

面向21世纪课程教材  
普通高等教育“十一五”国家级规划教材  
北京市高等教育精品教材  
教育部普通高等教育精品教材

# 算法与数据结构

## 第二讲：线性表

张乃孝等编著

# 《算法与数据结构—C语言描述》

## 2 线性表

- 2.1 线性表的概念
- 2.2 顺序表示
- 2.3 链接表示

## 2.1 基本概念与抽象数据类型

线性表（简称为表）是零个或多个元素的有穷序列。

$$\mathbf{L}=(k_0,k_1,\dots,k_{n-1})$$

- 线性表的逻辑结构： $\mathbf{L}=\langle \mathbf{K}, \mathbf{R} \rangle$

其中 $\mathbf{K}=\{k_0,k_1,\dots,k_{n-1}\}$ ,

$$\mathbf{R}=\{\langle k_i,k_{i+1} \rangle \mid 0 \leq i \leq n-2\},$$

$i$  称为元素 $k_i$ 的索引或下标。

表中的元素又称表目。

# 基本概念

- 表中所含元素的个数称为表的长度;
- 长度为零的表称为空表;
- $k_0$ 称为第一个元素;
- $k_{n-1}$ 称为最后一个元素;
- $k_i$  ( $0 \leq i \leq n-2$ ) 是 $k_{i+1}$ 的前驱;
- $k_{i+1}$ 是 $k_i$ 的后继。

# 符号说明

---

- 假设用List表示一个线性表类型，用DataType来表示其元素的类型，元素的下标用position类型的量表示；
- 则可以说明如下：  
List list;  
DataType x;  
position p;

# 线性表的抽象数据类型

**ADT List is**

**operations**

List createNullList ( void )

创建并且返回一个空线性表。

int insertPre ( List list, position p, DataType x )

在list中p位置前插入值为x的元素，并返回插入成功与否的标志。

int insertPost ( List list, position p, DataType x )

在list中p位置后插入值为x的元素，并返回插入成功与否的标志。

int deleteV (List list, DataType x )

在list中删除一个值为x的元素，并返回删除成功与否的标志。

# 线性表的抽象数据类型(续)

int deleteP (List list, position p )

在list中删除位置为p的元素，并返回删除成功与否的标志。

Position locate ( List list, DataType x )

在list中查找值为x的元素的位置。

int isNull ( List list )

判别list是否为空线性表。

**end ADT List;**

## 2.2 顺序表示

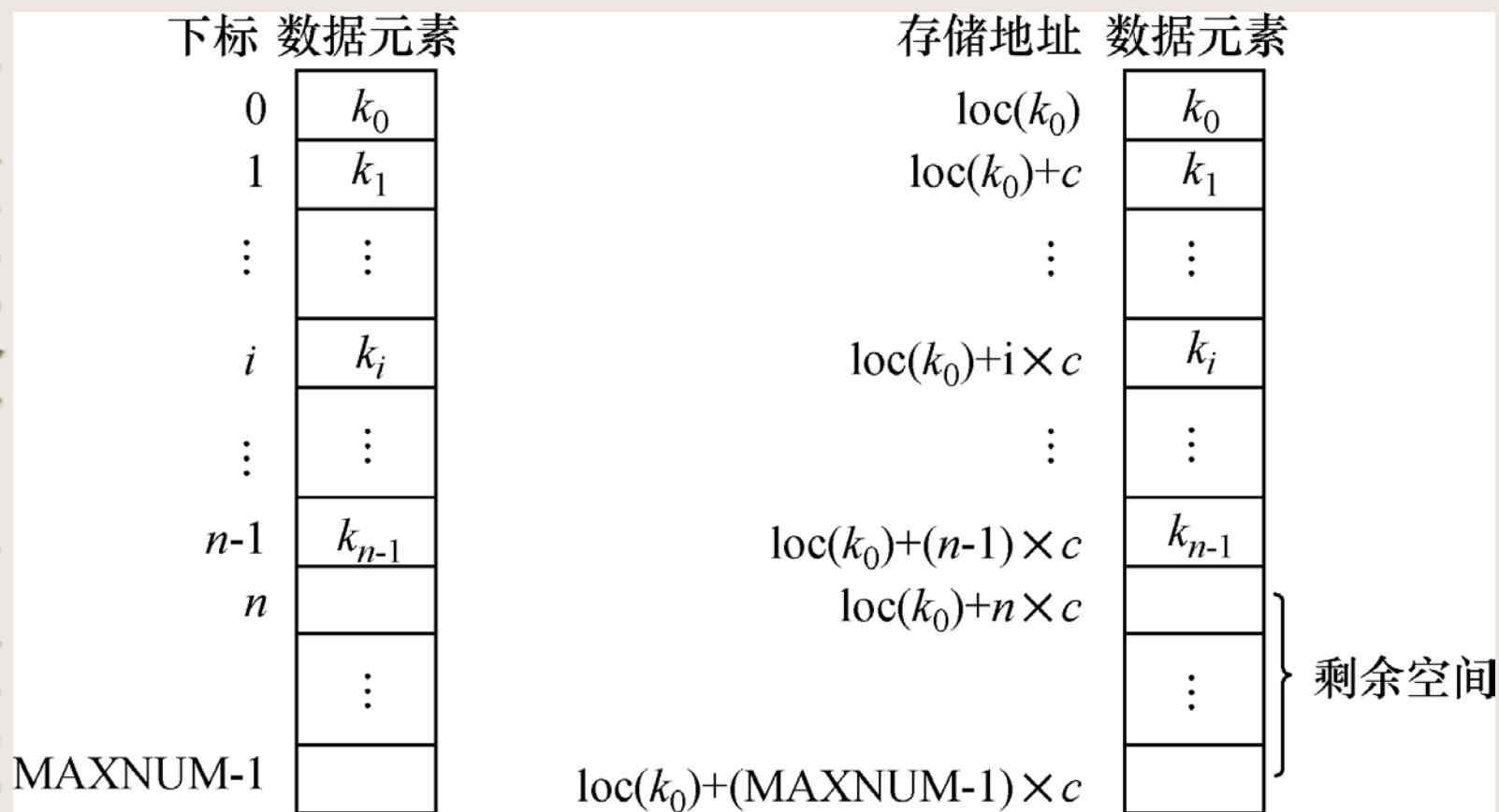
- 采用顺序存储是表示线性表最简单的方法
  - 将线性表中的元素依次存储在一片相邻的存储区域中
  - 元素之间逻辑上的相邻关系通过元素在计算机内物理位置上的相邻关系来表示
    - 逻辑上相邻 $\Leftrightarrow$ 存储位置相邻
  - 顺序表示的线性表也称顺序表



# 存储结构

- 线性表的**首地址**或**基地址**
  - 顺序表中 $k_0$ 的存储位置 $loc(k_0)$
- 假定顺序表中每个元素占用 $c$ 个存储单元
  - 下标为 $i$ 的元素 $k_i$ 的存储位置为:
$$loc(k_i) = loc(k_0) + i \times c$$
- 以数组为基础实现线性表
  - 考虑到线性表元素的变化, 创建一个大数组, 表元素连续存在数组前一段
    - 需要记录表中元素个数
    - 连续存放给删除操作带来不便
    - 数组固定大小可能导致插入操作失败

# 存储结构示意图



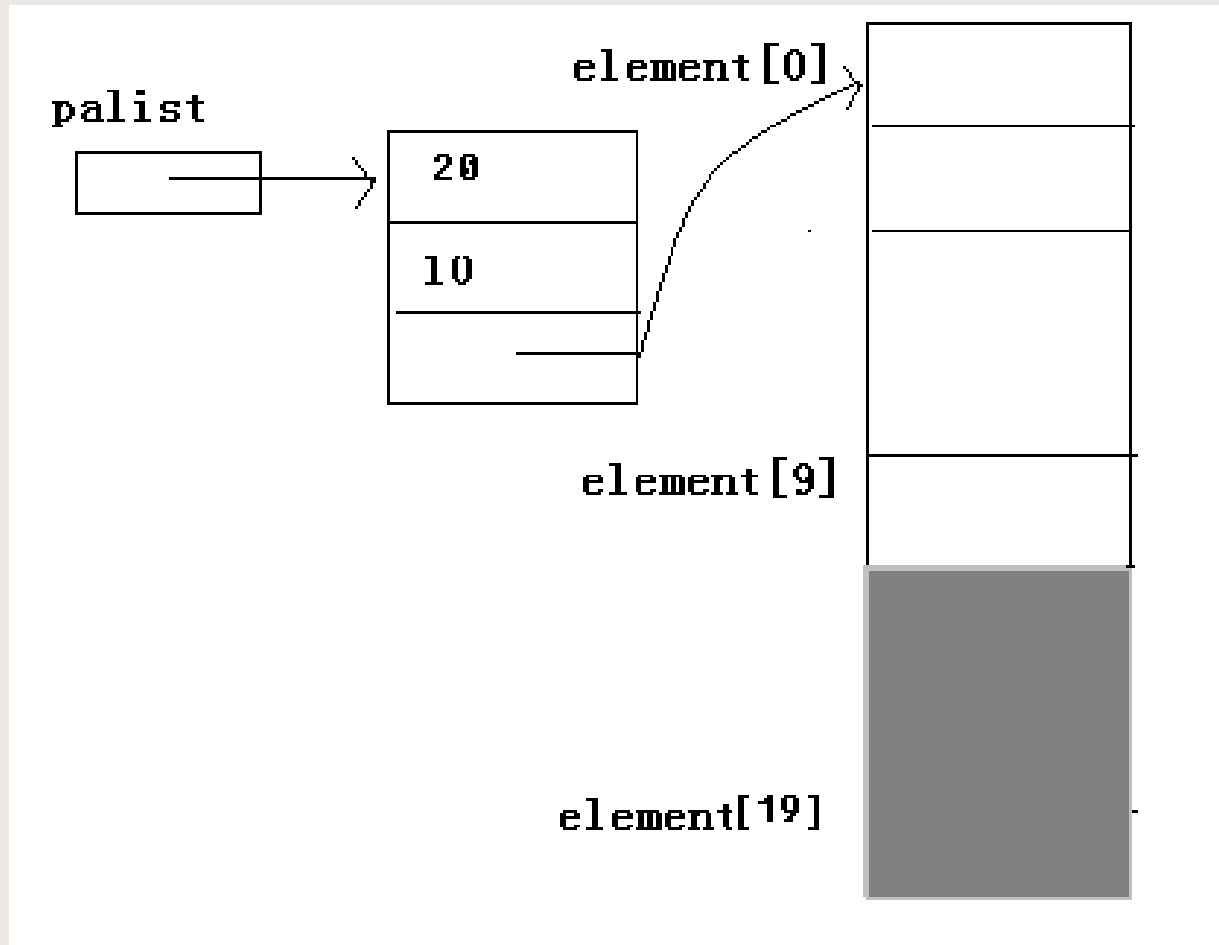
# 顺序表的C语言描述

```
struct SeqList{  
    int  MAXNUM;      /*顺序表中可以存放元素的个数*/  
    int   n;          /* 实际存放线性表中元素的个数, n≤MAXNUM */  
    DataType *element;  
    /* element[0], element[1], ..., element[n- 1]存放线性表中的元素 */  
};  
typedef struct SeqList  *PSeqList;
```

如果palist是一个PSeqList类型的指针变量, 则:

- palist->MAXNUM: 顺序表中可以存放元素的个数;
- palist->n: 顺序表中实际元素的个数;
- palist->element[0] , ..., palist->element[palist->n - 1]: 顺序表中的各个元素。

# 顺序表的存储示意图



# 顺序表基本操作

---

- 创建空顺序表
- 判断线性表是否为空
- 在顺序表中求某元素的下标
- 顺序表的插入
- 顺序表的删除

# 创建空顺序表

```
PSeqList createNullList_seq(int m ) {  
    /* 创建新的顺序表 */  
    PSeqList palist = (PSeqList)malloc(sizeof(struct SeqList));  
    if (palist!=NULL){  
        palist->element = (DataType*)malloc(sizeof(DataType)*m);  
        if (palist->element){  
            palist->MAXNUM=m;  
            palist ->n = 0;                /* 空表长度为0 */  
            return palist ;  
        }  
        else free palist;  
    }  
    printf("Out of space!!\n"); /* 存储分配失败 */  
    return NULL;  
}
```

# 判断线性表是否为空

```
int isNullList_seq( PSeqList palist ) {  
    /*判别palist所指顺序表是否为空表。*/  
    return ( palist->n == 0 );  
}
```

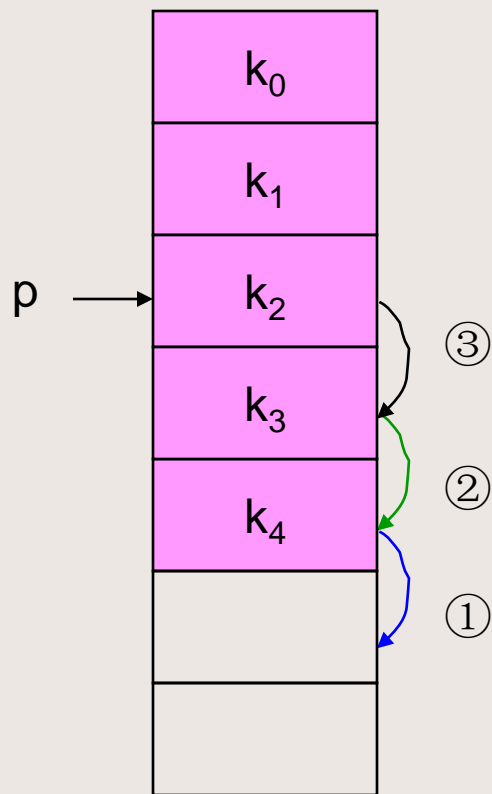
# 在顺序表中求某元素的下标

```
int locate_seq( PSeqList palist, DataType x ) {  
    /* 求x在palist所指顺序表中的下标 */  
    int q;  
    for ( q=0; q<palist->n ; q++)  
        if ( palist->element[q] == x) return q ;  
    return -1;  
}
```

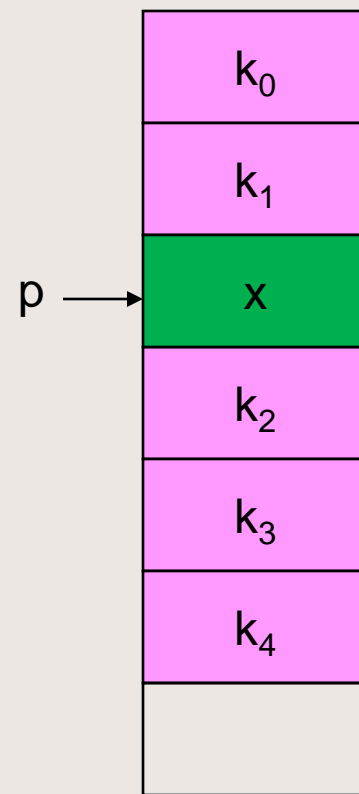
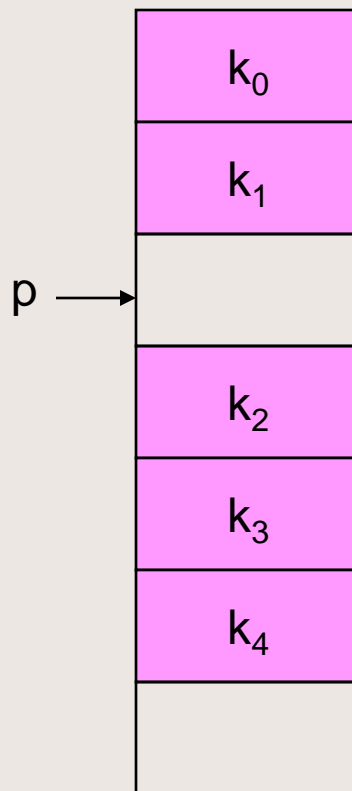


# 顺序表的插入

`int insertPre_seq(PSeqList palist, int p, DataType x)`



$n=5$



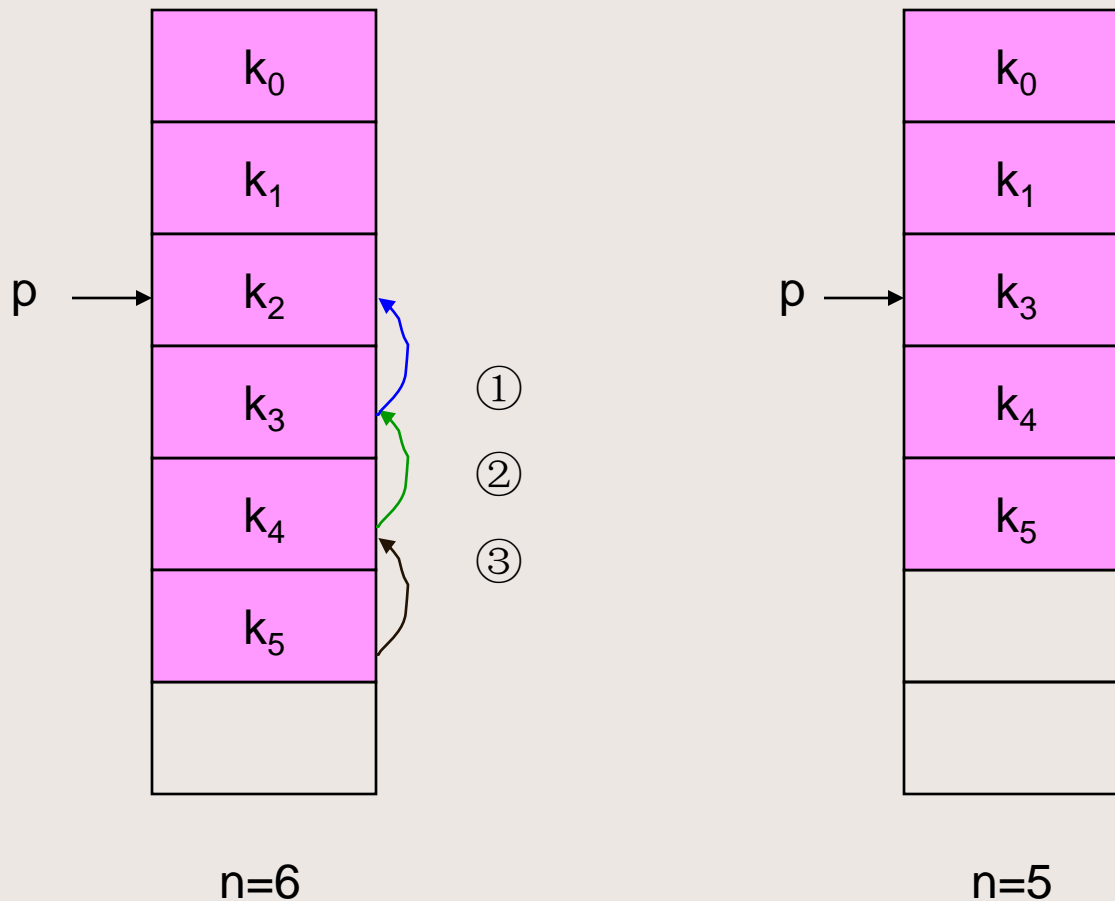
$n=6$

# 顺序表的插入

```
int insertPre_seq( PSeqList palist, int p, DataType x ) {  
    /* 在palist所指顺序表中下标为p的元素之前插入元素x */  
    int q;  
    if ( palist->n >= palist-> MAXNUM ) {                                /* 溢出 */  
        printf("Overflow!\n");  
        return 0 ;  
    }  
    if ( p<0 || p>palist->n ) {      /* 不存在下标为p的元素 */  
        printf("Not exist! \n"); return 0 ;  
    }  
    for(q=palist->n - 1; q>=p; q--) /* 插入位置及之后的元素均后移一个位置 */  
        palist->element[q+1] = palist->element[q];  
    palist->element[p] = x;          /* 插入元素x */  
    palist->n = palist->n + 1;       /* 元素个数加1 */  
    return 1 ;  
}
```

# 顺序表的删除

```
int deleteP_seq(PSeqList palist, int p)
```



# 顺序表中按下标删除元素

```
int deleteP_seq( PSeqList palist, int p ) {  
    /* 在palist所指顺序表中删除下标为 p 的元素 */  
    int q;  
    if ( p<0 || p>palist->n - 1 ) { /* 不存在下标为p的元素 */  
        printf("Not exist!\n ");  
        return 0 ;  
    }  
    for(q=p; q<palist->n-1; q++) /* 被删除元素之后的元素均前移一个位置 */  
        palist->element[q] = palist->element[q+1];  
    palist->n = palist->n - 1; /* 元素个数减1 */  
    return 1 ;  
}
```

# 顺序表中按值删除元素

```
int deleteV_seq( PSeqList palist, DataType x )
```

- 在palist所指顺序表中，删除一个值为x的元素，返回删除成功与否的标志。
- 实现的算法只要首先调用locate\_seq (palist, x )，在palist所指顺序表中寻找一个值为x的元素的下标，假设为p，然后调用deleteP\_seq(palist, p )即可。

# 算法分析与评价

- 在有n个元素的线性表里下标为i的元素前插入一个元素需要移动n-i个元素, 删除下标为i的元素需要移动n-i-1个元素。
- 若在下标为i的位置上插入和删除元素的概率分别是 $P_i$ 和 $P'_i$ , 则
  - 插入时平均移动元素数为:  $M_i = \sum_{i=0}^n (n-i) P_i$
  - 删除时平均移动元素数为:  $M_d = \sum_{i=0}^{n-1} (n-i-1) P'_i$
- 考虑在不同的下标位置上插入和删除元素的概率相等。  $P_i = 1/n$ ;  $P'_i = 1/(n+1)$ 。

# 算法分析与评价（续）

- 顺序表插入和删除操作的平均时间代价和最坏时间代价都是  $O(n)$  ；
  - 两种特殊情况：表后端插入/删除的时间代价为  $O(1)$  ；
- 根据元素值的定位操作(`locate_seq`)，需顺序与表中元素比较，当定位的概率平均分布在表的所有元素上时，一次定位平均需要和  $n/2$  个元素进行比较，时间代价为  $O(n)$  ；
  - 特殊情况：如果顺序表中的元素按照值的升(降)序排列，则可使用二分法使得定位操作的时间代价减少到  $O(\log_2 n)$  。

## 2.3 链接表示

- 链接表示: 实现线性表的另一种存储结构
  - 不连续的存储单元
    - 不要求逻辑关系相邻的两个元素在物理位置上也相邻存储
  - 附加信息
    - 通过增加指针来指示元素之间的逻辑关系和后继元素的位置





# 单链表表示

- 最简单的链表表示：只为每个数据元素关联一个链接，表示后继关系。
- 每个结点包括两个域：
  - 数据域：存放元素本身信息。
  - 指针域：存放其后继结点的存储位置。
    - 最后一个元素的指针不指向任何结点，称为空指针，在图示中用“^”表示，在算法中用“NULL”表示。
    - 指向链表中第一个结点的指针，称为这个链表的头指针。

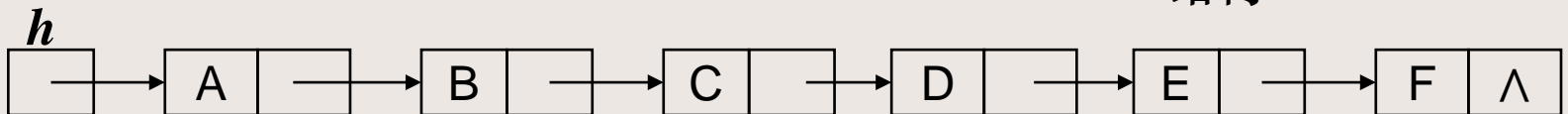
# 单链表的内部存储示意图

存储地址	数据域	指针域
100	D	131
107	B	113
113	C	100
119	F	NULL
125	A	107
131	E	119

头指针  $h$

125

实际中不需要关心结点的具体存储地址, 只关心表的逻辑结构



# 单链表的C语言描述

```
struct Node;                /*单链表结点类型*/
typedef struct Node * PNode; /*结点指针类型*/
struct Node {               /*单链表结点结构*/
    DataType info;
    PNode link;
};

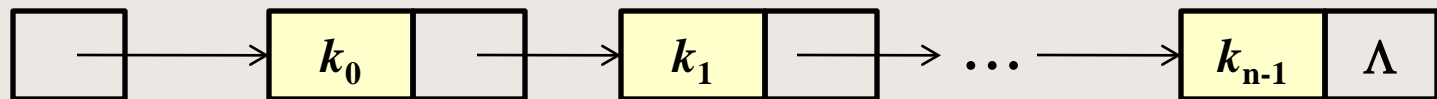
typedef struct Node * LinkList ; /*单链表类型*/

LinkList llist;                /*llist是一个链表的头指针*/
```

# 链表的头指针与头结点

假设llist是某单链表的头指针,类型是LinkedList, 存储结构如下所示,当单链表为空表时,llist的值为NULL。

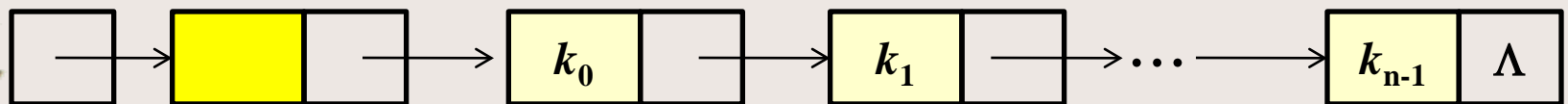
llist



- 为方便运算实现,可以在单链表的第一个结点之前另加一个**头结点**:
  - 辅助结点,不属于表的内容;
  - 可以不存信息,也可以保存与整个表相关的信息,如元素个数;
  - Link域指向表的第一个实际结点。

llist

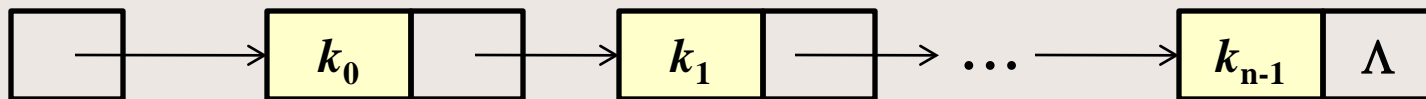
头结点



# 头结点的作用

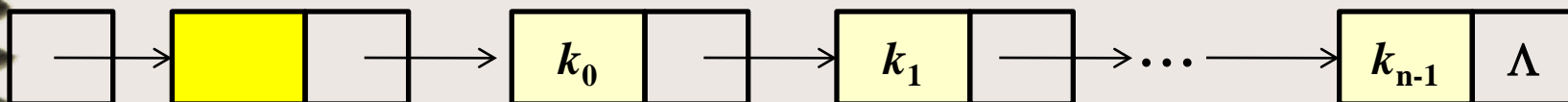
- 如果没有，在表头进行插入/删除时需要修改链表头指针，在其它位置插入/删除时修改其前驱结点的link，两种操作需要分别处理。
- 引进头结点使得这两种操作的实现可以统一处理，因为表中每个元素都链接在另一个结点的link域。
- 重点关注带头结点的单链表的操作实现算法。

llist



llist

头结点



# 创建空单链表

/\*创建一个带头结点的空单链表\*/

**LinkedList** createNullList\_link(void) {

LinkedList llist;

/\*申请表头结点空间\*/

llist=(LinkedList)malloc(sizeof(struct Node ));

if (llist!=NULL) llist->link=NULL;

else printf("Out of space!\n"); /\*创建失败\*/

return llist;

}

操作正常完成后的状态:

llist



# 判断单链表是否为空

```
int isNullList_link(LinkList llist) {  
    return (llist->link==NULL);  
}
```

/\*因为llist指向头结点,总是非空,所以算法中只要检查llist->link是否为空即可\*/

# 在单链表中求某元素存储位置

/\*在带头结点的单链表llist中找第一个值为x的结点存储位置\*/

**PNode locate\_link(LinkList llist, DataType x) {**

PNode p;

if(llist==NULL) return NULL; /\*找不到时返回空指针\*/

p=llist->link;

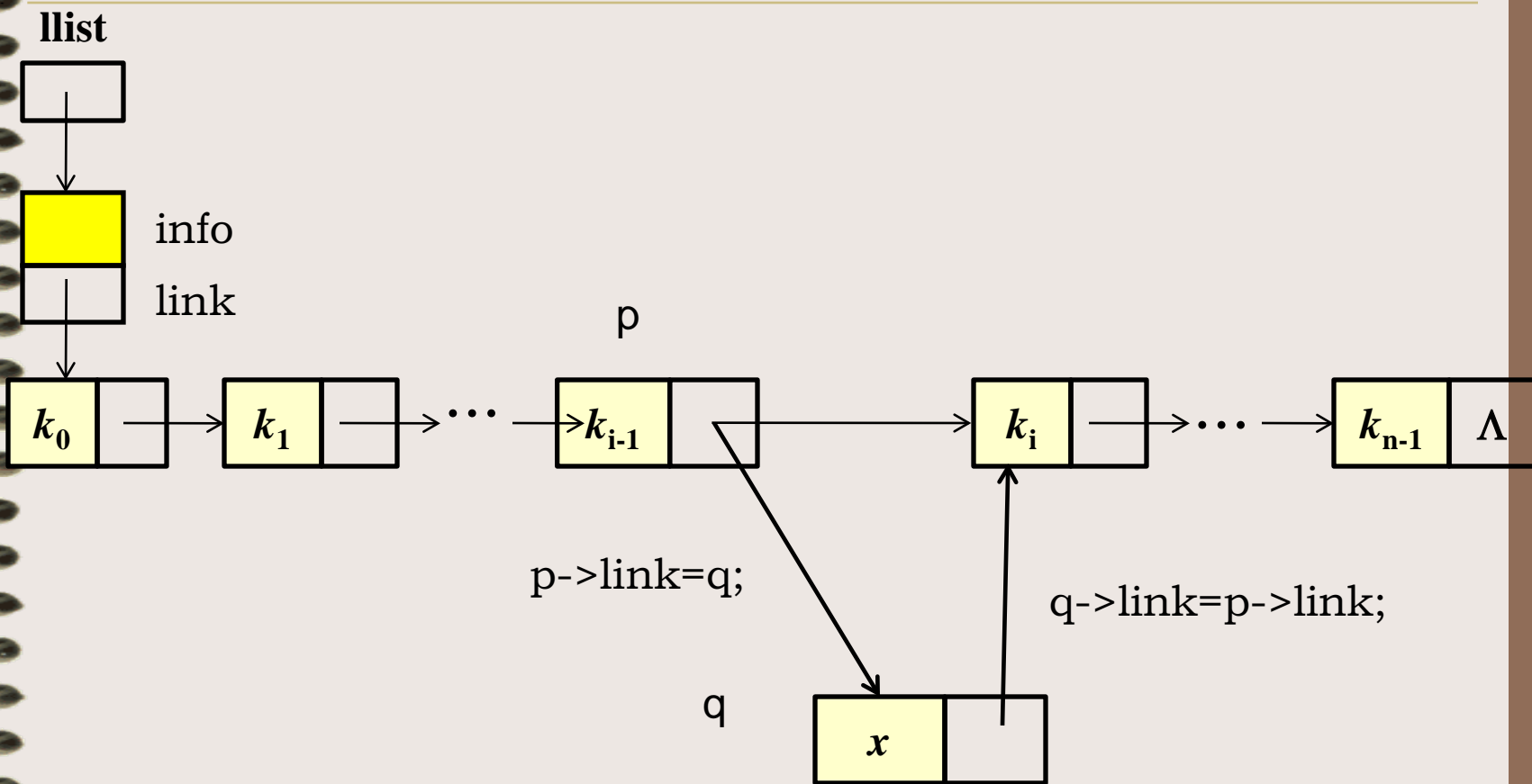
while (p!=NULL&& p->info!=x)p=p->link;

return p;

**}**



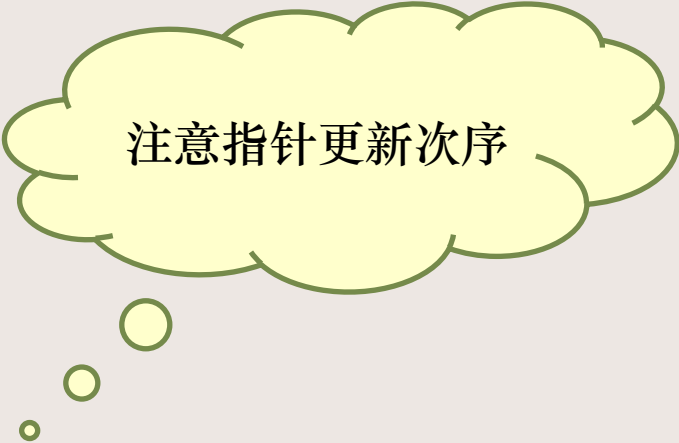
# 单链表的插入



插入算法示意

## 在带头结点的单链表llist中 下标为p的结点后插入值为x的新结点

```
int insertPost_link (LinkList llist, PNode p, DataType x) {  
    PNode q=(PNode)malloc(sizeof(Struct Node));    /*申请新结点*/  
    if (q==NULL) {  
        printf("Out of Space!\n");  
        return 0;  
    }  
    else {  
        q->info =x;  
        q->link=p->link;  
        p->link=q;  
        return 1;  
    }  
}
```

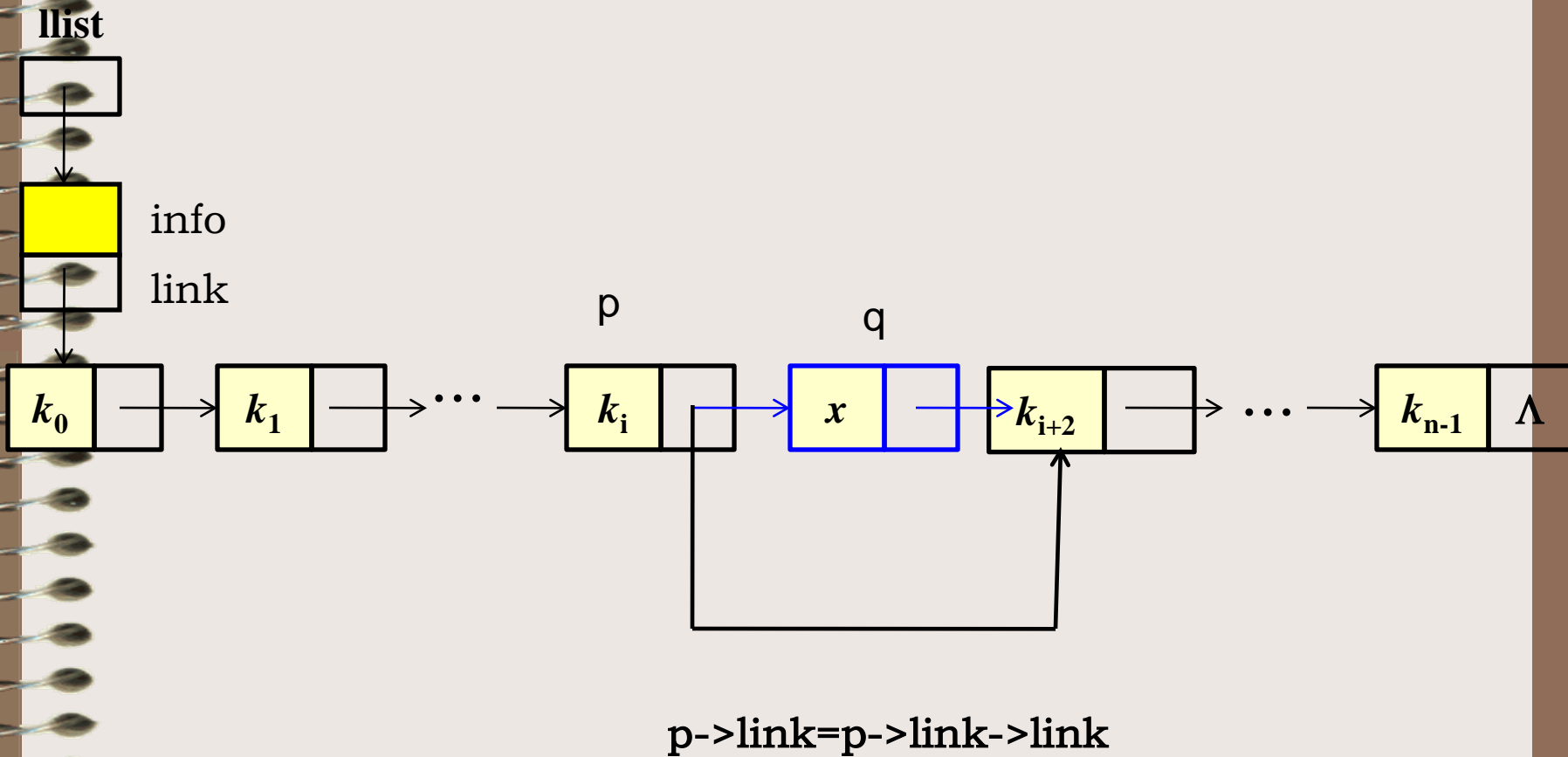


注意指针更新次序

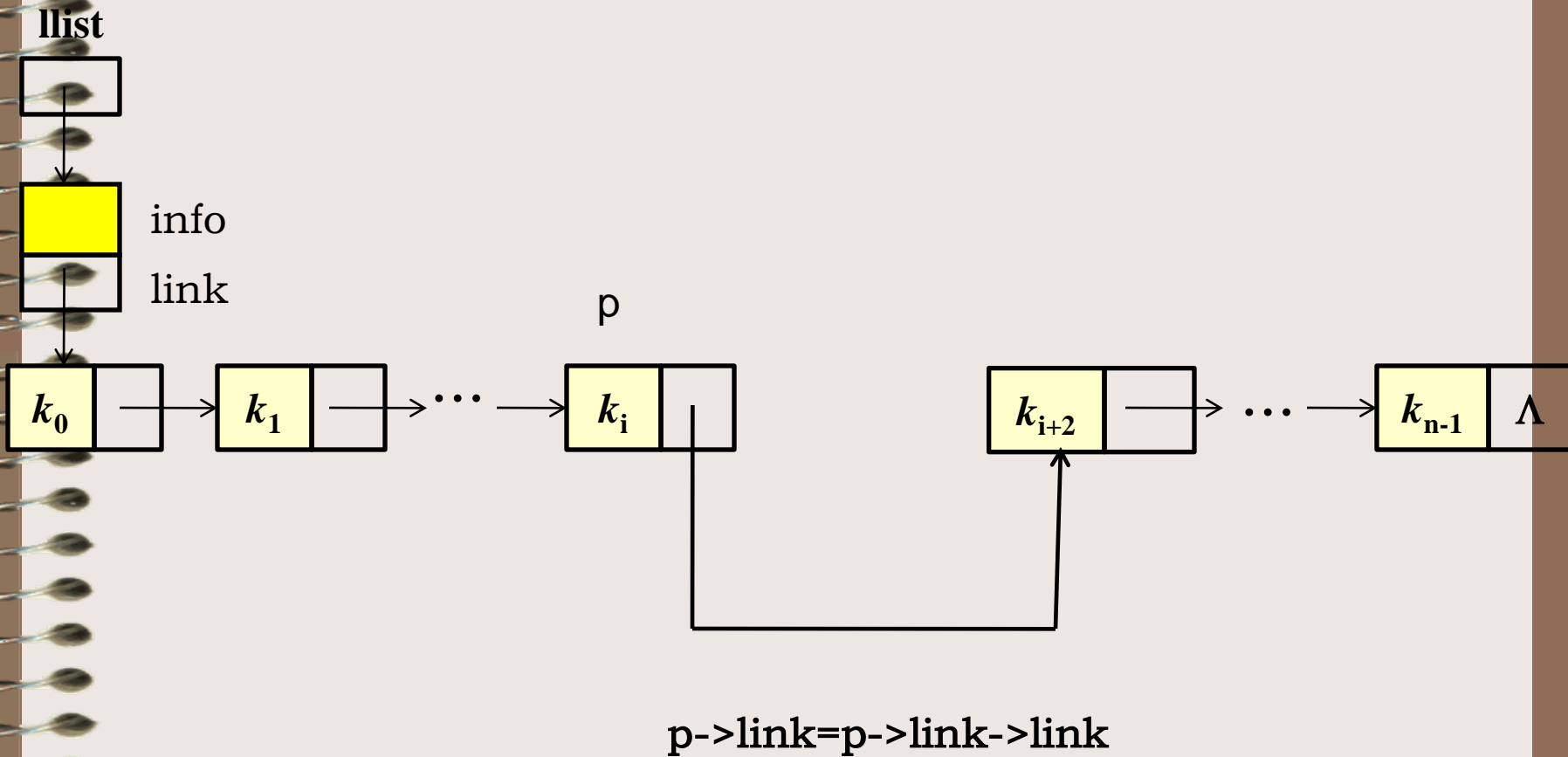
# 在单链表中求p所指结点的前驱

```
PNode locatePre_link (LinkList llist, PNode p) {  
    PNode p1;  
    if(llist==NULL) return NULL;  
    p1=llist;  
    while(p1!=NULL && p1->link!=p) p1=p1->link;  
    return p1;  
}
```

# 单链表的删除



# 单链表的删除



在带头结点的单链表llist中,删除第一个元素值为x的结点（这里要求**DataType** 可以用**!=**比较）

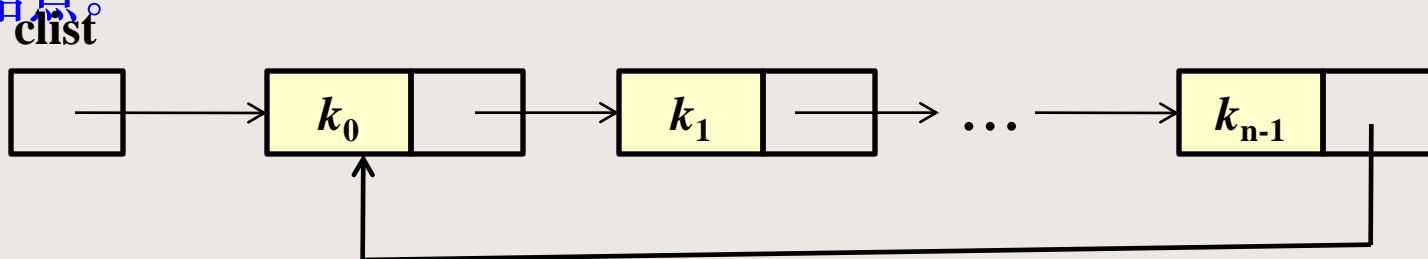
```
int deleteV_link(LinkList llist, DataType x) {
    PNode p, q;  p = llist;
    if(p==NULL) return 0 ;
    while( p->link != NULL && p->link->info != x )
        p = p->link; /*找值为x的结点的前驱结点的存储位置 */
    if( p->link == NULL ) { /* 没找到值为x的结点 */
        printf("Not exist!\n"); return 0 ;
    }
    else {
        q = p->link; /* 找到值为x的结点 */
        p->link = q->link; /* 删除该结点 */
        free( q ); return 1 ;
    }
}
```

# 分析与比较

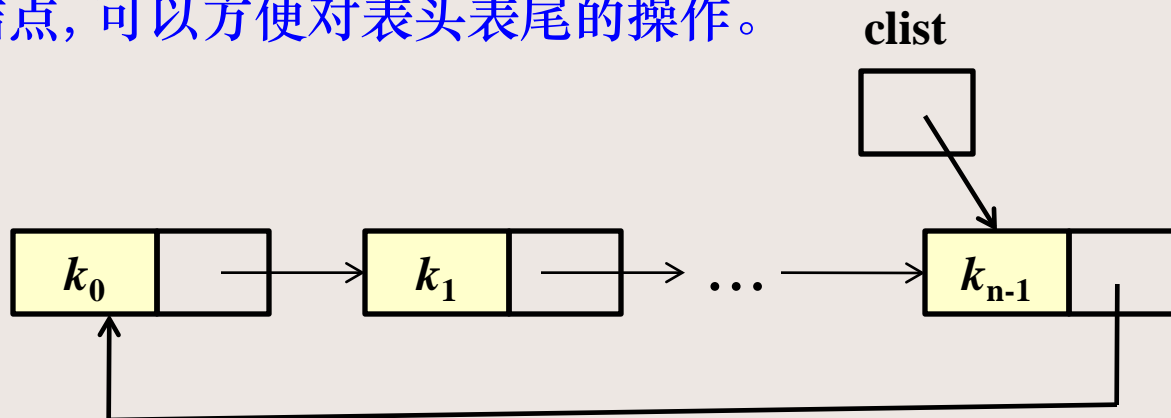
- 在有 $n$ 个结点的单链表中查找第 $i$ 个结点时，必须从链表的第一个结点开始往下查找，所需的时间与 $i$ 的大小成正比，最坏情况下要查找 $n$ 个结点，平均情况下需要查找 $n/2$ 个结点，因此时间复杂度为 $O(n)$ 。
- 在单链表中找第一个值为 $x$ 的结点存储位置；在单链表中， $p$ 所指结点前面插入值为 $x$ 的新结点；以及在单链表中删除第一个值为 $x$ 的结点 等算法的时间复杂度均为 $O(n)$ 。

# 循环链表(I)

- 将单链表的最后一个结点的指针不设置为空, 而是指向头一个结点, 这样就得到循环链表。
- 循环链表未增加新的存储空间, 但从任一结点出发都能访问所有结点。



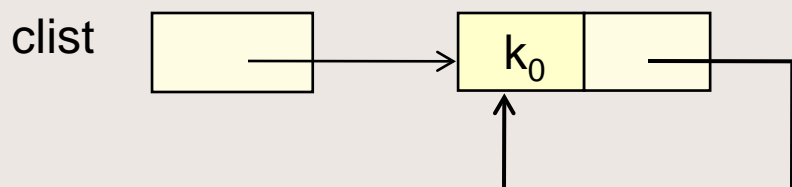
- 操作经常要访问、修改首结点和末结点, 让表头指针指向循环链表末结点, 可以方便对表头表尾的操作。



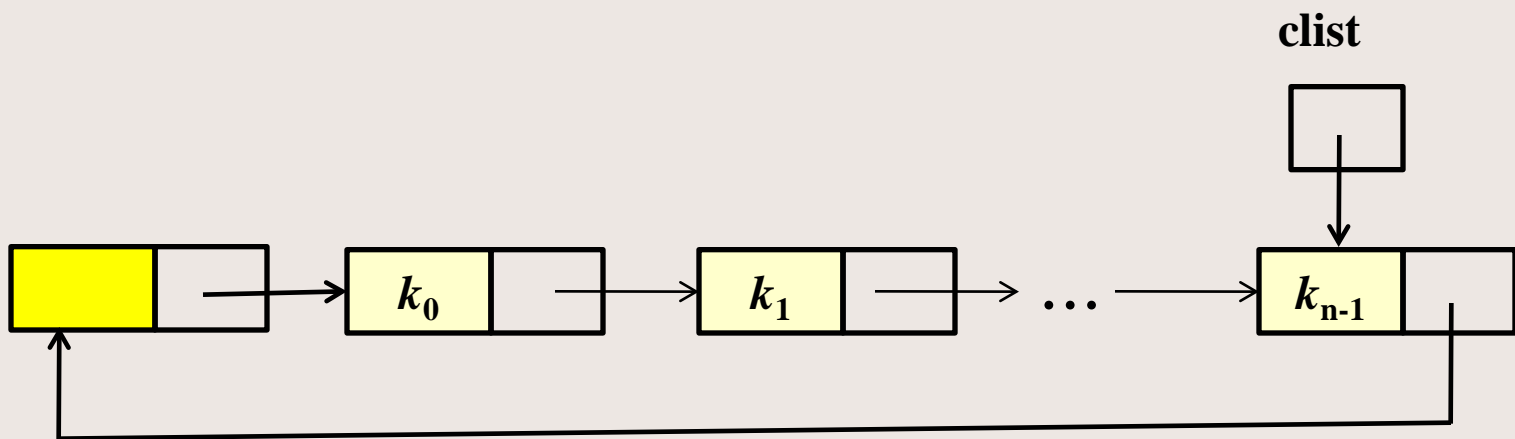


# 循环链表(II)

- 循环链表插入第一个元素时需要特别处理:



- 考虑带头结点的循环链表:



# 实现循环链表

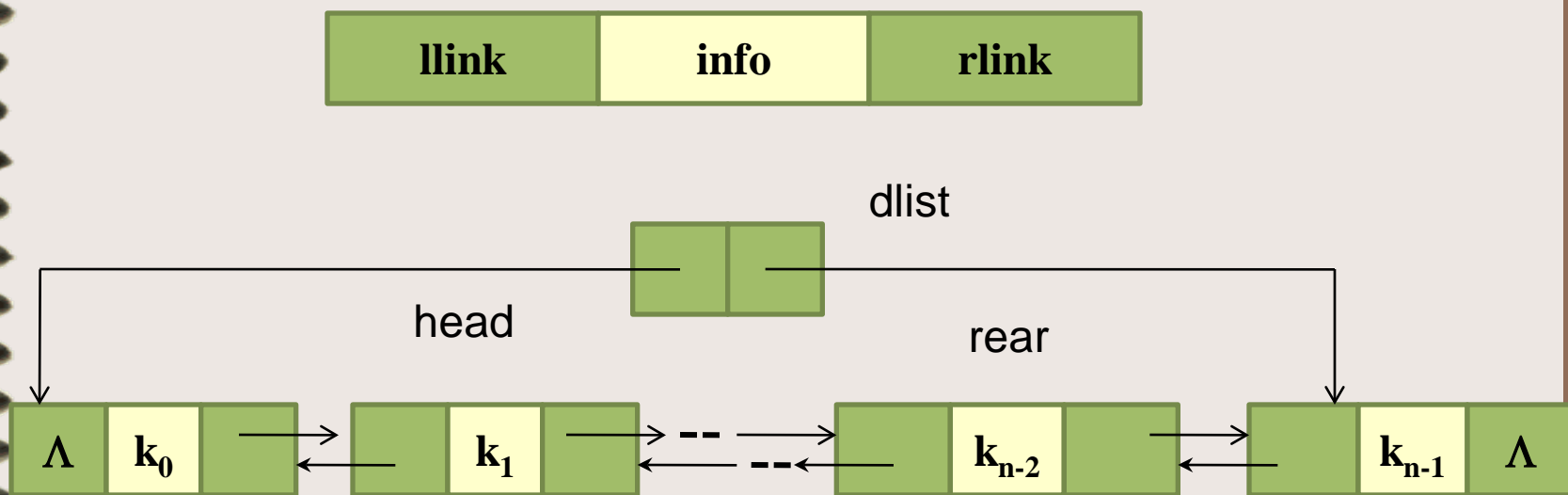
- 可以采用与单链表一样的数据类型定义。
- 创建链表的操作需要修改。如有头结点的链表：

```
llist = (LinkedList) malloc( sizeof( Node ) );  
if ( llist != NULL ) llist->link = llist;  
return llist;
```
- 无头结点的表插入第一个结点时，也需建立循环链接。
- 判断空表的操作（有头结点的表）：

```
return llist->link = llist;
```
- 其他操作基本上可以照搬单链表的操作。根据所采用的设计，可能需要做些小调整。

# 双链表

- 单链表找指定结点的前驱结点比较困难。
- 可以设计双链表来克服单链表的这个缺点：
  - 每个结点都设后继指针和前驱指针，既可以找到后继也可以直接找到前驱；
  - 双链表结点结构示意与带有头尾指针的双链表。



# 双链表的实现

- 数据类型定义

```
struct DoubleNode;      /*双链表结点*/
typedef struct DoubleNode * PDoubleNode; /*结点指针类型*/
struct DoubleNode{      /*双链表结点结构*/
    DataType info;
    PDoubleNode llink, rlink;
};
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*/
};
```

# 双链表的删除操作

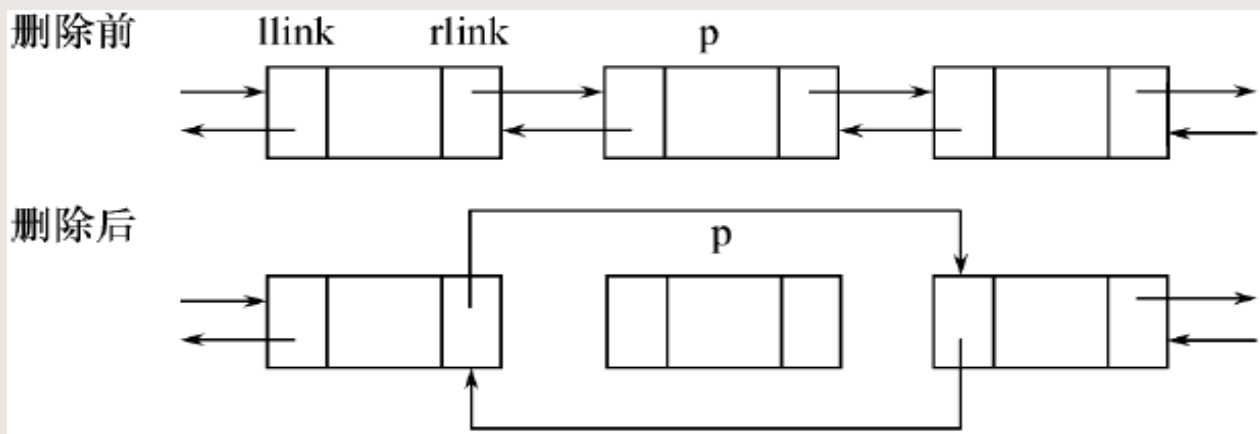
- 如果要删除指针变量p所指的结点,只需要修改该结点的前驱的rlink和后继的llink:

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

```
    p->llink->rlink=p->rlink;
```

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

```
    p->rlink->llink=p->llink;
```



# 双链表的插入操作

- 在p所指结点后插入一个新结点

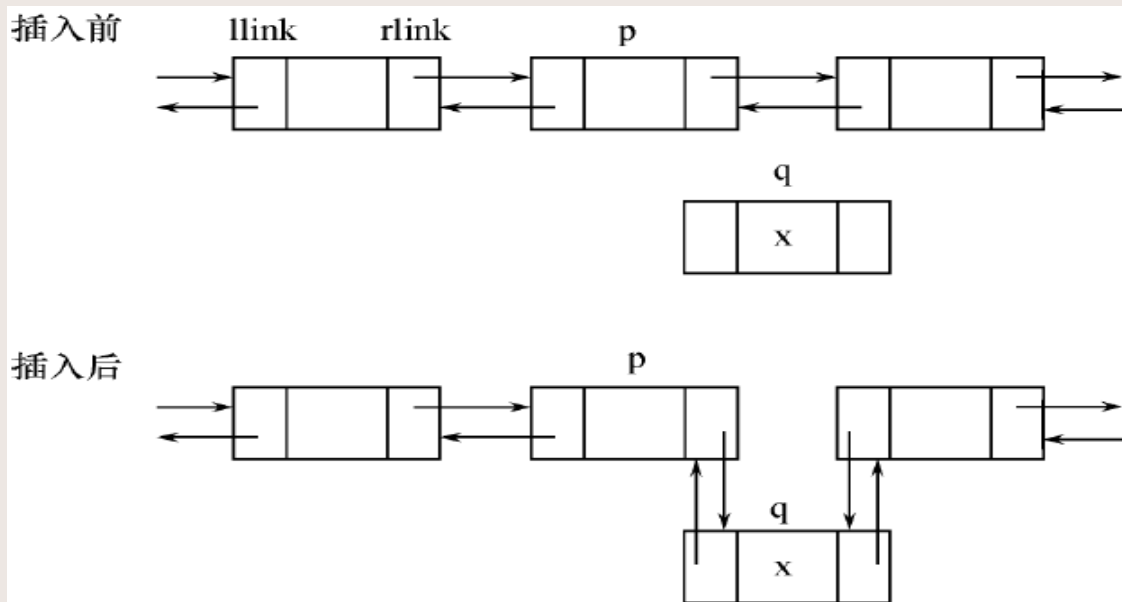
首先申请一个新结点  $q = (\text{PDoubleNode})\text{malloc}(\text{sizeof}(\text{struct DoubleNode}))$ ;  
修改p所指结点的rlink和原后继结点的llink:

$q \rightarrow \text{llink} = p$ ;

$q \rightarrow \text{rlink} = p \rightarrow \text{rlink}$ ;

$p \rightarrow \text{rlink} \rightarrow \text{llink} = q$ ;

$p \rightarrow \text{rlink} = q$ ;



# 小结

- 顺序存储结构的优点：
  - 元素之间的逻辑关系是通过存储位置直接反映的，顺序存储结构中只需存放数据元素自身的信息，因此，存储密度大、空间利用率高；
  - 元素的位置可以用元素的下标通过简单的解析式计算出来，因此可以随机存取。
- 顺序存储结构的缺点：
  - 元素的插入和删除运算可能需要移动许多其它元素的位置，
  - 一些长度变化较大的线性表必须按最大需要的空间分配存储。

## 小结（续）

- 链接存储结构的优点：
  - 不要预先按最大的需要分配连续空间；
  - 线性表的插入和删除只需修改指针域，而不需移动其他元素。
- 链接存储结构的缺点：
  - 每个结点中的指针域需额外占用存储空间，存储密度小；
  - 链式存储结构是一种非随机存储结构，查找任一结点都要从头指针开始，沿指针域一个一个地搜索，这增加了有些算法的时间代价。



# 讨论

- 时间代价和空间代价始终是数据结构与算法设计的最主要因素，然而它们往往是相互对立的。
- 一个好的设计总是在时间代价和空间代价之间作出了一个好的权衡，而这种权衡的标准需要根据实际计算机的资源情况和求解问题的特点来确定。
- 顺序表和单链表是两种最简单的数据结构，但是它们的应用非常广泛。