

第2章 线性表

2.1 线性表的定义及其基本操作

2.2 线性表的顺序存储结构

2.3 线性表的链式存储结构

2.4 数组与链表实现方法的比较

2.5 线性表应用举例

2.6 小结



2.1 线性表的定义及其基本操作

1. 什么是线性表（Linear List）？

由若干（**0**个或多个）同类型元素组成的线性序列。

定义：线性表

$$L = (a_0, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$$

其中：**L**为表名， a_i ($0 \leq i \leq n-1$)为数据元素；

n为表长。**n**>0时，L为**非空**表；否则为空表，记为 Φ 。



2.1 线性表的定义及其基本操作

2. 线性表的逻辑结构和特征

形式化描述：线性表

$$L = (D, R)$$

D：数据元素集合，R：D上的关系集合，其中：

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

$$R = \{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D, 0 \leq i \leq n-2 \}$$

关系符 $\langle a_i, a_{i+1} \rangle$ ：有序对，表示任意相邻的两个元素之间的一种先后次序关系

a_i 是 a_{i+1} 的直接前驱， a_{i+1} 是 a_i 的直接后继。

线性表的特征：对非空表， a_0 是表头，无前驱； a_{n-1} 是表尾，无后继；其它的每个元素 a_i 有且仅有一个直接前驱(a_{i-1})和一个直接后继(a_{i+1})。



2.1 线性表的定义及其基本操作

例2.1 线性表例子

$L1=(\text{'A'}, \text{'B'}, \dots, \text{'Z'})$ 元素为字符

$L2=(6, 7, \dots, 105)$ 元素为整数

学生记录表:

	学 号	姓 名	性 别	年 龄	班 级	...
a_0	J06001	丁兰	女	19	计06	...
a_1	J06002	王林	男	20	计06	...
..
..
a_{31}	J06032	马红	女	18	计06	...



2.1 线性表的定义及其基本操作

3. 线性表的抽象数据类型表示

ADT List {

数据元素集: $D = \{a_i | a_i \in \text{datatype}, i = 0, 1, 2, \dots, n-1, n \geq 0\}$

数据关系集: $R = \{ \langle a_i, a_{i+1} \rangle | a_i, a_{i+1} \in D, 0 \leq i \leq n-2 \}$

基本操作集: P

ListInit(&L)

操作结果: 构造一个空的线性表L。

ListDestroy(&L)

初始条件: 线性表L存在。

操作结果: 撤销线性表L。

ListClear(&L)

初始条件: 线性表L存在。

操作结果: 将L置为一张空表。

ListLength(L)

初始条件: 线性表L存在。

操作结果: 返回L中元素个数(即表长n)。



线性表的抽象数据类型表示

ListEmpty(L)

初始条件：线性表L存在。

操作结果：L为空表时返回TRUE，否则返回FALSE。

GetElem(L, i)

初始条件：线性表L存在，且 $0 \leq i \leq n-1$ 。

操作结果：返回L中第i个元素的值(或指针)。

LocateElem(L, e)

初始条件：线性表L存在，且 $e \in \text{datatype}$ 。

操作结果：若e在L中，返回e的序号(或指针)；否则返回e不在表中的信息（实际应用中如-1或NULL）。

PreElem(L, cur)

初始条件：线性表L存在，且 $\text{cur} \in \text{datatype}$ 。

操作结果：若cur在L中且不是表头，返回cur的直接前驱，否则返回NULL。

SuccElem(L, cur)

初始条件：线性表L存在，且 $\text{cur} \in \text{datatype}$ 。

操作结果：若cur在L中且不是表尾元素，返回cur的直接后继的值，否则返回NULL。



线性表的抽象数据类型表示

ListInsert(&L, i, e)

初始条件：线性表L存在，且 $e \in \text{datatype}$ 。

操作结果：若 $0 \leq i \leq n-1$ ，将e插入到第i个元素之前，表长增加1，函数返回TURE；
若 $i=n$ ，将e插入到表尾，表长增加1，函数返回TURE；i为其他值时函数返回FALSE，L无变化。

ListDel(&L, i)

初始条件：线性表L存在。

操作结果：若 $0 \leq i \leq n-1$ ，将第i个元素从表中删除，函数返回TURE，否则函数返回FALSE，L无变化。

ListTraverse(L)

初始条件：线性表L存在。

操作结果：依次对表中的元素利用visit()函数进行访问。

}ADT List;



线性表的抽象数据类型表示

以上给出的是线性表的一些基本操作，常用操作还有：

合并、拆分、复制、排序等。

例2.2 设线性表 $La=(a_0, a_1, \dots, a_{m-1})$, $Lb=(b_0, b_1, \dots, b_{n-1})$, 求 $La \cup Lb \Rightarrow La$ 。

算法思路：依次取 Lb 中的 $b_i(i=0,1,\dots,n-1)$ ，若 b_i 不属于 La ，则将其插入 La 。

算法描述：

```
void Union(List *La, List *Lb)
{
    int i,k ;
    datatype x;
    for (i = 0; i < ListLength(Lb); i++) {
        x = GetElem(Lb , i); k = LocateElem(La, x);
        if ( k == -1 ) ListInsert(La, ListLength(La), x);
    }
}
```




线性表的抽象数据类型表示

例2.3设计清除线性表 $L=(a_0, a_1, \dots, a_i, \dots, a_{n-1})$ 中重复元素的算法。

算法思路：对当前表 L 中的每个 a_i ($0 \leq i \leq n-2$)，依次与 a_j ($i+1 \leq j \leq n-1$) 比较，若与 a_i 相等，则删除之。

算法描述：

```
void Purge(list *L)
```

```
{ int i=0, j; datatype x, y;
```

```
  while(i<ListLength(*L)-1)
```

```
  { x=GetElem(*L,i); j=i+1;
```

```
    while ( j<ListLength(*L))
```

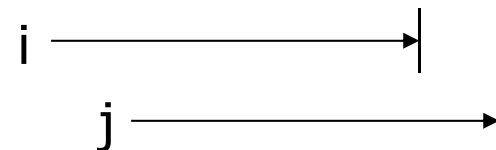
```
      {y=GetElem(*L,j);
```

```
        if (y==x) ListDel(L,j); else j++; }
```

```
    i++;}
```

```
}
```

初始： $L = (1, 3, 1, 5, 3, 5, 7)$



结果： $L = (1, 3, 5, 7)$



线性表的抽象数据类型表示

线性表的实现（存储结构）：

- ✓ 顺序表（数组）
- ✓ 链表

2.2 线性表的顺序存储结构

1. 顺序表

线性表 $L=(a_0, a_1, \dots, a_{n-1})$ 中的各元素依次存储于一片连续的存储空间



2.2 线性表的顺序存储结构

顺序表存储结构的C语言描述:

```
#define MAXSIZE 1024    //线性表的最大长度
typedef struct {         //表的类型
    datatype data[MAXSIZE]; //表的存储空间
    int last;            //当前表尾指针
}sqlist, *sqlink;       //表说明符
```

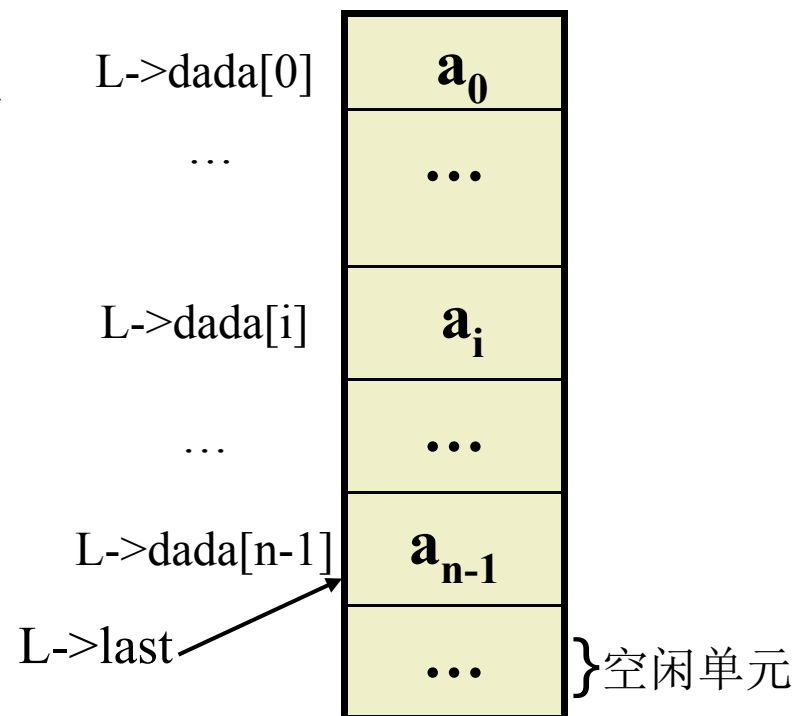
如果说明

```
sqlink L;
```

```
L=(sqlink)malloc(sizeof(sqlist));
```

则指针L指向一个线性表, 如右图所示。

a_i 表示为 $L->data[i]$ ($0 \leq i \leq L->last$)





2.2 线性表的顺序存储结构

顺序存储结构的特点：

- (1) 逻辑上相邻的元素 a_i, a_{i+1} ，其存储位置也是相邻的；
- (2) 对数据元素 a_i 的存取为随机存取或按地址存取。
- (3) 存储密度高。

存储密度 $D = (\text{数据结构中元素所占存储空间}) / (\text{整个数据结构所占空间})$

- (4) 不足：对表的插入和删除等操作的时间复杂度较差。



2.2 线性表的顺序存储结构

2. 顺序表的基本操作

置空表: ListClear (L), 令 $L \rightarrow \text{last} = -1$;

取 a_i : GetElem(L, i), 取 $L \rightarrow \text{data}[i]$ 之值;

求表长: ListLength(L), 取 $L \rightarrow \text{last}$ 之值加1即可。

(1) ListInsert(L, i, e): 将一给定值 e 插在元素 a_i 之前

算法思路: 若表存在空闲空间, 且参数 i 满足: $0 \leq i \leq L \rightarrow \text{last} + 1$, 则可进行正常插入。插入前, 将表中 ($L \rightarrow \text{data}[L \rightarrow \text{last}] \sim L \rightarrow \text{data}[i]$) 部分顺序下移一个数据单位, 然后将 e 插入 $L \rightarrow \text{data}[i]$ 处, $L \rightarrow \text{last}$ 之值加1即可。



顺序表的基本操作

算法描述:

```
int ListInsert(sqlink L, int i, datatype e)
{
    int j;
    if (L->last == MAXSIZE - 1) return -1; //表L溢出
    if (i < 0 || i > L->last + 1) return -2;    //非法参数i
    for (j = L->last; j >= i; j--)
        L->data[j+1] = L->data[j];
    L->data[i] = e;
    L->last++;
    return 0; //成功
}
```

时间复杂度=?



顺序表的基本操作

算法分析:

算法的主要时间耗费在数据元素的移动上, 即算法中的for 语句上,
故以每插入一个元素的平均移动次数刻画算法的时间复杂度 **$T(n)$**
(n 为表长)。

设元素 e 插入 a_i ($0 \leq i \leq n$)处的概率 p_i 均等, 即 $p_i = 1/(n+1)$,
插入 a_i 时的元素移动次数 $c_i = n - i$, 则平均移动次数为:

$$T(n) = \sum p_i c_i = n/2 = O(n)。$$



顺序表的基本操作

(2) ListDel(L, i): 将表中第 i 个元素 a_i 从表中删除

算法思路: 若参数 i 满足: $0 \leq i \leq L \rightarrow \text{last}$, 将表中

$L \rightarrow \text{data}[i+1] \sim L \rightarrow \text{data}[L \rightarrow \text{last}]$ 顺序向上移动一个元素位置, 挤掉
 $L \rightarrow \text{data}[i]$, 修改表长。



顺序表的基本操作

算法描述:

```
int ListDel(sqlink L, int i)
{
    int j;
    if (i < 0 || i > L->last) return -1;
    for (j = i+1; j <= L->last; j++)
        L->data[j-1] = L->data[j];
    L->last--;
    return 0;
}
```

算法分析:

设删除一元素 a_i ($0 \leq i \leq n-1$)的概率 p_i 均等, 即 $p_i = 1/n$, 删除 a_i 的元素移动次数 $c_i = n - (i+1)$, 则平均移动次数为:

$$T(n) = \sum p_i c_i = (n-1)/2 = O(n)$$



顺序表的基本操作

(3) LocateElem(L, x): 确定给定元素 x 在表 L 中第一次出现的位置（或序号）

算法思路：设一扫描变量 i （初值=0），判断当前表中元素 a_i 是否等于 x ，若相等，则返回当前 i 值（表明 x 落在表的第 i 位置）；否则 i 加1，继续往下比较。若表中无一个元素与 x 相等，则返回-1。



顺序表的基本操作

算法描述:

```
int LocateElem(sqlink L,datatype x)
{ int i=0;
  while (i<=L->last && L->data[i]!=x)
    i++;
  if (i<=L->last) return(i);
  else return(-1);
}
```

算法分析:

设元素 a_i ($0 \leq i \leq n-1$)与 x 相等的概率 p_i 均等, 即 $p_i = 1/n$, 查找 a_i 与 x 相等的比较次数 $C_i = i+1$, 则平均移动次数为:

$$T(n) = \sum p_i c_i = (n+1)/2 = O(n)$$



2.3 线性表的链式存储结构

1. 指针变量的物理意义

- ✓ 计算机的内存空间是由连续编号的内存单元构成，每个内存单元具有唯一的地址（编号）。在以字节编址的计算机系统中，每个内存单元存放1个字节。
- ✓ 变量的2个要素：地址，值
地址由系统自动分配，所占空间的大小由变量类型决定
- ✓ 指针变量与其它变量一样，在定义之后，系统自动分配相应的内存空间以存放其值。
- ✓ 指针变量的值是指针，即地址。



指针变量的物理意义

看下列C语言代码是否正确？

(1) `int *p;`

`*p = 1;`

✗

`p = (int*)malloc(sizeof(int));`

申请一块内存空间，以存放*p的值。

(2) `int *p, i;`

`p = &i;`

`*p = 1;`

✓

(3) `int *p;`

`p = 1;`

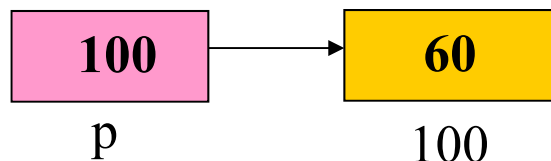
✓

指针变量的物理意义

变量p定义如下：

`int *p; // 表示p是一个指向int型数据的指针变量。`

假设某一时刻p的值是100，地址为100的内存单元存放着整数60



一般记作： p \longrightarrow 60

p是指针变量。

p的值是100（在这种情况下）。

值100被认为是地址。

地址100处保存着整数值60。

p所指对象的值为60。

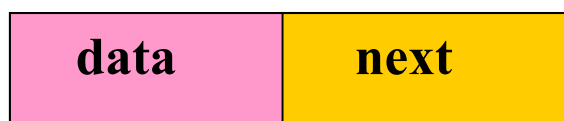
p的地址是什么？我们不知道，一般也不需要知道。（&p）

p的值一般也不关心。

2.3 线性表的链式存储结构

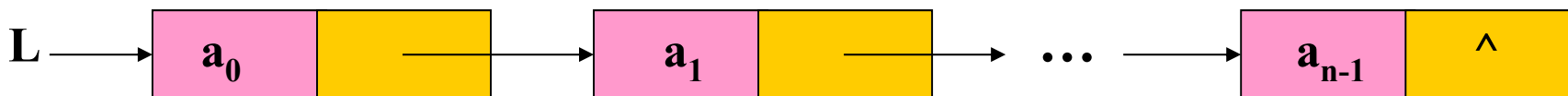
2. 单链表

节点：线性表中每个元素所占的存储块（带指针域）。形式：



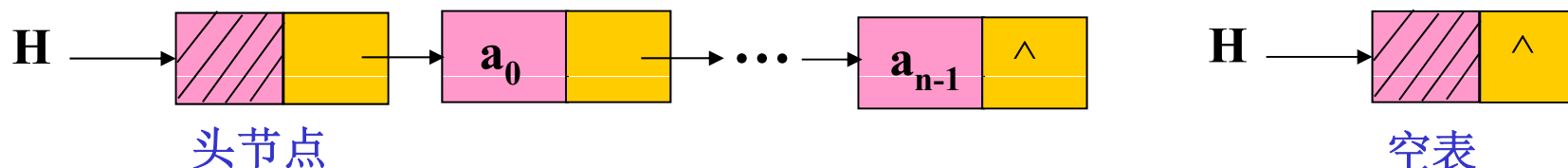
data域存放数据元素 a_i ，而**next**域是一个指针，指向 a_i 的直接后继 a_{i+1} 所在的节点。

线性表 $L=(a_0, a_1, \dots, a_{n-1})$ 的链式存储结构：



带头节点的单链表：在链表的节点 a_0 前加一个额外的头节点，方便操作。

除非特别指出，下面所讲的单链表都是带头节点的单链表



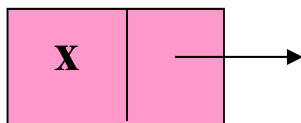
单链表

节点类型描述：

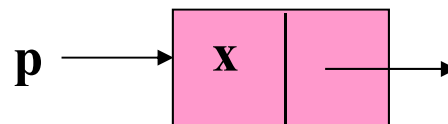
```
typedef int datatype;    //设数据元素为整型
typedef struct node {    //节点类型
    datatype data;       //节点的数据域
    struct node *next;   //节点的后继指针域
}linknode, *link;
```

约定两种提法：

节点x：数据元素为x的节点。



p节点：指针p所指向的节点。



单链表

可调用C语言中`malloc()`函数向系统申请节点的存储空间，若说明：

```
link p;
```

```
p = (link)malloc(sizeof(linknode));
```

则获得了一个类型为`linknode`的节点，且该节点的地址已存入指针变量`p`中。

```
free(p);
```

则释放`p`所指节点所占用的存储空间（由系统回收）。



`ai: p->data`

`ai+1: p->next->data`



2.3 线性表的链式存储结构

3. 单链表的基本操作：（1）创建

算法思路：依次读入表 $L=(a_0, \dots, a_{n-1})$ 中每一元素 a_i (设为整型)，若 $a_i \neq$ 结束符 (-1) ，则将 a_i 形成一新节点，链入表尾，最后返回链表的头指针。

算法描述：

```
link CreateList()
```

```
{
```

```
    int a; link H, p, r;
```

```
    H = (link)malloc(sizeof(linknode));
```

// 建立头节点

```
    r = H; scanf("%d", &a);
```

//输入数据元素

```
    while (a != -1) {
```

```
        p = (link)malloc(sizeof(linknode));
```

//申请新节点

```
        p->data = a; r->next = p; r = p;
```

//存入数据，将新节点链入表尾

```
        scanf("%d", &a);
```

```
    }
```

```
    r->next = NULL;
```

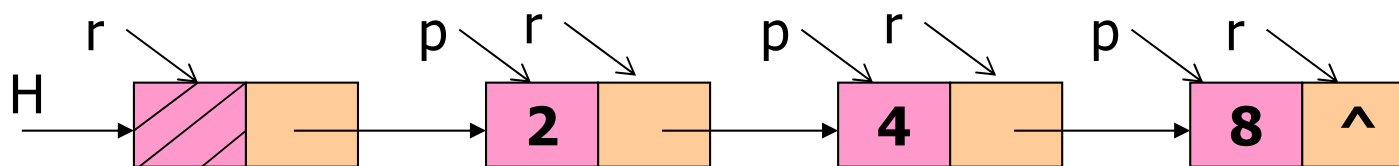
//表尾的后继置空

```
    return H;
```

```
}
```

单链表的基本操作

设 $L=(2, 4, 8, -1)$ ，则建表过程如下：



设表长为 n ，显然此算法的时间复杂度为 $T(n)=O(n)$ 。

从此算法可以看到，**链表的结构是动态形成的**，即算法运行前，表结构是不存在的，而通过算法的运行，得到一个如上图所示的单链表。



单链表的基本操作

(2) 查找

1) GetElem(L, i): 按序号查找

算法思路：从链表的 a_0 起，判断是否为第 i 个节点，若是则返回该节点的指针，否则查找下一节点，依此类推。

算法描述：

```
link GetElem(link H, int i)
{
    int j = -1; link p = H;
    if (i < 0) return NULL;
    while (p->next && j < i) {
        p = p->next; j++;
    }
    if (i == j) return p;
    else return NULL;           //查找失败，即i>表长
}
```



单链表的基本操作

2) LocateElem(L, e): 按值查找

算法思路：从节点 a_0 起，依次判断某节点是否等于 e 。若是，则返回该节点的地址，若不是，则查找下一节点 a_1 ，依此类推。若表中不存在 e ，则返回NULL。

算法描述：

```
link LocateElem(link H, datatype e)
{
    link p = H->next;
    while ( p && p->data != e)
        p = p->next;
    return p; //若p->data==e则返回指针p;否则p必为空，返回NULL
}
```

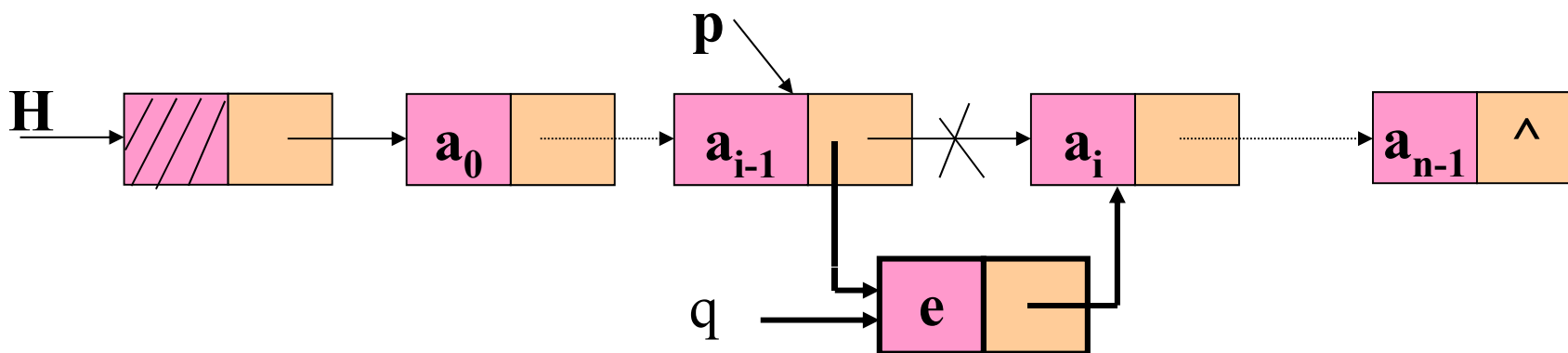
以上两个查找算法的时间复杂度都是 $O(n)$

单链表的基本操作

(3) **ListInsert(L, i, e):前插** (将一给定值 e 插在元素 a_i 之前)

算法思路：调用算法GetElem(H, i-1)，获取节点 a_{i-1} 的指针 $p(a_i$ 之前驱)，
然后申请一个 q 节点，存入 e ，并将其插入 p 节点之后。

插入时的指针变化如下：





单链表的基本操作

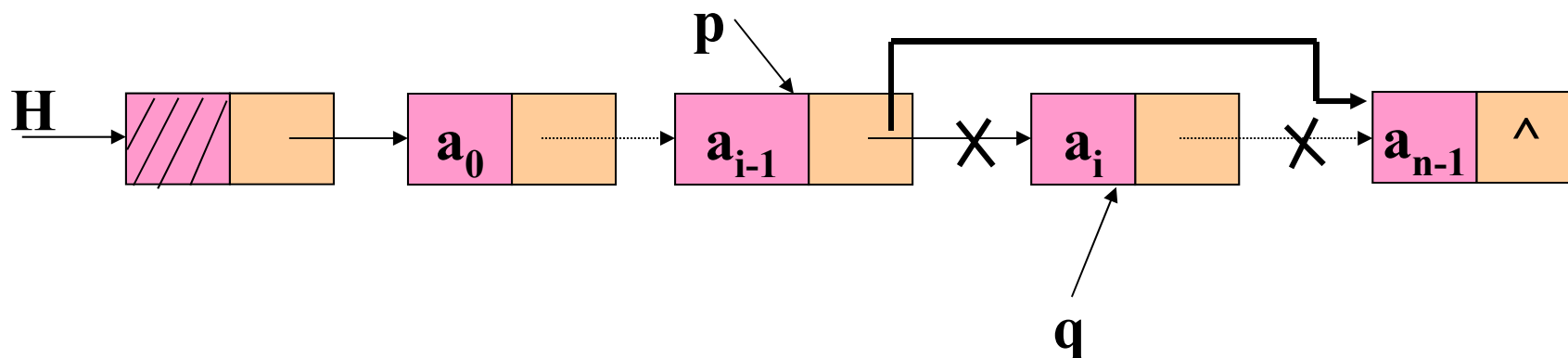
算法描述:

```
int ListInsert(link H, int i, datatype e)
{
    link p, q;
    p = GetElem(H, i-1); //取节点 $a_{i-1}$ 的指针
    if (p == NULL) return -1; //参数i出错
    q = (link)malloc(sizeof(linknode)); //申请插入节点
    q->data = e; //存入数据
    q->next = p->next; //插入新节点
    p->next = q;
    return 0;
}
```

此算法的时间主要花费在函数`GetElem(H,i-1)`上, 故 $T(n)=O(n)$, 但插入时未引起元素的移动, 这一点优于顺序结构的插入。

单链表的基本操作

(4) **ListDel(L, i):删除** (将表中第 i 个元素 a_i 从表中删除)



算法思路：同插入法，先调用函数**GetElem(H, i-1)**，找到节点 a_i 的前驱，然后将节点 a_i 删除之。



单链表的基本操作

算法描述:

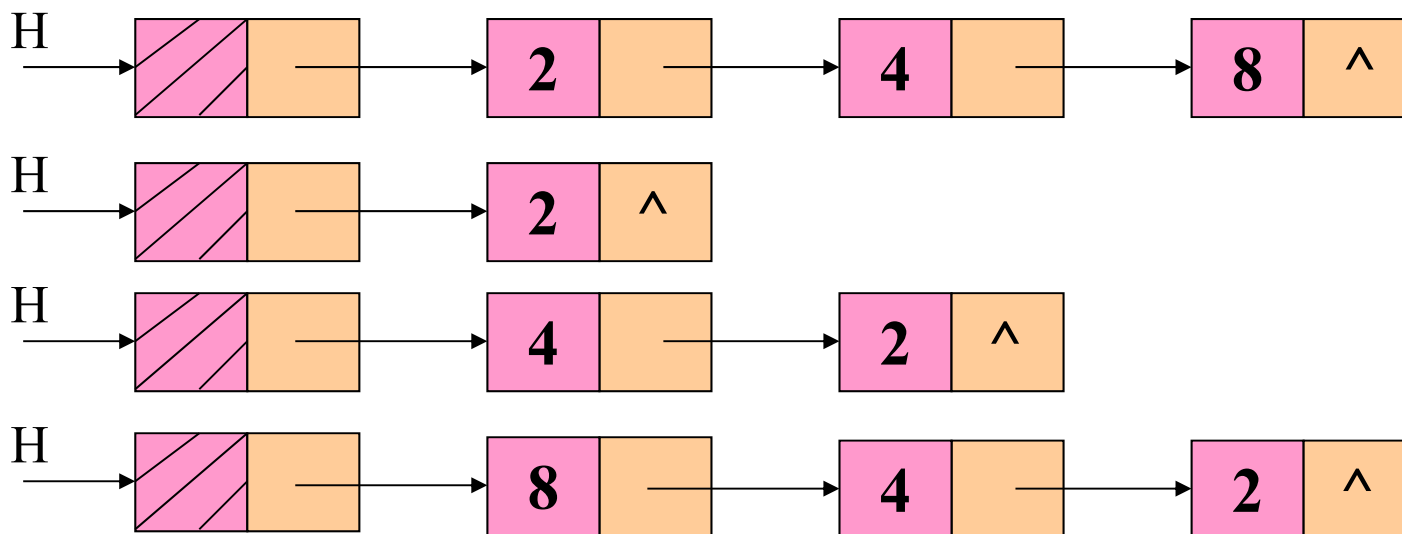
```
int ListDel (link H, int i)
{
    link p, q;
    if (i == 0) p = H;
    else p = GetElem(H, i-1); //取节点 $a_{i-1}$ 的地址
    if (p && p->next) {      //若p及p->next所在的节点存在
        q = p->next ; p->next = q->next; //删除
        free(q); return 0;
    }
    else return -1;          //参数i出错
}
```

同插入法，此算法的 $T(n)=O(n)$ 。

单链表的基本操作

例 2.5 设计算法，将单链表H倒置。

算法思路：依次取原链表中节点，将其作为新链表首节点插入H节点之后。





单链表的基本操作

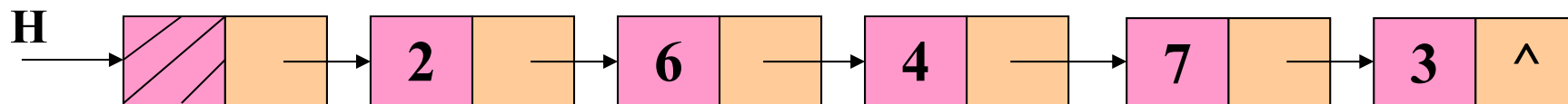
算法描述:

```
void L1nToLn1(link H)
{
    link p, q; p = H->next; //令指针p指向节点a0
    H->next = NULL; //将原链表置空
    while (p) {
        q = p; p = p->next; q->next = H->next; //将节点ai插入到头节点之后
        H->next = q;
    }
}
```

时间复杂度 $T(n)=O(n)$ 。

单链表的基本操作

例 2.6 设节点data域为整型，求链表中相邻两节点data值之和为最大的第一节点的指针。如下图所示的链表，它应返回值为4的节点所在的指针。



算法思路：设 p, q 分别为链表中相邻两节点指针，求 $p \rightarrow data + q \rightarrow data$ 为最大的那一组值，返回其相应的指针 p 即可。



单链表的基本操作

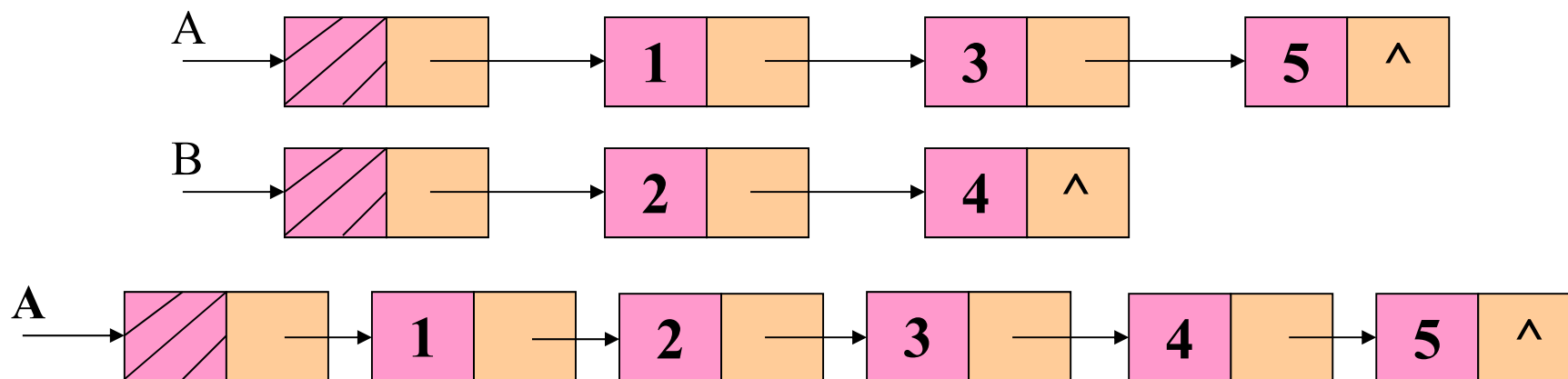
算法描述:

link Adjmax(link H)

```
{
    link p, p1, q; int m0, m1; p = H->next ;
    p1 = p; if (p1 == NULL) return NULL ;    //表空返回
    q = p->next; if (q == NULL) return p1;    //表长=1时返回
    m0 = p->data + q->data;    //相邻两节点data值之和
    while (q->next) {
        p = q; q = q->next;    //取下一对相邻节点的指针
        m1 = p->data + q->data;
        if (m1 > m0) { p1 = p; m0 = m1;} //取和为最大的第一节点指针
    }
    return p1;
}
```

单链表的基本操作

例 2.7 设两单链表A、B按data值（设为整型）递增有序，设计算法，将表A和B合并成一表A，且表A也按data值递增有序。如下图。



算法思路：设指针p、q分别指向表A和B中的节点，若 $p \rightarrow data \leq q \rightarrow data$ 则p节点进入结果表，否则q节点进入结果表。



单链表的基本操作

算法描述:

```
void Merge(link A, link B)
{
    link r, p, q;
    p = A->next; q = B->next; free(B); r = A;
    while (p && q) {
        if (p->data <= q->data) {
            r->next = p; r = p; p = p->next;
        } else {
            r->next = q; r = q; q = q->next;
        }
    }
    if (p == NULL) p = q;
    r->next = p; //收尾处理
}
```

时间复杂度=?

设原表A长度=m, 表B长度=n,
因算法中循环语句最多执行
m+n次, 故该算法的时间复
杂度为 $T(m, n) = O(m+n)$ 。



单链表的基本操作

说明：

对单链表进行操作时，只能插入到当前节点之后，只能删除当前节点的后继。

查找后继节点容易，时间复杂度是 $O(1)$ 。

若查找前驱节点，必须从表头开始，时间复杂度是 $O(n)$ 。

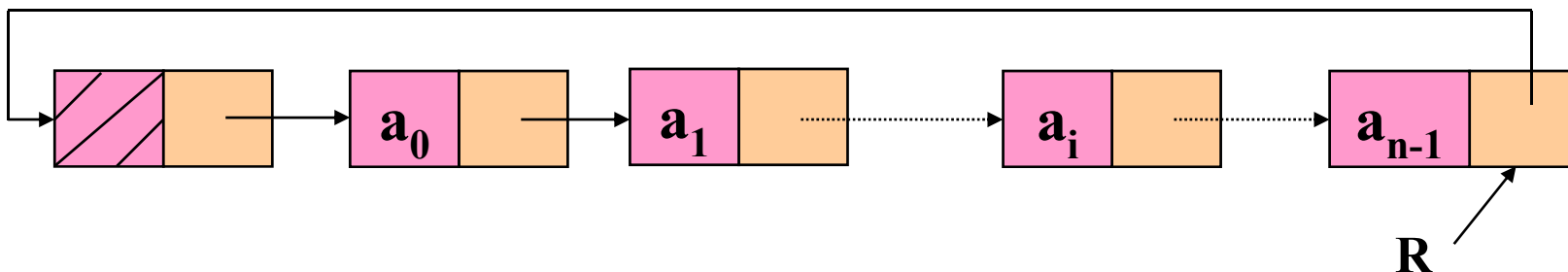
前面讲的单链表操作是带头节点的，带头节点有何好处？

如果不带头节点呢？

2.3 线性表的链式存储结构

4. 单向循环链表

单链表的首尾节点相连，如下图。



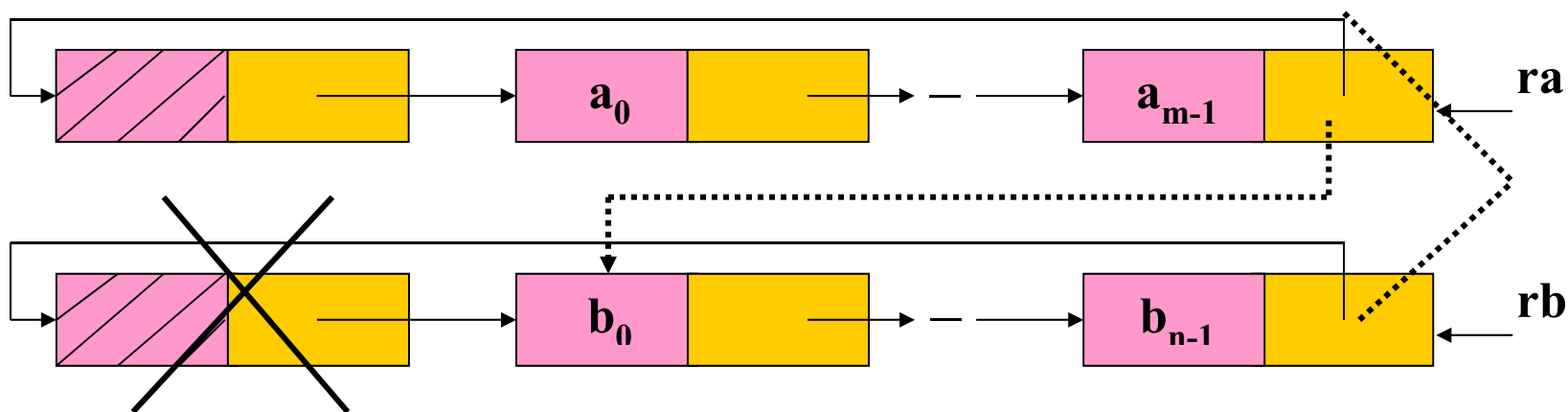
若运算频繁在尾部进行，有时可设一表尾指针 R （头指针 H 可省去）。这样，为获得表尾 a_{n-1} ，取 $R \rightarrow \text{data}$ 即可，不必遍历到表尾，而取 a_0 的操作为： $(R \rightarrow \text{next} \rightarrow \text{next}) \rightarrow \text{data}$

注意：

循环链表无明显的尾端，处理时注意不要进入死循环。

单向循环链表

例 2.8 设 ra 和 rb 分别为两循环链表的尾指针，设计算法，实现表 ra 和 rb 的简单连接。



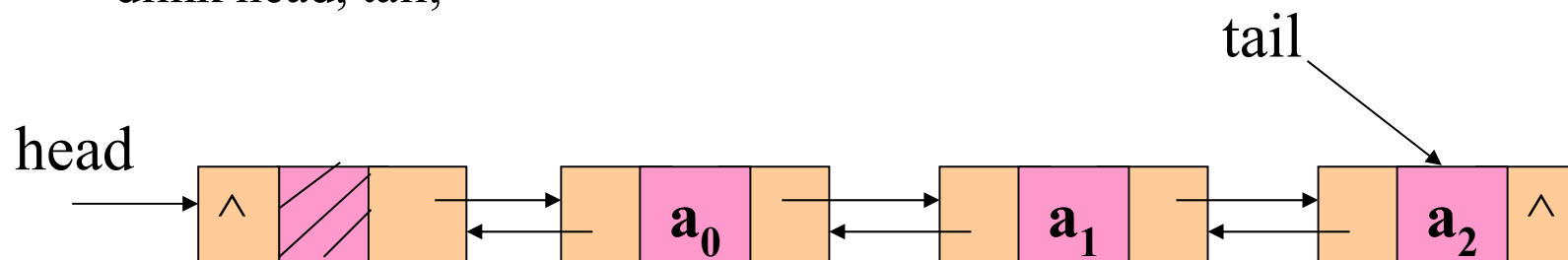
```
p = rb->next;  
rb->next = ra->next;  
ra->next = p->next;  
free(p);
```

2.3 线性表的链式存储结构

5. 双向链表

节点类型描述:

```
typedef struct dnode { //节点类型
    datatype data;      //节点的数据域
    struct dnode *prior, *next; //节点的前驱和后继指针域
} dlinknode, *dlink;
dlink head, tail;
```

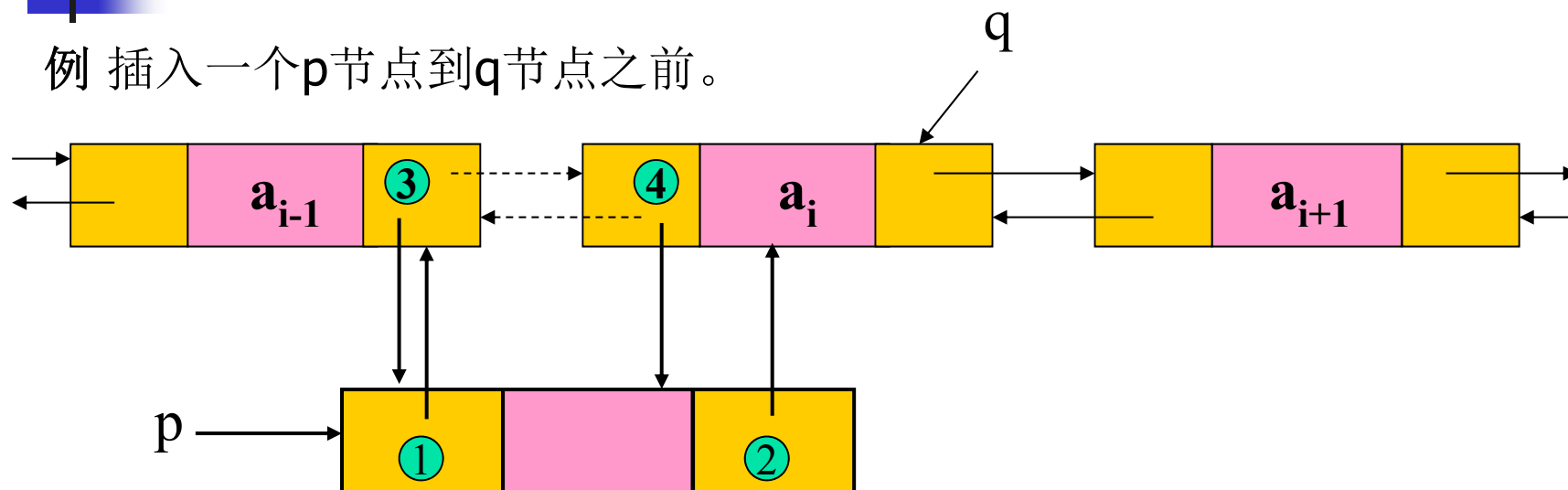


设p为链表中某节点的指针，且p有前驱和后继节点，则有对称性：

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

双向链表

例 插入一个p节点到q节点之前。



① $p \rightarrow \text{prior} = q \rightarrow \text{prior};$

② $p \rightarrow \text{next} = q;$

③ if ($q \rightarrow \text{prior} \neq \text{NULL}$) // 带头节点时，此条件一定满足

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

④ $q \rightarrow \text{prior} = p;$

注意次序：先链接新指针，再断开老指针。



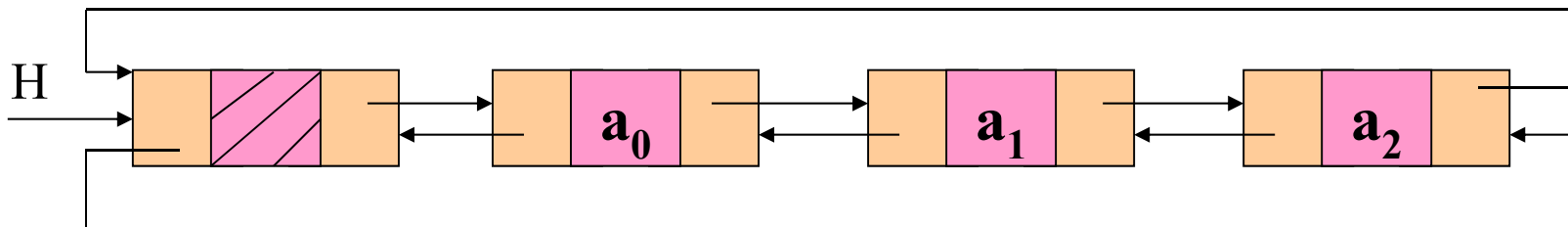
双向链表

插入一个p节点到q节点之后呢？

- ① $p \rightarrow \text{prior} = q;$
- ② $p \rightarrow \text{next} = q \rightarrow \text{next};$
- ③ $\text{if } (q \rightarrow \text{next} \neq \text{NULL}) \quad q \rightarrow \text{next} \rightarrow \text{prior} = p;$
- ④ $q \rightarrow \text{next} = p;$

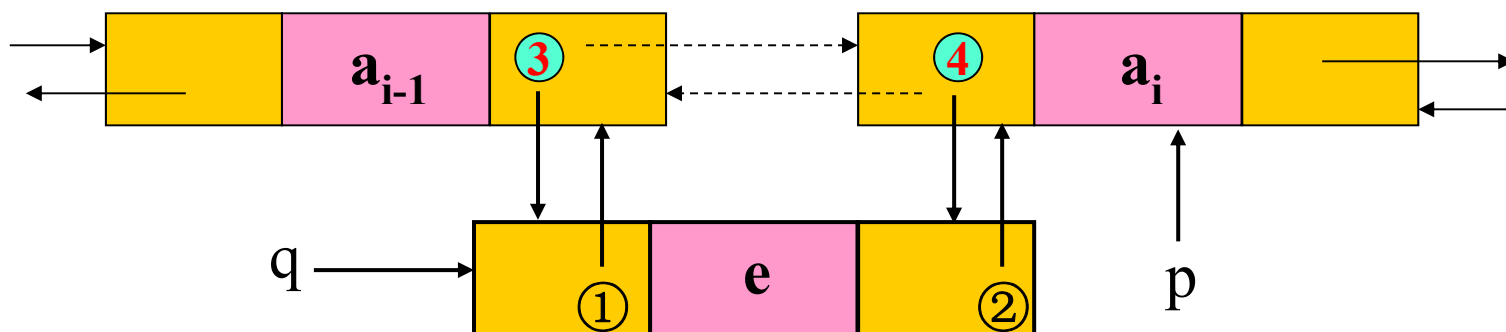
2.3 线性表的链式存储结构

6. 双向循环链表



双向循环链表

(1) 插入：在表L的节点 a_i 前插入一节点 e

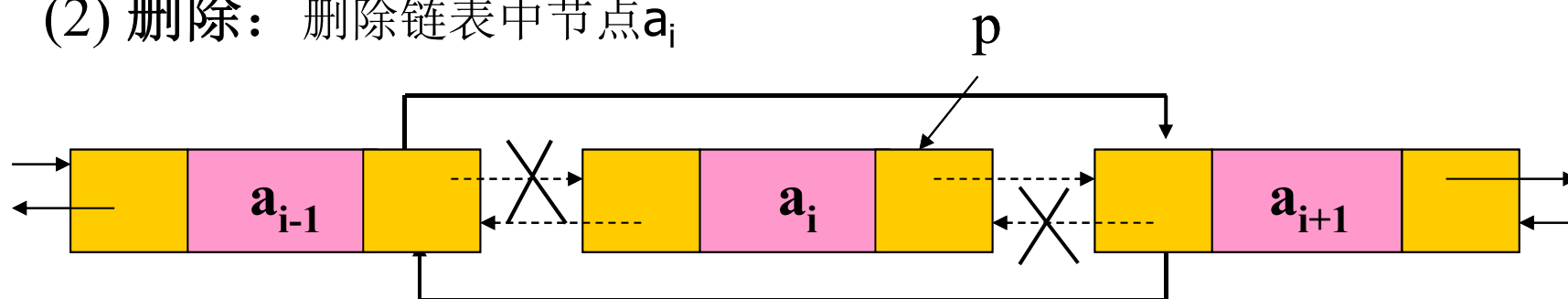


算法思路：首先获取节点 a_i 的指针 p 。若 p 存在，则申请一 q 节点，存入元素 e 。然后将 q 节点插入 p 节点之前。

```
if (p != NULL) {  
    q = (dlink)malloc(sizeof(dlinknode)); q->data = e;  
    q->prior = p->prior;  
    q->next = p;  
    p->prior->next = q;  
    p->prior = q;  
}
```


双向循环链表

(2) 删除：删除链表中节点 a_i



算法思路：首先调用查找算法获取节点 a_i 的指针 p 。若 p 存在，则删除之。

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

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

$\text{free}(p);$



2.3 线性表的链式存储结构

7. 静态链表

用数组模拟链表，以数组下标模拟指针

```
typedef struct {    //节点类型
    datatype data; //数据元素
    int next;
} SNODE;
SNODE space[MAXSIZE]; //静态链表存储空间
```

说明：可以用1个数组实现多个表。

静态链表

设有表

$L = (a, b, c)$

$M = (d, e)$

备用单元表available

设MAXSIZE = 10

插入：新单元来自available

删除：插入available以备后用

space	
0	////
1	d 7
2	4
3	c 0
4	6
5	a 8
6	0
7	e 0
8	b 3
9	2
	data next

Mhead=1 →

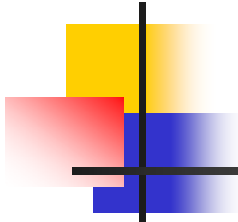
Lhead=5 →

avhead=9 →

不用，浪费掉。
或作为头节点。

0作为空指针。

当然用-1也可以，
但数组中next不能
达到等于-1的无符
号数。



2.4 数组与链表实现方法的比较

- ✓ 若用数组实现线性表，则必须在编译前知道表的最大长度。
对于数组，随机访问效率高。
数组的存储密度高。
- ✓ 链表需在每个节点附加指针，即存在结构性开销
- ✓ 但元素较少时，数组填不满，空间冗余。

一般来说，

当线性表元素个数未知或变化较大时，最好用链表；

若事先知道最大长度，可用数组；

若频繁插入/删除用链表较好。

2.5 线性表应用举例

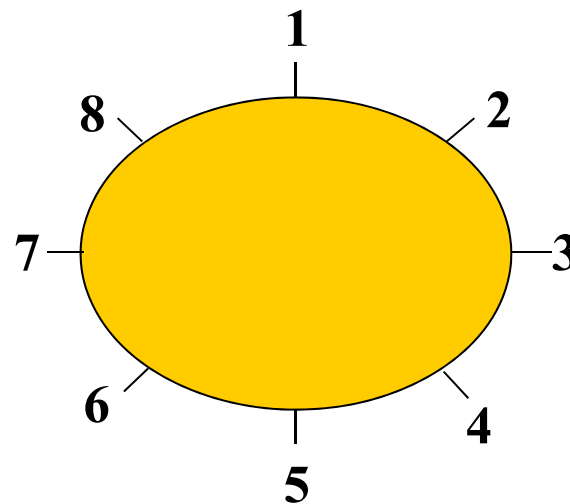
1. Josephu (约瑟夫) 问题

设编号分别为1, 2, ..., n的n个人围坐一圈。约定初始序号为k ($1 \leq k \leq n$) 的人从1开始计数, 数到m的那个人出列, 他的下一位又从1开始计数, 数到m的那个人又出列, 依此类推, 直到所有人出列为止。

设 $n=8$, $k=3$, $m=4$, 如右图。出列顺序为:
(6, 2, 7, 4, 3, 5, 1, 8)

采用何种数据结构来表示?

不带头节点的单向循环链表。





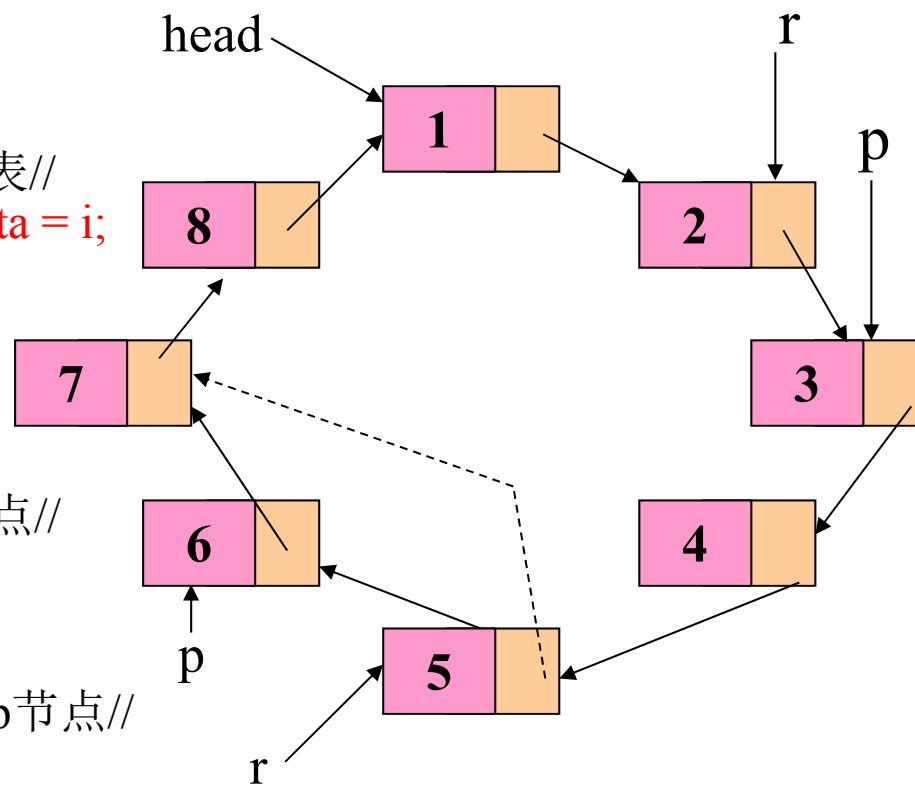
Josephu (约瑟夫) 问题

算法思路:

- 1) 根据 n 建立有 n 个节点的不带头节点的单向循环链表;
- 2) 从第 k 节点起从1计数, 计到 m 时, 对应节点从链表中删除: 注意: 最好有1个指针指向出列节点的前驱;
- 3) 从被删节点的下一个节点起又从1开始计数....., 直到所有节点都出列时算法结束。

Josephu (约瑟夫) 问题 --简化的算法描述

```
void Josephu(link head, int n, int k, int m)
{
    int i; link p, r; head = NULL; //置空表//
    for (i = 1; i <= n; i++) { //建立循环链表//
        p = (link)malloc(sizeof(linknode)); p->data = i;
        if (head == NULL) head = p;
        else r->next = p;
        r = p; }
    p->next = head; p = head; //环起来
    for (i = 1; i <= k-1; i++)
        r = p; p = p->next; } //找到第k个节点//
    while (p->next != p) //节点数>1时//
    { for (i = 1; i < m; i++)
        { r = p; p = p->next; } //报数//
        r->next = p->next; //删除当前出列p节点//
        printf ("%d", p->data); //输出序号//
        free(p); p = r->next; //取下一报数的起点指针//
    }
    printf ("%d\n", p->data); //输出最后一个节点的序号//
}
```



时间复杂度为: $T(n,m) = O(n) + O(k) + O(n*m) = O(n*m)$



2.5 线性表应用举例

2. 一元多项式的表示与相加

一元 n 次多项式: $p_n(x) = p_0 + p_1x^1 + \dots + p_ix^i + \dots + p_nx^n$

采用什么样的线性表来表示?

✓ $P(p_0, p_1, \dots, p_n)$

✓ $P((P_1, e1), (P_2, e2), \dots, (P_m, em))$

对应于 $P_n(x) = p_1x^{e1} + p_2x^{e2} + \dots + P_mx^{em}$

如果 $P_n(x)$ 中有许多系数为0的项, 如:

$$P_{2000}(x) = 1 + 3x^{1000} + 2x^{2000}$$

对应的线性表就是

哪一种表示好?

(1) $P(1, 0, \dots, 0, 3, \dots, 2)$

(2) $P((1, 0), (3, 1000), (2, 2000))$



一元多项式的表示与相加

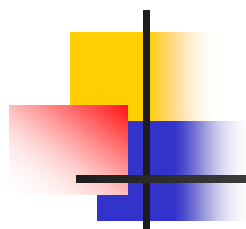
用单链表实现两个多项式相加：

节点类型描述：

```
typedef struct node {  
    float coef;  
    int exp;  
    struct node *next;  
}linknode,*link;
```



(系数域) (指数域) (下一项指针)



一元多项式的表示与相加

设两多项式A, B分别为:

$A_{16}(X)=5 + 2X + 8X^8 + 3X^{16}$, 对应的线性表:

$A((5, 0), (2, 1), (8, 8), (3, 16))$

$B_8(X)=6X + 23X^6 - 8X^8$, 对应的线性表:

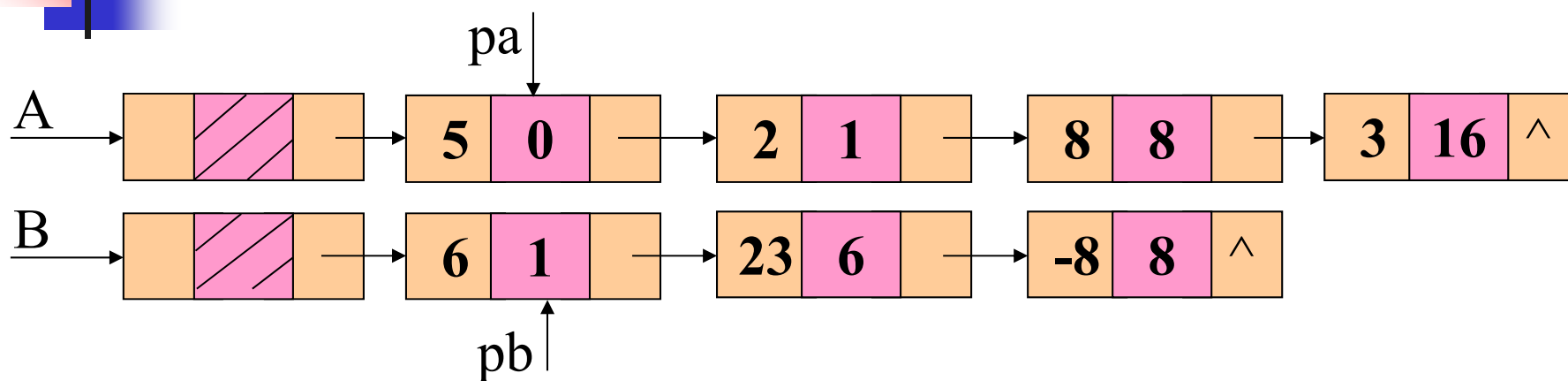
$B((6, 1), (23, 6), (-8, 8))$

A+B 的结果多项式C为:

$C_{16}(X)= 5 + 8X + 23X^6 + 3X^{16}$, 对应的线性表:

$C((5, 0), (8, 1), (23, 6), (3, 16))$

一元多项式的表示与相加



算法思路：设指针 pa , pb 分别指向两链表中的某节点（初始指向第一节点）：

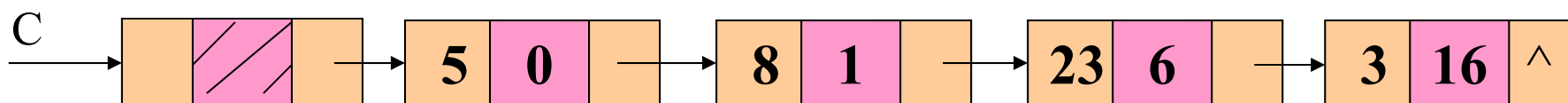
若 $pa \rightarrow \text{exp} < pb \rightarrow \text{exp}$ ，则 pa 节点应为和的一项；

若 $pa \rightarrow \text{exp} > pb \rightarrow \text{exp}$ ，则 pb 节点应为和的一项；

若 $pa \rightarrow \text{exp} = pb \rightarrow \text{exp}$ ，则两节点对应系数相加： $\text{sum} = pa \rightarrow \text{exp} + pb \rightarrow \text{exp}$

若 $\text{sum} \neq 0$ ，相加结果应为和的一项。

上图中两多项式的链表相加的结果如下图：



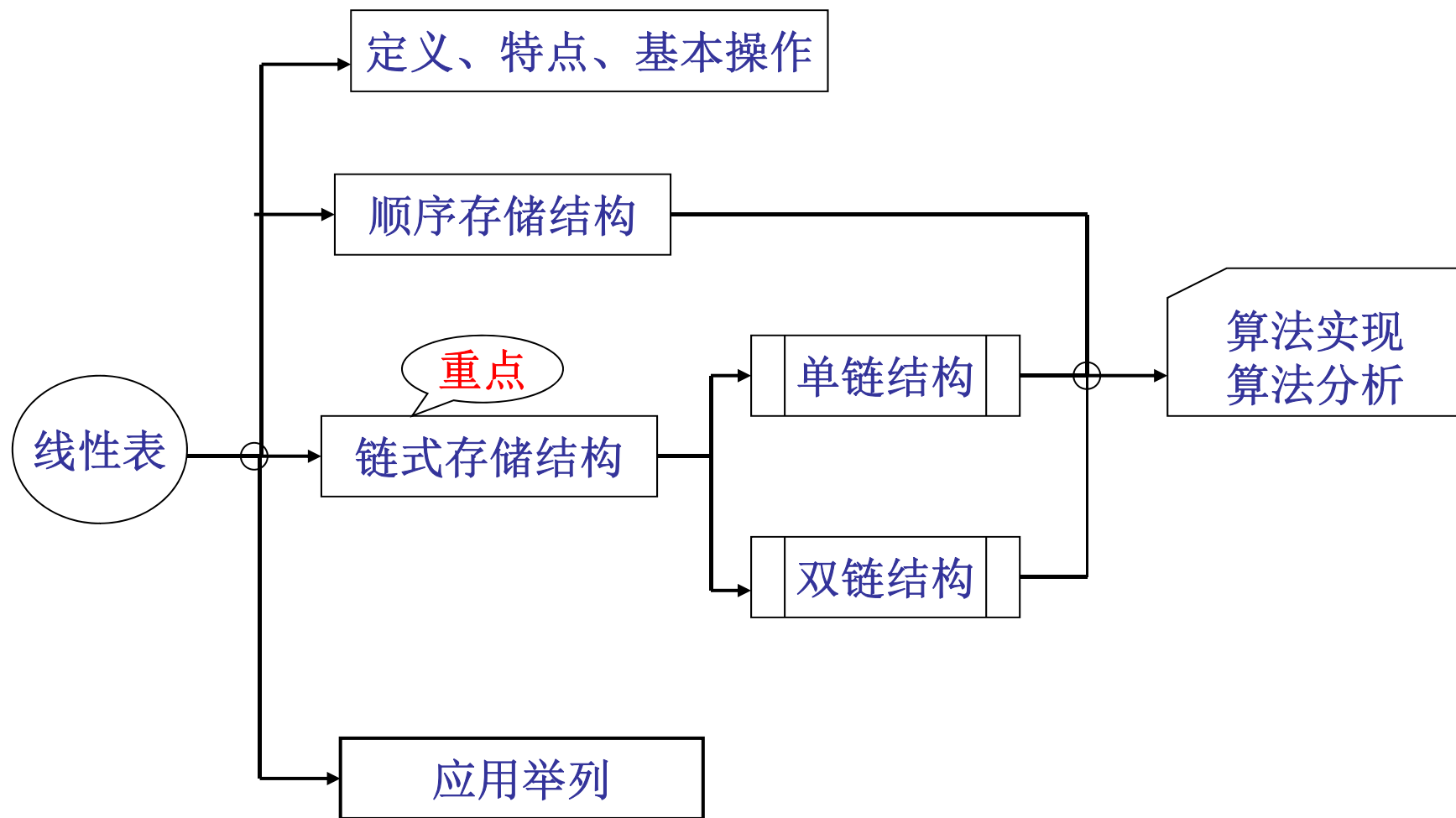


一元多项式的表示与相加

```
void Addpoly(link pa,link pb)
{ link pc,pre,u ; float sum;
  pc=pa; pre=pa; pa=pa->next ; u=pb; pb=pb->next; free(u);
  while (pa&&pb)
  { if (pa->exp< pb->exp) { pre=pa; pa=pa->next ;} // pa为和的一项
    else if (pa->exp>pb->exp) // pb为和的一项，插入到pre和pa之间
      {u=pb->next ; pb->next=pa; pre->next=pb; pre=pb;pb= u;}
    else { sum=pa->coef+pb->coef ;      //指数相同，系数相加
          if (sum!=0.0) { pa->coef=sum; pre= pa;} //修改指数
          else {pre->next=pa->next; free(pa);} //删除pa节点
          pa=pre->next;
          u=pb;                                //删除对应的pb节点
          pb=pb->next;
          free(u);}
    }
  if (pb) pre->next=pb; } //将pb的剩余项链入结果表
```

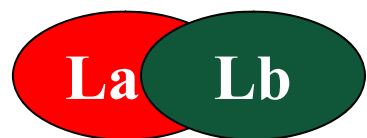
设两链表的表长分别为m和n，则此算法的时间复杂度为 $T(m,n)=O(m+n)$ 。

2.6 小结

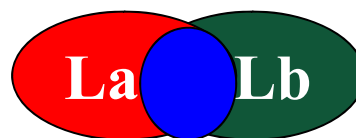


第2章 作业

1. 设线性表 $La=(a_0, a_1, \dots, a_{m-1})$, $Lb=(b_0, b_1, \dots, b_{n-1})$, 利用线性表基本操作, 写出求 $La - Lb \Rightarrow La$ 、 $La \cap Lb \Rightarrow Lc$ 操作的算法。



红: $La - Lb$



蓝: $La \cap Lb$

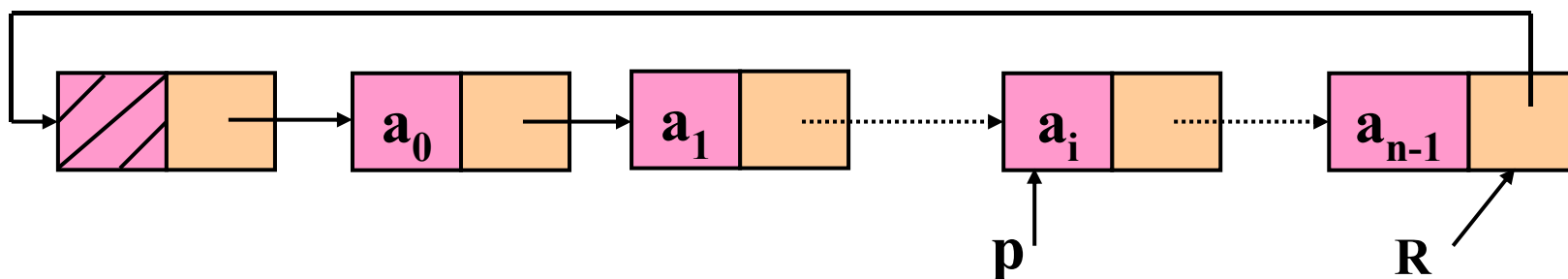
2. 设学生记录表S:
(按学号Sno有序)

Sno	Name[20]	sex	Class[20]
0001	丁一	男	计02
0002	王二	女	计02
0003	张三	男	计02
.....
0032	李四	男	计02

- (1)设计表S 的顺序存储结构;
- (2)写出将一学生记录x 插入到表中正确位置的算法:insert-s(S, x);
- (3)写出从表中删除Sno=y 的记录: delete-s(S, y)。

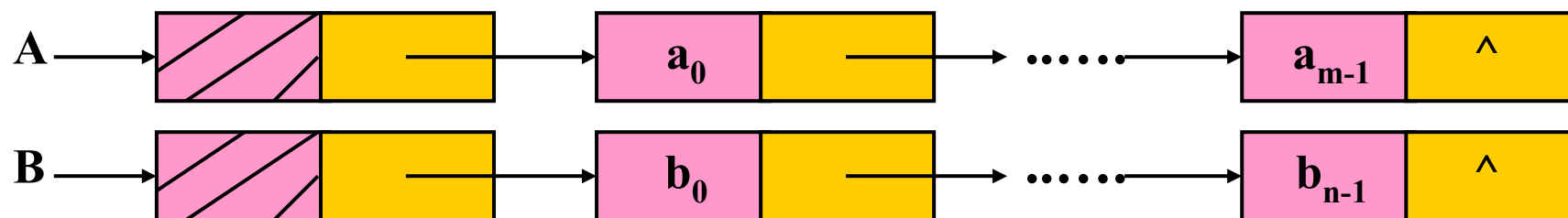
第2章 作业

3. 设循环链表:



试写出从表R中某 p 节点开始, 查找 $\text{data}=d$ 的节点指针的算法: $\text{search}(R,p,d)$.
(算法前应包括对节点的说明)

4. 设链表A、B 如下:



写出判断A 表和B 表是否相等的算法: $\text{equal}(A,B)$.

(两表相等的充分必要条件: 表长相等, 且两表中元素也对应相等。)