第2章 线性表

2.1 线性表的概念及运算

1. 线性表的定义

一个线性表是具有n个数据元素的有限序列。

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

2. 线性表的长度

线性表中元素的个数n (n>=0), n=0时,称为空表。

3. 位序

a_i是第i个元素，称i为数据元素a_i在线性表中的位序 。

4. 线性表的逻辑结构

例子：

- 英文字母表(A, B, ..., Z)；
- 车辆登记表。

车牌号	车　　名	车　　型	颜　　色
A13850	奥迪	卧车	黑色
B49271	福田	小卡	白色
A66789	东风	大卡	绿色
⋮	⋮	⋮	⋮

5. 线性表的特点

- 同一性：线性表由同类数据元素组成，每一个a_i必须属于同一数据对象。
- 有穷性：线性表由有限个数据元素组成， 表长度就是表中数据元素的个数。
- 有序性：线性表中相邻数据元素之间存在着序偶关系$<a_i, a_{i+1}>$。

6. 线性表的基本运算

- 初始化 InitList（&L） 建立一个空表。
- 求表长 ListLength（L） 返回线性表的长度。
- 读表元素 GetElem（L, i, &e) 用e返回L中第i个数据元素的值。
- 定位 LocateElem（L, e, compare()） 返回满足关系的数据元素的位序。
- 插入 ListInsert （&L, i, e） 在L中第i个位置之前插入新的数据元素e，线性表的长度增1。
- 删除 ListDelete （&L, i, & e） 删除L的第i个位置上的数据元素e，线性表的长度减1。
- 输出 ListDisplay （L） 按前后次序输出线性表的所有元素。

练习1：两个线性表LA和LB分别表示两个集合A和B，现求一个新的集合A=A∪B。

```
void union( List &La, List Lb)
{
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    for( i=1; i<=Lb_len; i++){
        GetElem(Lb, i, e);
        if(!LocateElem(La, e, equal))
            ListInsert(La, ++La_len, e);
    }
}
```

O(ListLength(La)×ListLength(Lb))

练习2：

两个线性表LA和LB中的数据元素按值非递减有序排列，现将LA和LB归并为一个新的线性表，LC中的数据元素仍按值非递减有序排列。

LA = (3, 5, 8, 11)

LB = (2, 6, 8, 9, 11, 15, 20)

LC = (2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)

$$c = \begin{cases} a, \text{当} a \le b \text{时} \\ b, \text{当a>b时} \end{cases}$$

```
void MergeList( List La, List Lb, List &Lc)
{    InitList(Lc);
     La_len = ListLength(La);  Lb_len = ListLength(Lb);
     i=j=1; k=0;
     while( (i<=La_len) && (j<=Lb_len)){
          GetElem(La, i, a); GetElem(Lb, j, b);
          if(a<=b) { ListInsert(Lc, ++k, a); ++i; }
          else { ListInsert(Lc, ++k, b); ++j; }
     }
     while(i<=La_len){
          GetElem(La, i++, a); ListInsert(Lc, ++k, a);}
     while(j<=Lb_len){
          GetElem(Lb, j++, b); ListInsert(Lc, ++k, b);}
}
```

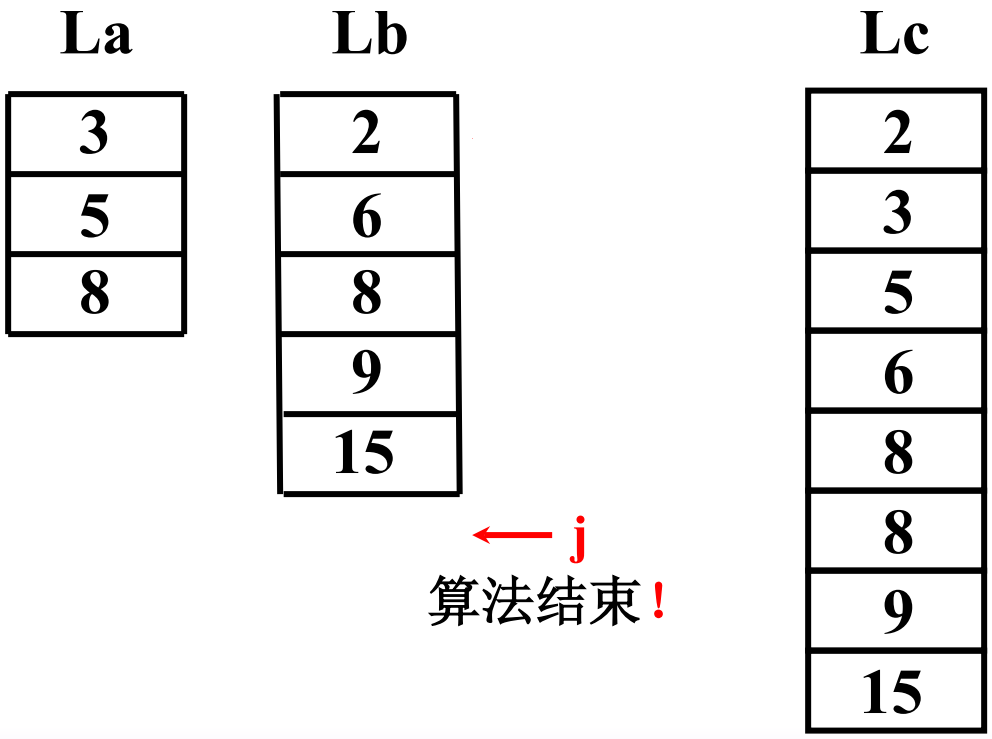
O(ListLength(La)+ListLength(Lb))

例，La = (3, 5, 8)

Lb = (2, 6, 8, 9, 15)

构造 Lc = (2, 3, 5, 6, 8, 8, 9, 15)

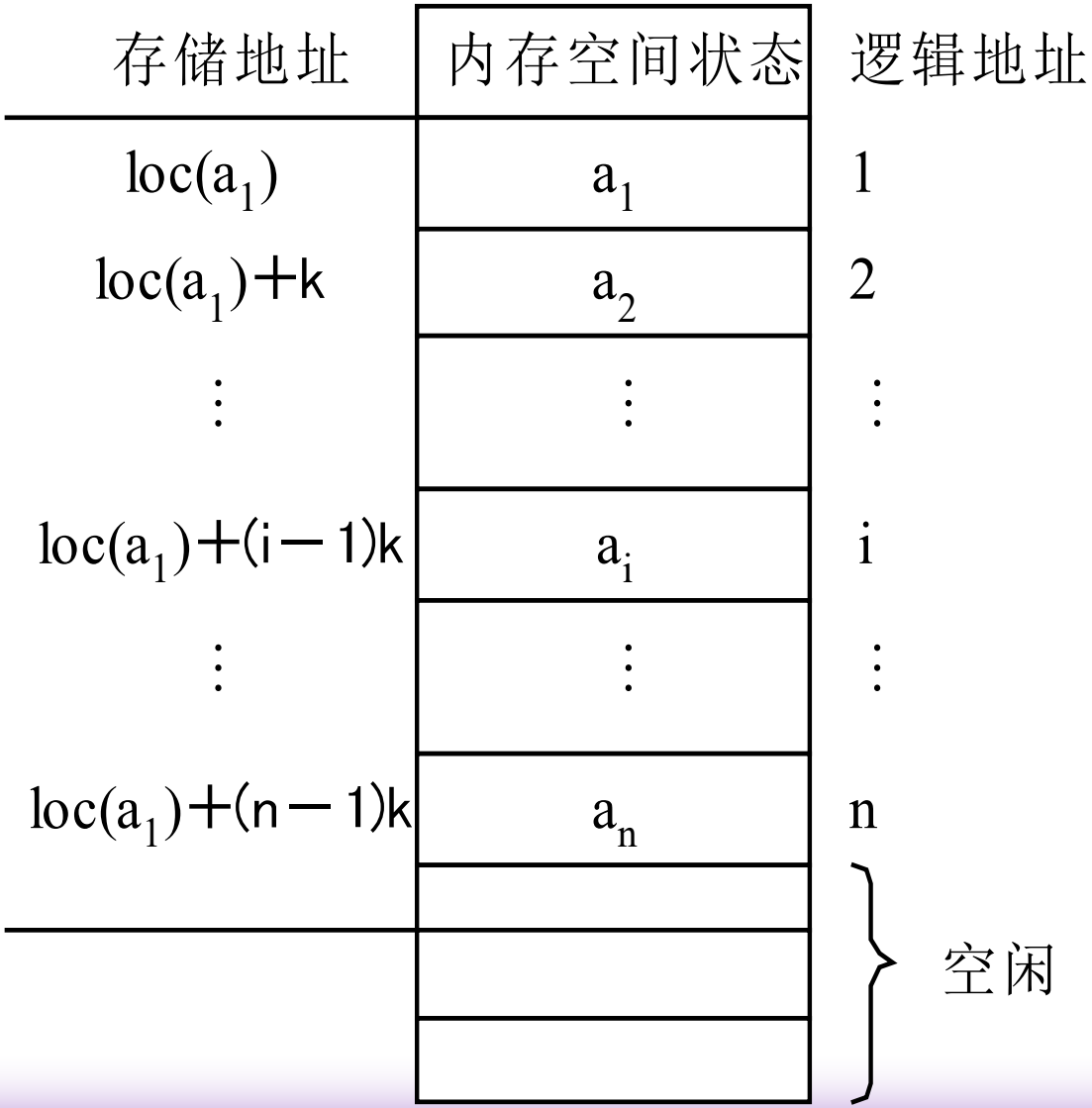
首先，La_len = 3；Lb_len = 5；



2.2 线性表的顺序表示和实现

1. 顺序表：

按顺序存储方式构造的线性表。



假设线性表中有n个元素，每个元素占k个单元，第一个元素的地址为$loc(a_1)$，则可以通过如下公式计算出第i个元素的地址$loc(a_i)$：

$$loc(a_i) = loc(a_1) + (i-1) \times k$$

其中$loc(a_1)$称为基地址。

2. 顺序表的特点：

- 逻辑结构中相邻的数据元素在存储结构中仍然相邻。
- 线性表的顺序存储结构是一种随机存取的存储结构。

3. 顺序表的描述：

```
typedef  struct
{
    ElemType  *elem;    //ElemType elem[MAXSIZE];
    int               length;  //当前长度
    int               listsize;  //分配的存储容量
 } SqList;
```

```
typedef   # ElemType;    #为根据具体问题确定的数据类型
typedef   int   Status;
```

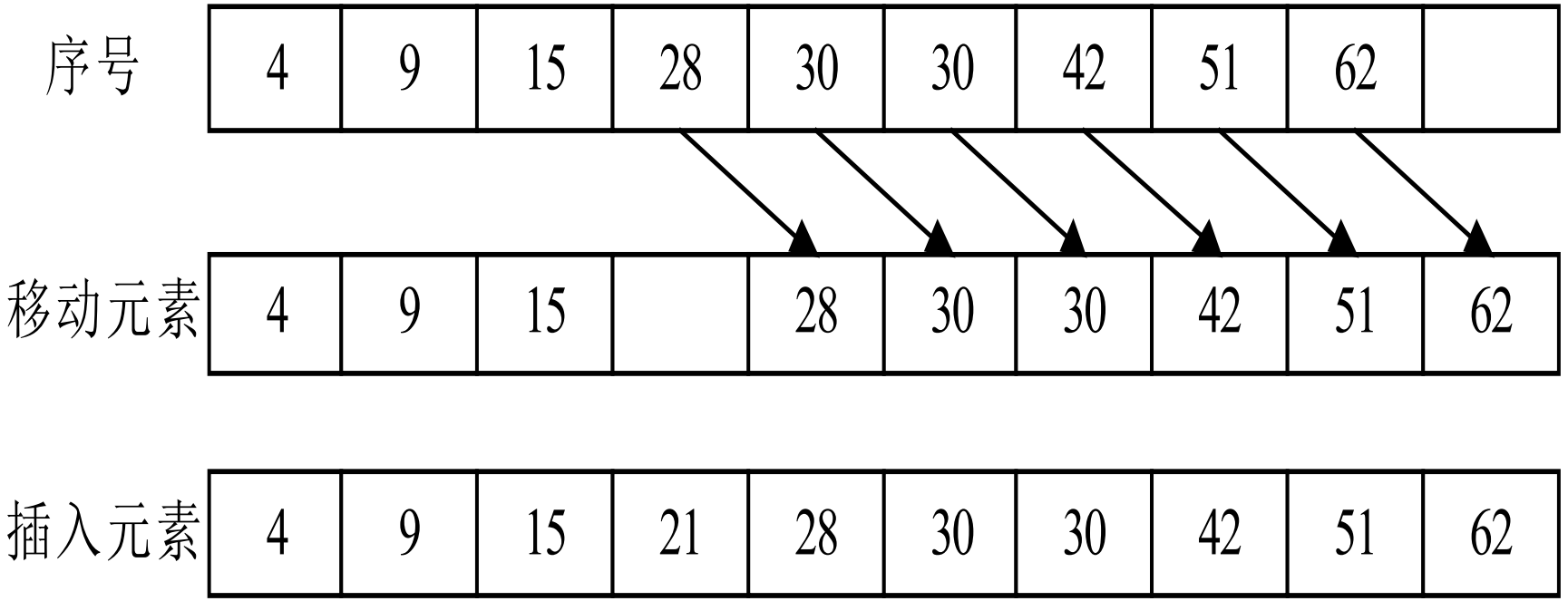
4. 顺序表上基本运算的实现

- 初始化 Status InitList_Sq (SqList &L)

```
{
    L.elem = (ElemType * )malloc(LIST_INIT_SIZE*
                                 sizeof(ElemType));
    if(!L.elem)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

L.elem = new ElemType[LIST_INIT_SIZE];

顺序表的插入：在表中第4个元素之前插入“21”。



顺序表中插入元素

- 插入 Status ListInsert_Sq (SqList &L, int i, ElemType e)

```
{
    if( (i<1) || (i>L.length+1) )
        return ERROR;
    if(L.length >= L.listsize){
        realloc(…); ….; // 越界处理 ;
    }
    q = &(L.elem[i-1]);
    for( p = &(L.elem[L.length-1]; p>=q; --p)
        *(p+1) = *p;
    *q = e;
    ++L.length;
    return OK;
}
```

```
// 越界处理

if  ( L.length >= L.listsize )  {

    newbase = ( ElemType * ) realloc ( L.elem ,
                ( L.listsize + LISTINCREMENT ) * sizeof(ElemType) );

    if  ( ! newbase )  exit(OVERFLOW) ;

    L.elem = newbase ;

    L.listsize += LISTINCREMENT ;

}
```

算法时间复杂度:

时间主要花在移动元素上，而移动元素的个数取决于插入元素位置。

i=1，需移动 n 个元素；

i=n+1，需移动 0 个元素；

i=i，需移动 n – i +1 个元素；

假设p_i是在第 i 个元素之前插入一个新元素的概率

则长度为 n 的线性表中插入一个元素所需移动元素次数的期望值为：$E_{is} = \sum_{i=1}^{n+1} p_i (n - i + 1)$

设在任何位置插入元素等概率，$p_i = \frac{1}{n+1}$

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

顺序表的删除：删除第5个元素，

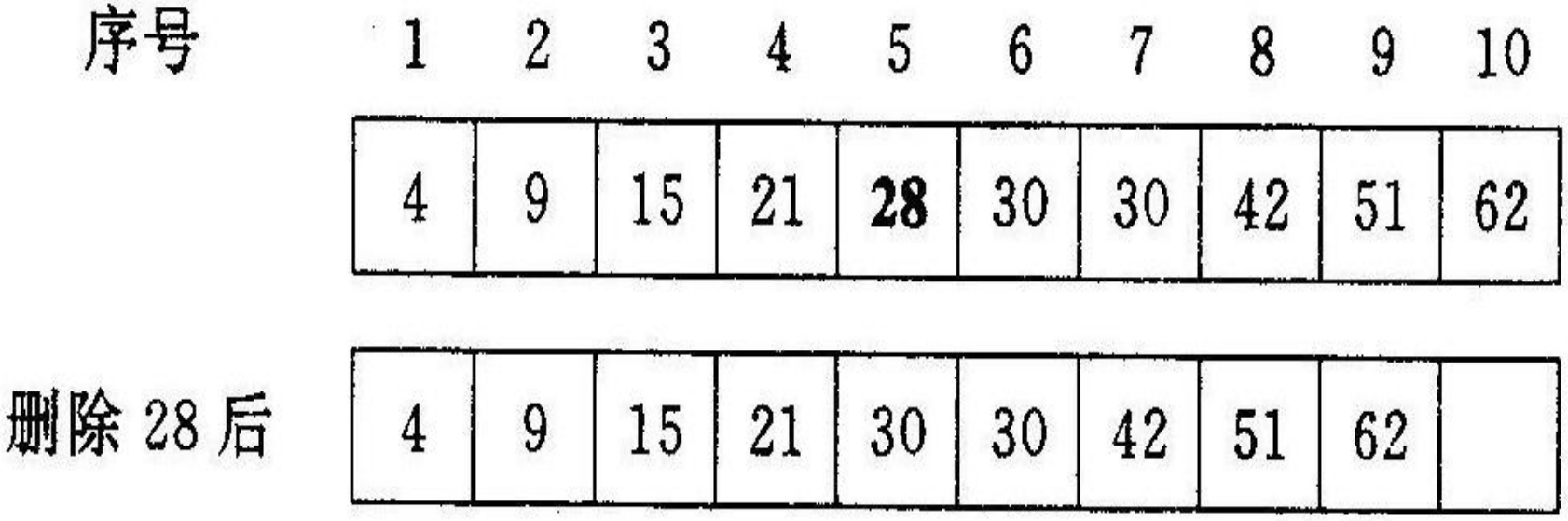


图 顺序表中删除元素

- 删除 Status ListDelete_Sq (SqList &L, int i, ElemType &e)

```
{
    if( (i<1) || (i>L.length) )
        return ERROR;
    p = &(L.elem[i-1]);
    e = *p;
    q = L.elem+L.length-1;
    for( ++p; p<=q; ++p)
        *(p-1) = *p;
    --L.length;
    return OK;
}
```

算法时间复杂度:

时间主要花在移动元素上，而移动元素的个数取决于删除元素位置。

i=1，需移动**n－1**个元素；

i=n，需移动**0**个元素；

i=i，需移动**n – i** 个元素；

假设q_i是删除第 i 个元素的概率

则长度为 n 的线性表中删除一个元素所需移动元素次数的期望值为：$E_{dl} = \sum_{i=1}^{n} q_i (n - i)$

设删除任何位置的元素等概率，$q_i = \frac{1}{n}$

$$E_{dl} = \frac{1}{n} \sum_{i=1}^{n} (n - i) = \frac{n-1}{2} \qquad O(n)$$

- 顺序表的归并，表中元素非递减排列。

```
void MergeList_Sq (SqList La, SqList Lb, SqList &Lc)
{
    pa = La.elem; pb = Lb.elem;
    Lc.listsize = Lc.length = La.length+Lb.length;
    pc = Lc.elem = (ElemType *)malloc(…);
    if(!Lc.elem) exit(OVERFLOW);
    pa_last = La.elem+La.length-1; pb_last = Lb.elem+Lb.length-1;
    while( (pa<=pa_last)&&pb<=pb_last)){
        if(*pa<=*pb)  *pc++ = *pa++;
        else  *pc++ = *pb++; }
    while(pa<=pa_last) *pc++ = *pa++;
    while(pb<=pb_last) *pc++ = *pb++;
}
```

顺序表的基础要点：

1. 无需为表示元素间的逻辑关系而增加额外的存储空间，存储密度大（100%）；
2. 可随机存取表中的任一元素。
3. 插入或删除一个元素时，需平均移动表的一半元素，具体的个数与该元素的位置有关，在等概率情况下，插入n/2，删除(n-1)/2；O(n)
4. 存储分配只能预先进行分配。
5. 将两个各有n个元素的有序表归并为一个有序表，其最少的比较次数是：n

- 作业：

  2.11

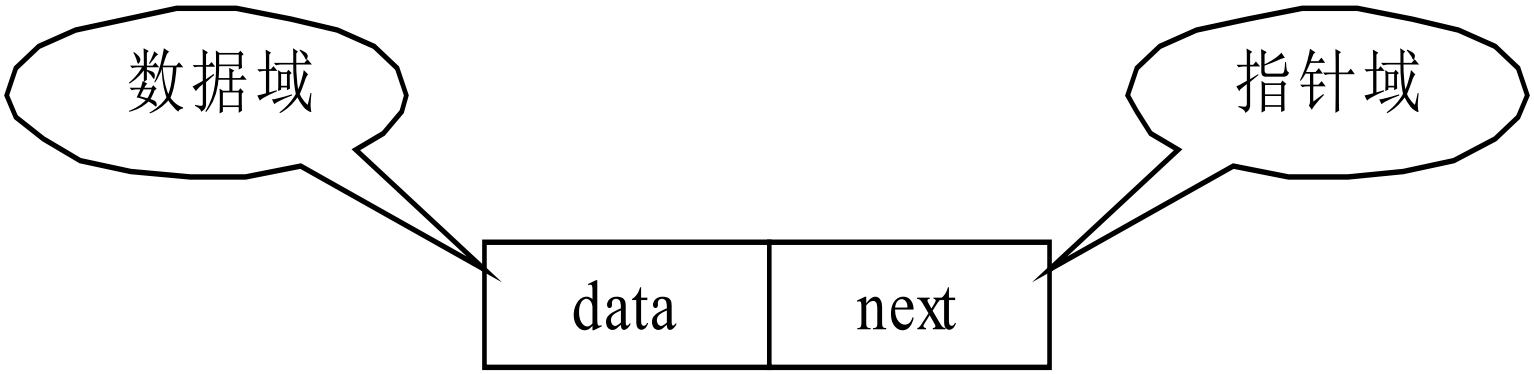
2.3 线性表的链式表示和实现

线性表链式存储结构的特点：

- 用一组任意的存储单元存储线性表的元素，不要求逻辑上相邻的元素在物理位置上也相邻；
- 插入删除时不需移动大量元素；
- 失去顺序表可随机存取的优点。

1. 线性链表（单链表）

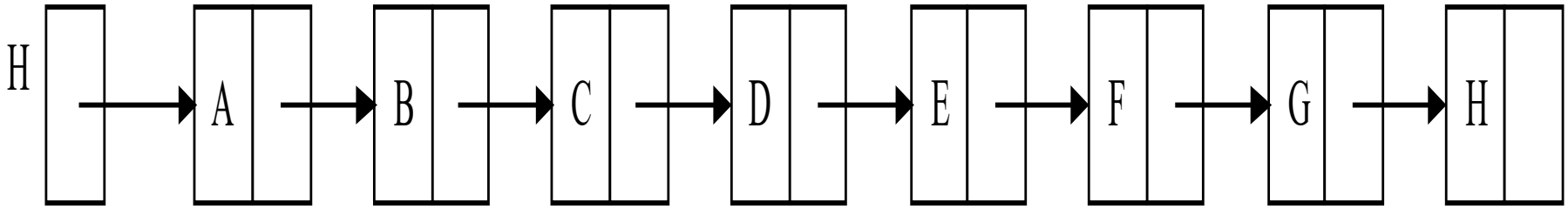
- 结点：数据元素的存储映象。



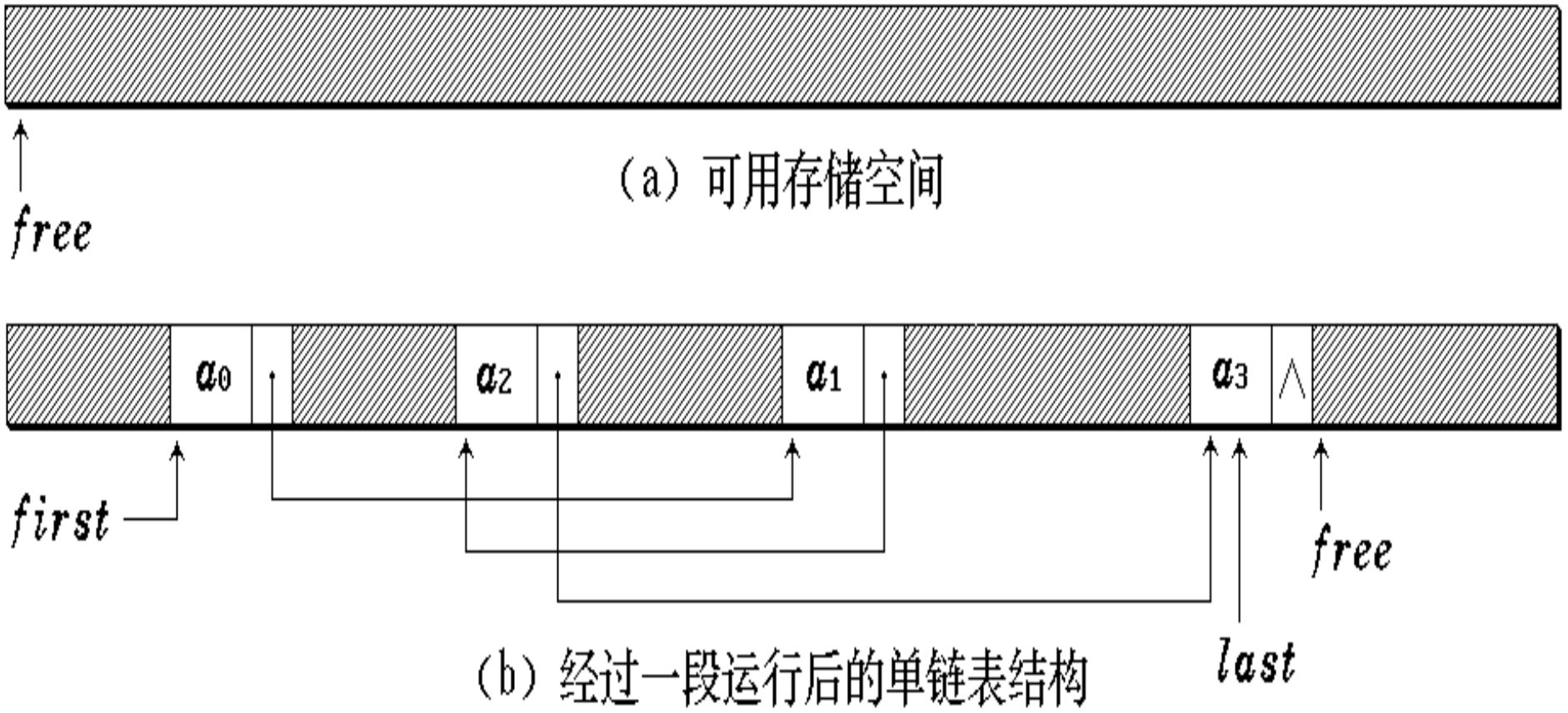
数据域用来存储结点的值；指针域用来存储数据元素的直接后继的地址（或位置）。

●头指针

指示链表中第一个结点的存储位置，单链表可由头指针唯一确定。



单链表的存储映象

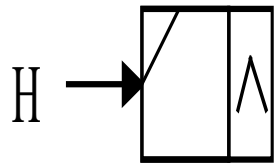


- **头结点**

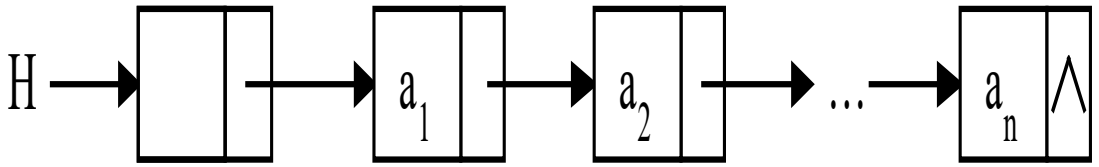
  在链表的第一个结点之前附设一个结点，头指针指向头结点。设置头结点的目的是统一空表与非空表的操作，简化链表操作的实现。

- **首元结点**

  链表中存储线性表中第一个数据元素的结点。



(a) 带头结点的空单链表



(b) 带头结点的单链表

- 单链表存储结构描述：

```
typedef struct LNode
{
    ElemType data;
    struct LNode  *next;
}LNode,  *LinkList;
```

单链表基本运算实现

(1)初始化单链表InitList(L)

该运算建立一个空的单链表,即创建一个头结点。

```
void InitList(LinkList &L)
{
    L=(LinkList)malloc(sizeof(LNode));
      /*创建头结点*/
    L->next=NULL;
}
```

(2)销毁单链表DestroyList(L)

释放单链表L占用的内存空间。即逐一释放全部结点的空间。

```
void DestroyList(LinkList L)
{       LinkList p=L, q=p->next;
        while (q!=NULL)
        {       free(p);
                p=q;q=p->next;
        }
        free(p);
}
```

(3)判断单链表是否为空表ListEmpty(L)

若单链表L没有数据结点,则返回真,否则返回假。

```
int ListEmpty(LinkList L)
{
    return(L->next==NULL);
}
```

(4)求单链表的长度ListLength(L)

返回单链表L中数据结点的个数。

```
int ListLength(LinkList L)
{       LinkList p=L;int i=0;
        while (p->next!=NULL)
        {     i++;
              p=p->next;
        }
        return(i);
 }
```

(5)输出单链表DispList(L)

逐一扫描单链表L的每个数据结点,并显示各结点的data域值。

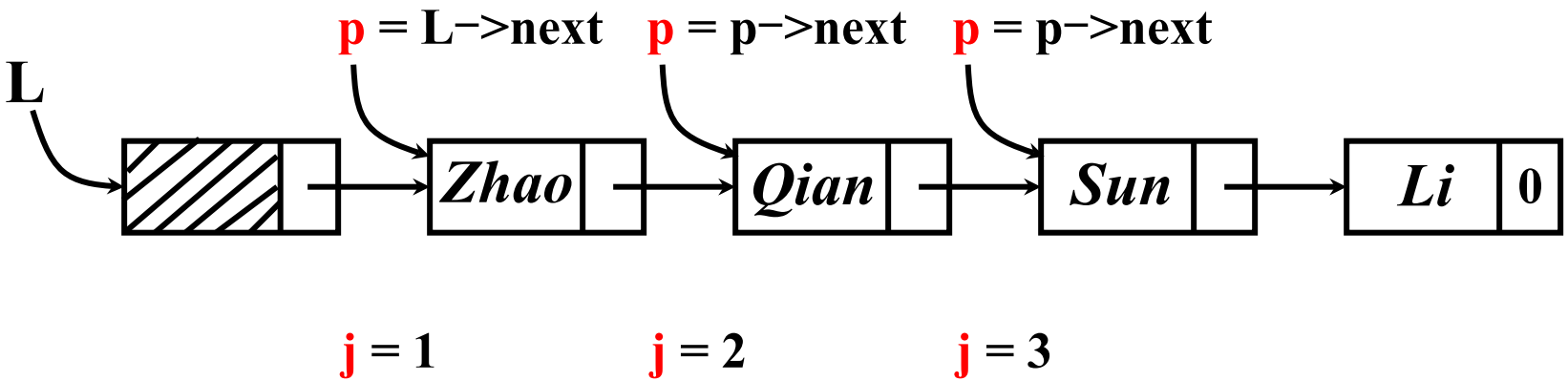
```
void DispList(LinkList L)
{       LinkList p=L->next;
        while (p!=NULL)
        {       printf("%c",p->data);
                p=p->next;
        }
        printf("\n");
}
```

(6) 取表元素

```
Status GetElem(LinkList L, int i, ElemType &e)
{
    p = L->next;   j = 1;
    while( p && j < i){
          p = p->next;  ++j;
    }
    if (!p || j>i) return ERROR;
    e = p->data;
    return OK;
}
```

- 从头指针L出发，从头结点（L->next）开始顺着链域扫描；
- 用j做计数器，累计当前扫描过的结点数，当j = i 时，指针p所指的结点就是要找的第i个结点。

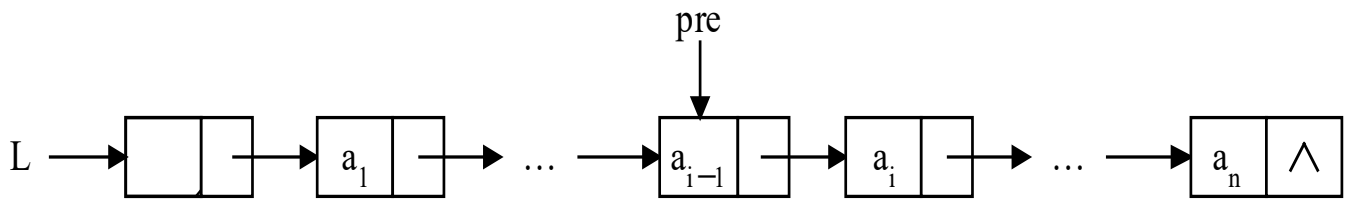
例，取第i=3个元素。



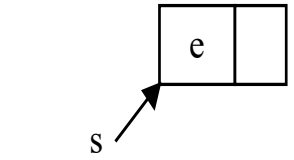
e = p->data = *Sun*

时间复杂度:　　*O*(n)

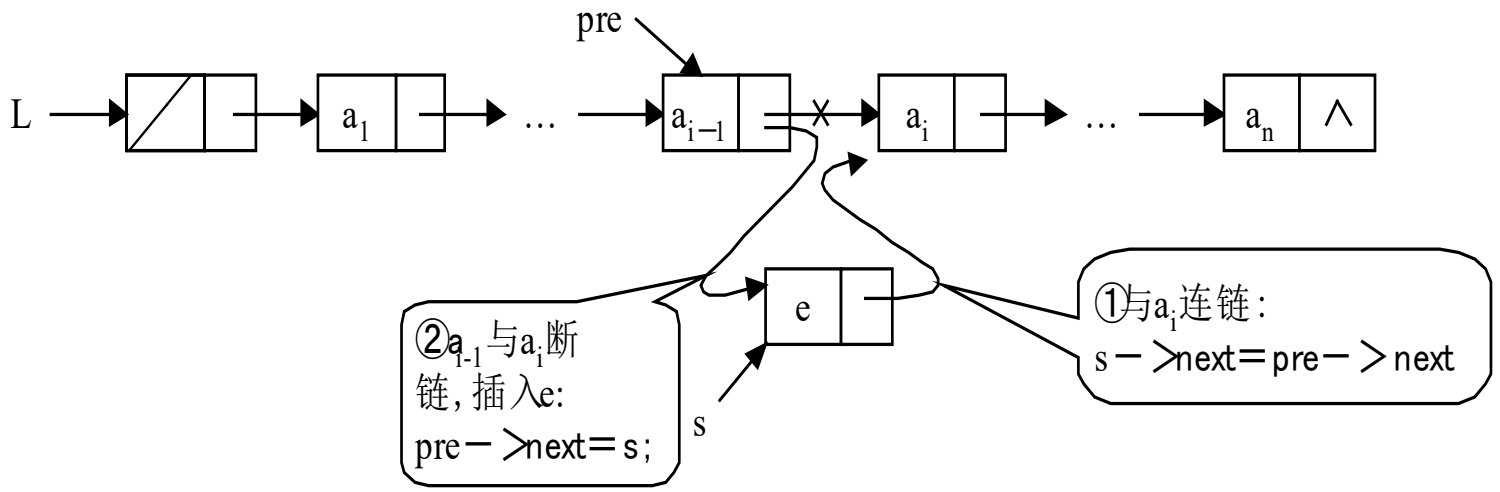
在单链表第i个结点前插入一个结点的过程



(a) 寻找第i−1个结点



(b) 申请新的结点

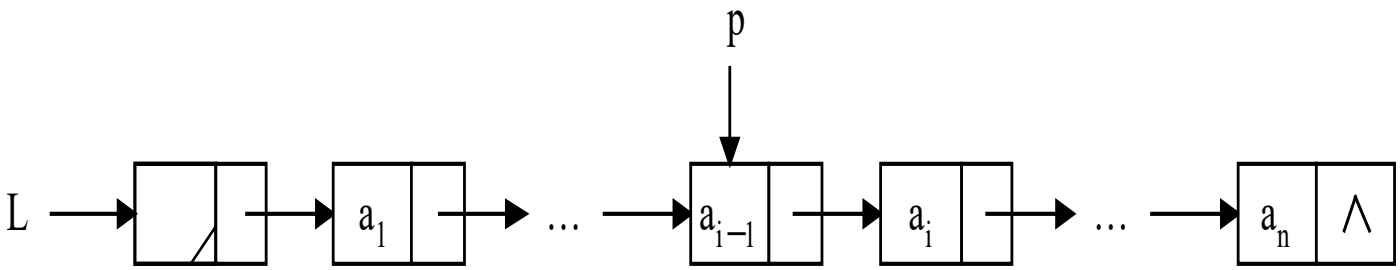


(c) 插入

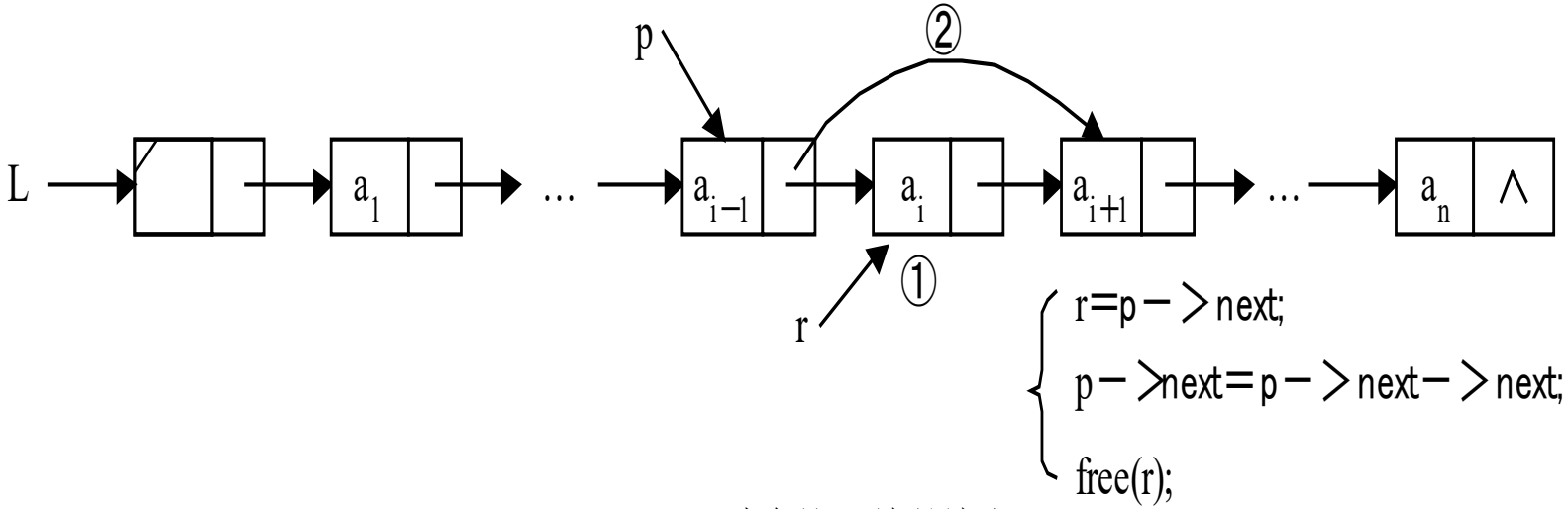
(7) 插入

```
Status ListInsert(LinkList &L, int i, ElemType e)
{
    p = L; j = 0;
    while (p && j < i-1) { p = p->next; ++j }
    if (!p || j>i-1)  return ERROR;
    s =  (LinkList) malloc( sizeof (LNode) );
    s->data = e;
    s->next = p->next;  ①
    p->next = s;  ②
    return OK;
}
```

- 删除单链表的第i个结点的过程



(a) 寻找第i−1个结点由p指向它

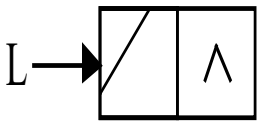


(b) 删除并释放第i结点

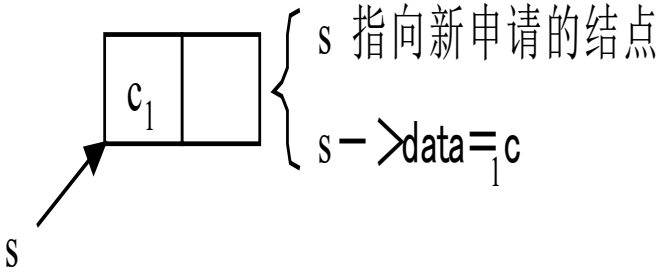
(8) 删除

```
Status ListDelete(LinkList &L, int i, ElemType &e)
{
    p = L; j = 0;
    while (p->next && j < i-1) { p = p->next; ++j }
    if (!(p->next) || j>i-1)  return ERROR;
    r = p->next;
    e = r->data;
    p->next = p->next->next;  //(p->next = r->next;) ①
    free(r);
    return OK;
}
```

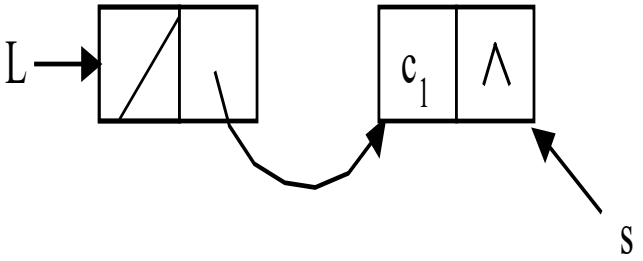
动态建立单链表的过程　　头插法建立单链表



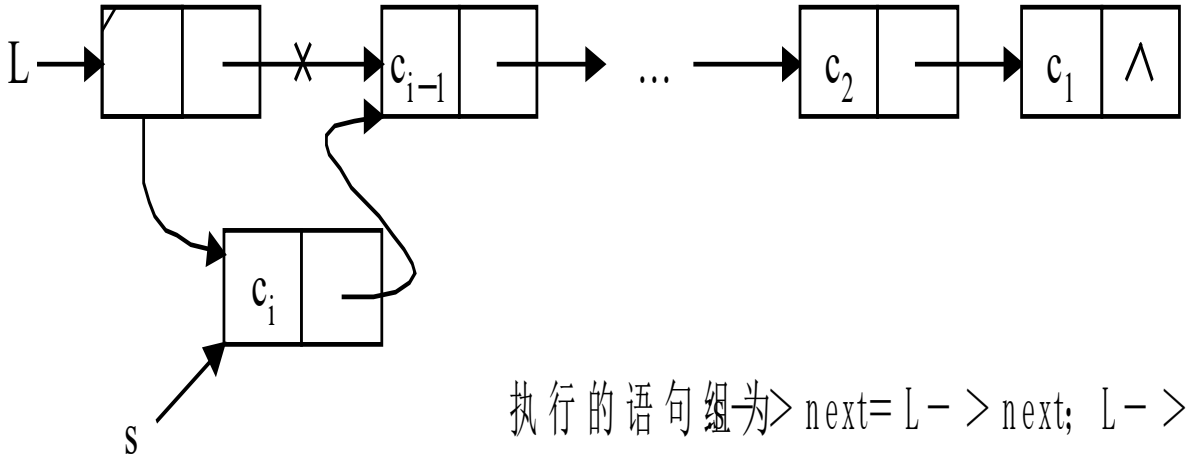
(a) 建空表



(b) 申请新结点并赋值



(c) 插入第一个结点



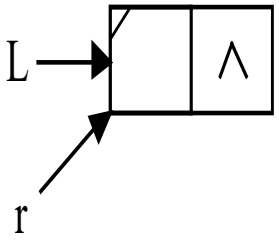
执行的语句组为：s->next=L->next; L->next=s;

(d) 插入第i个元素

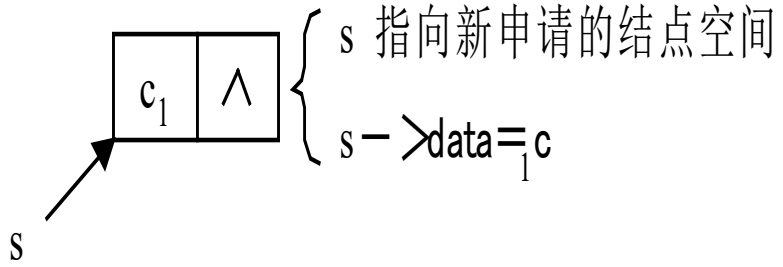
(9) 头插法建表

```
CreateList_H(LinkList &L, int n)
{
    L = (LinkList) malloc( sizeof (LNode) );
    L->next = NULL;
    for( i=n; i>0; --i){
        s = (LinkList) malloc( sizeof (LNode) );
        scanf( &s->data);
        s->next = L->next; ①
        L->next = s; ②
    }
}
```

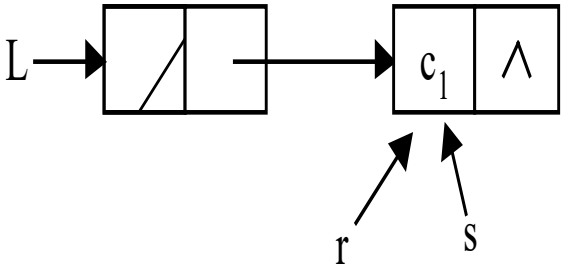
尾插法建表



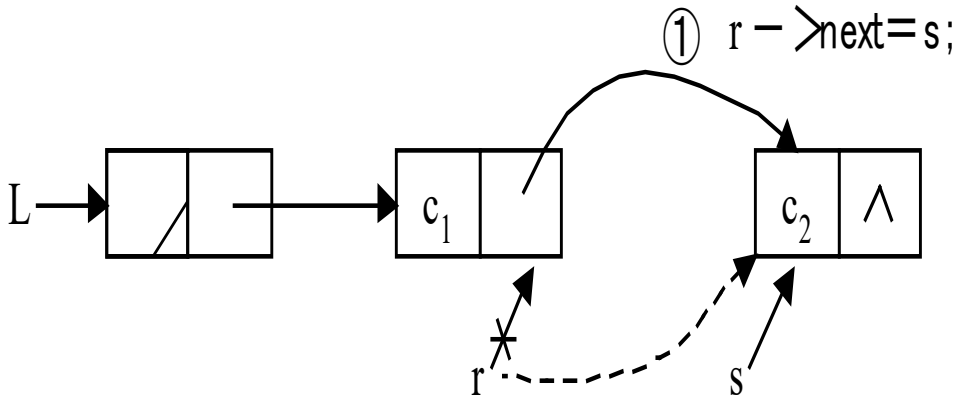
(a) 建空表



(b) 申请新结点并赋值



(c) 插入第一个结点



(d) 插入第二个结点

(10) 尾插法建表

```
CreateList_T(LinkList &L, int n)
{
    tail = L = (LinkList) malloc( sizeof (LNode) );
    L->next = NULL;
    for( i=n; i>0; --i){
        s = (LinkList) malloc( sizeof (LNode) );
        scanf( &s->data);
        s->next = NULL;
        tail->next = s; ①
        tail = s; ②
    }
}
```

(11)按元素值查找LocateElem(L,e)

思路：在单链表L中从头开始找第1个值域与e相等的结点,若存在这样的结点,则返回位置,否则返回0。

```
int LocateElem(LinkList L,ElemType e)
{       LinkList p=L->next;int n=1;
        while (p!=NULL && p->data!=e)
        {      p=p->next;  n++;  }
        if (p==NULL)  return(0);
        else  return(n);
}
```

练习：已知L是带头结点的非空单链表，指针p所指的结点既不是第一个结点，也不是最后一个结点

- 删除*p结点的直接后继结点的语句序列

```
q = p->next;
p->next = q->next;
free(q);
```

- 删除*p结点的直接前驱结点的语句序列

```
q = L;
while(q->next->next != p)   q = q->next;
s = q->next;
q->next = p;
 free(s);
```

- 删除*p结点的语句序列

```
q = L;
while( q->next != p)   q = q->next;
q->next = p->next;
free(p);
```

- 删除首元结点的语句序列

```
q = L->next;
L->next = q->next;
free(q);
```

- 删除最后一个结点的语句序列

```
while(p->next->next != NULL)   p = p->next;
q = p->next;
p->next = NULL;
free(q);
```

链式结构的特点

- 非随机存贮结构，所以取表元素要慢于顺序表。
  - 节约了大块内存
- 适合于插入和删除操作
  - 实际上用空间换取了时间，结点中加入了指针，使得这两种操作转换为指针操作;

线性表实现方法的比较

- 实现不同
  - 顺序表方法简单，各种高级语言中都有数组类型，容易实现；链表的操作是基于指针的，相对来讲复杂些。
- 存储空间的占用和分配不同
  - 从存储的角度考虑，顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模，也就是说事先对“MAXSIZE”要有合适的设定，过大造成浪费，过小造成溢出。而链表是动态分配存储空间的，不用事先估计存储规模。可见对线性表的长度或存储规模难以估计时，采用链表。
- 线性表运算的实现不同
  - 按序号访问数据元素，使用顺序表优于链表。
  - 插入删除操作 ，使用链表优于顺序表。

- 作业：

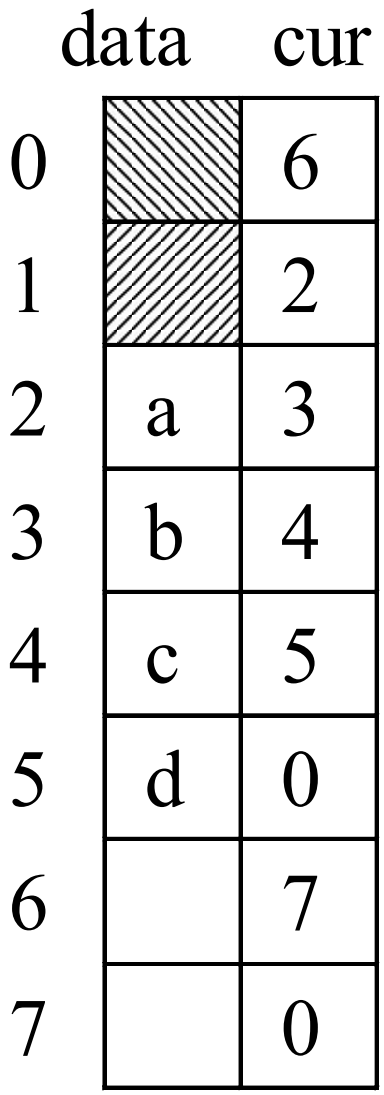
  2.4　2.19

3.静态链表

有些高级程序设计语言并没有指针类型，如FORTRAN和JAVA。我们可以用数组来表示和实现一个链表，称为静态链表。

可定义如下：

```
#define  MAXSIZE  1000   //最多元素个数
typedef  struct{
        ElemType data;
        int             cur; //游标，指示器
}component, SLinkList[MAXSIZE];
```



- **i = s[i].cur; 指针后移操作**
- **Malloc: i = s[0].cur; 第一个可用结点位置**

  if(s[0].cur) s[0].cur = s[i].cur;
- **Free: //释放k结点**

  s[k].cur = s[0].cur;

  s[0].cur = k;
- **Insert: //将i插在r之后**

  s[i].cur = s[r].cur;

  s[r].cur = i;
- **Delete: ;//p为k的直接前驱，释放k**

  s[p].cur = s[k].cur

  Free(k);

单链表基础要点

- 在单链表中，不能从当前结点出发访问到任一结点。
- 在单链表中，删除某一指定结点时，必须找到该结点的前驱结点。
- 线性表的链式存储结构是一种顺序存取的存储结构，不具有随机访问任一元素的特点。
- 设置头结点的作用：使在链表的第一个位置上的操作和表中其它位置上的操作一致，无需进行特殊处理，对空表和非空表的处理统一。

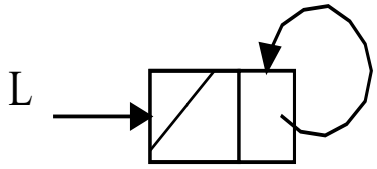
练习：2.22写一算法，对单链表实现就地逆置。

```
void Reverse_L( LinkList &L)
{
    p = L->next;
    L->next = NULL;
    while(p != NULL){
        q = p;
        p = p->next;
        q->next = L->next;
        L->next = q;
    }
}
```

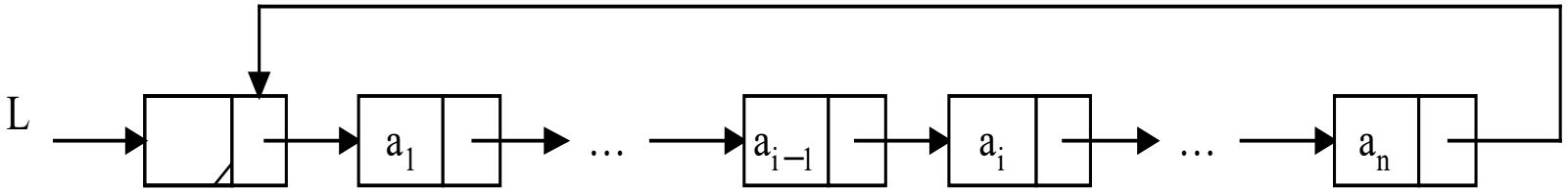
循环链表

- 循环链表是另一种形式的链式存储结构；
- 可从当前结点出发，访问到任一结点；
- 循环单链表；
- 多重循环链表。

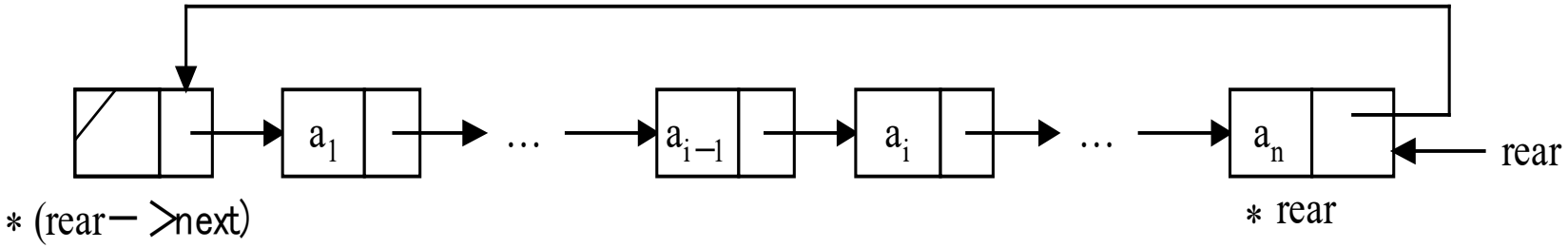
●单循环链表



(a) 带头结点的空循环链表



(b) 带头结点的循环单链表的一般形式



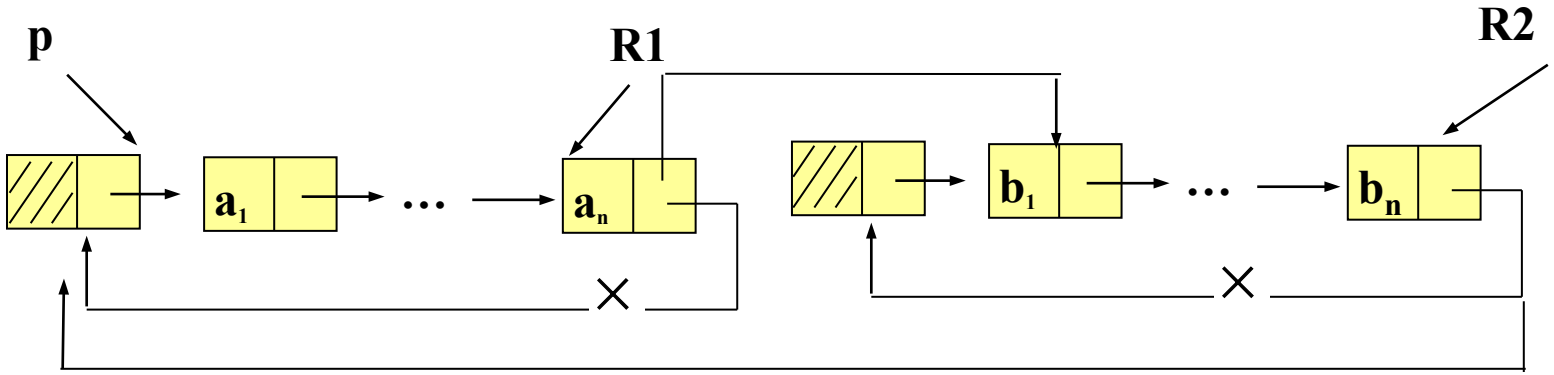
(c) 采用尾指针的循环单链表的一般形式

设置尾指针rear, 比设头指针更好。

连接两个只设尾指针的单循环链表L1和L2

操作如下：

```
p= R1 ->next;              //保存L1 的头结点指针
R1->next=R2->next->next;  //头尾连接
free(R2->next);           //释放第二个表的头结点
   R2->next=p;
```



操作与线性单链表基本一致，差别只是在于算法中的循环结束条件不是p是否为空，而是p是否等于头指针。

例，取循环链表第 i 个元素。

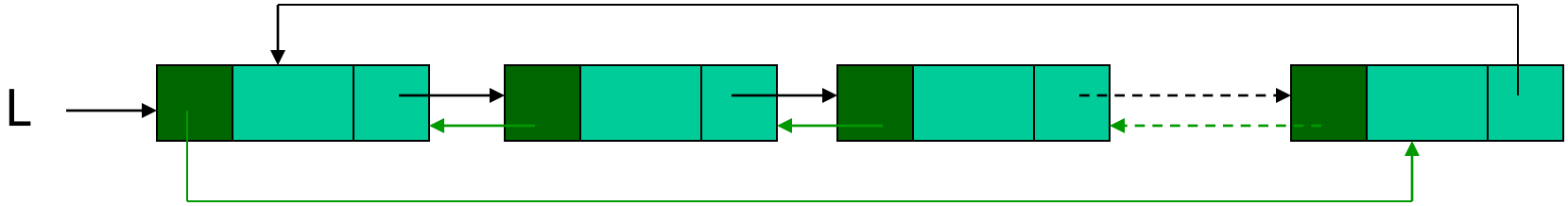
```
Status GetElem_L ( LinkList L，int i，ElemType &e ) {
    p = L->next ； j = 1 ；
    while ( p != L && j < i ) {
        p = p->next ； ++j ；
    }
    if ( p == L || j > i ) return ERROR ；
    e = p->data ；
    return OK ；
}
```

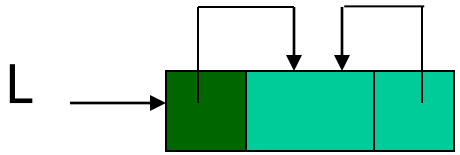
双链表

希望查找前驱的时间复杂度达到O(1)，我们可以用空间换时间，每个结点再加一个指向前驱的指针域，使链表可以进行双方向查找。用这种结点结构组成的链表称为双向链表。

结点的结构图：

双向链表的逻辑表示

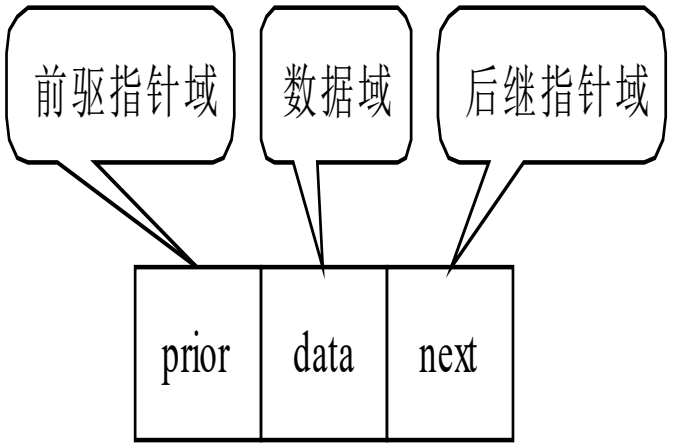




2. 双向链表(Double Linked List)

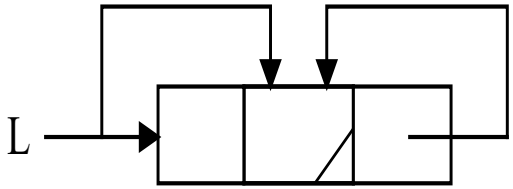
- 类型描述

```
typedef struct DuLNode{
    ElemType          data;
    struct DuLNode    *prior;
    struct DuLNode    *next;
}DuLNode,  *DuLinkList;
```

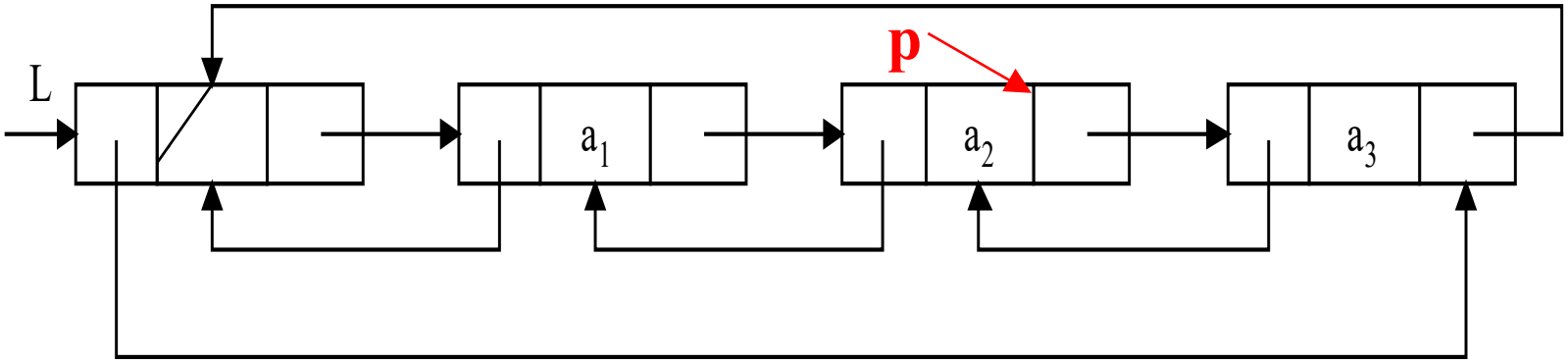


●双向循环链表

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

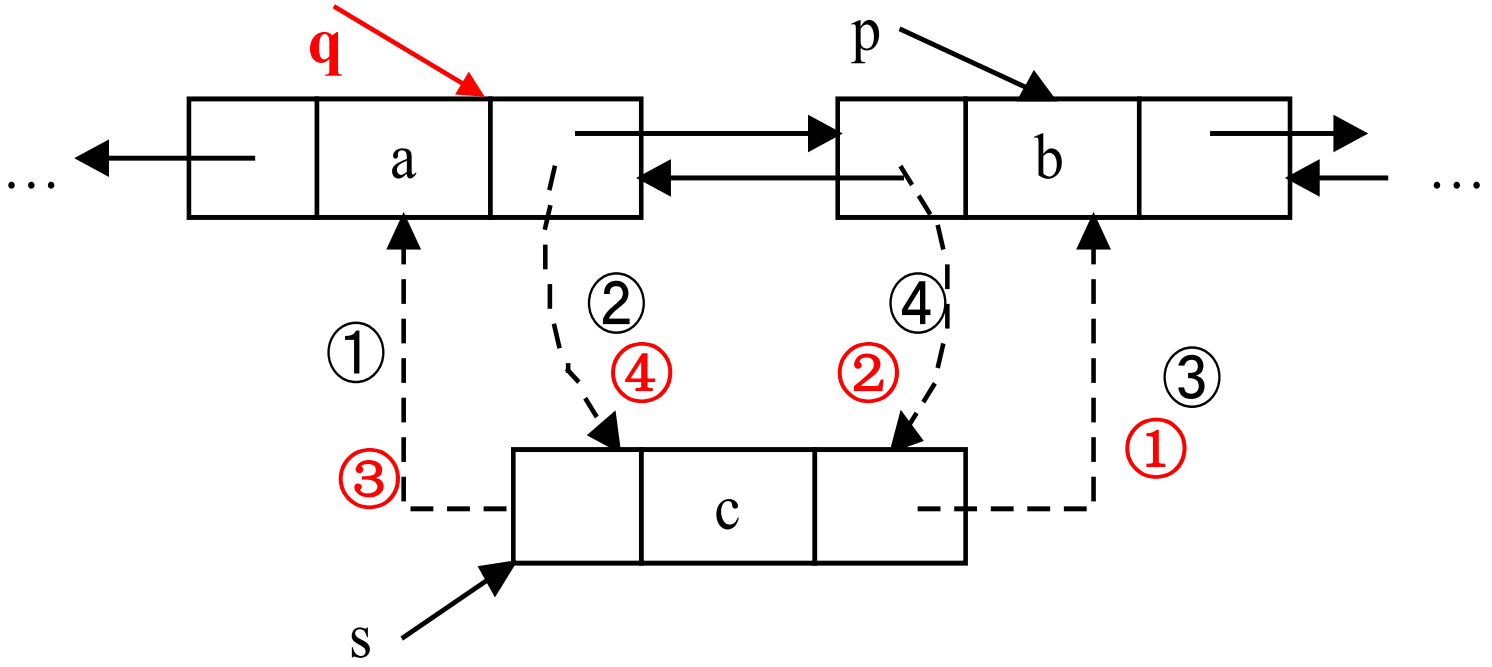


(a) 空的双向循环链表



(b) 非空的双向循环链表

- 双向链表的前(后)插入操作



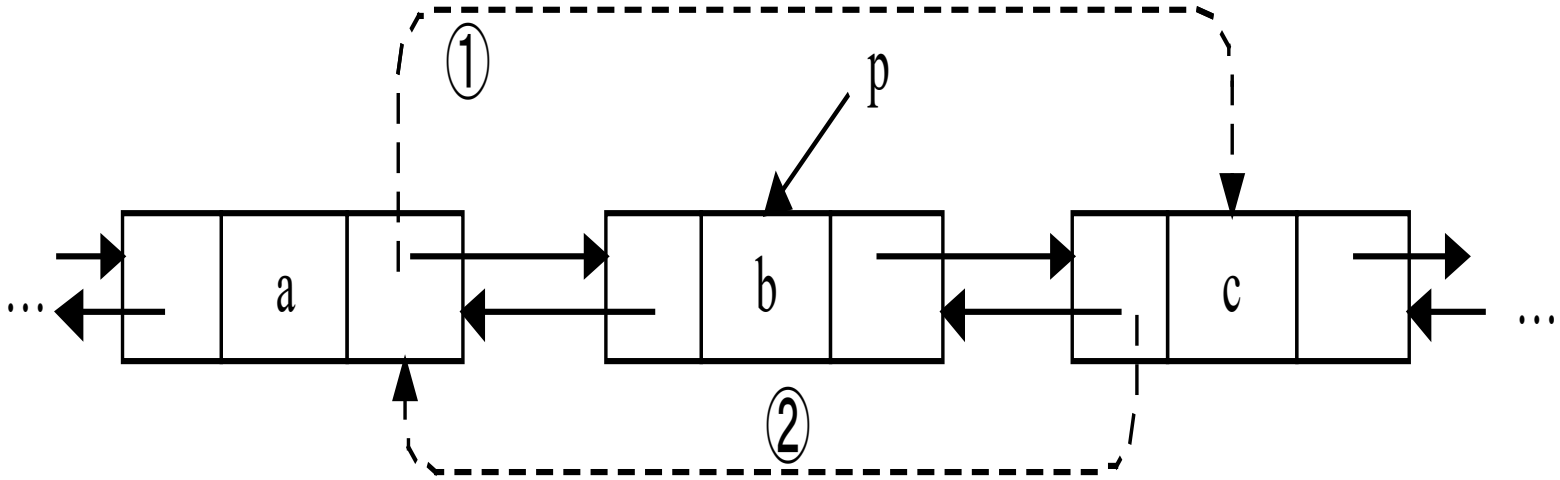
①s->prior = p->prior; ②p->prior->next = s;

③ s->next = p; ④p->prior = s;

①s->next = q->next; ②q->next->prior= s;

③s->prior = q; ④q->next = s;

- 双向链表的删除操作



①p->prior->next = p->next;

②p->next->prior = p->prior;

- 删除*p的直接后继结点的语句序列

```
q = p->next;
p->next = p->next->next;
p->next->prior = p;
free(q);
```

- 删除*p的直接前驱结点的语句序列

```
q = p->prior;
p->prior = p->prior->prior;
p-> prior->next = p;
free(q);
```

- 作业：

  2.8　　2.9

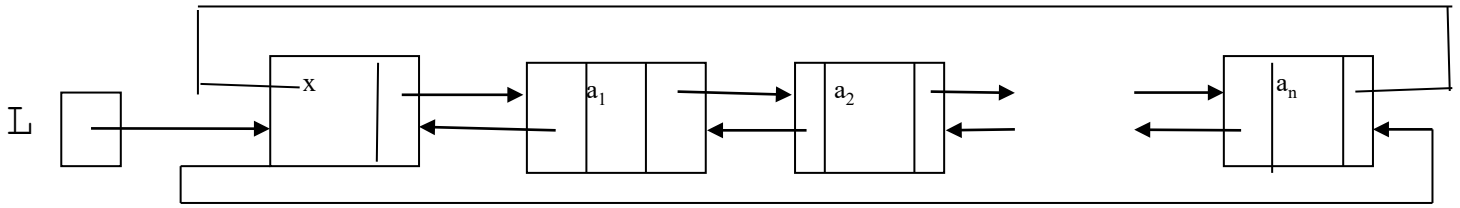
循环链表算法举例 (1)

假设一个单循环链表，其结点含有三个域pre、data、link。其中data为数据域；pre为指针域，它的值为空指针（null）；link为指针域，它指向后继结点。请设计算法，将此表改成双向循环链表。

```
void  SToDouble（DuLinkList  la）
// la是结点含有pre，data，link三个域的单循环链表。其中
   data为数据域；  pre为空指针域，link是指向后继的指针域。
   本算法将其改造成双向循环链表。
{while（la->link->pre==null）
    {la->link->pre=la//将结点la后继的pre指针指向la
     la=la->link;      //la指针后移
    }
 }  //算法结束
```

循环链表算法举例（2）

已知一双向循环链表，从第二个结点至表尾递增有序，（设a1<x<an）如下图。试编写程序，将第一个结点删除并插入表中适当位置，使整个链表递增有序



```
void DInsert（DuLinkList &L）
  //L是无头结点的双向循环链表，自第二结点起递增有序。本算法将第一结点（a1<x<an）插入到链表中，使整个链表递增有序
  {s=L;          //s暂存第一结点的指针
   t=L->prior;  // t暂存尾结点指针
   p=L->next;   //将第一结点从链表上摘下
   p->prior=L->prior;  p->prior->next=p;
   x=s->data;
   while（p->data<x） p=p->next;   //查插入位置
   s->next=p;  s->prior=p->prior;  //插入原第一结点s
   p->prior->next=s;  p->prior=s;
   L=t->next;
  } // 算法结束
```

循环链表算法举例（3）

例 编写出判断带头结点的双向循环链表L是否对称相等的算法。

解:p从左向右扫描L,q从右向左扫描L,若对应数据结点的data域不相等,则退出循环,否则继续比较,直到p与q相等或p的下一个结点为*q为止。对应算法如下:

```
int Equal(DuLinkList L)
{     int same=1;
      DuLinkList p=L->next;   /*p指向第一个数据结点*/
      DuLinkList q=L->prior;  /*q指向最后数据结点*/
      while (same==1)
           if (p->data!=q->data)  same=0;
           else
           {
                if (p==q) break;  /*数据结点为奇数的情况*/
                q=q->prior;
                if (p==q) break;  /*数据结点为偶数的情况*/
                p=p->next;
           }
       return same;
}
```

顺序表和链表的比较

- 基于空间的考虑

  存储密度=元素本身所占的存储量/实际分配的存储总量

- 基于时间的考虑

  顺序表：随机存取结构，**O(1)**。

  链表：　顺序存取结构，**O(n)**。

- 基于语言的考虑

2.4 一元多项式的表示及相加

$$P_n(X) = P_0 + P_1X + P_2X^2 + P_3X^3 + \cdots + P_nX^n$$

用一个线性表P来表示：

$$P = (P_0, P_1, P_2, \cdots, P_n)$$

假设m<n，则两个多项式相加的结果
$R_n(x)=P_n(x)+Q_m(x)$， 也可以用线性表R来表示：

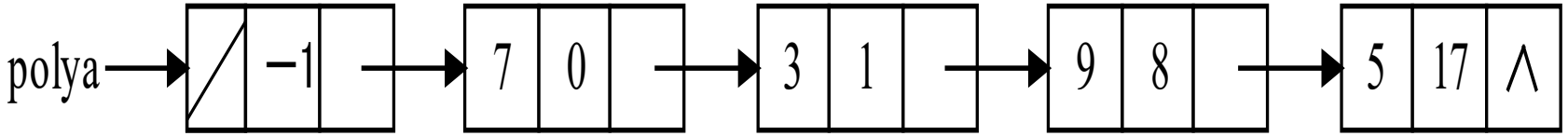
$$R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \cdots, p_m + q_m, p_{m+1}, \cdots, p_n)$$

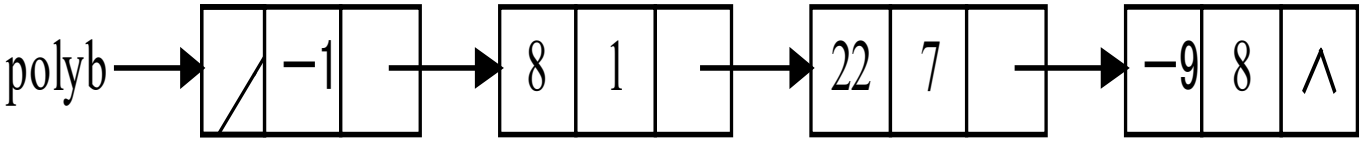
$$R(x) = 1 + 5x^{10000} + 7x^{20000}$$

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \ldots + p_m x^{e_m}$$

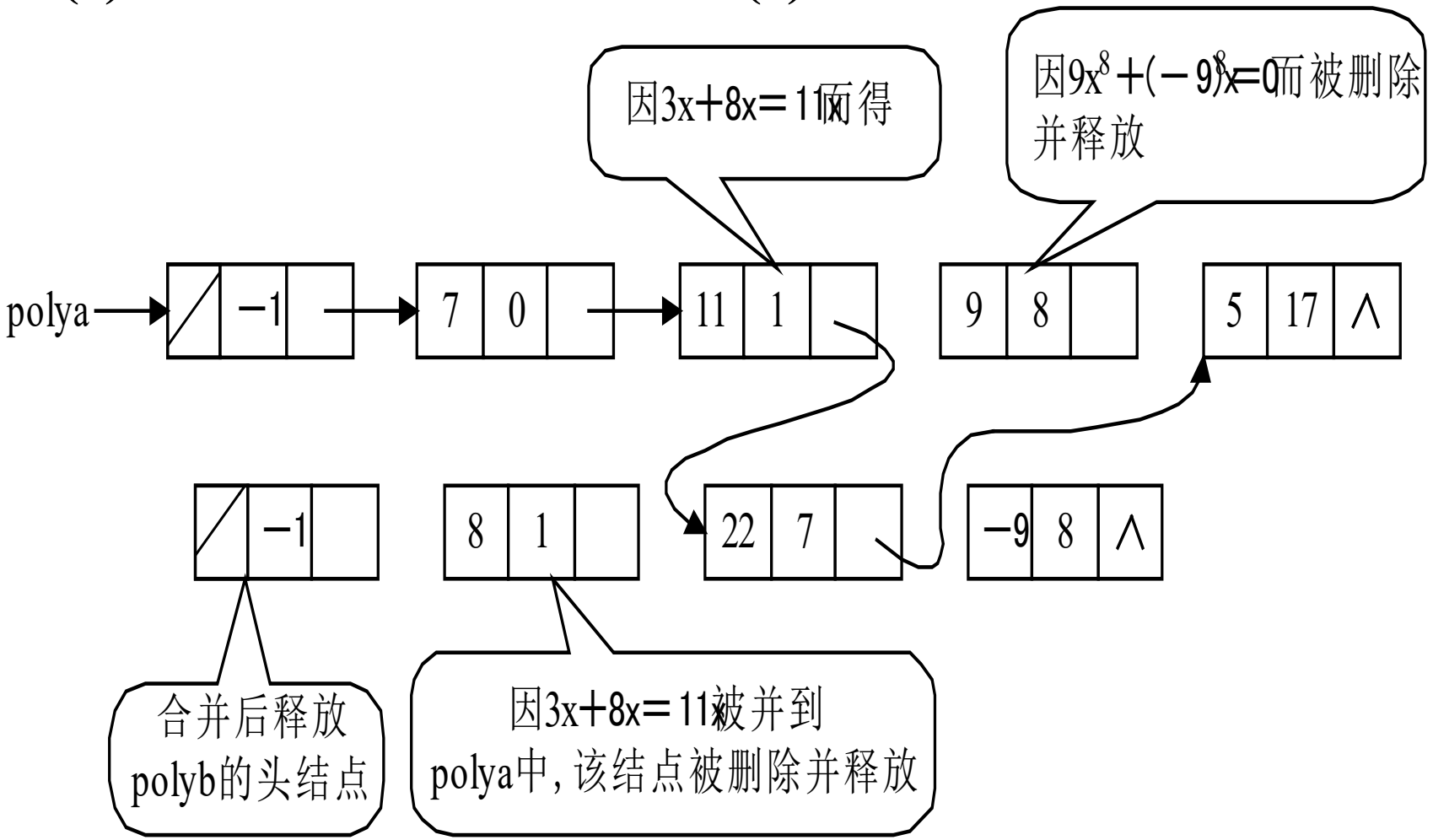
$$((p_1, e_1), (p_2, e_2), \cdots, (p_m, e_m))$$

A(x)=7+3x+9x^8+5x^{17}和多项式B(x)=8x+22x^7-9x^8





$A(x)=7+3x+9x^8+5x^{17}$和多项式$B(x)=8x+22x^7-9x^8$



多项式相加得到的多项式和

练习1：某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用哪种存储方式最节省时间

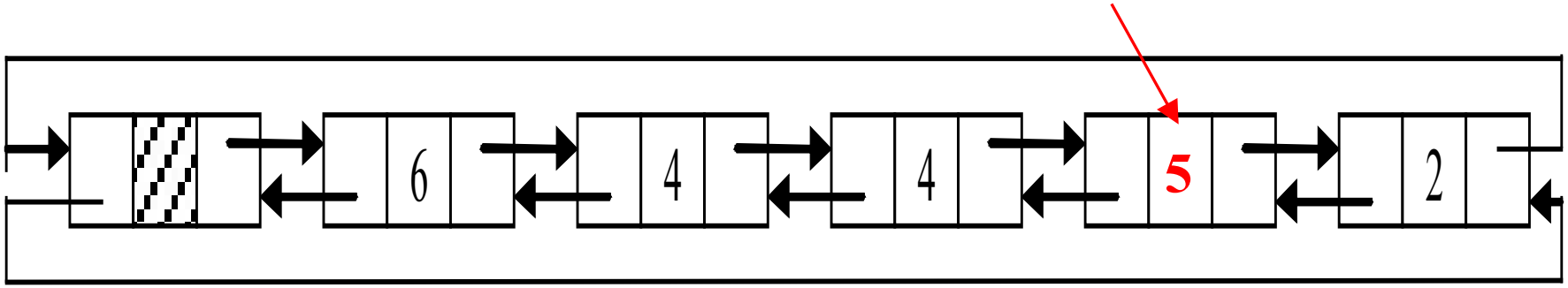
A 单链表　　B 仅有头指针的单循环链表

C 双链表　　D 仅有尾指针的单循环链表

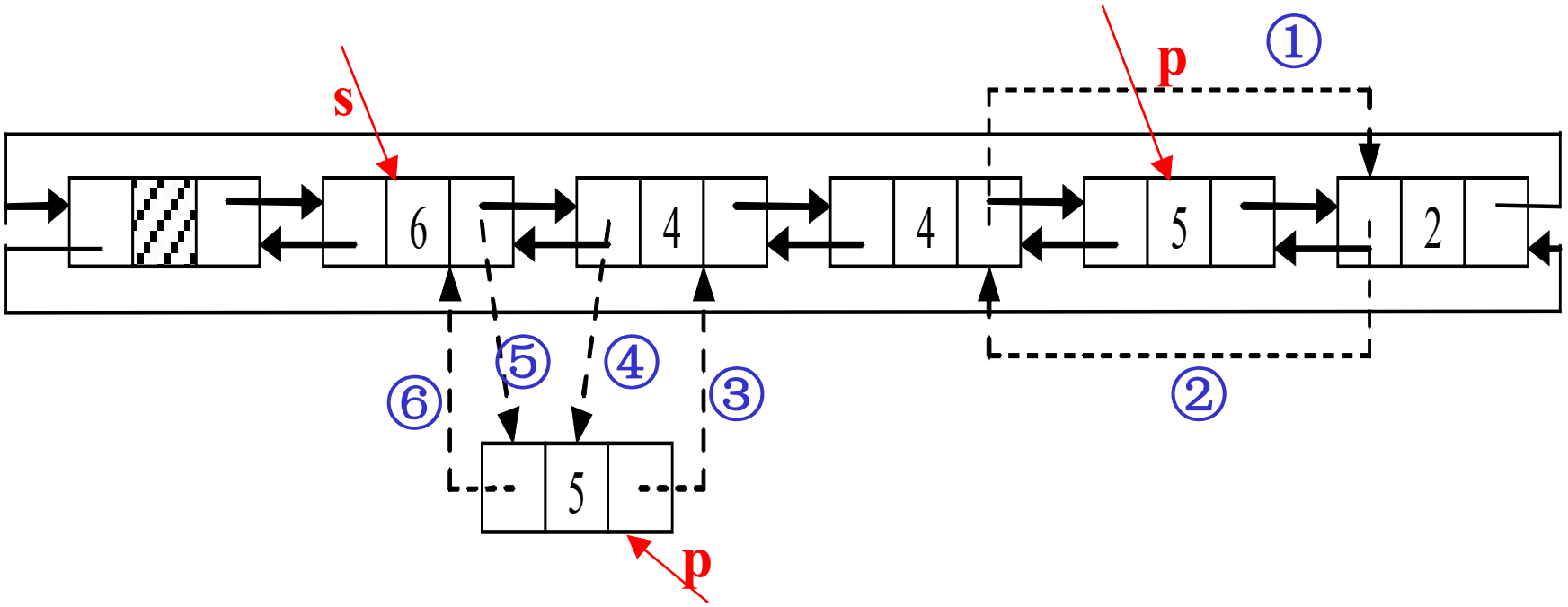
练习2：2.38 设有一个双向循环链表，每个结点中除有pre, data和next三个域外，还增设一个访问频度域freq。在链表被起用之前，频度域freq的值均初始化为零，而每当对链表进行一次Locate(L,x)的操作后，被访问的结点的频度域freq的值增1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以使被频繁访问的结点总是靠近表头结点。试编写符合上述要求的Locate(L,x)操作的算法。

DuLinkList Locate(DuLinkList &L, ElemType x）

```
typedef struct DuLNode{
    ElemType            data;
    int                 freq;
    struct DuLNode     *prior;
    struct DuLNode     *next;
}DuLNode,  *DuLinkList;
```



```
DuLinkList  Locate( DuLinkList &L, ElemType x）
{
    p=L->next;
    while(p->data!=x && p!=L)
          p=p->next;
    if(p==L)   return NULL; //没找到
    p->freq++;    s=p->pre;
    while(s!=L && s->freq < p->freq)
          s=s->pre; //查找插入位置
    if(s!=p->pre){
          ……
    }
    return p;
}
```




```
if(s != p->pre){
    ①p->pre->next=p->next;  ②p->next->pre=p->pre;
    ③p->next=s->next;       ④s->next->pre=p;
    ⑤s->next=p;             ⑥p->pre=s;
}
```