

第二章 线性表

线性结构的特点： $K_1 K_2 K_3 \dots K_n$

在数据元素的非空有限集中，

- (1) 存在唯一的一个被称为“**第一个**”的数据元素；
- (2) 存在唯一的一个被称为“**最后一个**”的数据元素；
- (3) 除第一个之外，集合中的每个数据元素均只有一个“**直接前驱**”；
- (4) 除最后一个之外，集合中的每个数据元素均只有一个“**直接后继**”；

常用的线性结构：**线性表、栈、队列、串等**

本章主要内容:

- 线性表的概念（逻辑结构）
- 线性表的顺序存储结构（顺序表）
- 线性表的链接存储结构（链表）
 - 单链表
 - 静态链表
 - 循环链表
 - 双向链表
- 线性表的应用（**Josephus**问题）
- 矩阵
- 广义表与动态存储管理

2.1 线性表的概念

- **线性表**：简称为表，是零个或多个元素的有穷序列，通常可以表示成 k_0, k_1, \dots, k_{n-1} ($n \geq 1$)。
- **表长**：线性表中所含元素的个数 n 。
- **空表**：长度为零的线性表($n=0$)。
- **表目**：线性表中的元素（可包含多个数据项，**记录**）。

线性表的表示： $A=(k_0, k_1, \dots, k_{n-1})$

k_0 是第一个元素， k_{n-1} 是最后一个元素

k_i 是 k_{i+1} 的前驱， k_{i+1} 是 k_i 的后继

k_0 无前驱， k_{n-1} 无后继

线性表的基本运算(逻辑上的定义):

- 创建空线性表;
- 插入一个元素;
- 删除某个元素;
- 查找某个特定元素;
- 查找某个元素的后继元素;
- 查找某个元素的前驱元素;
- 判别一个线性表是否为空表。

线性表特点: 关系简单、操作灵活, 其长度可以增长、缩短
存储结构: 顺序、链接存储

2.2 线性表的顺序表示和实现

定义：将线性表中的元素一个接一个地存储在一片地址连续的存储单元中。

逻辑关系表达(存储)：以元素在计算机内存中的“**物理位置相邻**”来表示线性表中数据元素之间的逻辑关系，如下所示：

$$\text{Locate}(k_{i+1}) = \text{Locate}(k_i) + \text{sizeof}(\text{DataType})$$

$$\text{Locate}(k_i) = \text{Locate}(k_0) + \text{sizeof}(\text{DataType}) * i$$

只要确定了首地址，线性表中任意数据元素都可以随机存取。因此，顺序表为“**随机存取的存储结构**”。

k_0
k_1
k_2
...
k_i
...
k_{n-2}
k_{n-1}

2.2.1 线性表的顺序存储结构示意图

逻辑地址

数据元素

0	k_0
1	k_1
...	...
i	k_i
...	
n-1	k_{n-1}

存储地址

数据元素

$\text{Loc}(k_0)$	k_0
$\text{Loc}(k_0)+c$	k_1
...	...
$\text{Loc}(k_0)+i*c$	k_i
...	
$\text{Loc}(k_0)+(n-1)*c$	k_{n-1}

2.2.2 顺序表的定义

在C语言中可以用以下方式定义一个顺序表：

```
#define    MAXNUM  <最多允许的数据元素个数>
DataType  element[MAXNUM];
int        n;
```

定义的缺陷：没有反映出element和n的内在联系，即没有指出n是顺序表element本身的属性。在这个定义中，n和element完全处于独立平等的地位，所以程序中完全可以将n作为一个自由变量来使用。

为此，引进SeqList类型，它的定义为：

```

struct SeqList
{
    DataType element[MAXNUM];
    int  n;          /* n < MAXNUM */
};

```

在实际应用中，为了方便，通常定义一个struct SeqList类型的指针类型和别名：

```

typedef struct SeqList    *PSeqList;
typedef struct SeqList    SeqList;

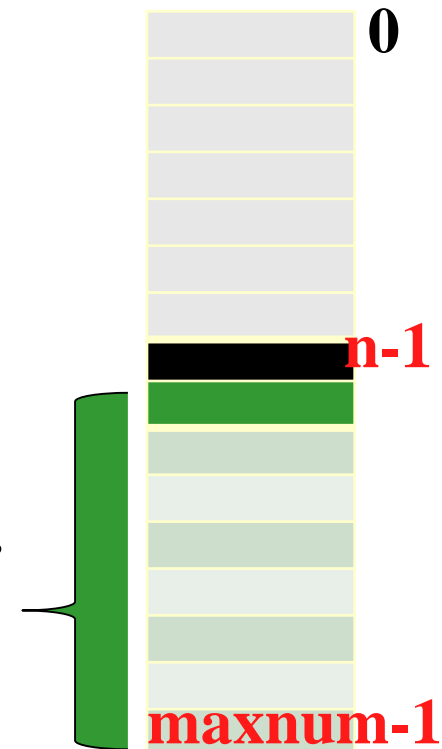
```

例如：PSeqList palist;

palist->n :顺序表中数据元素个数

palist->element[0], palist->element[1],...

剩余空间： n到MAXNUM-1



- 创建
- 插入
- 删除
- 查找
- 取值
- 判空

2.2.3 顺序表的基本运算

2.2.4 顺序表基本运算的实现[p33~36]

参看书P33~36页，理解插入、删除运算的实现

插入: $(k_0, k_1, \dots, k_p, k_{p+1}, \dots, k_{n-1})$ n
 $(k_0, k_1, \dots, k_p, \mathbf{x}, k_{p+1}, \dots, k_{n-1})$ $n+1$

删除: $(k_0, k_1, \dots, k_{p-1}, \mathbf{k_p}, k_{p+1}, \dots, k_{n-1})$ n
 $(k_0, k_1, \dots, k_{p-1}, k_{p+1}, \dots, k_{n-1})$ $n-1$

12
111
22
8888
333
123
56
99

333前插入234



12
111
22
8888
234
333
123
56
99

删除22元素



12
111
8888
234
333
123
56
99



插入、删除元素时的时间效率分析:

设 p_i 是在第 i 个元素之前插入一个元素的概率, 则在长度为 n 的线性表中插入一个元素时所需移动元素的次数的期望值(平均次数)为:

$$E_{is} = \sum_{i=0}^n p_i (n - i)$$

等概率情况下: $p_i = \frac{1}{n+1}$

$$E_{is} = \frac{1}{n+1} \sum_{i=0}^n (n - i) = \frac{n}{2}$$

$$E_{ds} = \sum_{i=0}^{n-1} q_i (n - i - 1)$$

$$q_i = \frac{1}{n}$$

$$E_{dl} = \frac{1}{n} \sum_{i=0}^{n-1} (n - i - 1) = \frac{n-1}{2}$$

删除

插入、删除元素时的时间效率分析：

插入、删除的完成是通过数据元素的移动完成的；

插入、删除一个数据元素平均需要移动大约一半数量的数据元素，效率低。

改进：使用链接存储结构

定位运算：通过比较完成，完成一次定位也平均需要 $n/2$ 次比较，时间复杂度为 $O(n)$ ；

对于**有序表**的改进：采用**折半查找**方法， $O(\log_2 n)$

顺序表空间的扩展

```

struct SeqList
{
    int MAXNUM;           //最多允许元素数目
    DT *element;         //元素数组
    int n;                //  $n < \text{MAXNUM}$ 
} *pSeq;

```

```
//假定扩展1倍
```

```
pSeq->MAXNUM = 2 * pSeq->MAXNUM;
```

//申请新的空间

```
DT *NE = (DT *) malloc(pSeq->MAXNUM* sizeof(DT));
```

if (! NE)

```
{ printf("Overflow\n"); return; }
```

//原成员拷贝

```
memcpy( NE, pSeq->element, pSeq->n* sizeof(DT));
```

//释放原空间并设置新空间

```
free(pSeq->element);
```

pSeq->element = NE;

#####
#####
#####
#####
#####
#####

[illegible]

2.3 线性表的链接表示及实现

线性表的顺序存储结构的特点： 逻辑相邻的数据元素通过物理存储位置相邻表达，可以随机存取表的任意数据元素，并且物理元素的存取可以通过一个简单的公式来表示。

缺点： 插入、删除数据元素需要大量的数据**元素移动**，时间效率低。另外，需要预先按照最大空间分配连续的物理存储空间，容易造成空间浪费（当数据元素个数远远少于最大空间时）和存储空间不够（当数据元素个数超过最大空间时），**不易扩展**。

改进： 按照链接存储结构存储线性表（不要求逻辑相邻物理也相邻，通过附加指针表达逻辑关系），克服顺序存储的不足，但失去了随机存取的优点。

基本内容和思路

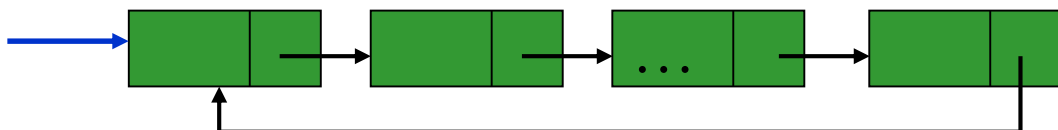
- 单链表



静态链表

- 单链表的改进和扩展

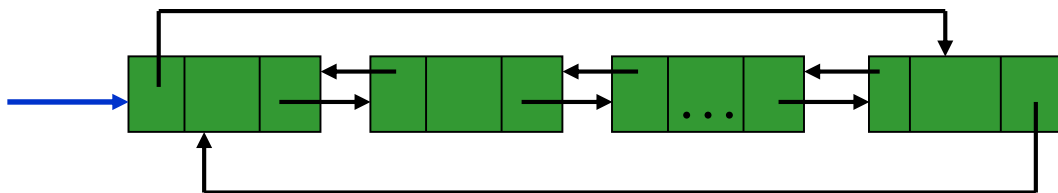
循环链表



双向链表



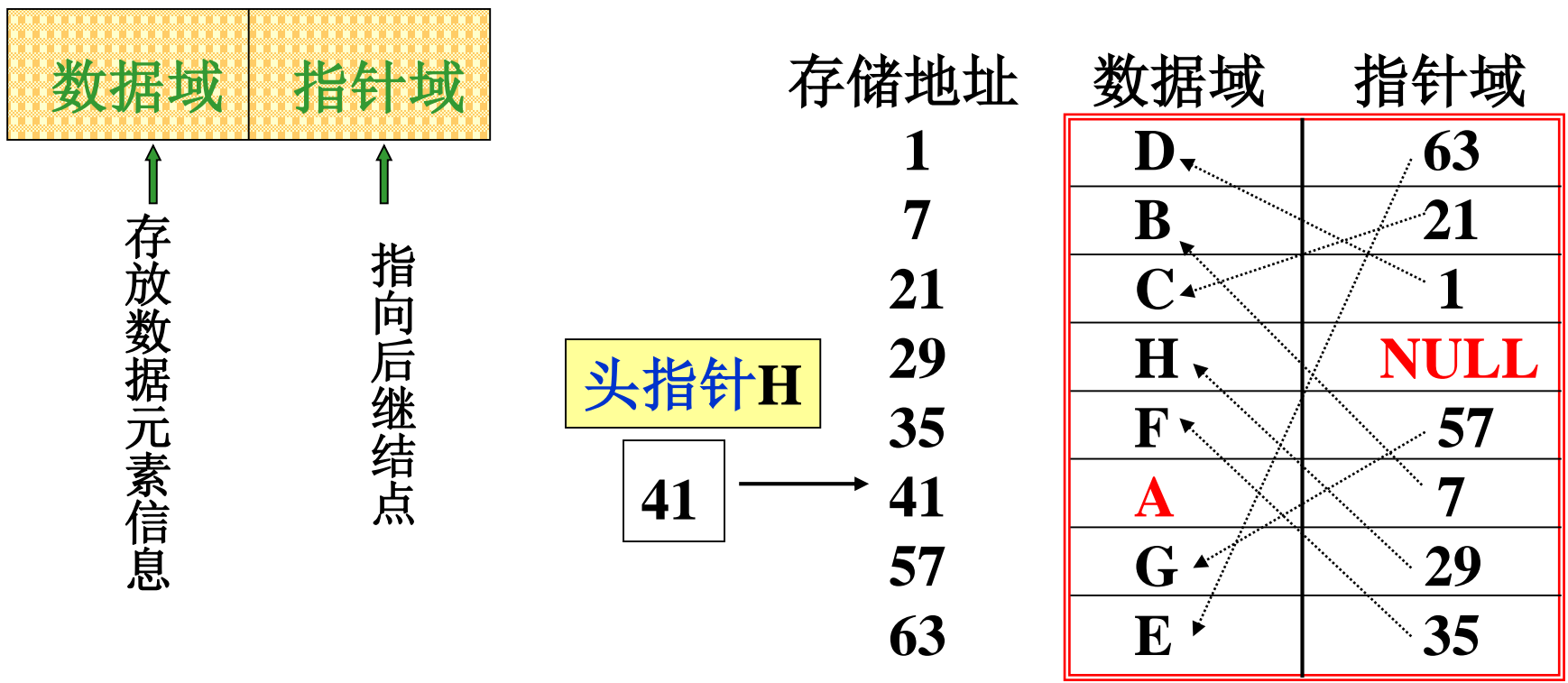
双向循环链表



2.3.1 单链表

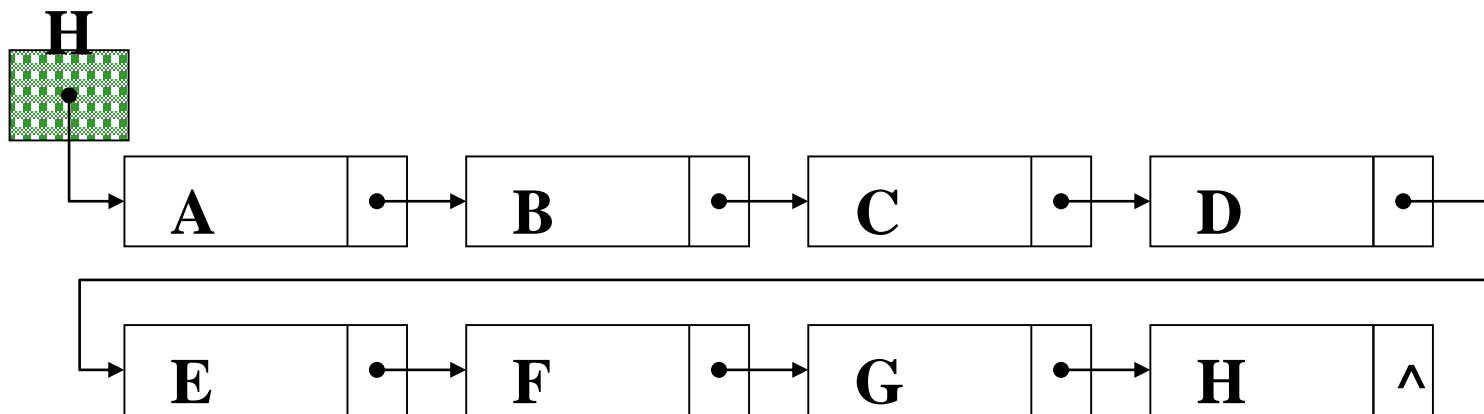
1. 结点定义：数据元素的存储映象，由数据域和指针域构成。

(A, B, C, D, E, F, G, H)

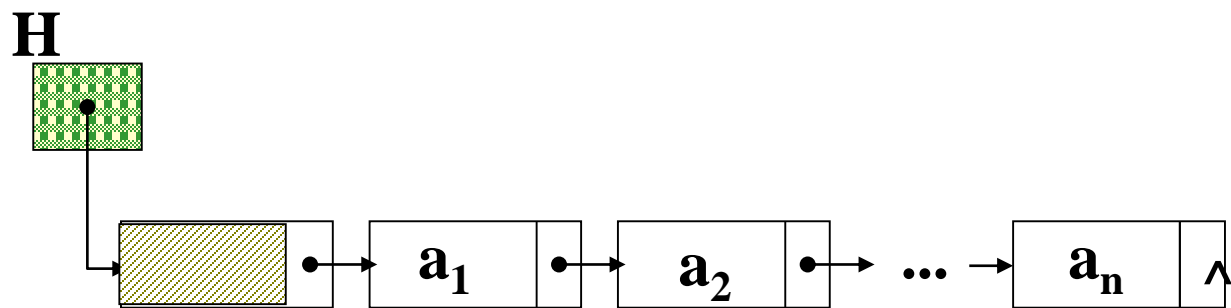


最后结点无后继，因此其指针域为空

2. 单链表的图示（数据元素的逻辑顺序，不是存储位置）



单链表的逻辑状态（不带头结点）



头结点

带头结点

3. 单链表存储结构的定义

```
typedef struct Node      /* 单链表结点结构 */  
{   DataType  info;  
    struct Node *link; /* struct Node *next; */  
}Node, *PNode;
```

```
typedef struct LinkList /* 单链表类型封装定义 */  
{  
    PNode  head; /* 无头结点：指向第一个结点 */  
            /* 带头结点：指向头结点 */  
    ..... /* 有关单链表的其它信息 */  
}LinkList, *PLinkList;  
PlinkList plist; /* 单链表定义 */
```

对于线性表(A, B, C, D, E, F), 假定p指向结点C, 则:

p->link指向C的后继D

p->link->link指向?

p->link->link->link指向?



无头结点

整个单链表: **plist**

第一个结点: **plist->head**

空表判断: **plist \neq NULL, plist->head = NULL**

带头结点

整个单链表: **plist**

头结点: **plist->head**

第一个结点: **plist->head->link**

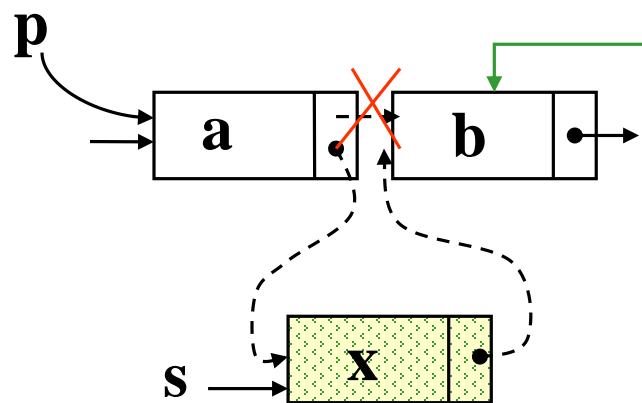
空表判断: **plist \neq NULL, plist->head->link = NULL**

4. 单链表基本运算及其实现

- 创建空的单链表 **CreateNull_Link(void)**
- 插入结点 **Insert_Link(PLinkList plist, int I, DataType x)**
- 删除结点 **Delete_Link(PLinkList plist, DataType x)**
- 定位运算 **Locate_Link(PlinkList plist, DataType x)**
- 求存储地址运算 **Find_Link(PLinkList plist, int I)**
- 判空运算 **IsNull_Link(PLinkList plist)**
- 单链表的建立 **Create_Link(DataType data[], int num)**

存储结构及基本运算实现

5. 结点插入和删除图示

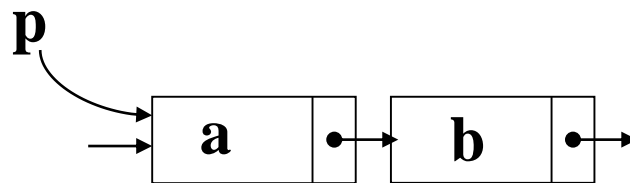


单链表插入结点时的情况

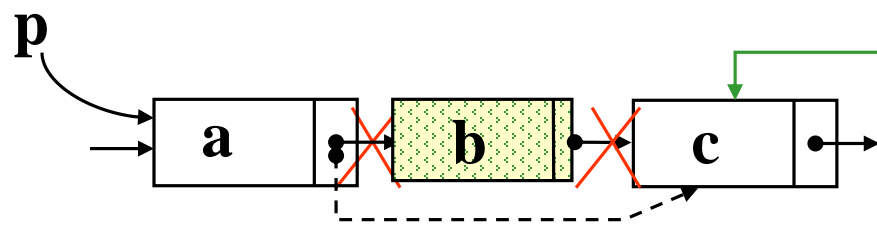
$s \rightarrow \text{link} = p \rightarrow \text{link};$ (1)

$p \rightarrow \text{link} = s;$ (2)

注: (1),(2)顺序不能反



单链表



单链表删除结点时的情况

$p \rightarrow \text{link} = p \rightarrow \text{link} \rightarrow \text{link};$

对于单链表，必须找到插入或删除结点的前驱

结点插入和删除运算中的几个问题:

- 1) 带头结点与不带头结点问题
- 2) 结点的生成与删除（动态生成malloc，删除free）
- 3) 时间复杂度问题

由于“条件插入、删除运算”首先需要定位，而定位必须从头结点开始顺链一个一个结点进行比较，因此其“基本的操作”为比较运算。比较运算的执行次数与结点的位置有关，最好情况时为1次（第一个结点），最坏为n次（所有结点皆比较一次），因此条件插入、删除运算的时间复杂度仍然为 $O(n)$ 。

对于在p所指结点后插入新的结点，无定位问题，因此其时间复杂度为 $O(1)$ 。

6. 单链表与顺序表的比较

$$d = \text{数据空间} / \text{总空间}$$

- (1) 单链表的**存储密度**比顺序表低；但在许多情况下链式的分配比顺序分配有效；顺序表为静态结构，而链表为动态结构。
- (2) 在单链表里进行插入、删除运算比在顺序表里容易得多；顺序表通过数据元素移动完成，而链表通过修改指针完成。
- (3) 对于顺序表，可通过简单的定位公式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，顺序表为随机存取结构，而链表不是。

7. 单链表与顺序表的选择

(1) 有利于基本运算的实现

频繁访问：顺序表（随机存取）； 频繁插入、删除：链表

(2) 有利于数据的特性

顺序表：难于事先估计数据元素个数，过大浪费空间，过小易产生溢出；

链表：动态申请，但存储密度比顺序表低；

(3) 有利于软件环境（程序设计语言）

程序设计语言是否支持动态分配？ Fortran、Basic等不支持动态分配，此时只能选择顺序表。

2.3.2 静态链表

整个结构采用顺序表。

顺序表中，每个元素包含两项：数据项和存储后继位置的游标项。

通过游标表达数据元素之间的逻辑关系。逻辑上相邻元素，存储位置不一定相邻。

D1	C1
D3	C3
D4	C4
D2	C2

数据

后继在数组中的位置（游标）

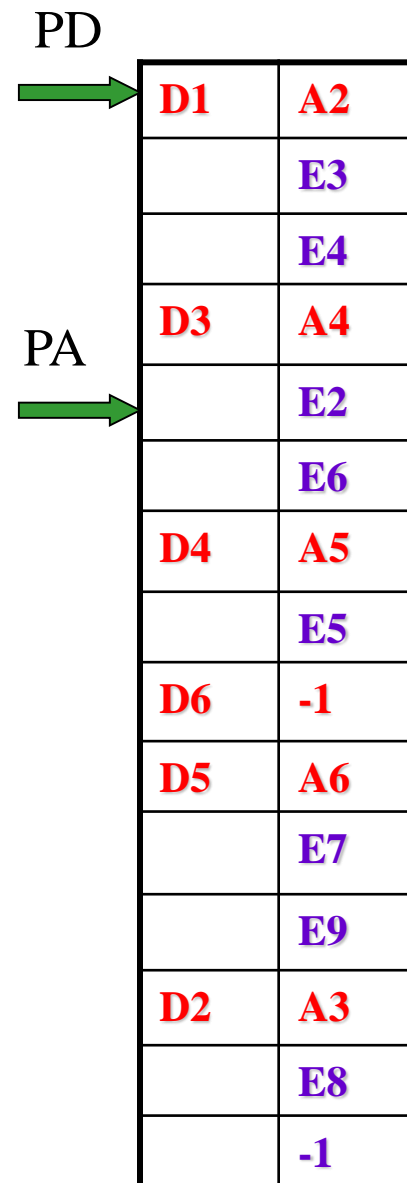
```
#define MaxSize    1000    /* 供分配空间的大小 */
typedef struct
{
    DataType data;
    int    cursor;    （游标代替链表中的指针）
}Component, SLinkList[MaxSize];
```

游标代替链表中的指针，插入、删除时仅需要修改游标项完成。

此时的静态顺序表具有链表的优点。

另外，将整个存储空间分成两部分：数据元素组成的**工作空间**和备用元素构成的**备用空间**，这两部分分别设计一个链表（**静态链表**）进行管理。

当需要插入一个结点时，从备用链中取出第一个结点作为待插入的结点（类似于**Malloc**）；当从工作链中删除一个结点时，将被删除的结点连接到备用链上（类似于**Free**）。



The diagram illustrates a static linked list structure using a table with two columns. The first column represents the 'Work Space' (工作空间) and the second column represents the 'Backup Space' (备用空间). Two pointers, PD and PA, are shown on the left, each with a green arrow pointing to a specific row in the table.

PD		
→	D1	A2
		E3
		E4
	D3	A4
PA	→	E2
		E6
	D4	A5
		E5
	D6	-1
	D5	A6
		E7
		E9
	D2	A3
		E8
		-1

假定S为SlinkList型变量， $s[0].\text{cursor}$ 指示第一个结点在数组中的位置，若设 $i = s[0].\text{cursor}$ ，则 $s[i].\text{data}$ 保存第一个数据元素， $s[i].\text{cursor}$ 指示第二个结点在数组中的位置。如第 i 个分量表示链表的第 k 个结点，则 $s[i].\text{cursor}$ 指示第 $k+1$ 个结点在数组中的位置。

$i = s[i].\text{cursor}$ 类似链表中的 $p = p \rightarrow \text{link}$
最后结点的 $\text{cursor} = 0$ (类似链表中的NULL)。

线性表: (A, B, C, D, E, F)

S[0]		3
S[1]	C	5
S[2]		
S[3]	A	6
S[4]	E	7
S[5]	D	4
S[6]	B	1
S[7]	F	-1

→

0		9
1	A	2
2	B	3
3	C	4
4	D	5
5	E	6
6	F	7
7	G	8
8	H	0
9		10
10		0

修改前

→

0		10
1	A	2
2	B	3
3	C	4
4	D	9
5	E	6
6	F	7
7	G	8
8	H	0
9	D'	5
10		0

插入D'

→

0		6
1	A	2
2	B	3
3	C	4
4	D	9
5	E	7
6	F	7
7	G	8
8	H	0
9	D'	5
10		0

10

删除F

静态链表示例(插入D', 删除zheng)

// 初始化静态链表，将所有结点连接成一个备用链

void InitList(SLinkList list); **/* 用 ‘0’表示空 */**

//从备用链中申请空间，返回申请到的空间的下标

int Malloc(SLinkList list);

// 释放 ‘k’指向的空间，将其连接到备用链中

void Free(SLinkList list, int k);

void InitList(SLinkList list)

{ **//所有空间给备用链**

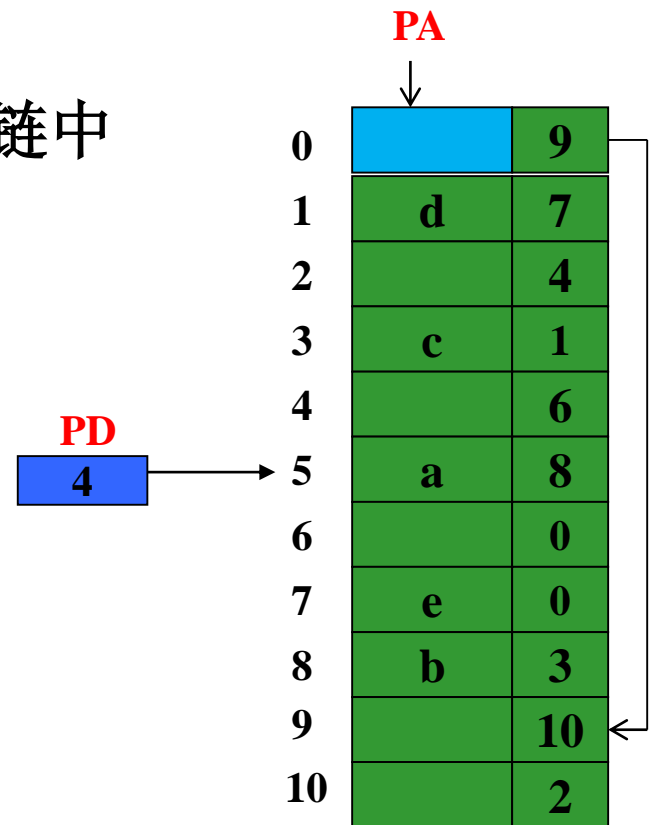
int i;

for(i = 0; i < MaxSize - 1; i++)

list[i].cursor = i + 1;

list[MaxSize-1].cursor = 0;

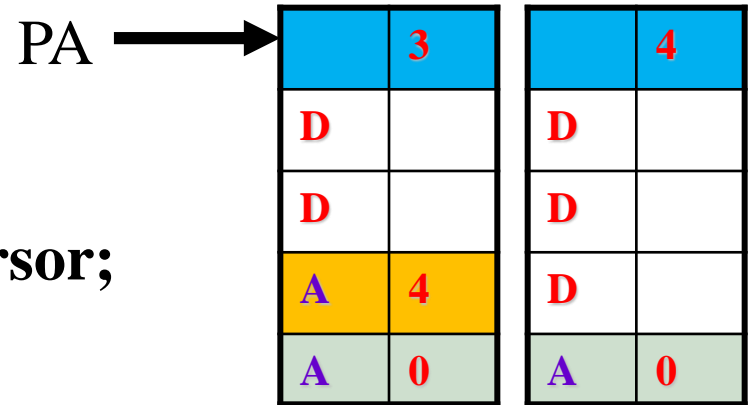
}



工作链表+备用链表

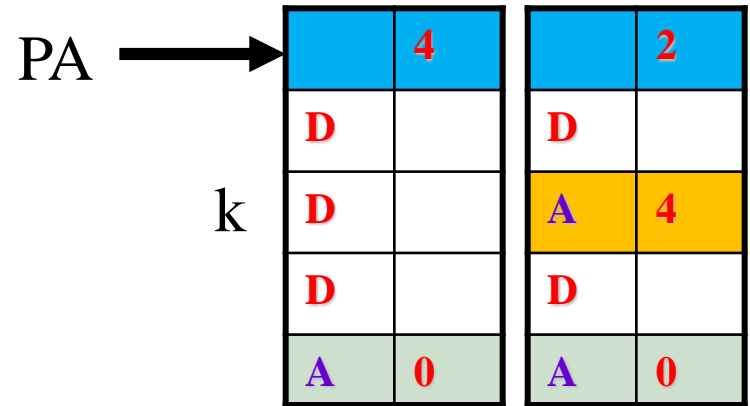
//从备用链中取第一个

```
int Malloc(SLinkList list)
{   int i = list[0].cursor;
    if(i != 0)    list[0].cursor = list[i].cursor;
    return i;
}
```



//插入到备用链的第一个位置

```
void Free(SLinkList list, int k)
{   list[k].cursor = list[0].cursor;
    list[0].cursor = k;
}
```



```
int Locate_SL(SLinkList S, int d, DataType e)
{   int i = s[d].cursor;
    while (i && s[i].data != e)    i = s[i].cursor;
    return i;
}
```

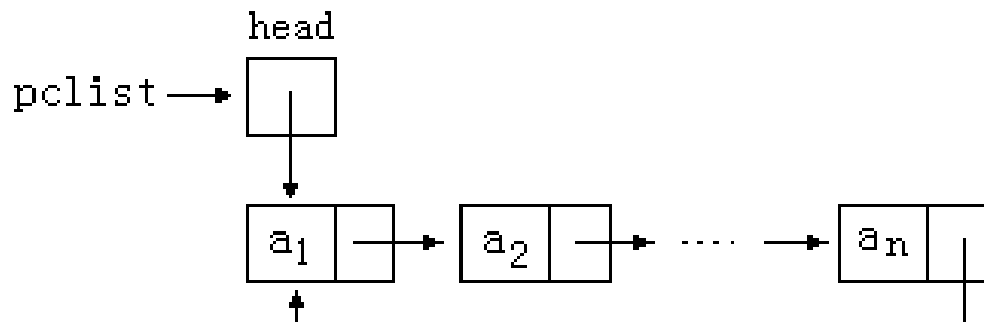
//工作链检索
//由d开始

2.3.3 循环链表

建立： 将最后一个结点的指针项指向第一个结点（或头结点），构成一个循环链。

循环链表的优点：

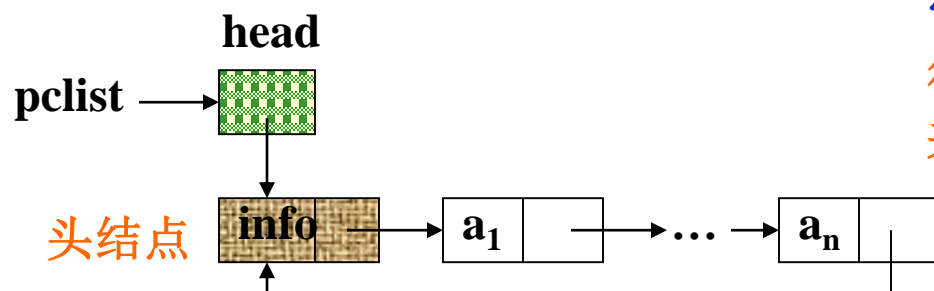
从任何一个结点出发，可以访问表中任何一个结点元素。



无头结点：

循环条件： $p \rightarrow \text{link} = \text{pclist} \rightarrow \text{head}$

表空条件： $\text{pclist} \rightarrow \text{head} = \text{NULL}$



带头结点：

循环条件： $p \rightarrow \text{link} = \text{pclist} \rightarrow \text{head}$

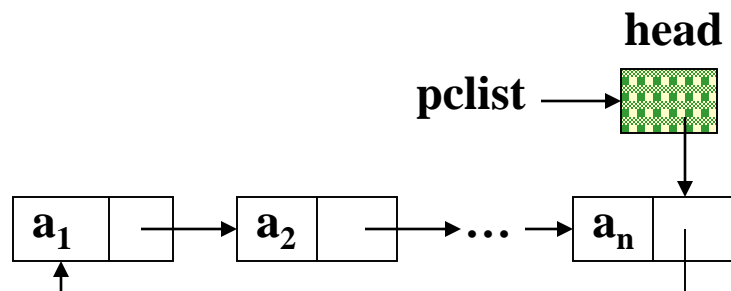
表空条件： $\text{pclist} \rightarrow \text{head} \rightarrow \text{link} = \text{NULL}$

上面循环链表的缺点：

如果要访问最后一个结点，仍要访问表中所有的结点。

改进：

修改`pclist->head`指向最后一个结点，方便某些操作(如两个链表合并等)。



最后结点： `pclist->head`
第一个结点： `pclist->head->link`
循环条件： `p = pclist->head`
空表判断： `p->head = NULL`

思考问题：

如何将一个单循环链表（无头结点，且`pclist->head`指向第一个结点）倒置？

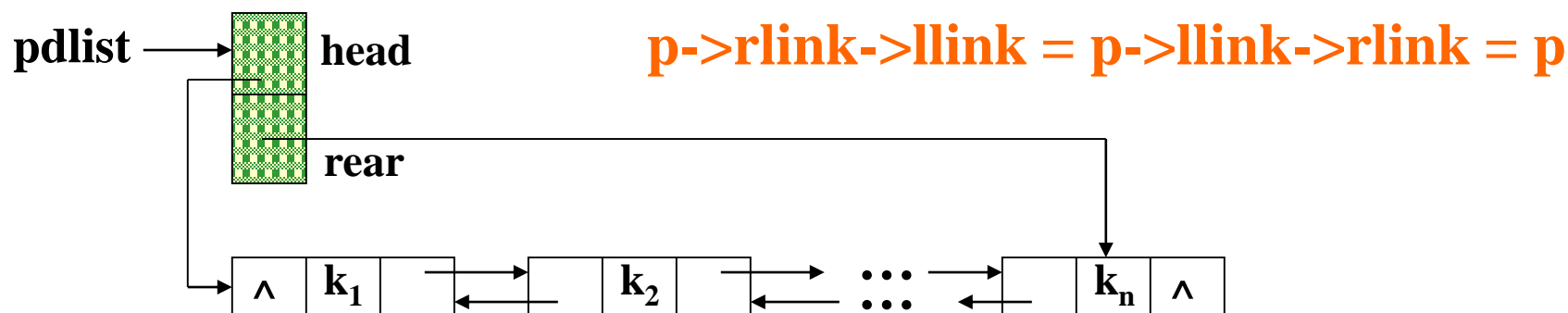
$$(a_1, a_2, \dots, a_n) \Rightarrow (a_n, a_{n-1}, \dots, a_1)$$

2.3.4 双向链表和双向循环链表

单链表缺点：找后继容易，找前驱必须从头开始查找。

双向链表：既可以找前驱，也可以找后继。

结点结构：



不带头结点

空表判断 : $pdlist \rightarrow head = NULL$

最后结点判断 : $p \rightarrow rlink = NULL$

第一个结点 : $pdlist \rightarrow head$

最后结点 : $pdlist \rightarrow rear$

描述双链表及其结点类型的说明为：

```
typedef struct DoubleNode /* 双链表结点结构 */
```

```
{    DataType   info;
```

```
    struct DoubleNode *llink, *rlink;
```

```
}DoubleNode, *PDoubleNode;
```

```
typedef struct DoubleList /* 双链表类型 */
```

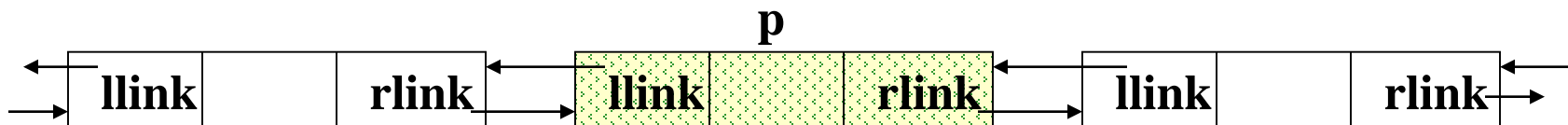
```
{    PDoubleNode head;           /* 指向第一个结点 */
```

```
    PDoubleNode rear;          /* 指向最后一个结点 */
```

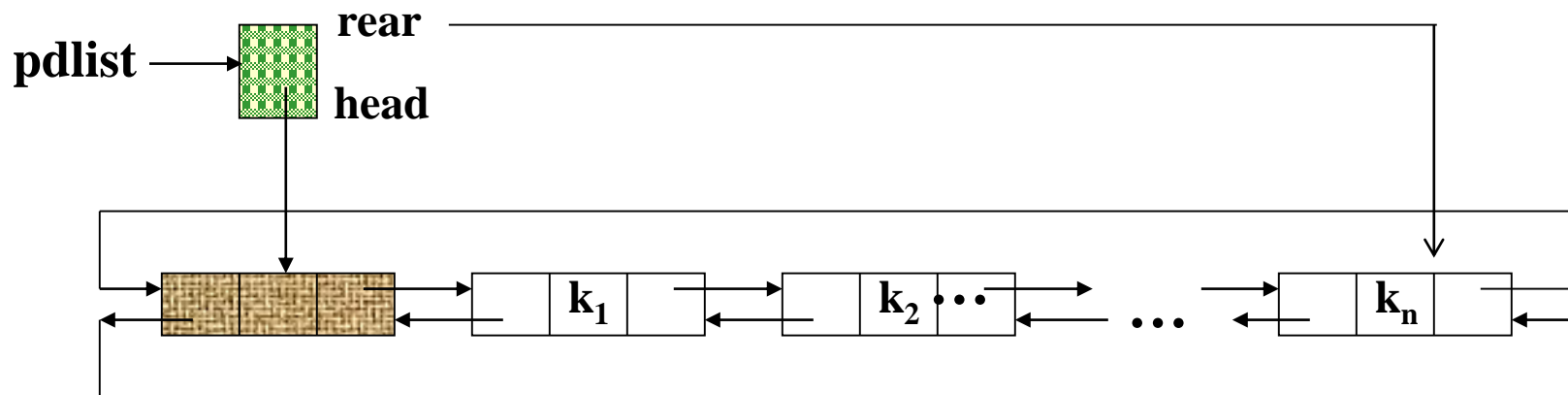
```
    .....                     /* 其它信息 */
```

```
}DoubleList, *PDoubleList;
```

重要特性: $p \rightarrow rlink \rightarrow llink = p \rightarrow llink \rightarrow rlink = p$



双向循环链表



带头结点的双向循环链表

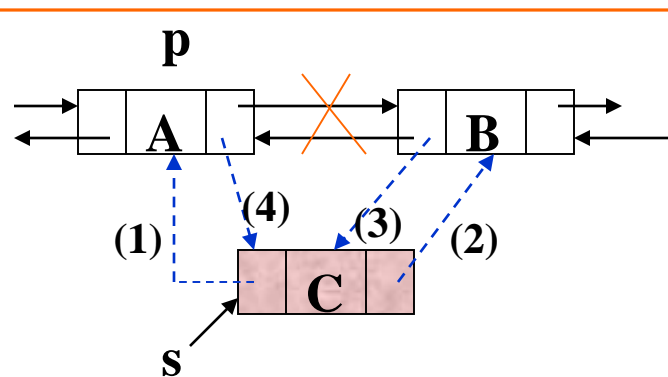
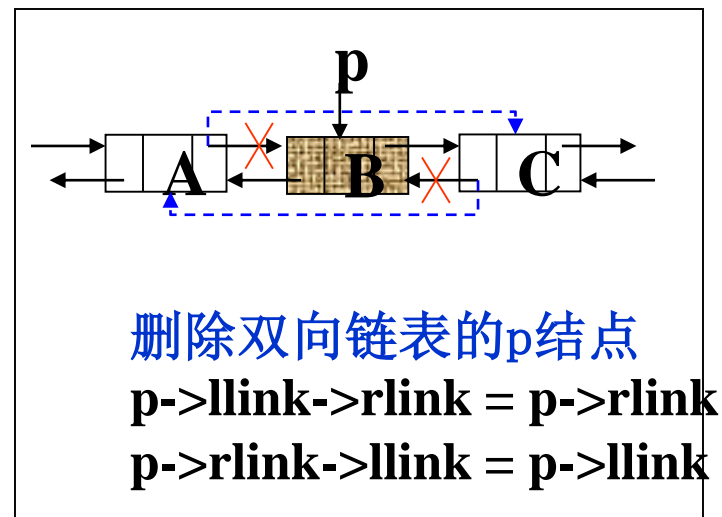
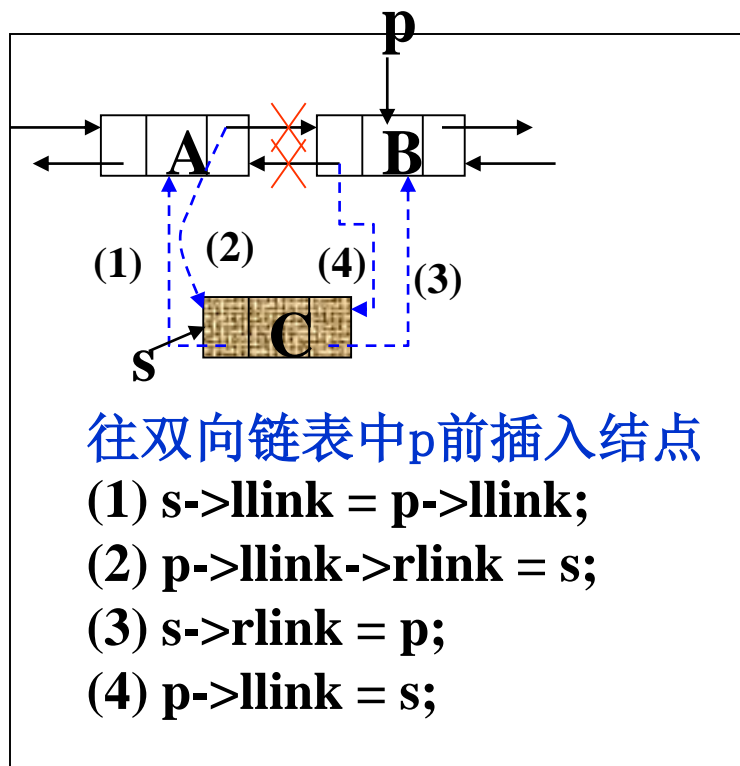
空表判断 : `pdlist->head->rlink = NULL`

最后结点判断: `p->rlink=pdlist->head` 或
`p =pdlist->head->llink`

第一个结点 : `pdlist->head->rlink`

最后结点 : `pdlist->head->llink`

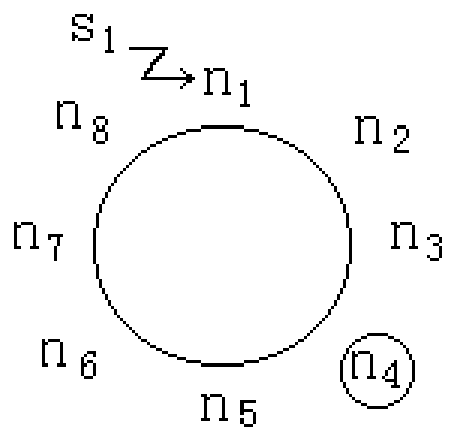
双向链表结点插入和删除的图示：



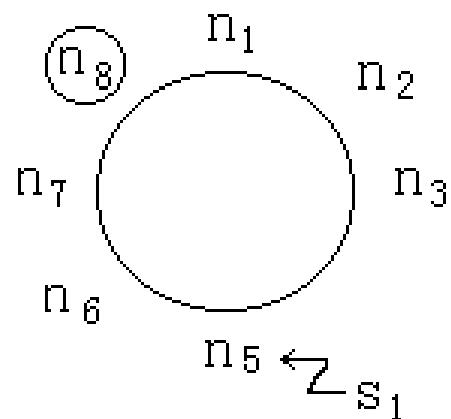
2.4 应用举例—Josephus问题

n 个人围坐在一圆桌周围，现从第 s 个人开始报数，数到第 m 的人出列，然后从出列的下一个入重新开始报数，数到第 m 的人又出列，如此反复直到所有的人全部出列为止。Josephus问题是：对于任意给定的 n, s 和 m ，求出按出列次序得到的 n 个人员的序列。

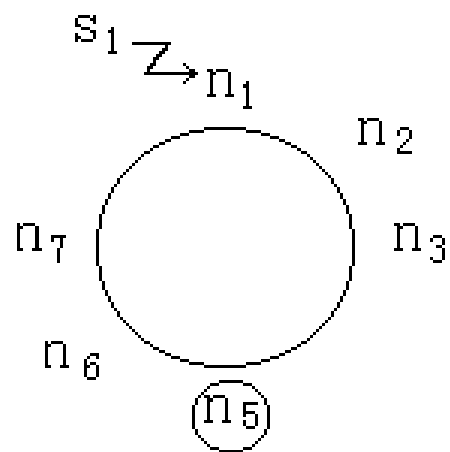
现以 $n=8$, $s=1$, $m=4$ 为例，问题的求解过程如下列图所示。图中 s_1 指向开始报数位置，带圆圈的是本次应出列人员。若初始顺序为 $n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8$ ，则问题的解为 $n_4, n_8, n_5, n_2, n_1, n_3, n_7, n_6$ 。



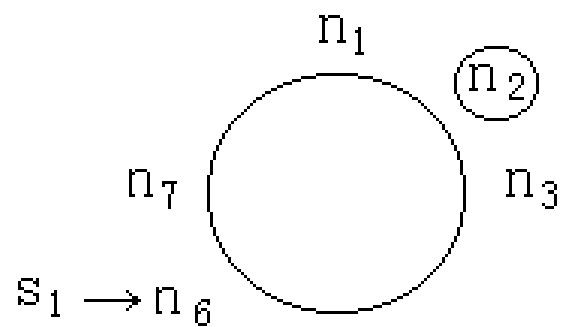
(a) n4



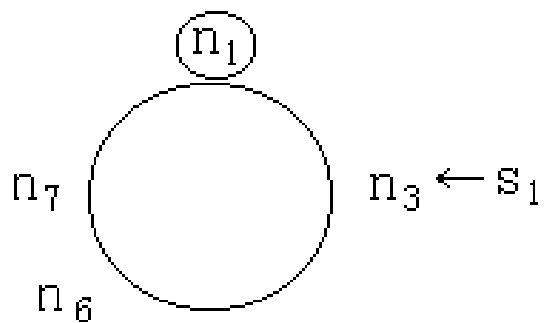
(b) n4 n8



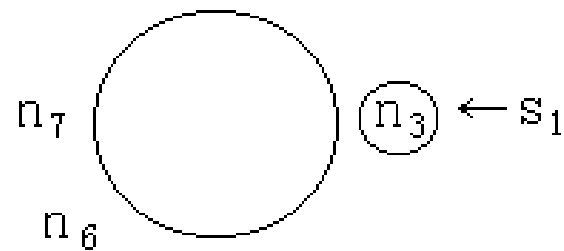
(c) n4 n8 n5



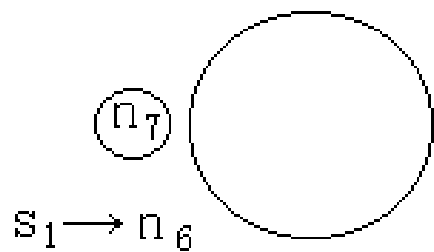
(d) n4 n8 n5 n2



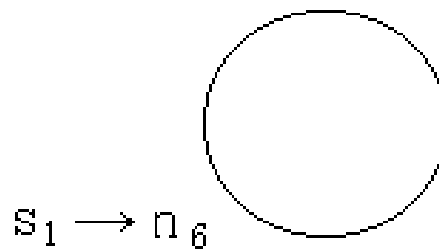
(e) n_4 n_8 n_5 n_2 n_1



(f) n_4 n_8 n_5 n_2 n_1 n_3



(g) n_4 n_8 n_5 n_2 n_1 n_3 n_7



(h) n_4 n_8 n_5 n_2 n_1 n_3 n_7 n_6

2.4.1 顺序表方式实现

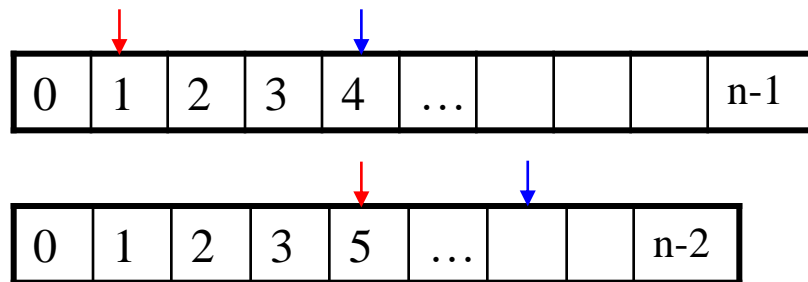
步骤：1: 建立顺序表

2: 出列

时间复杂度分析：出列元素的删除（移动实现）

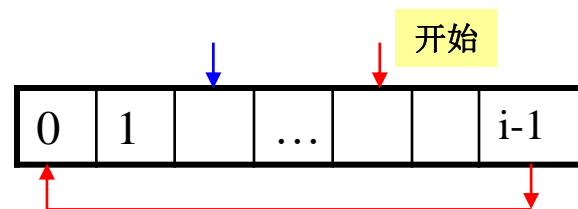
为基本运算（每次最多*i-1*个元素移动，需要*n-1*次）

$$(n-1)+(n-2)+\dots+1 = n(n-1)/2 \Rightarrow O(n^2)$$



当前剩下*i*个人,从*s1*数第*m*个的位置:

$$s1 = (s1+m-1)\%i$$

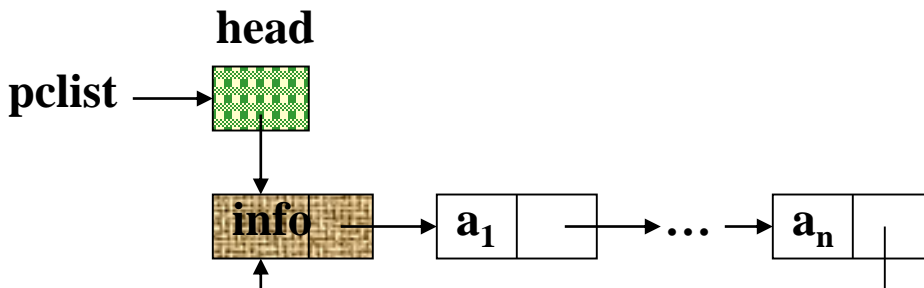


2.4.2 循环链表方式实现

步骤：1: 建立带头结点的循环单链表

2: 寻找第*s*个结点，输出并删除第*m*个结点。

时间复杂度分析：三部分时间（创建链表： $O(n)$ +求第*s*个结点： $O(s)$ +求第*m*个应出列的元素： $O(m*n)$ ）




典型应用：一元多项式表达与实现

一元多项式： $P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_n x^n$

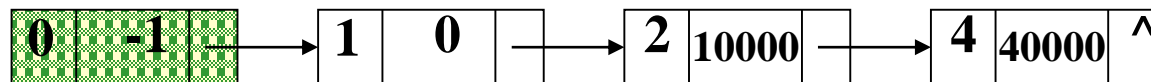
线性表表示： $P = (p_0, p_1, p_2, \dots, p_n)$

顺序表表示：只存系数（第*i*个元素存 x^i 的系数）

特殊问题： $p(x) = 1 + 2x^{10000} + 4x^{40000}$  浪费空间

链表表示：

结点结构



相加运算：相同指数对应结点的系数项相加，如“和”为0，删除结点，否则必定为“和链表”的一个结点。（**实质**上就是两个单链表的合并问题）

0	1
1	0
2	0
3	0
...	0
10000	2
...	0
	0
	0
40000	4

车辆管理

某单位最多100辆车辆，每辆车的基本信息包括：车号、车型、购买日期、购买价格、驾驶员名称、违章记录等。其中，各辆车的违章次数可能差别很大（有的一次没有，有的可能有数十次）。每次违章记录中包括：违章时间、违章地点、违章代码（如闯红灯、违规停车等）、违章处理等。

现思考：

- a) 数据结构？哪种存储结构能够有效地表达上述车辆管理问题；
- b) 如何查找某辆车的违章信息？
- c) 如何统计某一个时间段内所有车的违章记录？
- d) 如何统计某一个地点的所有违章记录？
- e) 如何统计某一违章代码的所有违章记录？
- f) 新买车，车辆报废问题；
- g) 新违章记录加入，违章处理后的修改？

统计违章记录时，需要给出车号、违章记录数目等。

实现上述车辆管理的程序，包括车辆信息的文件导入/导出和显示等。

一种可能的结构

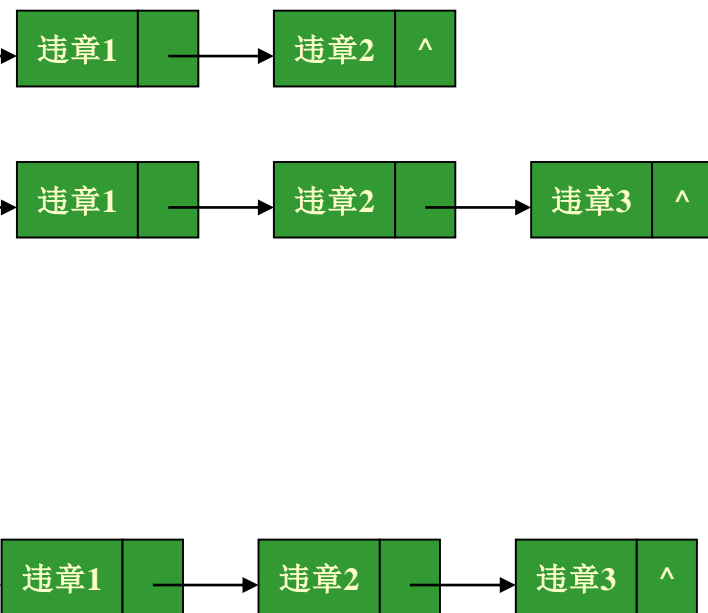
```
//车辆定义
typedef struct
{
    int id;           //车号
    char type[32];    //车型
    float price;      //价格
    char driver[32];  //驾驶员
    WZ *pWZ;          //违章记录
} CAR
```

```
//车辆顺序表
define MAX_NUM_CARS 100
typedef struct
{
    CAR cars[MAX_NUM_CARS];
    int m;           //车辆数目
} CarList;
```

顺序表

Car ₁	—
Car ₂	^
Car ₃	—
.....
Car _m	—

单链表



```
//违章结点定义
typedef struct _tagWZ WZ;
struct _tagWZ
{
    int time;           //时间
    char place[64];     //地点
    int code;           //代码
    bool ok;            //是否处理
    WZ *next;           //下一个
};
```

小结

本章主要讨论了线性表的概念、存储表示以及基本运算的实现，需要重点掌握。线性表是一种最简单的数据结构，它是 n 个元素的有限序列。常用存储方式：顺序、链接存储结构；

顺序存储结构中，物理位置相邻表达逻辑关系相邻；只存储数据元素自身信息，存储密度大、空间利用率高；元素定位通过简单公式，是随机存取结构；但数据元素的插入、删除是通过大量的元素移动完成，时间效率低；必须事先估计最大元素数。

链接存储结构中，结点空间是动态申请和释放的；数据元素之间的逻辑关系通过附加指针表达；插入、删除只需要修改指针即可，不必移动大量的数据元素；缺点是需要附加的指针，存储密度降低；元素的访问需要从头结点开始顺链一个一个结点进行，是非随机存取的存储结构。

链表：**单链表、循环链表、双向链表、静态链表**（游标）

结合Josephus问题和一元多项式表达讨论了顺序表和链表的应用。

另外，常用算法的时间复杂度分析也是需要掌握的。

书面作业：书p67复习(4、6、7)，算法（3、12、16）

上机作业：车辆管理问题