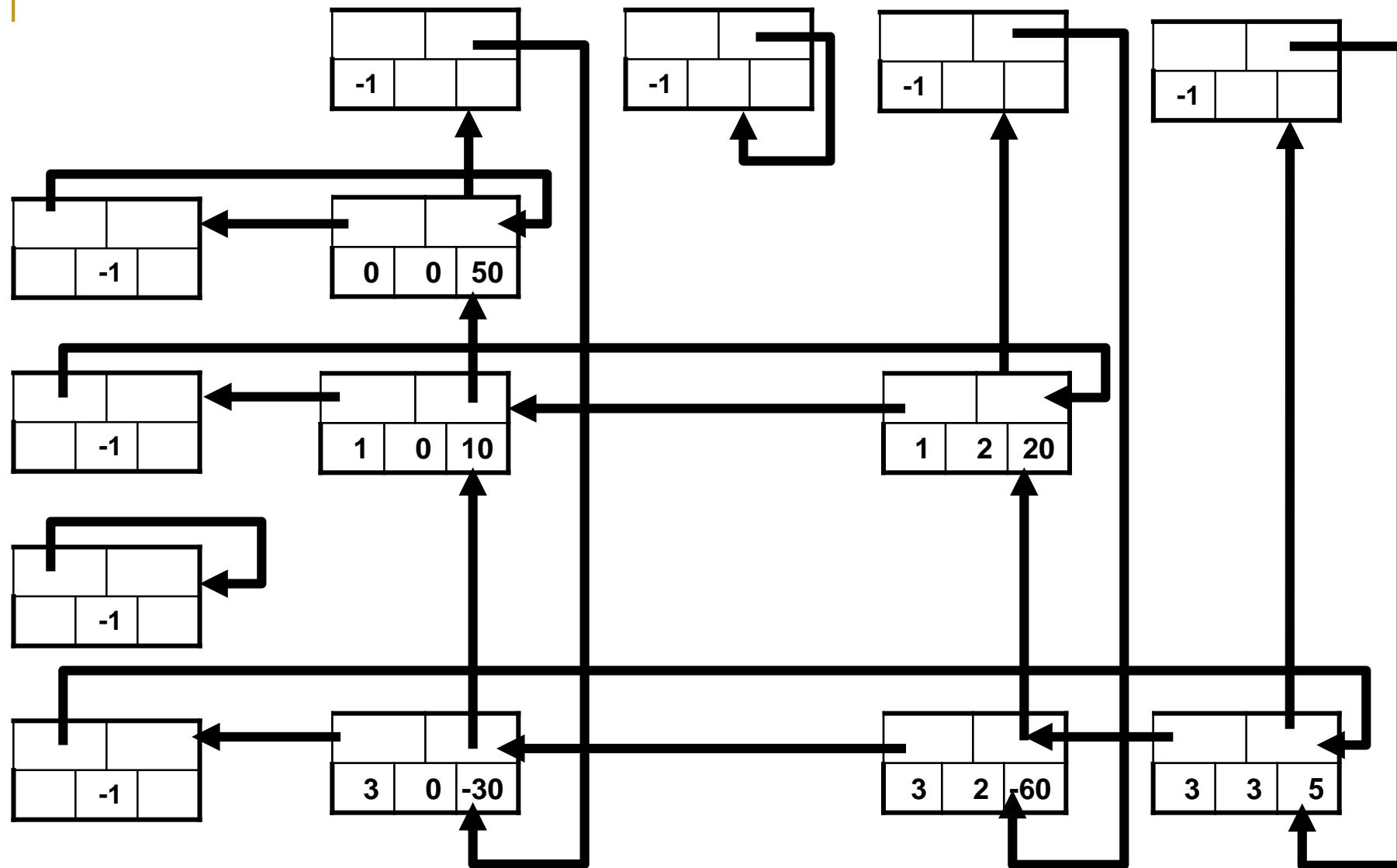
LEFT	UP	
ROW	COL	VOL

结点类型：

```
struct node {  
    node *LEFT, *UP;  
    int ROW, COL;  
    valuetype VAL;  
};
```

二维数组A（矩阵）的链接表示如下。

$$A = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -3 & 0 & -6 & 0 & 5 \end{pmatrix}$$



2.7 广义表 (General Lists)

广义表示线性表的一种推广结构，线性表要求是相同的元素类型，而广义表中的元素可以取不同类型，可以是最基本的不可再分割的“原子”，也可以是广义表本身。

广义表是由零个原子，或若干个原子或若干个广义表组成的有穷序列。

通常将广义表表示为：

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

其中， A 是名称，元素个数 n 是表的长度；若 a_i 不是原子，则称其为 A 的子表。

注意广义表的递归性。

例如：

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

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。

结论：

- ① 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；
- ② 一个广义表可以被其他广义表所共享。
- ③ 广义表是一个递归的表，即广义表可以是其本身的子表。

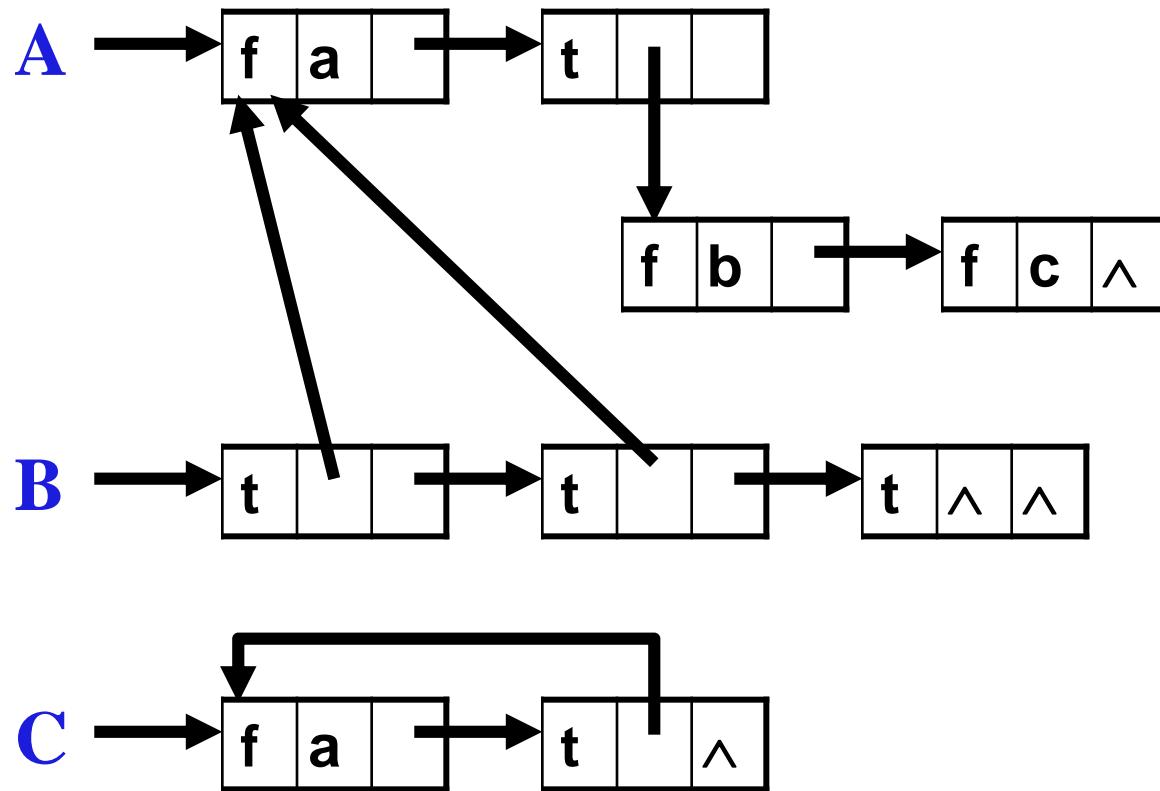
操作：

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

广义表的存储结构

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

广义表实例



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

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

$$C = (a, C)$$

广义表的操作

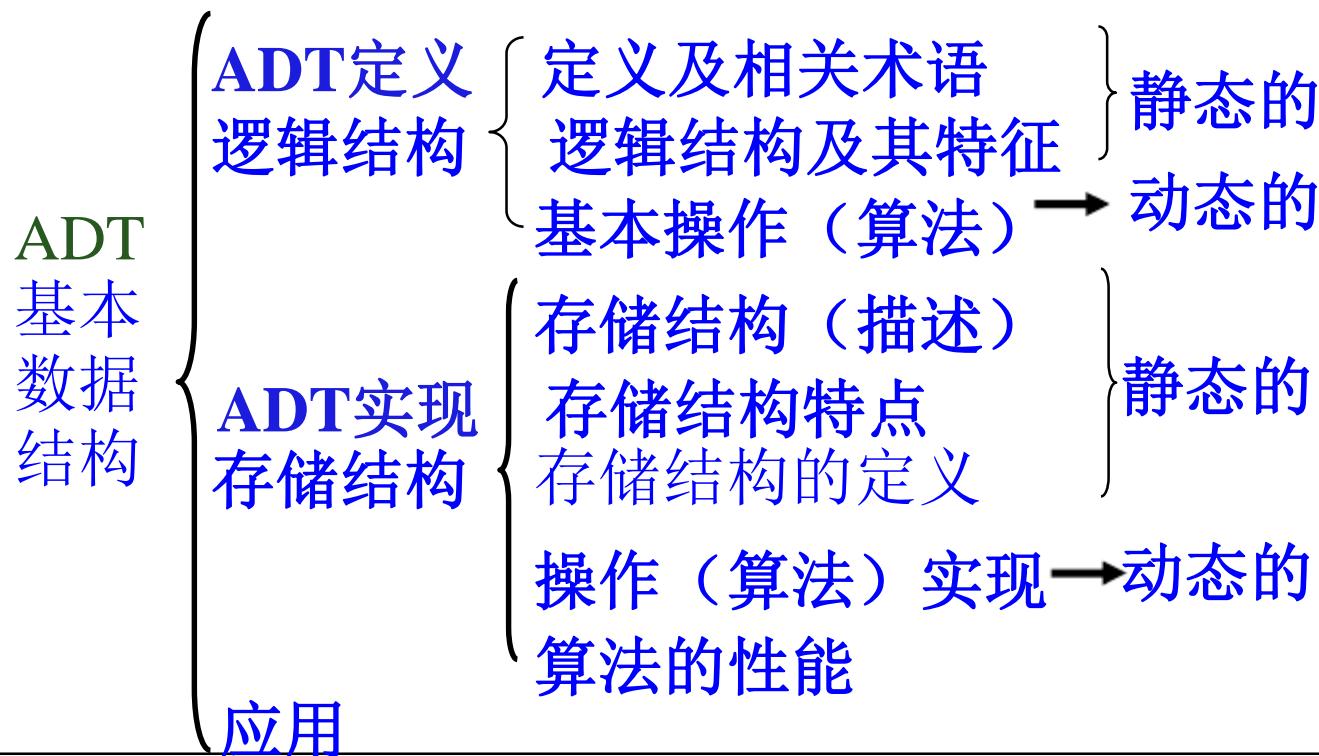
```
boolean Equal( listpointer S, T )
{  boolean x, y ;
   y = False ;
   if ( ( S == NULL ) && ( T == NULL ) )
      y = True ;
   else if ( ( S != NULL ) && ( T != NULL ) )
      if ( S→tag == T→tag )
         { if ( S→tag == FALSE
              { if ( S→element.data == T→element.data )
                  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 ;
        }
```

本章小结

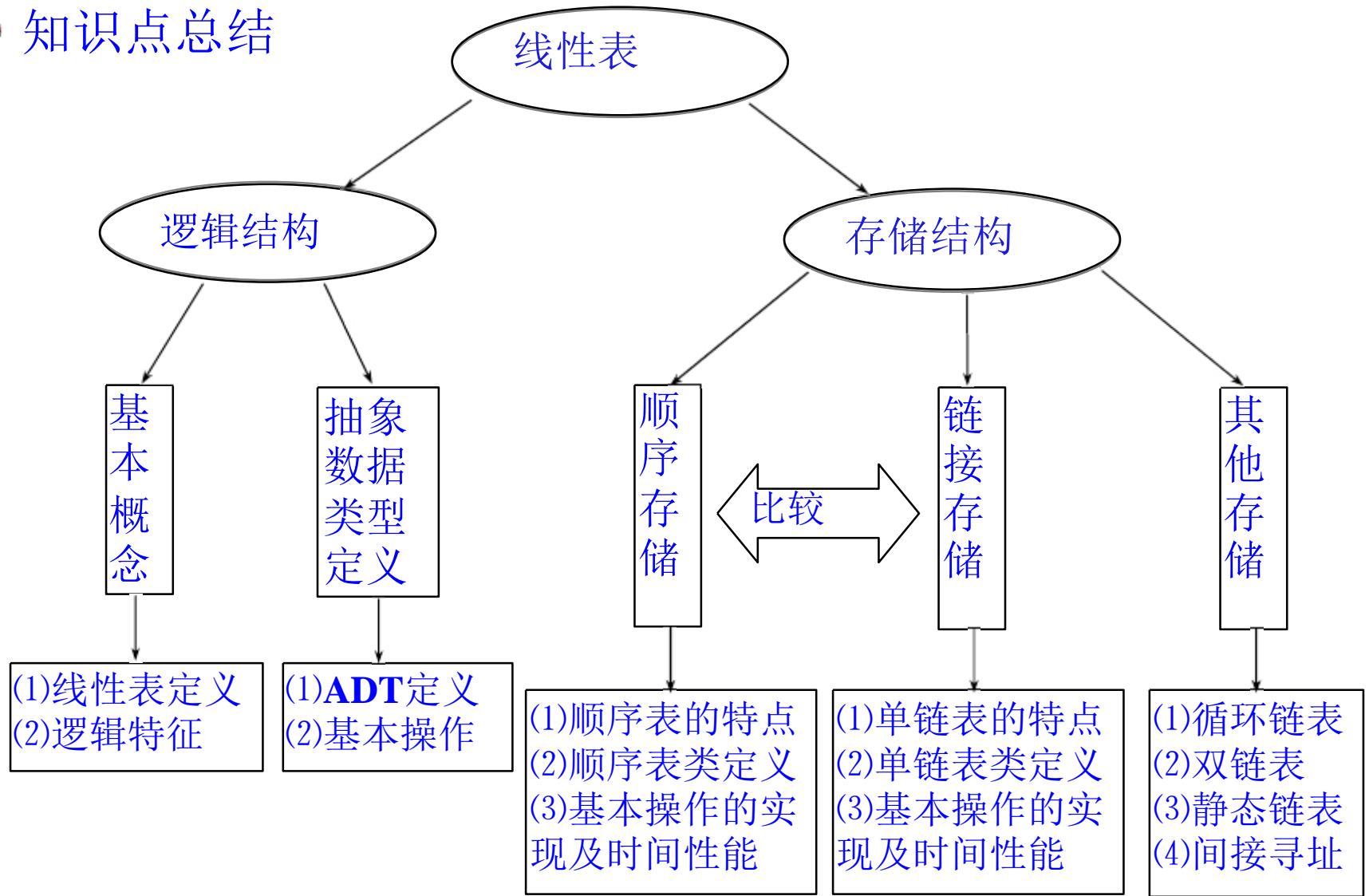
知识点：

- 线性表、栈、队列、串、（多维）数组、广义表

知识点体系结构

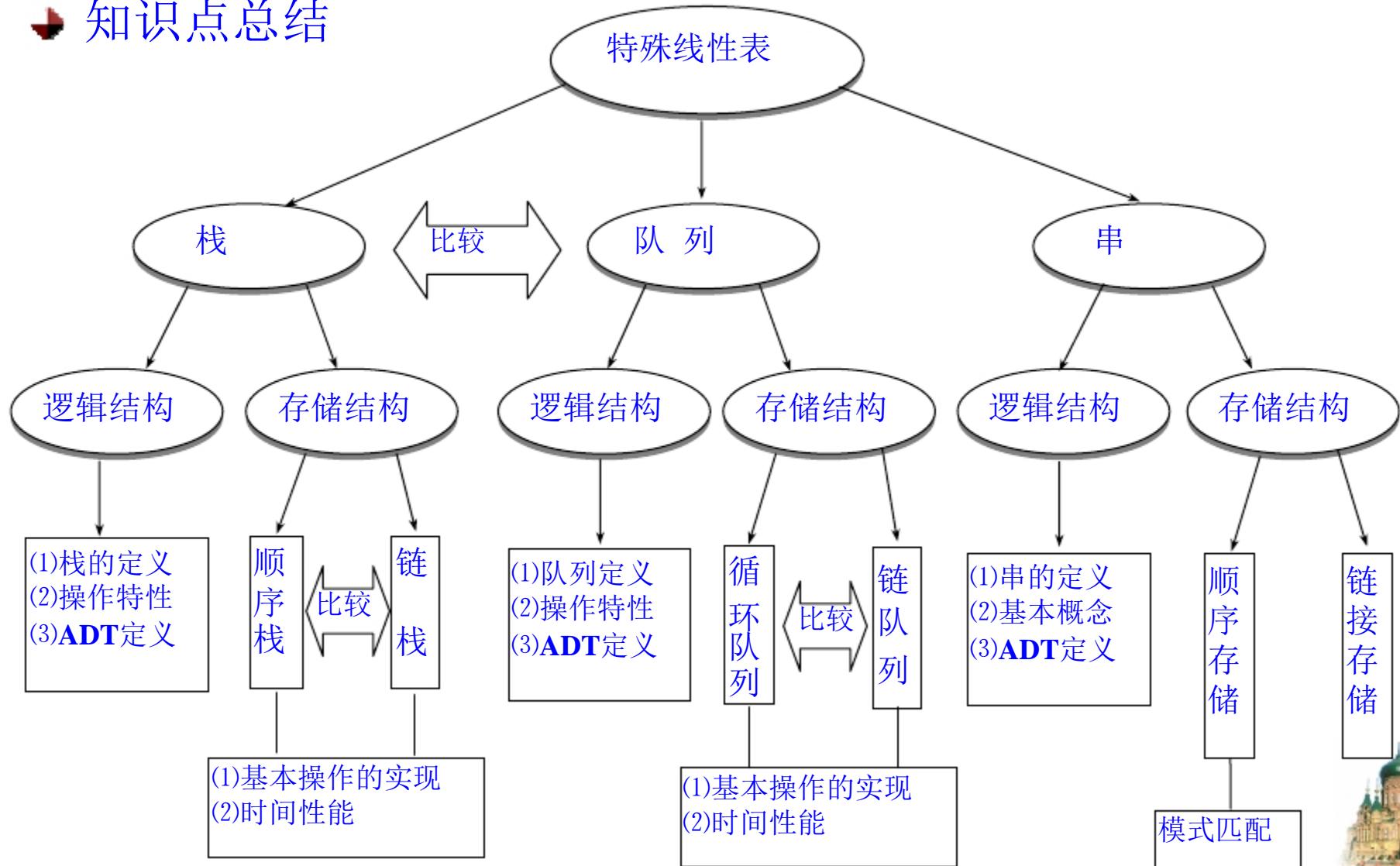


知识点总结





知识点总结





知识点总结

