

第二章 线性表

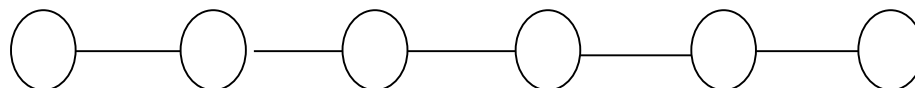
四类基本结构：

1. 线性结构

2. 树形结构

3. 图状结构

4. 集合



- 1) 线性表
- 2) 栈和队列
- 3) 串
- 4) 数组与广义表

线性表的逻辑特征:



在线性表 $(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$ ($n>0$)中:

1. 存在唯一的 **“第一元素”** a_1 ;
2. 存在唯一的一个 **“最后元素”** a_n ;
3. 除最后元素 a_n 在外, 均有 **唯一的后继**;
4. 除第一元素 a_1 之外, 均有 **唯一的前驱**。

第二章 线性表

2.1 线性表的概念及类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

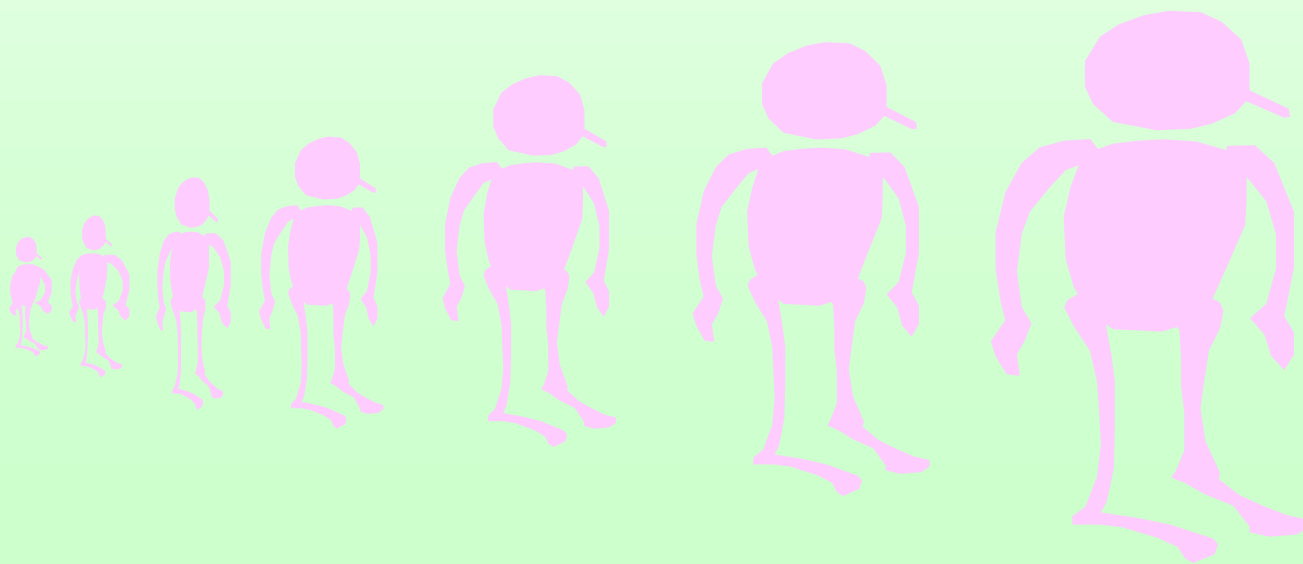
2.4 一元多项式的表示及相加

2.1

线性表的概念及类型定义

基本概念

“线性表”简称表，由 n ($n \geq 0$)个数据元素组成的**有限序列**，通常可以表示成 $(a_1, a_2, \dots, a_i, \dots, a_n)$ ($n \geq 0$)。表中所含元素的个数 n 称为表的“**长度**”； $n=0$ 的表称为“**空表**”。



基本概念

线性表中的数据元素可以是各式各样的（如一个数、一个字符，也可能是其它更复杂的信息），但在同一线性表中必须属于同一数据对象。

线性表示例

- 例1. 某公司在2002年1至6月份的月销售情况统计表(单位: 万元)

(380 , 420 , 400 , 390 , 410 , 430)

数据元素: **数字** 表长度: **n=6**

- 例2. 英文大写字母表

(A , B , C , …… , X , Y , Z)

数据元素: **字符** 表长度: **n=26**

线性表示例

例3. 某班级的学生成绩登记表

学号	姓名	高数	英语	线代	C语言
0001	张三	92	80	98	98
0002	李四	95	79	100	95
数据元素：若干数据项组成					
表长度：n=该班级人数					
0003	王五	89	82	99	91
...
...

线性表的特点

同一性：线性表由同类数据元素组成，即数据元素 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 n 。

有序性：线性表中相邻数据元素之间存在着序偶关系 $\langle a_i, a_{i+1} \rangle$ 。

位序：设线性表为 $(a_1, \dots, a_i, \dots, a_n)$ ，称 i 为 a_i 在线性表中的**位序**。

线性表的抽象数据类型定义

ADT **List** {

数据对象:

$D = \{ a_i \mid a_i \in D_0, i = 1, 2, \dots, n, n \geq 0, \}$
 D_0 为某一数据对象}

数据关系:

$R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 2, 3, \dots, n \}$

基本操作:

结构初始化操作

结构销毁操作

引用型操作

加工型操作

} ADT List

初始化操作

InitList(&L)

操作结果： 构造一个空的线性表 L。

结构销毁操作

DestroyList(&L)

初始条件： 线性表 L 已存在。

操作结果： 销毁线性表 L。

引用型操作:

ListEmpty(L)

ListLength(L)

PriorElem(L, cur_e, &pre_e)

NextElem(L, cur_e, &next_e)

GetElem(L, i, &e)

LocateElem(L, e)

ListTraverse(L, visit())

ListEmpty(L) (线性表判空)

初始条件: 线性表 L 已存在。

操作结果: 若 L 为空表, 则返回
1, 否则0。

ListLength(L) (求线性表的长度)

初始条件: 线性表 L 已存在。

操作结果: 返回 L 中元素个数。

PriorElem(L, cur_e, &pre_e) (求前驱)

初始条件: 线性表 L 已存在。

操作结果: 若cur_e 是L的元素, 且不是第一个, 则用pre_e 返回它的前驱, 否则操作失败, pre_e无定义。

NextElem(L, cur_e, &next_e) (求后继)

初始条件: 线性表 L 已存在。

操作结果: 若cur_e 是L的元素, 且不是最后一个, 则用next_e返回它的后继, 否则操作失败, next_e无定义。

GetElem(L, i, &e) (求某个数据元素)

初始条件： 线性表 L 已存在，
且 $1 \leq i \leq \text{LengthList}(L)$

操作结果： 用 e 返回 L 中第 i 个元素的值。

LocateElem(L, e) (定位函数)

初始条件： 线性表 L 已存在，e 为给定值。

操作结果： 返回 L 中第 1 个与 e 相等的元素的位序。
若这样的元素不存在，则返回值为 0。

ListTraverse(L, visit()) (遍历操作)

初始条件： 线性表 L 已存在。

操作结果： 依次对 L 的每个元素调用函数 visit () , 一旦 visit () 失败, 则操作失败。

加工型操作

SortList(&L)

ClearList(&L)

ListInsert(&L, i, e)

ListDelete(&L, i, &e)

SortList(&L) (**线性表排序**)

初始条件: **线性表 L 已存在。**

操作结果: **按某一数据项的值对线性表中的元素进行排序。**

ClearList(&L) (**线性表置空**)

初始条件: **线性表 L 已存在。**

操作结果: **将 L 重置为空表。**

ListInsert(&L, i, e) (插入数据元素)

初始条件: 线性表 L 已存在,

且 $1 \leq i \leq \text{LengthList}(L) + 1$

操作结果: 在 L 的第 i 个元素之前插入新的元素 e, L 的长度增1。

ListDelete(&L, i, &e) (删除数据元素)

初始条件: 线性表 L 已存在且非空,

$1 \leq i \leq \text{LengthList}(L)$

操作结果: 删除 L 的第 i 个元素, 并用 e 返回其值, L 的长度减1。

End: Definition of ADT List

例2.1

已知有两个集合 A 和 B 分别用两个线性表 LA 和 LB 表示，线性表中的数据元素即为集合中的成员。

现要求一个新的集合 $A = A \cup B$ 。

问题分析:

**扩大线性表 LA, 将存在于线性表LB
中而不存在于线性表 LA 中的数据元素
插入到线性表 LA 中去。**

算法思路:

1、从线性表LB中依次取出每个数据元素;

$\text{GetElem}(\text{LB}, i) \rightarrow x$

2、判断线性表LA中是否存在该数据元素

$\text{LocateElem}(\text{LA}, x)$

3、如果不存在, 则插入; 否则转第1步。

$\text{ListInsert}(\text{LA}, n+1, x)$

```

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))
            ListInsert(La, ++La_len, e);
    }
}
    
```

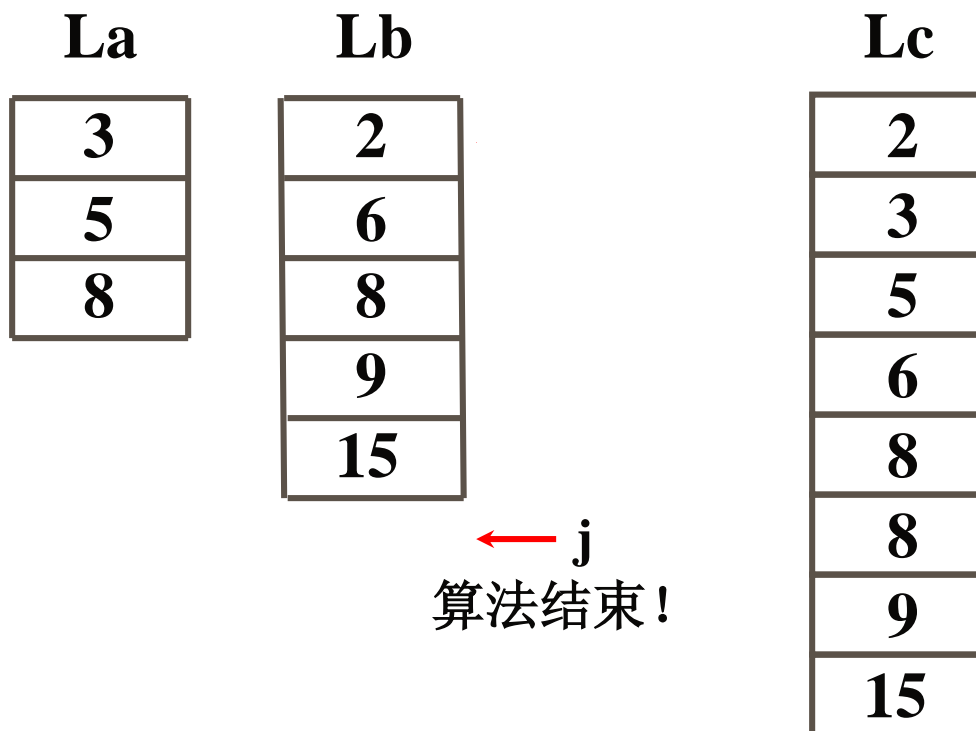

例2.2 已知线性表LA和线性表LB中的数据元素按值非递减有序排列，现要求将LA和LB归并为一个新的线性表LC，且LC中的元素仍按值非递减有序排列。

例如, $L_a = (3, 5, 8)$

$L_b = (2, 6, 8, 9, 15)$

构造 $L_c = (2, 3, 5, 6, 8, 8, 9, 15)$

首先, $L_a_len = 3$; $L_b_len = 5$;



已知线性表LA和线性表LB中的数据元素按值非递减有序排列，现要求将LA和LB归并为一个新的线性表LC，且LC中的元素仍按值非递减有序排列。

此问题的算法如下：

```
void MergeList(List La, List Lb, List &Lc)
{
    InitList(Lc);
    i=j=1;k=0;
    La_len=ListLength(La);
    Lb_len=ListLength(Lb);
```

```
while((i<=La_len)&&(j<=Lb_len))
```

```
{ //La和Lb均非空
```

```
    GetElem(La, i, ai); GetElem(Lb, j, bj);
```

```
    if(ai<=bj) { ListInsert(Lc, ++k, ai); ++i; }
```

```
    else { ListInsert(Lc, ++k, bj); ++j; }
```

```
}
```

```
while(i<=La_len) //如果线性表La中还有剩余的元素
```

```
{ getelem((La, i++, ai);ListInsert(Lc, ++k, ai); }
```

```
while(j<=lb_len) //如果线性表Lb中还有剩余的元素
```

```
{ getelem((lb, j++, bj);listinsert(lc, ++k, bi); }
```

```
}
```

分析上面两个算法:

算法2.1的时间复杂度为

$$O(\text{ListLength}(La) * \text{ListLength}(Lb))$$

算法2.2的时间复杂度为

$$O(\text{ListLength}(La) + \text{ListLength}(Lb))$$

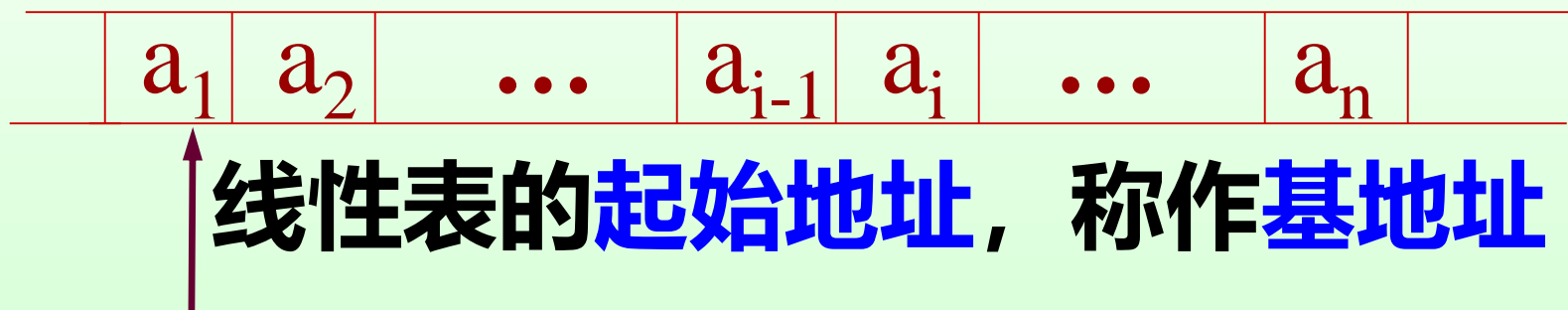
2.2

线性表的顺序表示和实现

通常，称线性表的顺序存储结构为顺序表。

1. 顺序表的定义

用一组地址连续的存储单元依次存放线性表的各个数据元素。



特点：

在顺序表中逻辑结构上相邻的数据元素，其物理位置也是相邻的。

2. 顺序表中数据元素的存储地址

“存储位置相邻” 的有序对 $\langle a_{i-1}, a_i \rangle$ 之间的

位置关系为: $LOC(a_i) = LOC(a_{i-1}) + C$

其中 C 为一个数据元素所占据的存储空间

数据元素的存储位置均取决于基地址和
该数据元素在线性表中的位置

$$LOC(a_i) = \underbrace{LOC(a_1)}_{\uparrow \text{基地址}} + (i-1) \times C$$

存储密度

$$d = \frac{\text{数据元素的值所需的存储量}}{\text{该数据元素所需的存储总量}}$$

顺序分配的存储密度 $d=1$

单链表一般的存储密度 <1

3.顺序表的描述:

方式一，采用静态数组直接定义

```
#define MAXSIZE = 线性表可能的最大长度  
typedef struct  
{  
    ElemType elem[MAXSIZE];  
        /*线性表占用的数组空间*/  
    int length;    /*线性表当前长度*/  
} SqList;
```

3.顺序表的描述:

方式二，动态分配存储空间

```
#define LIST_INIT_SIZE    100 //存储空间的初始分配量
#define LISTINCREMENT     10  //存储空间的分配增量

typedef struct
{
    ElemType *elem; // 存储空间基址
    int length; // 顺序表当前长度
    int listsize; // 顺序表分配的存储容量
} SeqList;
```

动态分配存储空间的初始化函数，即构造一空表

```
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;
} //InitList_Sq
```

4.顺序表的基本操作——查找算法

线性表有两种基本的查找运算：

按序号查找GetData(L, i):

查找线性表L中第i个数据元素

$\text{GetData}(L, i) = L.\text{elem}[i]$

按内容查找Locate(L, e):

若找到相等的元素,则返回元素在表中的序号

若找不到, 则返回一个“空序号”, 例如: 0

4.顺序表的基本操作——按内容查找算法

算法思想

采用顺序查找实现，即从第一个元素开始，依次将表中元素与e比较，若相等则查找成功，否则失败。

```
int LocateElem_Sq(SeqList L, ElemType e,
                  Status(*compare)(ElemType, ElemType) )
{ //找到后返回位序，否则返回0
  i=1; //i的初始值是第一个元素的位序
  p=L.elem; //p的初始值是第一个元素的存储位置
  while(i<=L.length &&!(*compare)(*p++ , e)) ++i;
  if (i<=L.length) return i;
  else return 0; //返回的是存储位置
} //LocateElem_Sq
```

4.顺序表的基本操作——插入算法

功能描述:

在线性表第 i ($1 \leq i \leq n+1$)个元素之前插入一个新的数据元素 x 。

插入 x 前 $(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$ 长度= n



插入 x 后 $(a_1, a_2, \dots, a_{i-1}, x, a_i, \dots, a_n)$ 长度= $n+1$

4.顺序表的基本操作——插入算法

算法思路:

1. 判断插入位置的合法性，判断条件是

$$1 \leq i \leq n+1$$

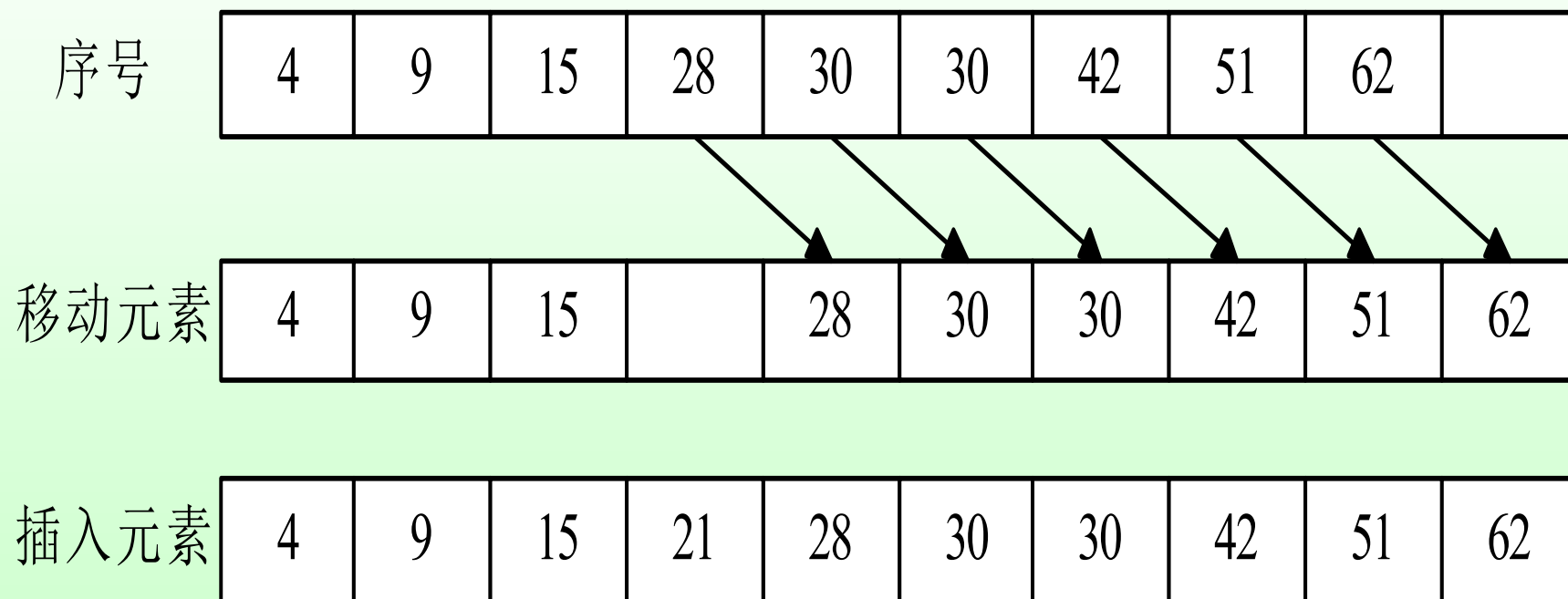
2. 从表尾到*i*逐个依次向右移动数据元素
($n-i+1$ 个元素)

$$a_{j+1} = a_j; \quad j = n, n-1, \dots, i$$

3. 插入数据元素*x*并修改表的长度

$$a_i = x; \quad n = n+1;$$

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

4.顺序表的基本操作——插入算法

```

Status ListInsert_Sq(Sqlist L, int i, ElemType e)
{ if (i<1 || i>L.length+1) return ERROR;
  if (L.length>=L.listsize)//当前存储空间已满
  { newbase=(ElemType *) realloc(L.elem,
    (L.listsize +LISTINCREMENT)*sizeof(ElemType));
    if (!newbase) exit(OVERFLOW);//分配不成功
    L.elem= newbase;
    L.listsize += LISTINCREMENT; }
  q=&(L.elem[i-1]); //q所指向的是第 i 个元素
  for(p=&(L.elem[L.length-1]);p>=q;--p) //从最后一个元素移动
    *(p+1)=*p;
    *q=e;
    ++L.length;
    return OK;
  }//ListInsert_Sq
    
```

插入算法性能分析:

主要时间消耗在移动数据元素上，因此我们将其作为基本操作对插入算法进行时间复杂度估计。

当插入位置为*i*时，移动次数

$$f(n) = n - i + 1$$

若*i=n+1*，无需移动结点，直接插入即可

$$T(n)=O(1)$$

若*i=1*，需移动全部*n*个结点

$$T(n)=O(n)$$

设在第*i*个元素之前插入一个元素的概率为 p_i ，则插入一个元素所需移动元素次数的期望值为：

$$E_i = \sum_{i=1}^{n+1} p_i(n - i + 1)$$

若假定在线性表中任何一个位置上进行插入的概率都是相等的，则移动元素的期望值为：

$$\begin{aligned} E_i &= \sum_{i=1}^{n+1} p_i(n - i + 1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) \\ &= \frac{1}{n+1} \frac{(n-1+1) + (n-(n+1)+1)}{2} (n+1) \\ &= \frac{n}{2} \end{aligned}$$

$$T(n) = O(n)$$

4.顺序表的基本操作——删除算法

功能描述：

删除线性表中第 i ($1 \leq i \leq n$)个元素。

($a_1, a_2, \dots, a_{i-1}, \cancel{a_i}, a_{i+1}, \dots, a_n$) 长度= n



($a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n$) 长度= $n-1$

顺序表的删除：删除第5个元素，

序号

1 2 3 4 5 6 7 8 9 10

4	9	15	21	28	30	30	42	51	62
---	---	----	----	-----------	----	----	----	----	----

删除 28 后

4	9	15	21	30	30	42	51	62	
---	---	----	----	----	----	----	----	----	--

顺序表中删除元素

4.顺序表的基本操作——删除算法

```

Status ListDelete_Sq(Sqlist &L, int i, ElemType &e)
{ //在顺序表L中删除第i个元素,并用e返回其值
  //i的合法值为  $1 \leq i \leq \text{ListLength\_Sq}(L)$ 
  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;
  }//ListDelete_Sq
    
```

删除算法性能分析:

主要时间消耗在移动数据元素上，因此我们将其作为基本操作对删除算法进行时间复杂度估计。

当删除位置为*i*时，移动次数

$$f(n) = n - i$$

若*i=n*，无需移动结点,直接将表长度-1即可

$$T(n)=O(1)$$

若*i=1*，需移动全部*n-1*个结点

$$T(n)=O(n)$$

设删除第*i*个元素的概率为 Q_i ，则删除一个元素所需移动元素次数的期望值为：

$$E_i = \sum_{i=1}^n Q_i(n - i)$$

若假定在线性表中任何一个位置上进行删除的概率都是相等的，则移动元素的期望值为：

$$E_{del} = \sum_{i=1}^n Q_i(n - i) = \frac{1}{n} \sum_{i=1}^n (n - i) = \frac{n-1}{2}$$

$$T(n)=O(n)$$

例2.3 顺序表合并

已知线性表 LA 和线性表 LB 中的数据元素按值非递减有序排列，现要求将 LA 和 LB 归并为一个新的线性表 LC ，且 LC 中的元素仍按值非递减有序排列。

其实就是例2.2通过顺序表的具体实现。

介绍一个概念：有序表

若线性表中的数据元素相互之间可以比较，并且数据元素在线性表中依值非递减或非递增有序排列，即 $a_i \geq a_{i-1}$ 或 $a_i \leq a_{i-1}$ ($i = 2, 3, \dots, n$)，则称该线性表为有序表(Ordered List)。

问题分析：

在归并的过程中，**不断**将线性表 LA 和 LB 中待归并的数据元素之间的**最小值**并入到 LC 中，直到归并完 LA 和 LB 的所有数据元素。

算法思路:

1. 设置两个指针 i, j 分别标记 LA 和 LB 中待归并数据元素的当前位置, 指针 k 标志 LC 长度;

$i = 1; j = 1; k = 0;$

2. 分别取出当前 LA 与 LB 中待归并元素;

$GET(LA, i, ai); GET(LB, j, bj);$

3. 比较 ai 与 bj 的大小, 将其中较小的元素并入 LC 中, 并使相应的待排序元素指针后移。转第2步;

$\left\{ \begin{array}{ll} INSERT(LC, ++k, ai) & ai \leq bj \\ INSERT(LC, ++k, bj) & ai > bj \end{array} \right.$

```

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(Lc.listsize*sizeof(ElemType));
    if(!Lc.elem) exit(OVERFLOW);
    pa_last=La.elem+La.length -1;  //指向顺序表La最后一个元素
    pb_last=Lb.elem+Lb.length -1; //指向顺序表Lb最后一个元素
    while( pa<=pa_last && pb<=pb_last )
    {
        if (*pa<=*pb) *pc++=*pa++;
        else *pc++=*pb++;
    }
    while(pa<=pa_list) *pc++=*pa++;
    while(pb<=pb_list) *pc++=*pb++;
} //MergeList_Sq
    
```

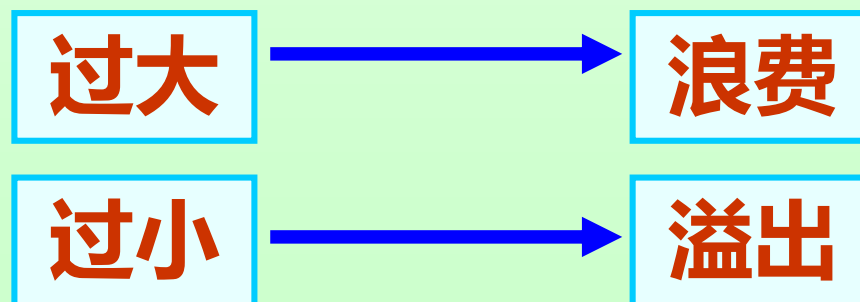
顺序存储结构的优缺点分析

优点:

1. 无需为表示结点间的逻辑关系而增加额外的存储空间;
2. 可方便地随机存取表中的任一元素。

缺点:

1. 插入或删除平均需要移动一半的结点;
2. 存储分配只能预先进行静态分配



课堂练习

已知A、B、C为三个元素值递增有序的顺序表，现要求对A作如下运算，删去那些既在B中出现又在C中出现的元素，实现上述算法并分析时间复杂度。

$$A = A - (B \cap C)$$

$$A = (1, 2, 6, 6, 8, 9, 10, 10, 11, 15)$$

$$B = (1, 2, 6, 6, 7, 9, 10, 15)$$

$$C = (3, 4, 6, 7, 7, 9, 9, 9, 10, 12)$$

$$A = (1, 2, 8, 11, 15)$$

分析:

- 先从B和C中找出公有元素,记为same;
- A中从当前位置开始, 凡小于same的元素均保留(存到新的位置),等于same的跳过;
- 大于same时就再找下一个same.

```

void SqList_Intersect_Delete( SqList &A, SqList B, SqList C)
{
    pa = A.elem; pa_last; pb; pb_last; pc; pc_last; p0 = A.elem;
    while((pa<=pa_last) && (pb<=pb_last) && (pc<=pc_last)) {
        if (*pb < *pc)  pb++;
        else if (*pb > *pc)  pc++;
        else {
            same = *pb;
            while((pb<=pb_last) && (*pb == same))    pb++;
            while((pc<=pc_last) && (*pc == same))    pc++;
            while((pa<=pa_last) && (*pa < same))
                *p0++ = *pa++;
            while((pa<=pa_last) && (*pa == same))  pa++;
        }//else
    }//while
    while(pa <= pa_last)
        *p0++ = *pa++;
    A.length = p0 - A.elem;
} // p0 = A.elem 就地利用原空间
    
```

2.3

线性表的链式表示和实现

2.3 线性表的链式表示和实现

2.3.1 线性链表

2.3.2 循环链表

2.3.3 双向链表

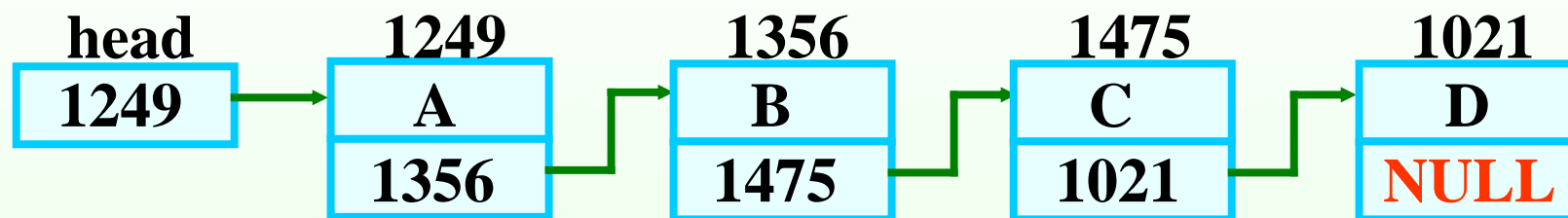
2.3.1 线性链表

1. 基本概念

定义：采用链式存储结构的线性表称为链表，其采用一组任意的存储单元来存放线性表的数据元素（可零散的分布于内存单元的任何位置上）。

特点：链表中结点的逻辑次序和物理次序不一定相同。即：逻辑上相邻未必在物理上相邻。

链表概述：链表是一种常见的重要的数据结构。
它是动态地进行存储分配的一种结构。



1. **“头指针”**，以变量**head**表示，它存放一个地址，该地址指向一个称为**“结点”**的元素；
2. 每个结点都应包括**两部分**：用户需要用的**实际数据**和下一结点的地址(**指针域**)；
3. **最后一个结点**，该结点的指针域不再指向其他结点，它称为**“表尾”**，它的指针域放一个**“NULL”**(表示**“空地址”**)，链表到此结束。

2. 结点(Node)组成



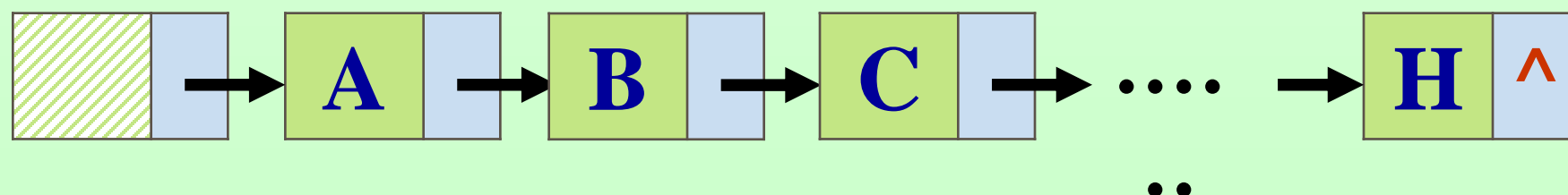
数据域——存储数据元素本身

指针域——存储邻接元素的存储位置

通过指针域，不论结点的物理次序如何，都可将其按逻辑顺序依次链接在一起构成线性表。

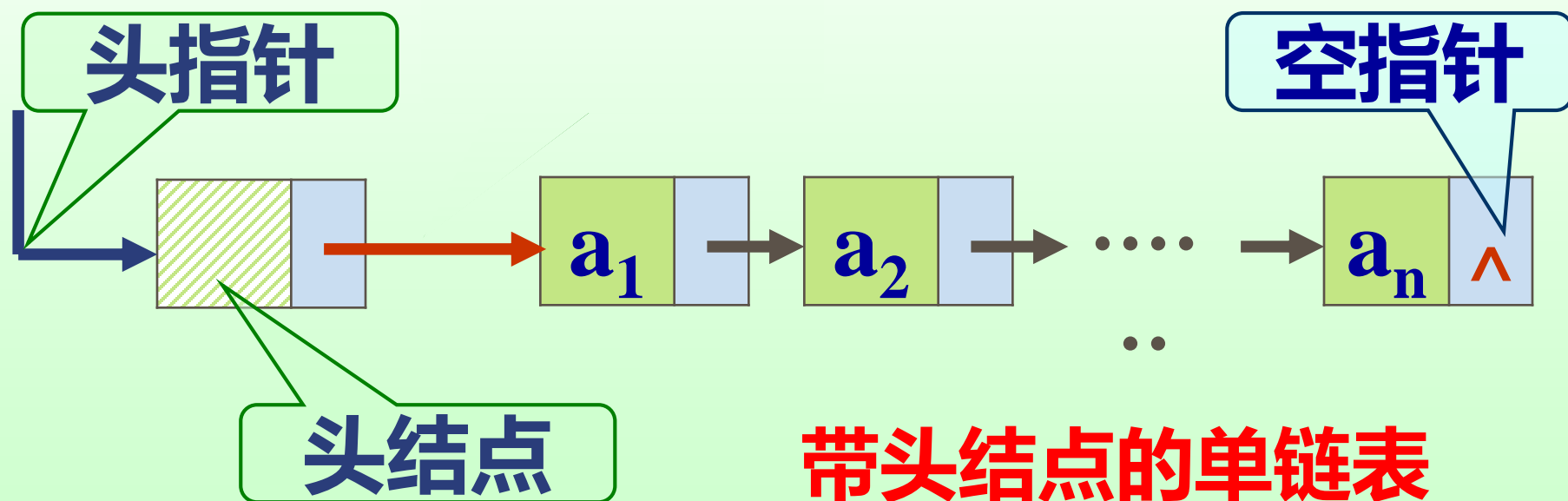
线性链表的物理状态示意图

	存储地址	数据域	指针域
	1	D	43
	7	B	13
头指针H	13	C	1
<div>31</div>	19	H	NULL
	25	F	37
	31	A	7
	37	G	19
	43	E	25



3. 单链表(线性链表)

定义：所有结点的指针域中只包含一个指针
(存储直接后继结点的存储位置)的链表。




带头结点的单链表

单链表常用**头指针**来命名，如 La , $Head$.

4. 单链表示例($a_1, a_2, \dots, \dots, a_n$)

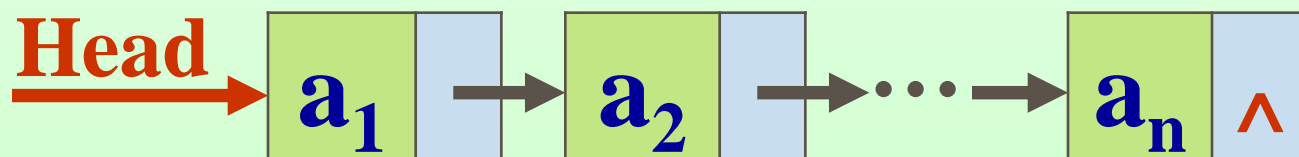
空表($n=0$):

不带头结点: **Head=NULL**

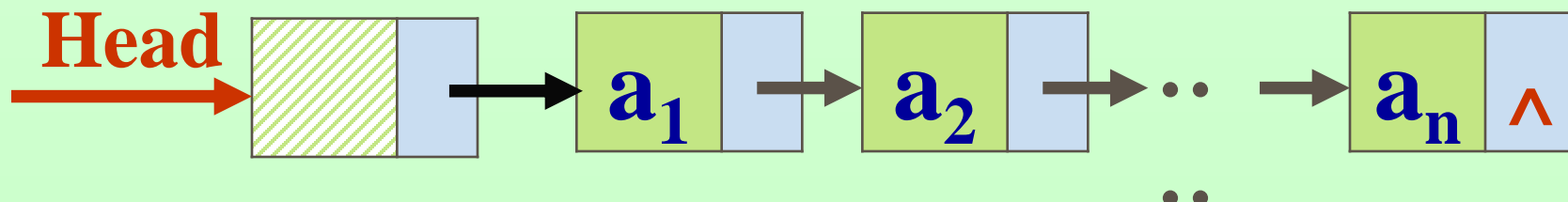
带 头结点: **Head** 

非空表($n>0$):

不带头结点:



带 头结点:



存储空间的分配和释放

它们的原型说明在 “**stdlib.h**” 头文件和 “**alloc.h**” 头文件中，使用下面的函数时，应把头文件包含到源程序中。

❖ malloc 函数

```
void * malloc(unsigned int size);
```

❖ free 函数

```
void free(void *p);
```

申请一个结点

如 `LinkList s;` //注意：此时的s是指针变量

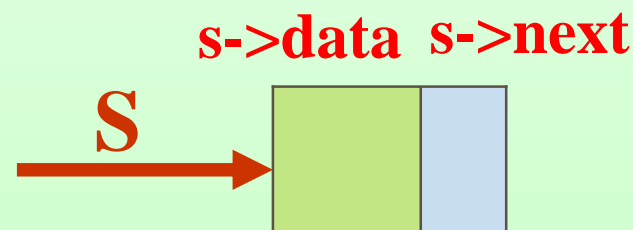
则语句：

`s=(linklist) malloc(sizeof(LNode))` 表示申请了一个结点空间

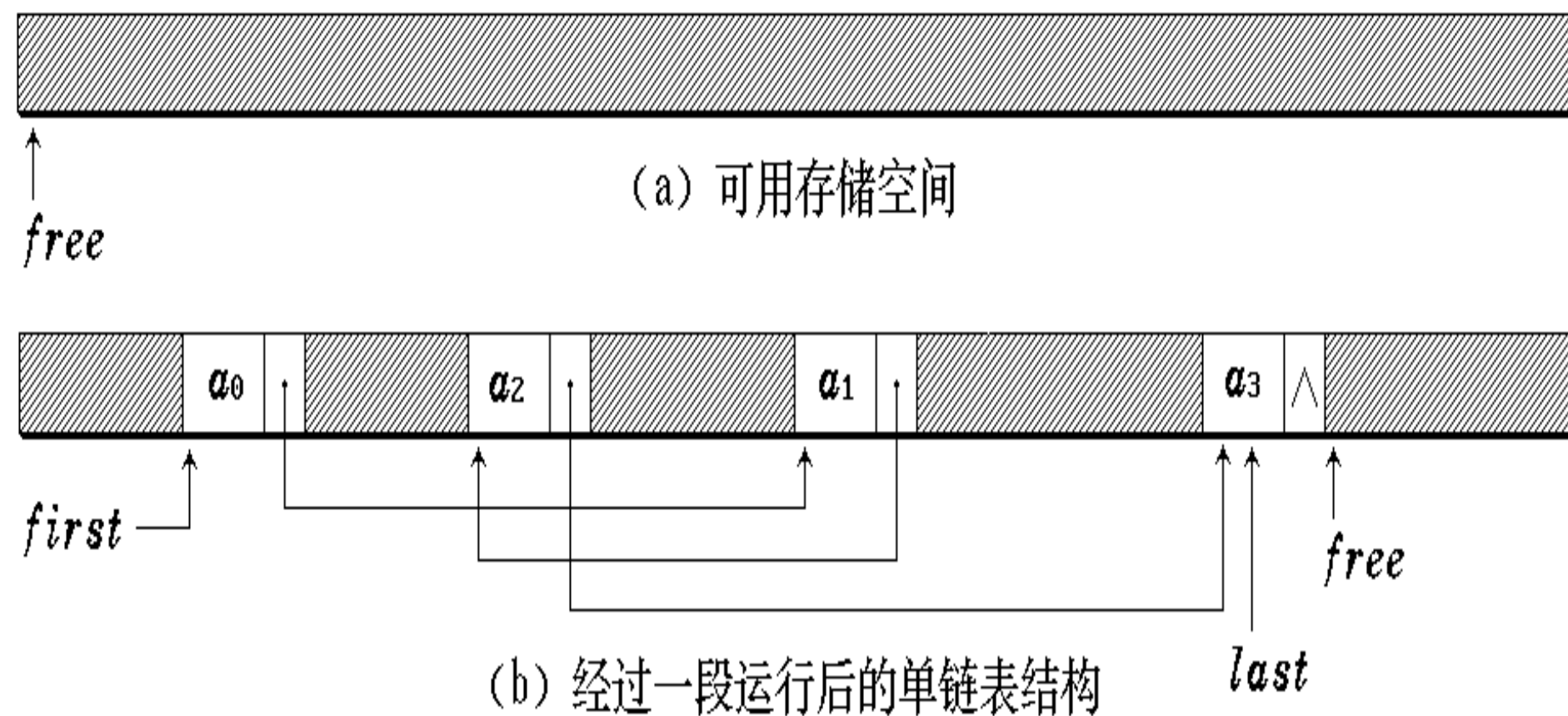
`free(s)` 表示释放 s所指的结点

数据域为 `(*s).data` 或 `s->data`

指针域为 `(*s).next` 或 `s->next`



单链表的存储映象



5. 单链表的描述:

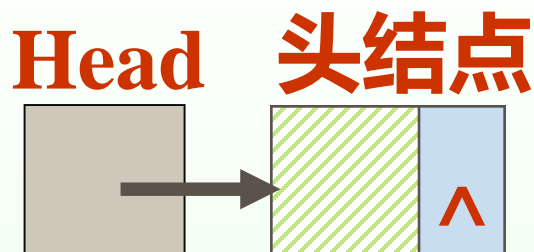
在C语言中, 可以用 “**结构指针**” 来描述

```
typedef struct LNode
{
    ElemType data; //数据域, 存储数据元素
    struct LNode * next;
                    //指针域, 记录后继结点的存储位置
} LNode, *LinkList;
```

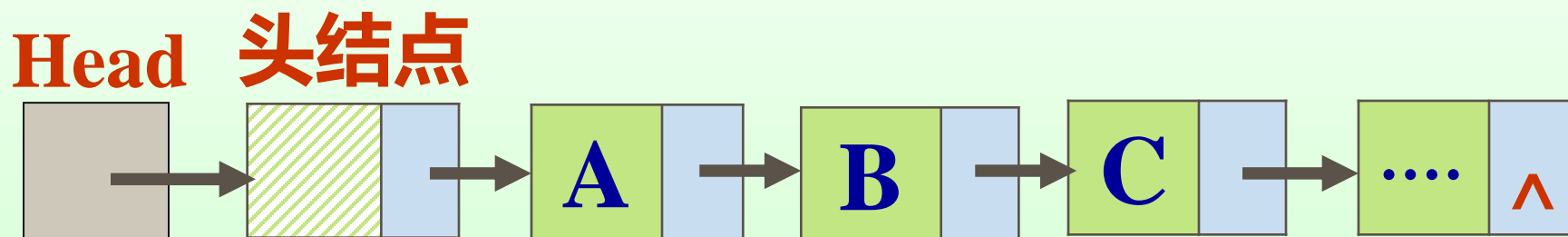
数据元素类型根据需要而定

思考：如何建立带头结点的单链表？

空链表



非空链表



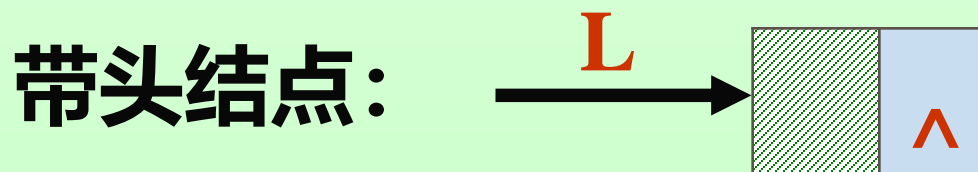
1. 头指针 **Head** 始终是指向头结点的 **非空指针**;
2. 头结点的指针始终指向链表的第一个结点。
如果链表为空，则其为NULL;

6. 单链表基本操作——初始化单链表

```

LinkedList init_linklist(LinkedList L) //对带头结点
                                         单链表进行初始化
{
    L=(LinkedList)malloc(sizeof(Node)); //申请空间
    L->next=NULL;                        //置为空表
    return L;
}
    
```

空表($n=0$):



6. 单链表基本操作——销毁表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);
}
```

6. 单链表基础

(1) 尾插法

建立链表就是根据需要一个一个地开辟新结点，在结点中存放数据并建立结点之间的链接关系。

算法思路：从一个空表开始，每读入一个数据，生成新结点，将读入数据存放在新结点的数据域中，然后将新结点**插入到当前链表的最后一个结点之后**，依此类推，直到处理完所有数据。

生成的结点次序和输入的顺序相同

例：(A, B, C, D, ..., Z)



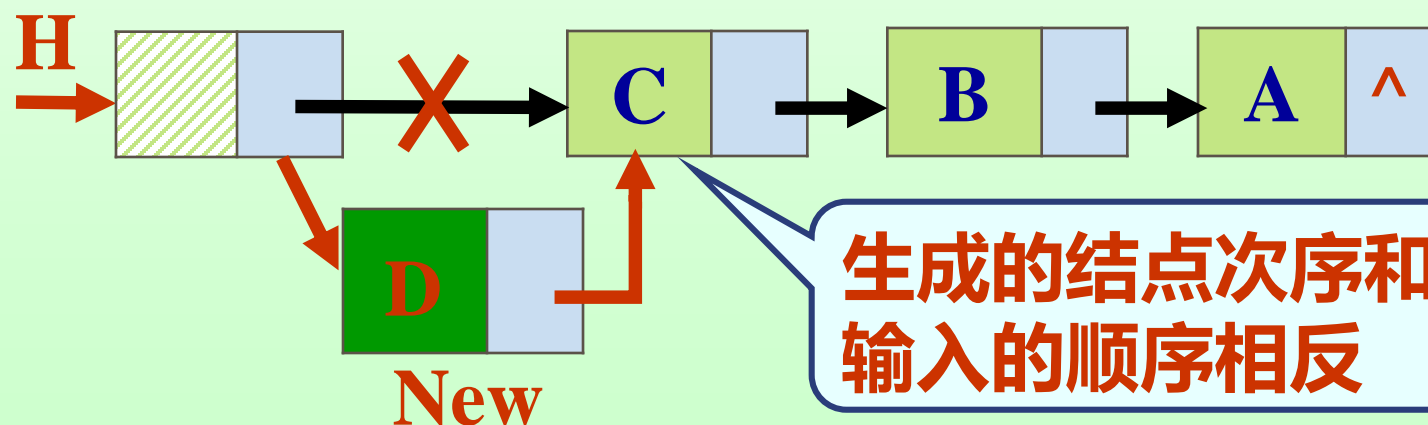
6. 单链表基本操作——尾插法建立单链表

```
void CreateFromTail(LinkList L)
{
    LNode *r, *s;   char c;   int flag=1;   r=L;
    while(flag)
    {
        c=getchar();
        if(c!='$')
        {
            s=(LNode*)malloc(sizeof(LNode));
            s->data=c;
            r->next=s;
            r=s;
        }
        else
        {
            flag=0;
            r->next=NULL;
        }
    }
}
```

6. 单链表基本操作——建单链表

(2) 头插法

算法思路： 从一个空表开始，每读入一个数据，生成新结点，将读入数据存放在新结点的数据域中，然后将新结点**插入到当前链表的第一个结点之前**，依此类推，直到处理完所有数据。例：(A, B, C, D, ..., Z)



6. 单链表基本操作——头插法建立单链表

```

void CreateFromHead(LinkList L)
{
    LNode *s;  char c;  int  flag=1;
    while(flag)
    {
        c=getchar();
        if(c!='$')
        {
            s=(LNode*)malloc(sizeof(LNode));
            s->data=c;
            s->next=L->next
            L->next=s;  }
        else
            flag=0;
    }
}
    
```

6. 单链表基本操作——取元素

在单链表中，取得第 i 个元素必须从头指针出发寻找，因此它是非随机存取的存储结构

算法思路：从链表的第一个元素结点起，判断当前结点是否是第 i 个，若是，则返回该结点的指针，否则继续后一个，直到表结束为止。没有第 i 个结点时返回空。

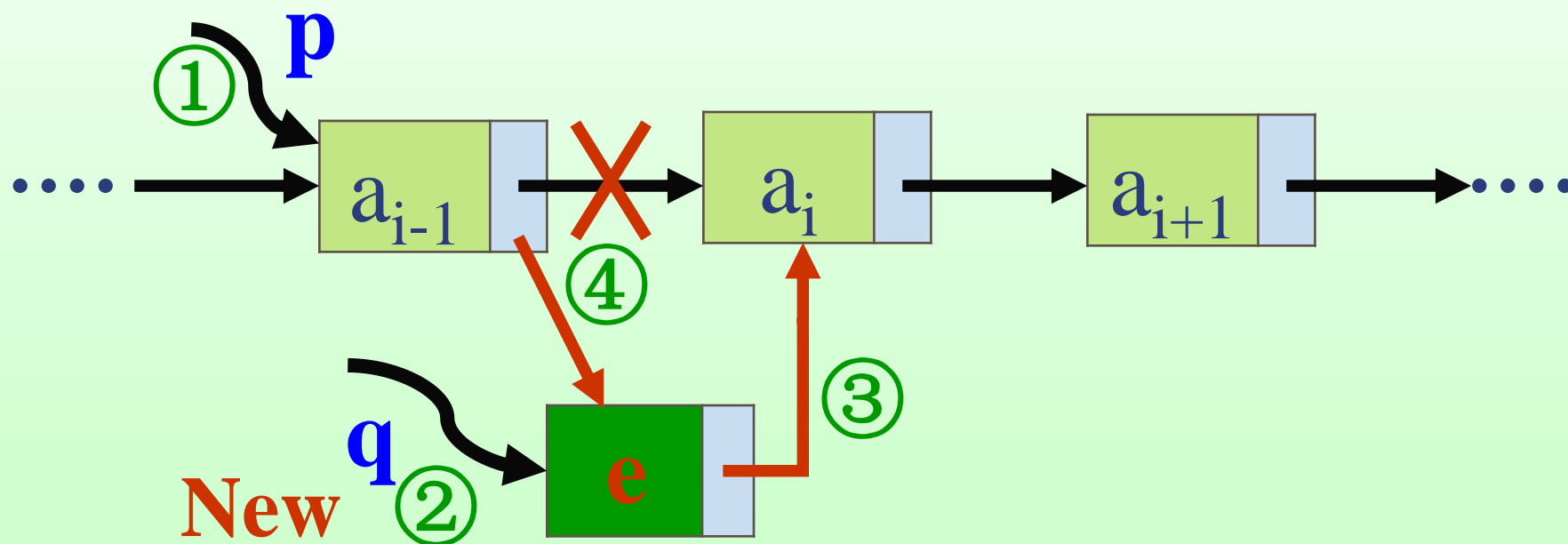
6. 单链表基本操作——取元素

```
Status GetElem_L(LinkList L,int i ,ElemType &e)
{ //L为带头结点的单链表的头指针
    p=L->next; j=1;
    while( p && j<i)
    {
        p=p->next; ++j;
    }
    if (!p|| j>i) return ERROR; //第i个元素不存在
    e = p->data;
    return OK;
} //GetElem_L           // 时间复杂度为O(n)
```

6. 单链表基本操作——插入结点

操作名称: **ListInsert(L, i, e)**

算法思路:



6. 单链表基本操作——插入结点

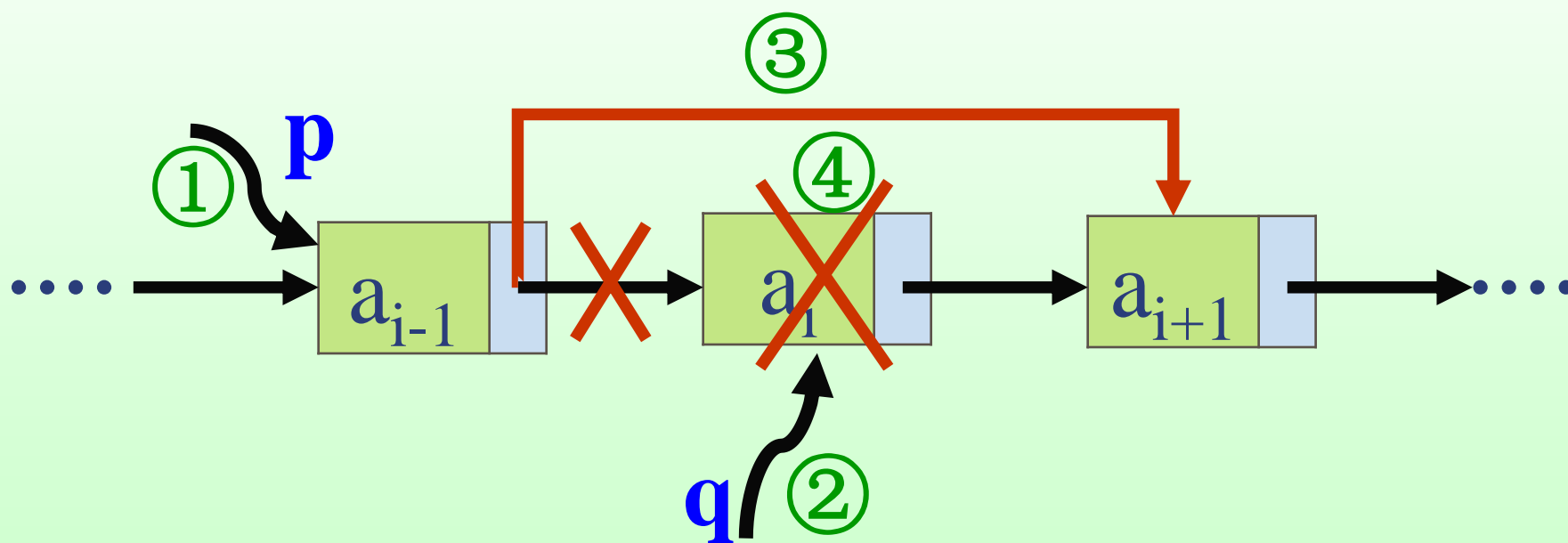
```

Status ListInsert_L(LinkList &L, int i, ElemTyp e)
{ //在带头结点的单链表L中第i个位置之前插入元素e
    p=L;j=0;
    while (p && j<i-1) //当循环结束时p可能指向第i-1个元素
    {
        p=p->next;
        ++j;
    }
    if(!p || j>i-1) return ERROR;
    s=(LinkList ) malloc(sizeof(LNode));
    s->data=e; s->next=p->next; p->next=s;
    return OK;
} //ListInsert_L
    
```

6. 单链表基本操作——删除结点

操作名称: **ListDelete(L, i, e)**

算法思路:



6. 单链表基本操作——删除结点

```

Status ListDelete_L(Linklist &L, int i, ElemTyp  &e )
//在带头结点单链表 L 上删除第i个结点,并由e返回其值
{
    p=L;j=0;
    while (p->next  && j<i-1 )
    {
        p=p->next;++j;
    }
    if(!p->next || j>i-1 ) return ERROR;
    q=p->next; p->next=q->next;
    e=q->data;  free(q);
    return OK;
}
//ListDelete_L
    
```

思考1：

试以带头结点的单链表为例，分析**插入结点**算法的时间复杂度 $T(n)$ ？

问题分析：

在第 i 结点之前插入时，主要时间花费在寻找第 $i-1$ 个结点上，为基本操作，执行次数为

$$f(n) = i-1, 1 \leq i \leq n+1$$

若 $i=1$ ，无需寻找，直接插入即可

$$T(n)=O(1)$$

若 $i=n+1$ ，需遍历全部 n 个结点

$$T(n)=O(n)$$

具体同顺序表

思考2：

采用链式存储方式时，插入与删除算法的效率并没有提高($O(n)$)，关键问题在于需要寻找第 $i-1$ 个结点的位置。假如我们将插入(删除)点位置不是以逻辑序号 i 的形式给出，而是以第 i 个结点的地址 p 形式给出，那么可以使得算法效率提高为 $O(1)$ 。

如何将两个有序链表合并为一个有序链表？

已知单链表 LA 和 LB 中的数据元素按值非递减有序排列，现要求将 LA 和 LB 归并为一个新的单链表 LC，且 LC 中的数据元素仍按值非递减有序排列。

```
void MergeList_L (LinkList &La, LinkList &Lb, LinkList &Lc)
```

```
{ // 已知单链线性表La和Lb的元素按值非递减排列。
```

```
  pa = La->next;  pb = Lb->next;
```

```
  Lc = pc = La;    // 用La的头结点作为Lc的头结点
```

```
  while (pa && pb)
```

```
  {
```

```
    if (pa->data <= pb->data)
```

```
      {   pc->next = pa;   pc = pa;   pa = pa->next;   }
```

```
    else {   pc->next = pb;   pc = pb;   pb = pb->next;   }
```

```
  }
```

```
  pc->next = pa ? pa : pb; // 插入剩余段
```

```
  free(Lb);           // 释放Lb的头结点
```

```
} // MergeList_L
```

存在的问题及解决方案

存在的问题：

1. 单链表的表长是一个隐含的值；
2. 在单链表的最后一个元素之后插入元素时，需遍历整个链表；
3. 在链表中，元素的“位序”概念淡化，结点的“位置”概念加强。

改进链表的设置：

1. 增加“表长”、“表尾指针”和“当前位置的指针”三个数据域；
2. 将基本操作中的“位序 i ”改变为“指针 p ”。

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

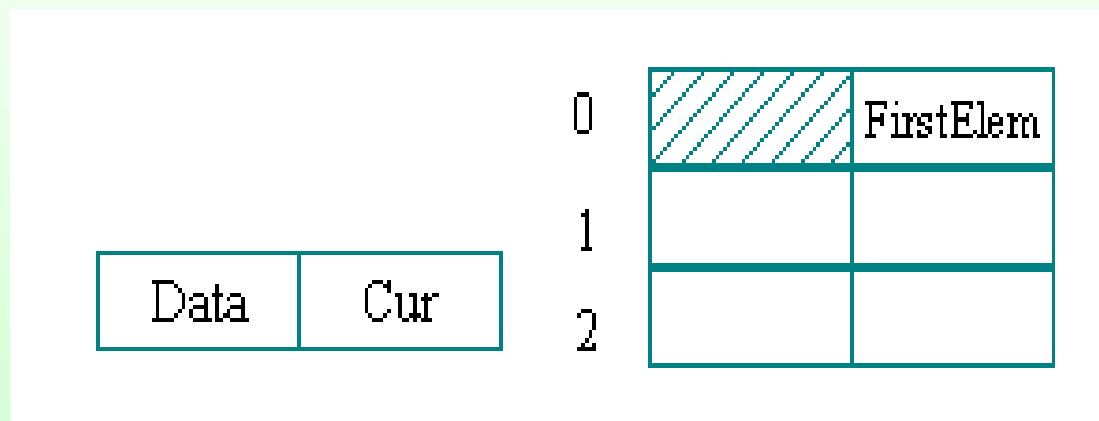
```
q = L->next;
L->next = q->next;
free(q);
```

□ 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);
```

7. 静态链表

定义：用数组描述的链表称为**静态链表**。其中链表中的结点为数组中的一个分量，游标(Cursor)代替指针表示结点在数组中的相对位置。



数组的第零个分量作为头结点，这种结构仍需**预先分配**一个较大的空间，但插入和删除不需移动结点，仅需修改游标。

线性表的静态单链表存储结构

```
#define MAXSIZE 1000
typedef struct
{
    ElemType data;
    int cur;
} component, SLinkList[ MAXSIZE];
```

下图显示了插入元素 “SHI”和删除元素 “ZHENG”前后的变化

修改前

0		1
1	Zhao	2
2	Qian	3
3	Sun	4
4	Li	5
5	Zhou	6
6	Wu	7
7	Zheng	8
8	Wang	0
9		
10		

修改后

0		1
1	Zhao	2
2	Qian	3
3	Sun	4
4	Li	9
5	Zhou	6
6	Wu	8
7	Zheng	8
8	Wang	0
9	Shi	5
10		

静态链表中的运算——按值定位

```
int LocateElem_SL( SLinkList S, ElemType e)
{
    // 在静态单链线性表S中查找第1个值为e的元素。
    // 若找到，则返回它在S中的位序，否则返回0。

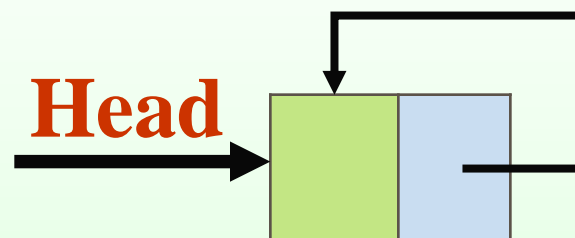
    int i;
    i = S[0].cur;                // i指示表中第一个结点
    while (i && S[i].data != e) i = S[i].cur; // 在表中顺链查找
    return i;

} // LocateElem_SL
```

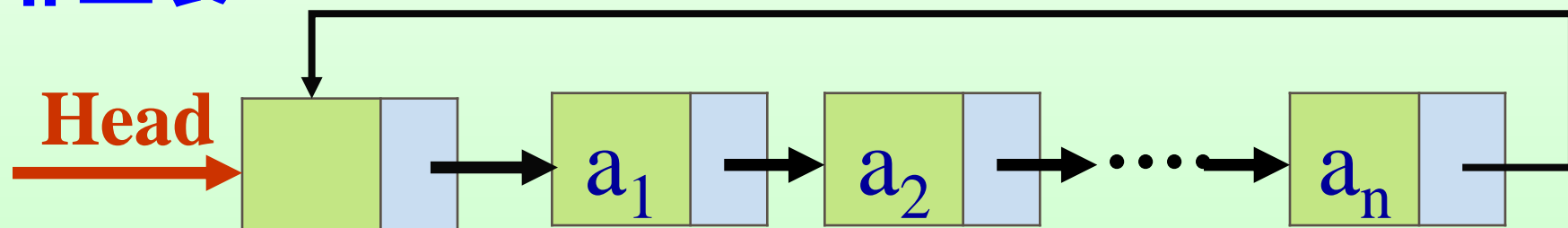
2.3.2 循环链表

特点：最后一个结点的指针指向头结点，整个链表构成了一个封闭的环。

空表：



非空表：



基本操作同线性链表，区别在于判别链表中最后一个结点的条件是“后继是否为头结点”

例：有两个带头结点的循环单链表LA, LB，编写算法，将两个循环单链表合并为一个循环单链表，其头指针为LA。LA, LB均是循环单链表的头指针。

```

LinkedList merge_1 (LinkedList LA, LinkedList LB)
{
    Node *p, *q;    p=LA;    q=LB;
    while (p->next!=LA)    p=p->next; /*找到LA的表尾*/
    while (q->next!=LB)    q=q->next; /*找到LB的表尾*/
    q->next=LA;
    p->next=LB->next;
    free(LB);
    return(LA);
}
    
```

$$T(n)=O(n)$$

例：有两个带头结点的循环单链表RA, RB，编写算法，将两个循环单链表合并为一个循环单链表，其尾指针为RB。RA, RB均是循环单链表的尾指针。

```
LinkList merge_2 (LinkList RA, LinkList RB)
```

```
{  Node *p;
    p=RA->next;          /*保存链表RA的头结点地址*/
    RA->next=RB->next->next
    free(RB->next);       /*释放链表RB的头结点*/
    RB->next=p;
    return RB;           /*返回新循环链表的尾指针*/
}
```

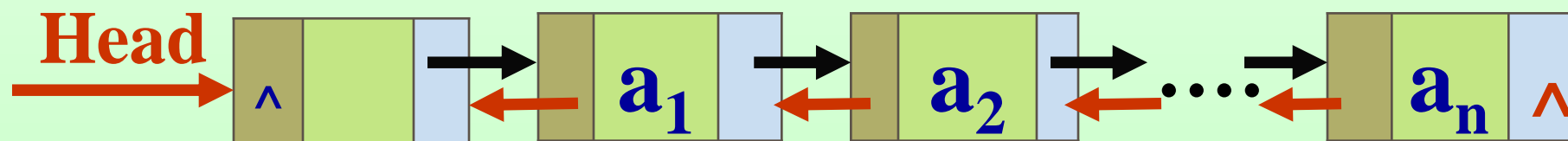
$T(n)=O(1)$

2.3.3 双向链表

特点：每个结点的指针域里包含两个指针，其中一个指向前驱结点，另外一个指向后继结点。



非空表：



双向链表的描述

```
typedef struct DuLNode
{
    ElemType data;
    struct DuLNode *prior ;
    struct DuLNode *next;
}DuLNode, * DLinkList;
dlinklist *head;
```

和单链表类似，双链表一般也是由头指针head唯一确定。

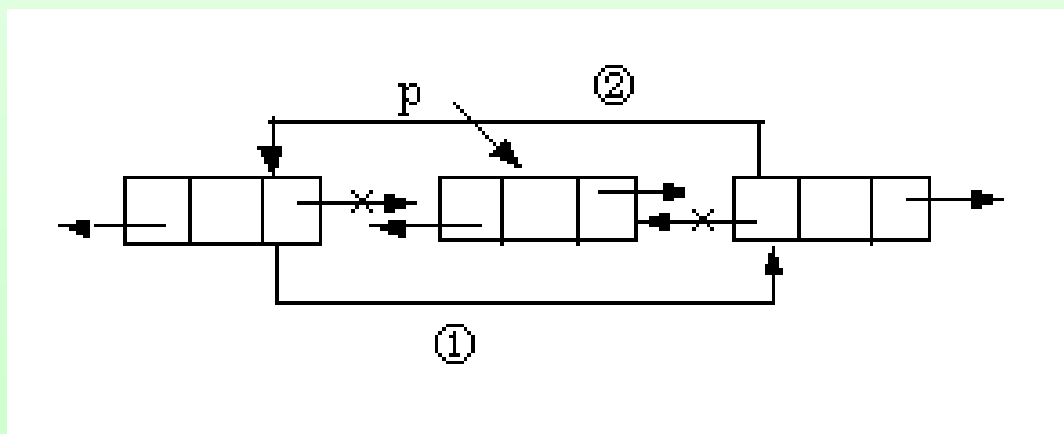
双向链表操作的特点:

1. “查询” 和单链表相同。
2. “插入” 和 “删除” 时需要同时修改两个方向上的指针。

双向链表中结点的删除

设 p 指向双向链表中某结点，删除 $*p$ 。
操作如下：

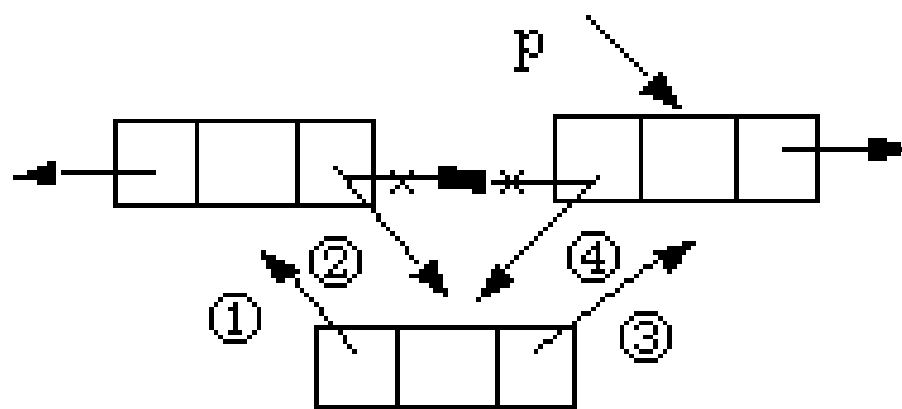
- ① $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$
 - ② $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$
- $\text{free}(p);$



双向链表中结点的插入：

设 p 指向双向链表中某结点， s 指向待插入的值为 x 的新结点，将 $*s$ 插入到 $*p$ 的前面，插入示意图如下图所示。

- ① $s \rightarrow \text{prior} = p \rightarrow \text{prior};$
- ② $p \rightarrow \text{prior} \rightarrow \text{next} = s;$
- ③ $s \rightarrow \text{next} = p;$
- ④ $p \rightarrow \text{prior} = s;$

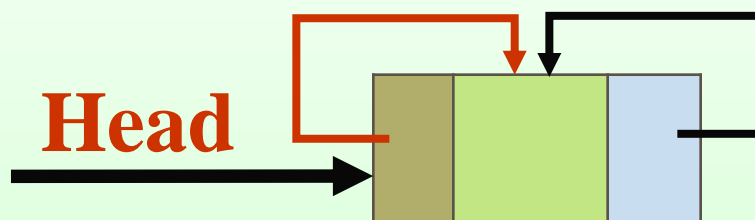


双向链表中的结点插入

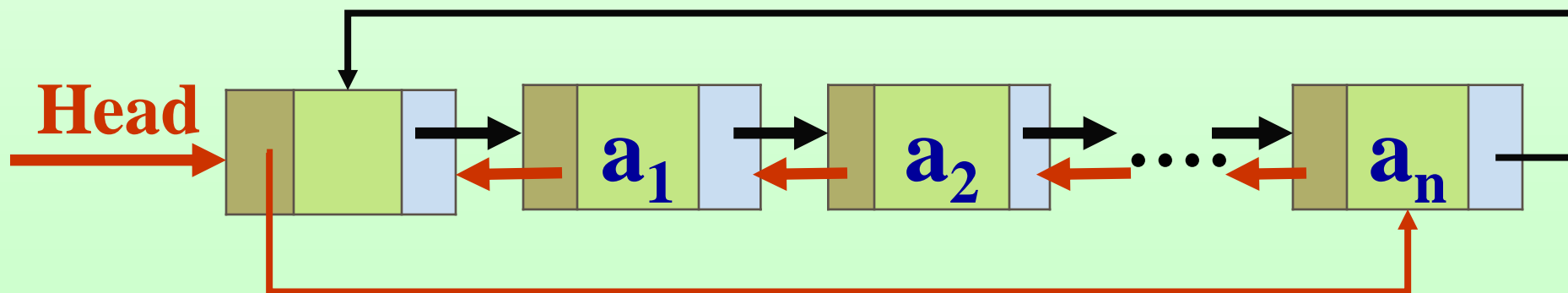
双向循环链表

特点：在双向链表的基础上，最后一个结点的next指针指向头结点，而头结点的prior指针指向最后一个结点。

空表：



非空表：



双向循环链表

$p \rightarrow \text{prior} \rightarrow \text{next} = p = p \rightarrow \text{next} \rightarrow \text{prior}$

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。

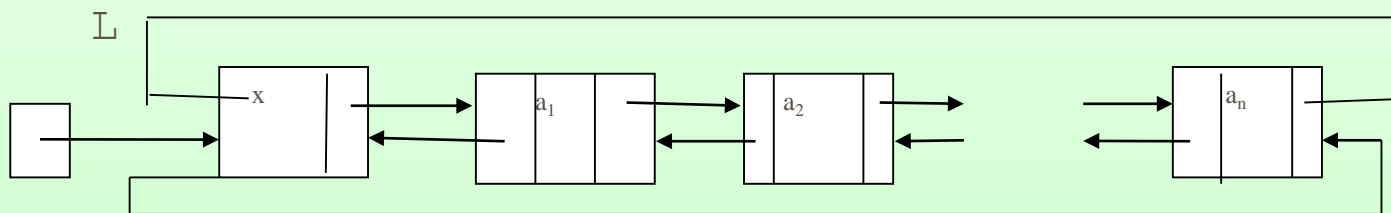
```
void   SToDouble (DuLinkList   la)
```

```
// 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 $a_1 < x < a_n$ ）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 ($a_1 < x < a_n$) 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)

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;
}
    
```

链表存储的特点

1. 基于存储的考虑

链表不用事先估计存储规模，但链表的存储密度较低。

2. 基于运算的考虑

链表中按序号访问的时间性能 $O(n)$

3. 基于环境的考虑

顺序表容易实现，任何高级语言中都有数组类型，链表的操作是基于指针的

一般原则：

- “较稳定”的线性表选择顺序存储
- 频繁做插入删除（即动态性较强）的线性表宜选择链式存储。

练习：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

2.4

一元多项式的表示及相加

主要讨论利用线性链表的基本操作
来实现一元多项式的运算

1、一元多项式

$$p_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

在计算机中，可以用一个线性表来表示：

$$P = (p_0, p_1, \dots, p_n)$$

但是对于形如

$$S(x) = 1 + 3x^{10000} - 2x^{20000}$$

的多项式，上述表示方法是否合适？

一般情况下的一元稀疏多项式可写成

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$$

其中： p_i 是指数为 e_i 的项的非零系数，

$$0 \leq e_1 < e_2 < \dots < e_m = n$$

可以下列线性表表示：

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

例如:

$$P_{999}(x) = 7x^3 - 2x^{12} - 8x^{999}$$

可用表示的线性表为:

$$((7, 3), (-2, 12), (-8, 999))$$

抽象数据类型一元多项式的定义如下:

ADT Polynomial {

数据对象:

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

**TermSet 中的每个元素包含一个
表示系数的实数和表示指数的整数 }**

数据关系:

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n$

且 a_{i-1} 中的指数值 $<$ a_i 中的指数值 }

基本操作:

CreatPolyn (P, m)

操作结果: 输入 m 项的系数和指数,
建立一元多项式 P 。

DestroyPolyn (P)

初始条件: 一元多项式 P 已存在。

操作结果: 销毁一元多项式 P 。

PrintPolyn (P)

初始条件: 一元多项式 P 已存在。

操作结果: 打印输出一元多项式 P 。

PolynLength(P)

初始条件：一元多项式 P 已存在。

操作结果：返回一元多项式 P 中的项数。

AddPolyn (Pa, Pb)

初始条件：一元多项式 Pa 和 Pb 已存在。

操作结果：完成多项式相加运算，即：

$Pa = Pa + Pb$ ，并销毁一元多项式 Pb。

SubtractPolyn (Pa, Pb)

MultiplyPolyn (Pa, Pb)

} ADT Polynomial

2、一元多项式的实现

用带表头结点的有序链表表示多项式

结点的数据元素类型定义为:

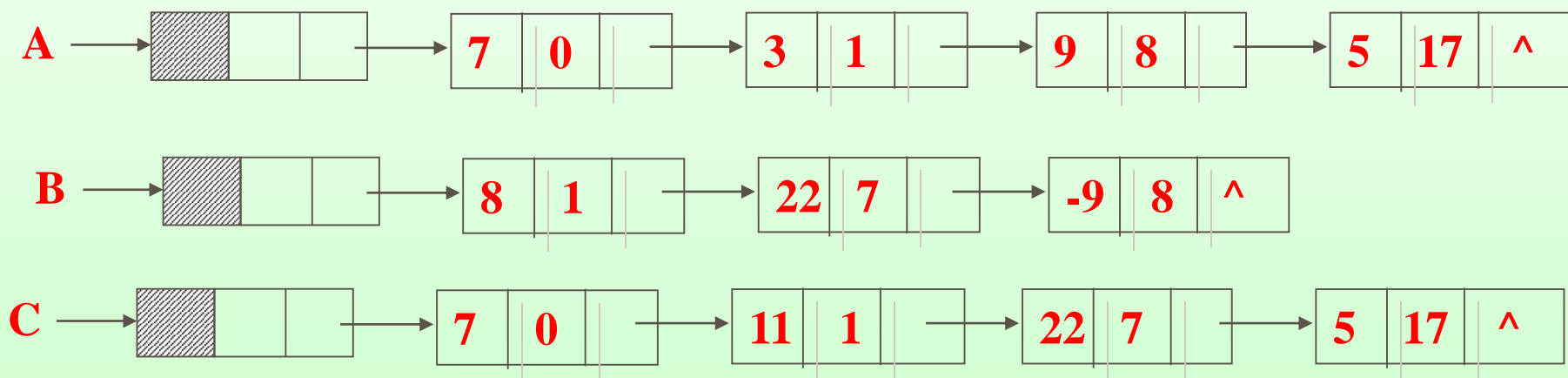
```
typedef struct Polynode
{
    int  coef ; //系数coefficient
    int  exp;   //指数exponent
    struct Polynode *next; //后继结点指针
} Polynode, *Polylist;
```

一元多项式相加

$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

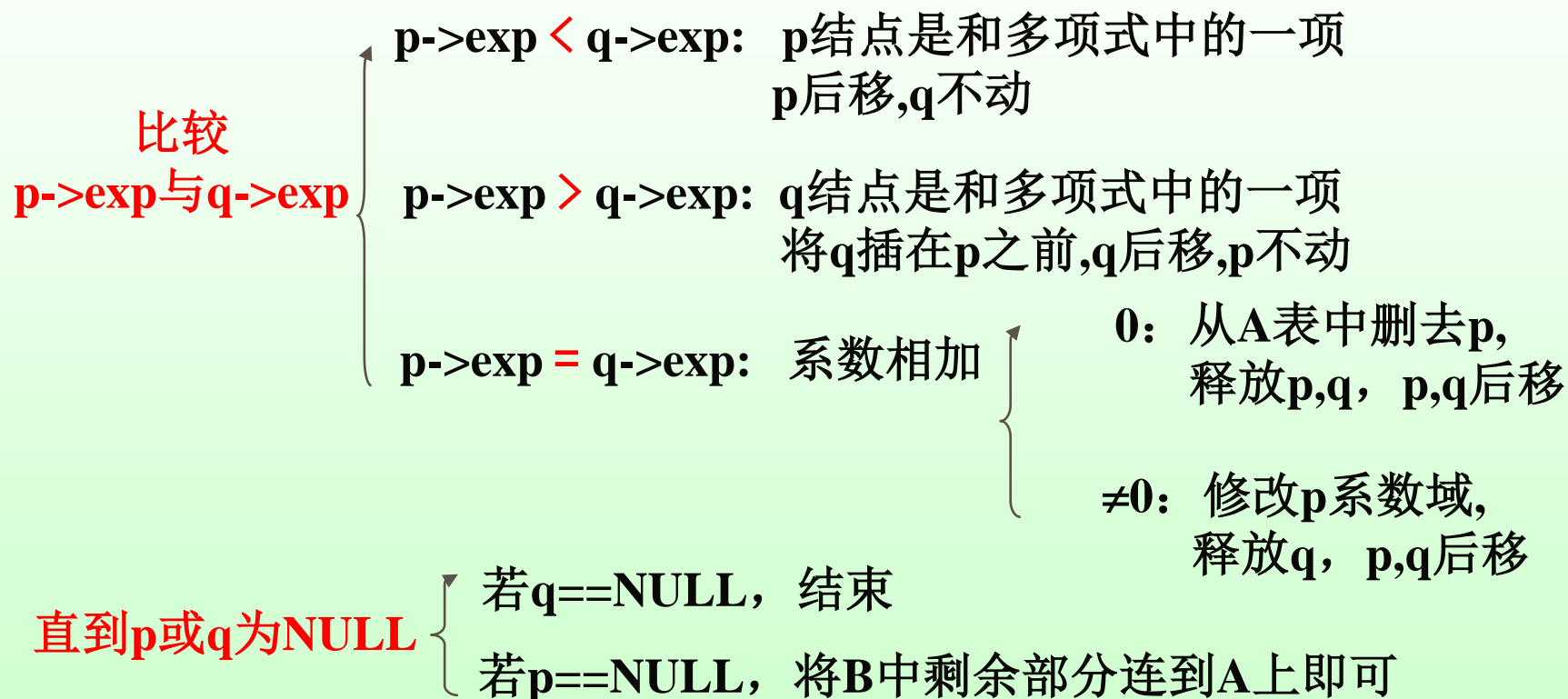
$$B(x) = 8x + 22x^7 - 9x^8$$

$$C(x) = A(x) + B(x) = 7 + 11x + 22x^7 + 5x^{17}$$



运算规则

设 p, q 分别指向 A, B 中某一结点, p, q 初值是第一结点



3.一元多项式相加——主框架

```

Polylist polycrate () ;
void polyprint(Polylist head) ;
void polydestroy(Polylist head) ;
void polyadd(Polylist ha, Polylist hb ) ;
void main(void )
{
    Polylist HA , HB;
    HA=polycrate();
    HB=polycrate();
    polyadd(HA, HB);
    polyprint(HA);
    polydestroy(HA);
}

```

/*A,B的头结点*/

/*创建A*/

/*创建B*/

/*A,B相加*/

/*打印多项式*/

/*删除多项式*/

带头结点的有序链表表示

```

Polylist polycrate( )
{ Polynode *head, *rear, *s;      int c,e;
  head=(Polynode*)malloc(sizeof(Polynode)); /*建立头结点*/
  rear=head; /*rear始终指向单链表的尾,便于尾插法建表*/
  scanf("%d,%d",&c,&e); /*键入多项式的系数和指数项*/
  while(c!=0) /*若c=0, 则代表多项式的输入结束*/
  { s=(Polynode*)malloc(sizeof(Polynode));
    s->coef=c;          s->exp=e;
    rear->next=s;      /*在当前表尾做插入*/
    rear=s;
    scanf("%d,%d",&c,&e);
  }
  rear->next=NULL;      /*表示链表的结束*/
  return (head);
}
    
```

3.一元多项式相加——删除多项式

```
void polydestroy (Polylist head)
{
    Polynode *p;           /*p始终指向第一个结点*/
    p=head->next;
    while(p!=NULL)         /*当p为空，表示链表删除完成*/
    {
        head->next=p->next; /*第二个变第一*/
        free(p);           /*释放第一个结点空间*/
        p=head->next;       /*p始终指向第一个结点*/
    }
}
```

3.一元多项式相加——输出多项式

```
void polyprint (Polylist head)
{
    Polynode *p;           /*临时指针*/
    printf("\n The polynomial is :\n") ;
    p=head->next;
    while(p!=NULL)
    {
        printf("%dx^%d",p->coef,p->exp);
        p=p->next;
        if(p!=NULL)
            printf("+") ;
    }
}
```

```

void polyadd (Polylist ha, Polylist hb)
{  Polynode *p, *q,*tail,*temp;  int sum;
   p=ha->next;    q=hb->next;
   tail=ha;      /* tail指向和多项式的尾结点*/
   while(p!=NULL && q!=NULL)
   {    if (p->exp < q->exp)
        小于情况下处理
        else if (p->exp == q->exp)
            相等情况下处理
        else
            大于情况下处理
   }
   if(p!=NULL)      /*多项式A中还有剩余, 则将剩余的
                     结点加入到和多项式中*/
       tail->next=p;
   else tail->next=q; /*否则, 加入B的结点加入*/
}
    
```

3.一元多项式相加——小于情况下处理

/*如果p指向的多项式项的指数小于q的指数，将p结点加入到和多项式中*/

```
{  
    tail->next=p;  
    tail=p;  
    p=p->next;  
}
```

3.一元多项式相加——相等情况下处理

```

sum=p->coef + q->coef;
/*若系数和非零,则系数和置入结点 p,释放q,并指针后移*/
if (sum != 0)
{
    p->coef=sum;
    tail->next=p;  tail=p;  p=p->next;
    temp=q;  q=q->next;  free(temp);
}
/*若系数和为零,则删除结点p与q,并将指针后移*/
else
{
    temp=p;  p=p->next;  free(temp);
    temp=q;  q=q->next;  free(temp);
}
    
```


3.一元多项式相加——大于情况下处理

/*如果p指向的多项式项的指数大于q的指数，将q结点加入到和多项式中*/

```
{  
    tail->next=q;  
    tail=q;  
    q=q->next;  
}
```

本章小结

1. 顺序表与链表的比较

- 顺序是用**数组**实现的，而链表是用**指针**或“**游标**”来实现的。
- 顺序表的存储空间是**静态分配**的，而算法的执行前必须明确的规定其存储规模；链表的存储空间是**动态分配**的。当线性表的长度变化较大或问题规模难以确定时，采用动态链表较好。

1. 顺序表与链表的比较

- 对于顺序表，可**随机访问**任一个元素，而在单链表中，需要顺着链逐个进行查找，因此单链表适合于在成批地、顺序地处理线性表中的元素时采用。
- 在单链表里进行**插入、删除运算**比在顺序表里容易得多。

2. 本章学习重点

(1) 了解线性表的逻辑结构特性是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是顺序存储结构(线性表)和链式存储结构(链表)。

(2) 熟练掌握这两类存储结构的描述方法，以及线性表的各种基本操作（主要是查找、插入、删除）的实现。

(3) 能够从时间和空间复杂度的角度综合比较线性表两种存储结构的不同特点及其适用场合。

作业

试写一算法，将带头结点的单链表逆置。

