数据结构

第二讲 线性表

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年9月18日

课程信息

- 课程主页：
    - http://www.math.pku.edu.cn/teachers/sunm/ds2017/
    - 作业通过course.pku.edu.cn提交

课程内容

- 线性表的概念和抽象数据类型
- 顺序表示
- 链接表示

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。

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

- 线性表的逻辑结构：$L =< K, R >$

  其中

  $K = \{k_0, k_1, \ldots, k_{n-1}\},\ R = \{< k_i, k_{i+1} > \mid\ 0 \leq i \leq n-2\}$

  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；
- $k_i\ (0 \le i \le n-2)$ 是 k_{i+1} 的前驱；
- k_{i+1}是k_i的后继。

数据结构的操作

- 作为一种包含元素（可以没有，也可以有许多）的数据结构，通常都需要提供一些“标准”操作，例如：
  - 创建和销毁这种数据结构（的实例）；
  - 判断一个数据结构是否空（没有元素），如果数据结构的容量有限制，还需判断它是否满（不能再加入新元素）；
  - 向结构中加入元素或从中删除元素；
  - 访问结构里的元素。

数据结构的操作

- 从作用看，数据结构的操作可以分为三类：
  - 构造操作：构造出该数据结构的一个新实例
  - 访问操作：从已有数据结构中提取某些信息，但不创建新结构，也不修改被操作的结构
  - 变动操作：修改已有的数据结构

符号说明

- 假设List表示一个线性表类型，DataType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。*/
  int insertPost ( List list, position p, DataType x )  /*在list中p位置后插入值为x的元素。*/
  int deleteV (List list, DataType x )                  /*在list中删除一个值为x的元素。*/
  int deleteP (List list, position p )                  /*在list中删除位置为p的元素。*/
  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;
```

线性表数据结构的实现

- 线性表的实现主要考虑两方面的情况
  - 计算机内存的特点，以及保存元素和元素顺序信息的需要
  - 重要操作的效率。其中使用最频繁的操作通常是：（定位）元素访问，元素加入，元素删除，元素遍历
- 元素遍历就是依次访问表里的所有（或一批）元素
  - 操作效率与访问元素的个数有关
  - 遍历所有元素的操作，希望其复杂性不超过 $O(n)$
- 加入/删除/访问元素的操作效率与表的实现结构有关
- 基于各方面考虑，人们提出了两种基本实现模型
  - 将表元素顺序存放在的一大块连续的存储区里，这样实现的表也称为顺序表（或连续表），元素顺序有自然的表示
  - 将表元素存放在通过链接构造起来的一系列存储块里（链接表）

顺序表示

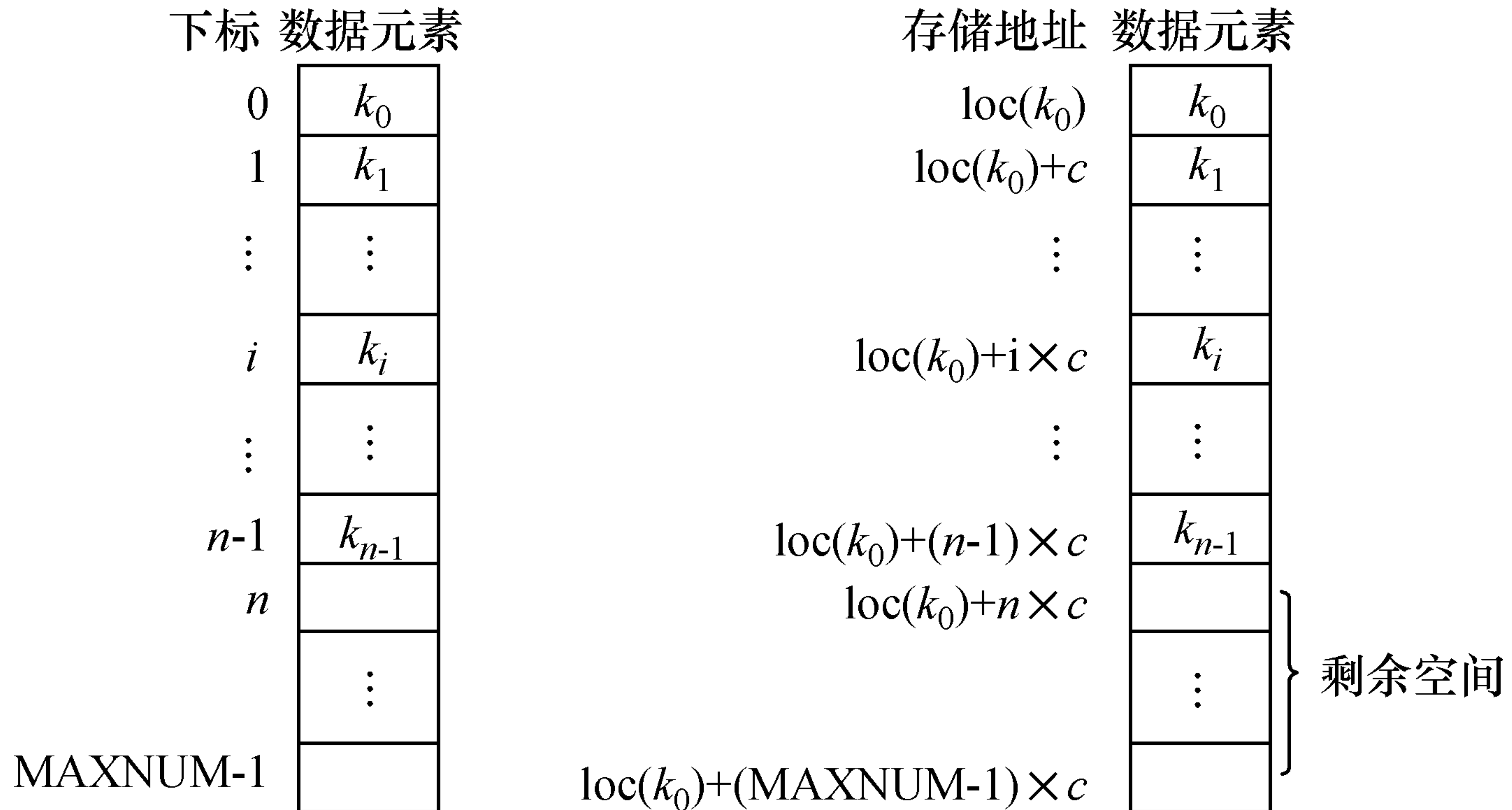
- 采用顺序存储是表示线性表最简单的方法
  - 将线性表中的元素依次存储在一片相邻的存储区域中
  - 元素之间逻辑上的相邻关系通过元素在计算机内物理位置上的相邻关系来表示
    - 逻辑上相邻⇔存储位置相邻
  - 顺序表示的线性表也称顺序表

存储结构

- 一般情况下表元素所需存储量相同，因此顺序表中任一元素的位置都可简单计算出来，存取操作可以在 $O(1)$ 时间内完成
- 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:

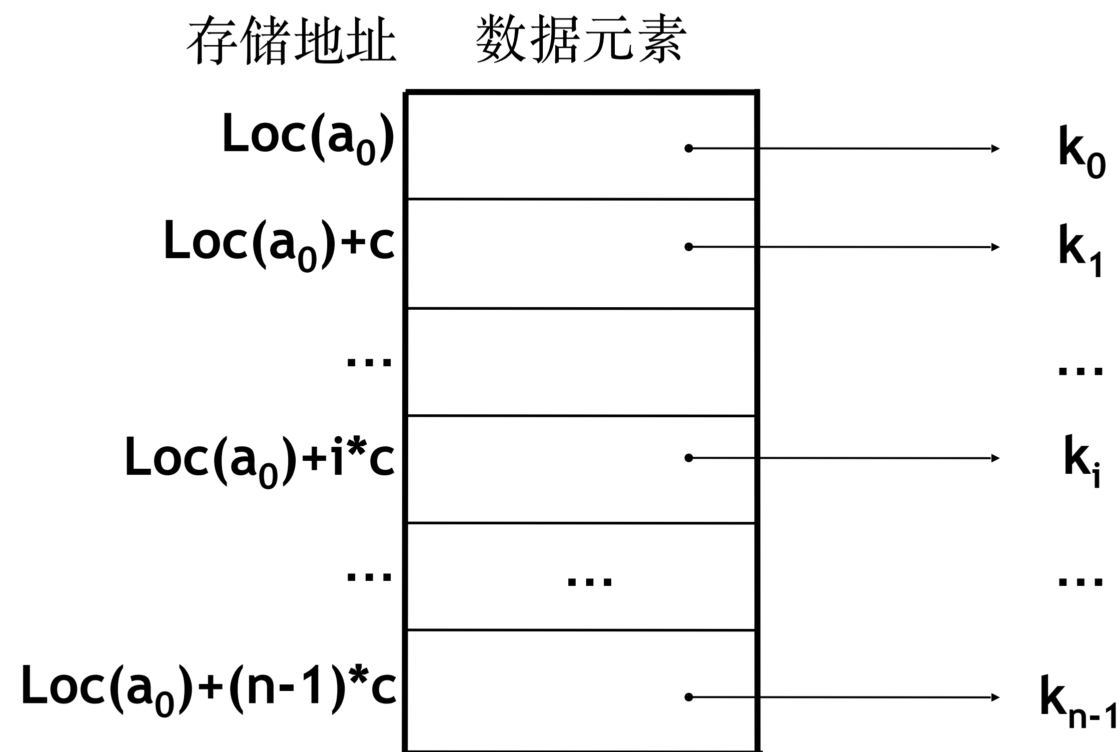
$$loc(k_i) = loc(k_0) + i \times c$$

存储结构示意图



存储结构

- 如果表中要保存的元素的情况复杂，大小不一，或者还有复杂的内部结构，可以采用链接方式，在表中保存元素链接（链接的大小相同）



c 是链接的大小。元素另外表示，为另外的简单元素或复杂结构

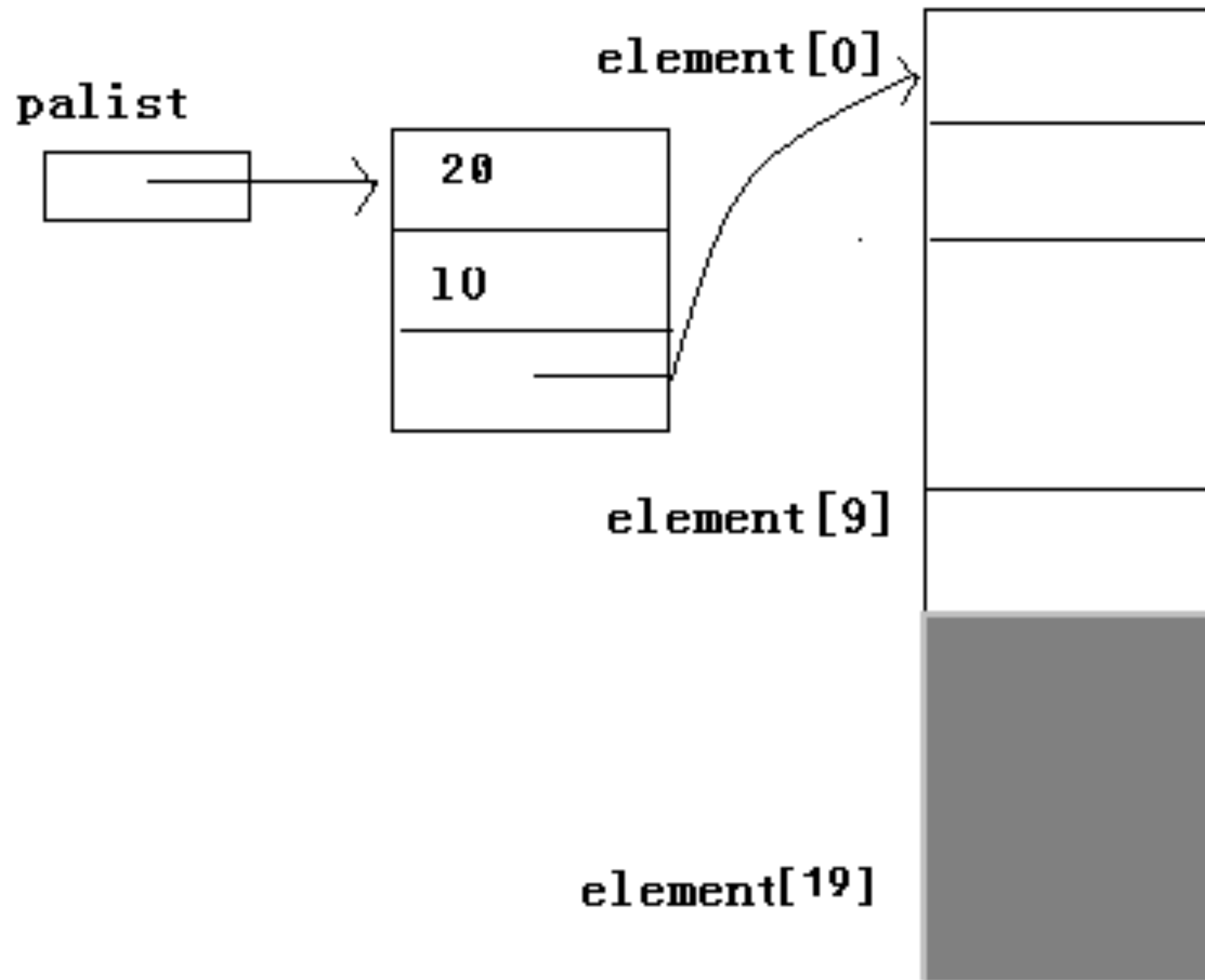
顺序表的C语言描述

```
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[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]：顺序表中各个元素。

存储结构示意图



顺序表的基本操作

- 创建空顺序表
- 判断表是否为空
- 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;
}
```

判断顺序表是否为空

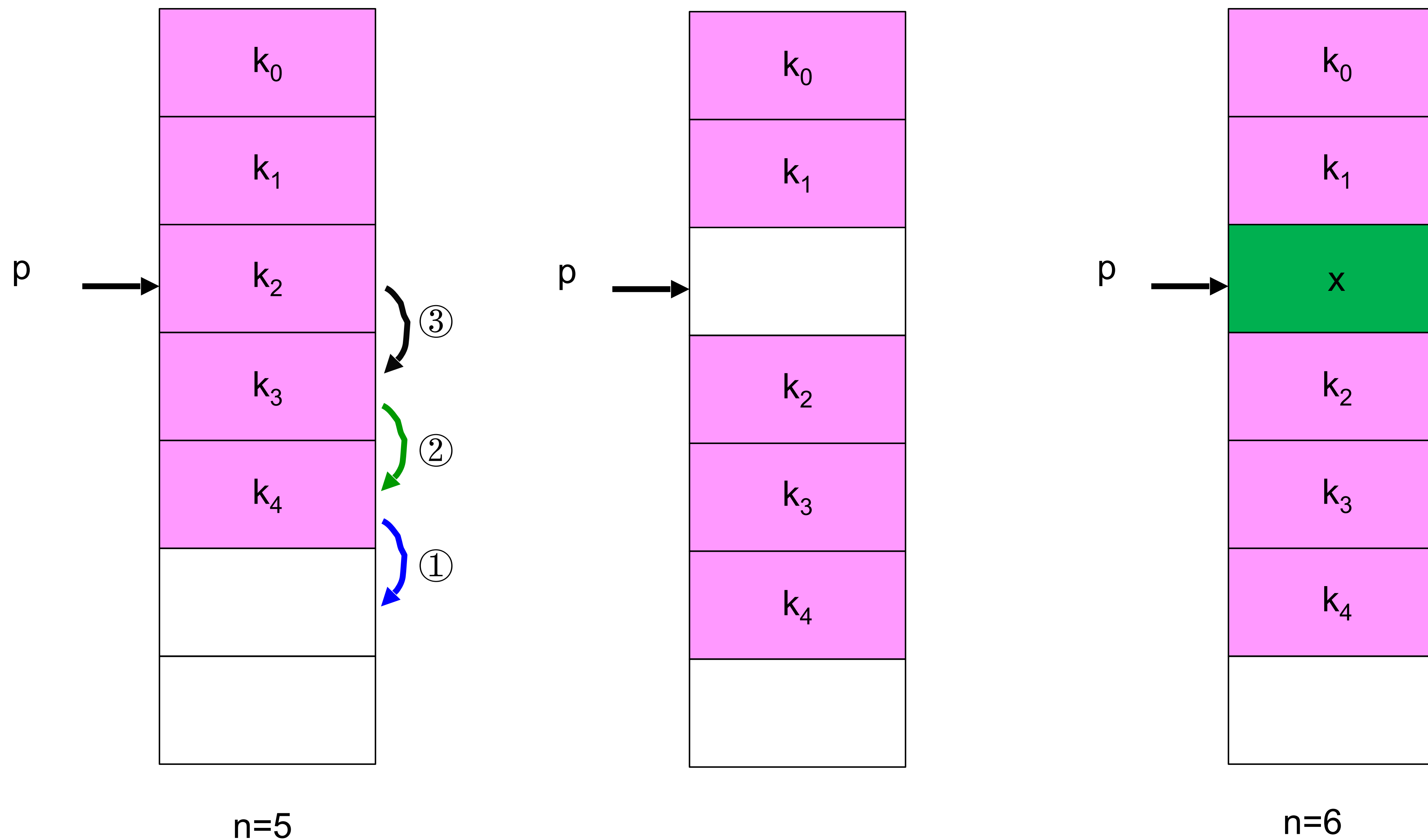
```
int  isNullList_seq(PSeqList palist) {
/*判别palist所指顺序表是否为空表。*/
   return ( palist->n == 0 );
}
```

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)

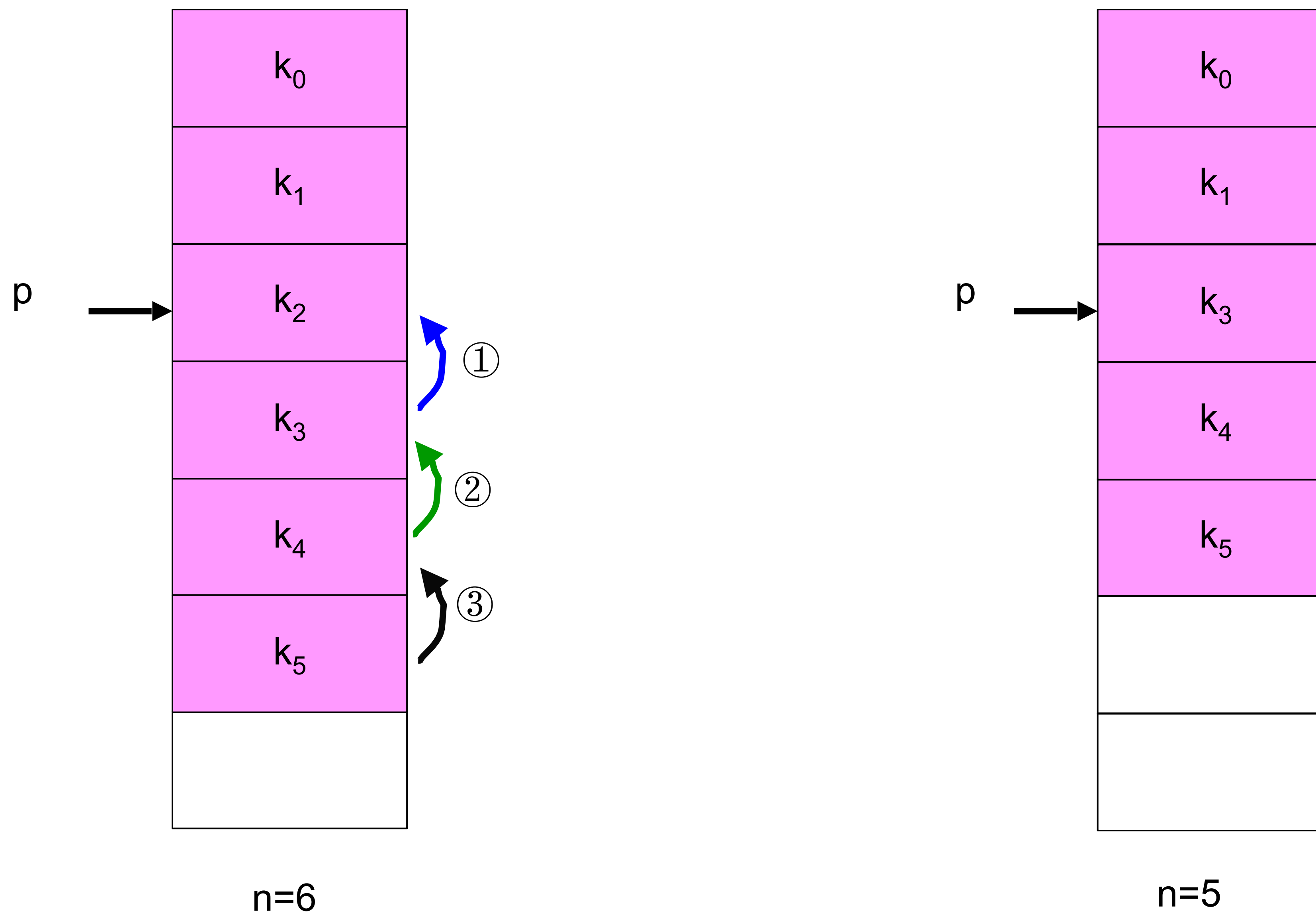


顺序表的插入

```
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 $n\text{-}i$ 个元素，**删除**下标为 i 的元素需要移动 $n\text{-}i\text{-}1$ 个元素。
- 若在下标为 i 的位置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。

$$P_i = \frac{1}{n}; \quad P'_i = \frac{1}{n+1}$$

算法分析与评价

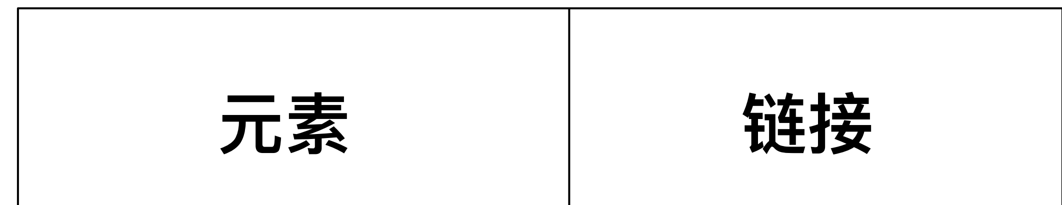
- 顺序表插入和删除操作的平均时间代价和最坏时间代价都是 $O(n)$。
  - 两种特殊情况：表后端插入/删除的时间代价为$O(1)$ 。
- 根据元素值的定位操作(locate_seq)，需顺序与表中元素比较，当定位的概率平均分布在表的所有元素上时，一次定位平均需要和n/2个元素进行比较,时间代价为$O(n)$ 。
  - 特殊情况：如果顺序表中的元素按照值的升（降）序排列，则可使用二分法使得定位操作的时间代价减少到$O(log_2n)$ 。

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
    - 通过增加指针来显式地指示元素之间的逻辑关系和后继元素的位置
- 基于链接技术实现的线性表称为链接表或链表

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。



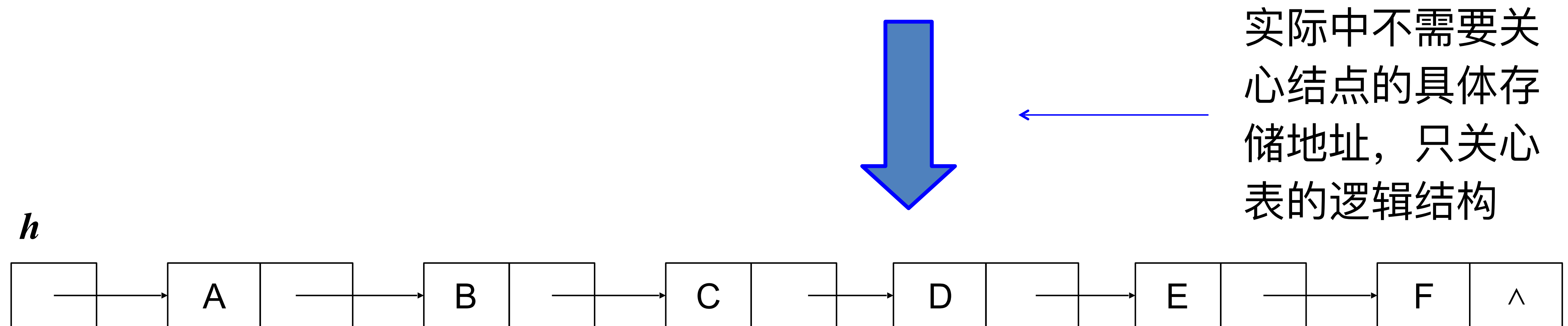
- 在单链表里，与表里的 n 个元素对应的 n 个结点通过指针形成一条结点链。从表里的任一个结点都可以找到保存下一个元素的结点

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

表头变量*h*

125

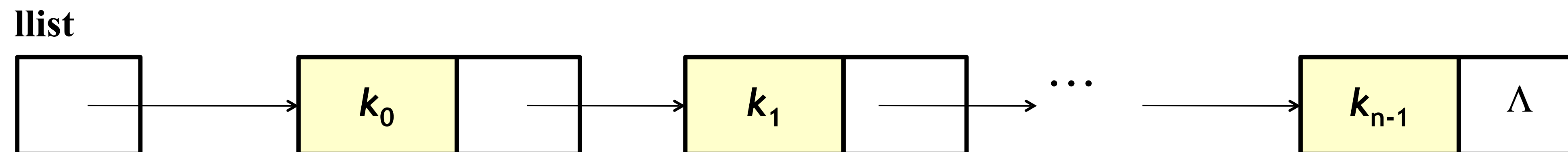


单链表的C语言描述

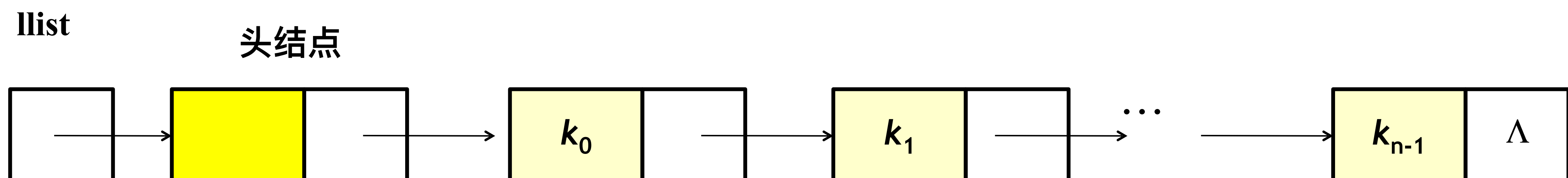
```
struct Node;                          /*单链表结点类型*/
typedef struct Node * PNode;    /*结点指针类型*/
struct Node {                         /*单链表结点结构*/
    DataType info;
    PNode link;
};

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LinkList，存储结构如下所示，当单链表为空表时，llist的值为NULL。

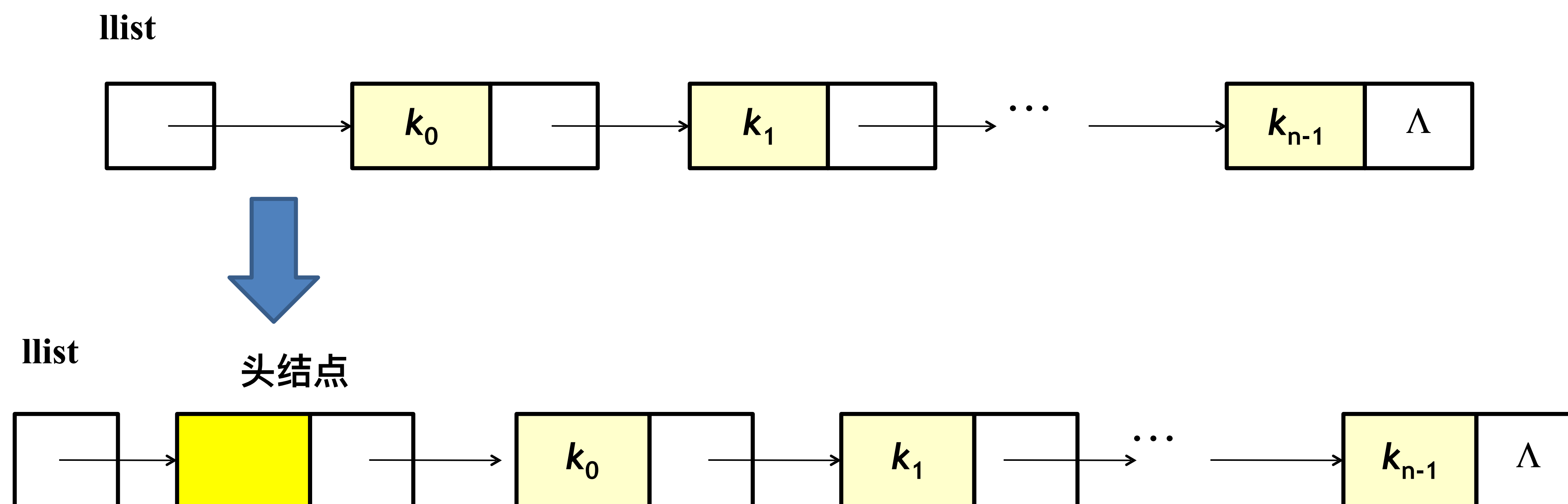


- 为方便运算实现，可以在单链表的第一个结点之前另加一个**头结点**。
  - 辅助结点，不属于表的内容
  - 可以不存信息，也可以保存与整个表相关的信息，如元素个数
  - Link域指向表的第一个实际结点



头结点的作用

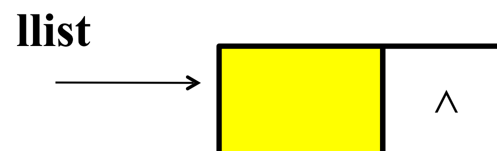
- 如果没有,在表头进行插入/删除时需要修改链表头指针， 在其它位置插入/删除时修改其前驱结点的link, 两种操作需要分别处理
- 引进头结点使得这两种操作的实现可以统一处理, 因为表中每个元素都链接在另一个结点的link域
- 重点关注带头结点的单链表的操作实现算法



创建空单链表

```
/*创建一个带头结点的空单链表*/
LinkList  createNullList_link(void) {
    LinkList llist;
   /*申请表头结点空间*/
    llist=(LinkList)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:



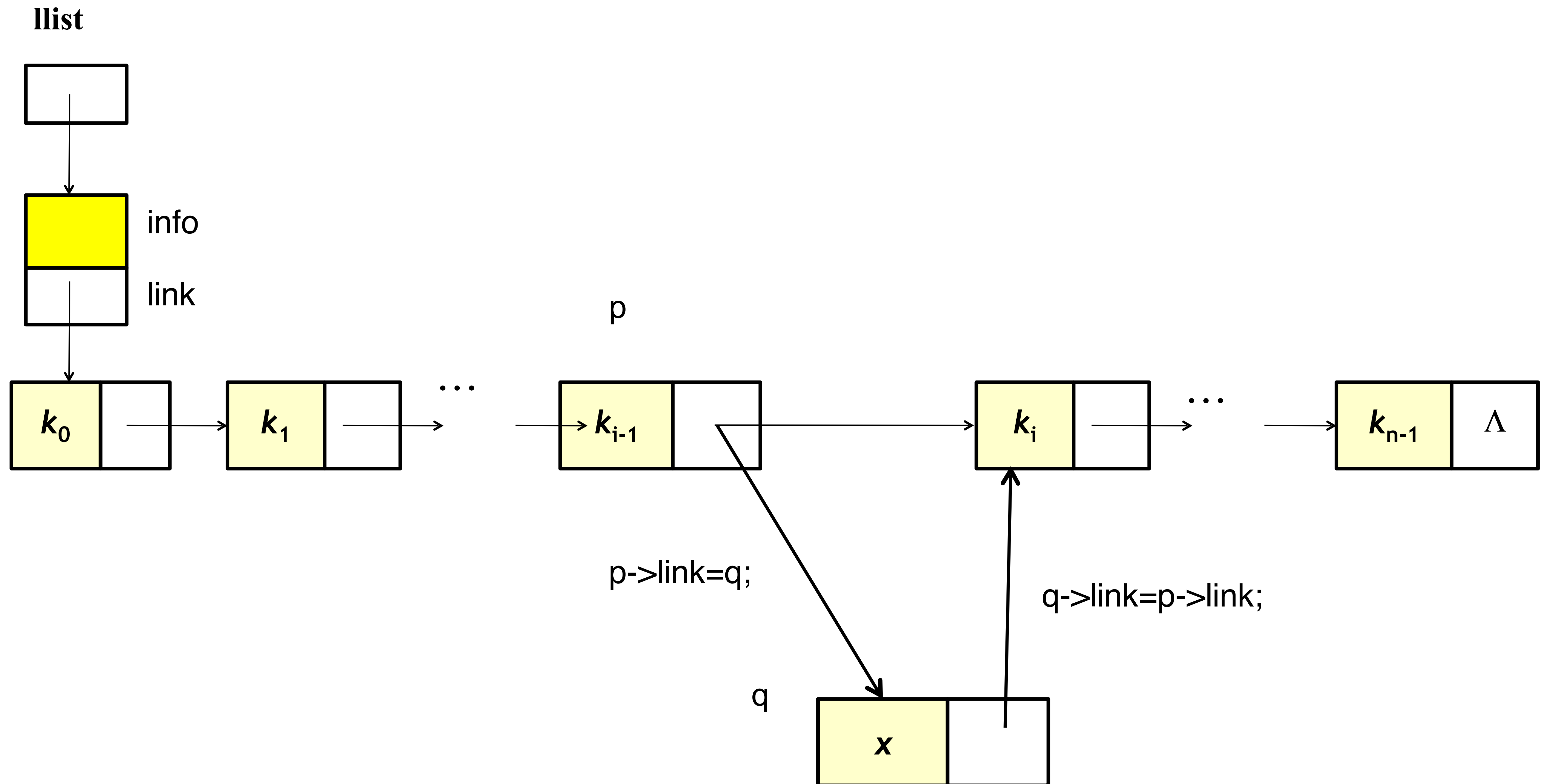
判断单链表是否为空

```
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;
}
```

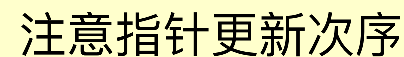
单链表的插入



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

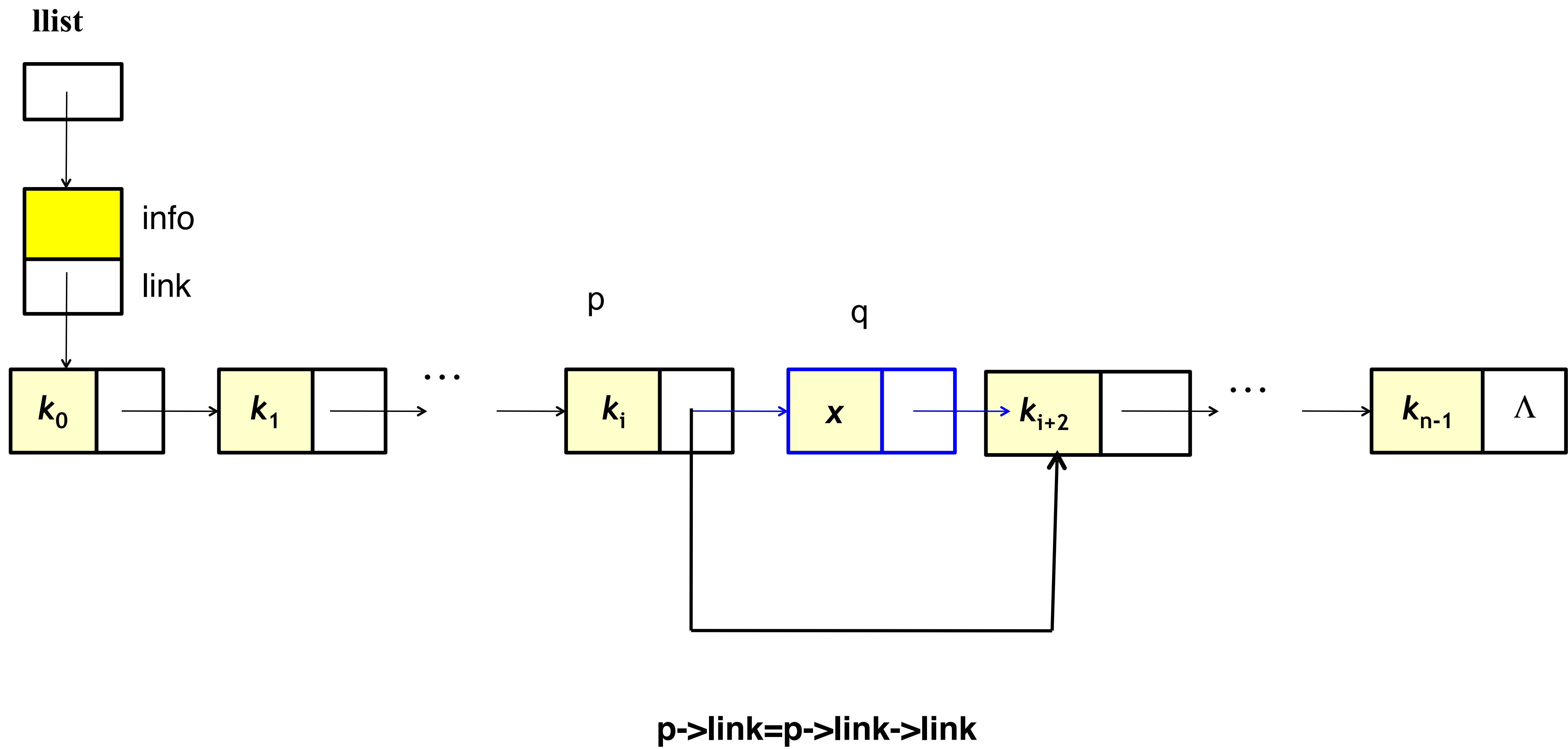
```
int  insertPost_link (LinkList  llist, PNode p, DataType  x) {
        PNode  q=(PNode)malloc(sizeof(Struct Node));     /*申请新结点*/
        if (q==NULL) {
                printf("Out of Space!\n") ;
                 return 0;
          }
         else {
                  q->info =x;
              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



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



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)

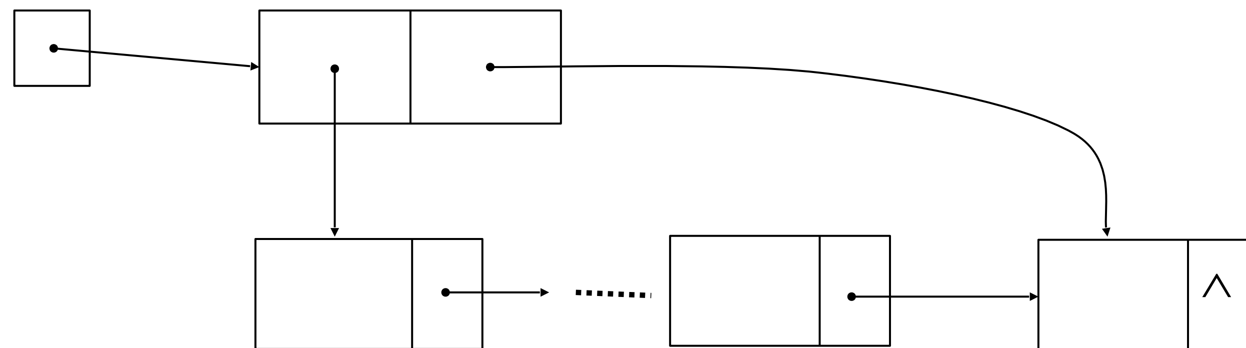
```
int deleteV_link(LinkList llist, DataType x) {
PNode p, q;     p = llist;
  if(p==NULL) return 0 ;
  while( p->link != NULL && p->link->info != x )
      p = p->link;                          /*找值为x的结点的前驱结点的存储位置 */
   if( p->link == NULL ) {                  /* 没找到值为x的结点 */
         printf(”Not exist!\n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

单链表的变形

- 单链表并非只有一种设计，可以根据需要和认识修改设计，例如
  - 前面实现的一个缺点是尾端加入操作的效率低
  - 实际中可能经常需要频繁地在表的两端加入元素
- 一种可能是采用下面的结构，给表对象增加一个对表尾结点的指针：

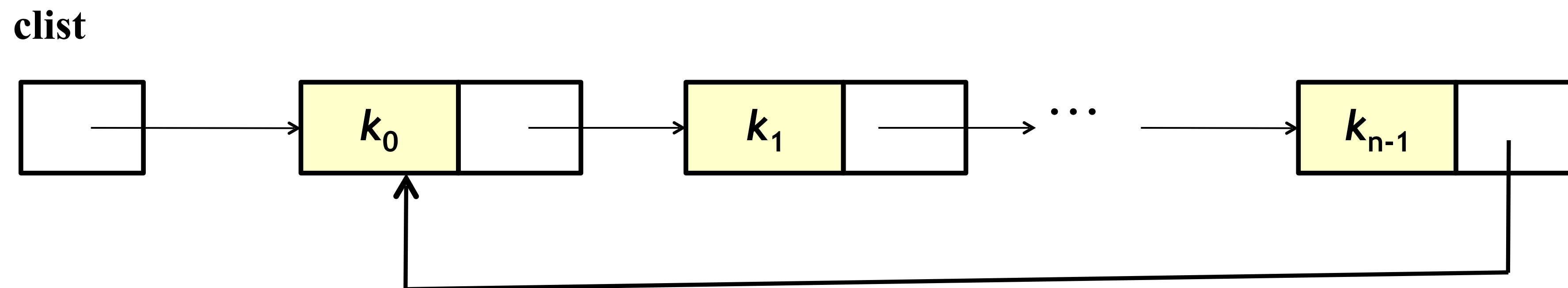


这样，在尾端加入元素，也能做到 $O(1)$

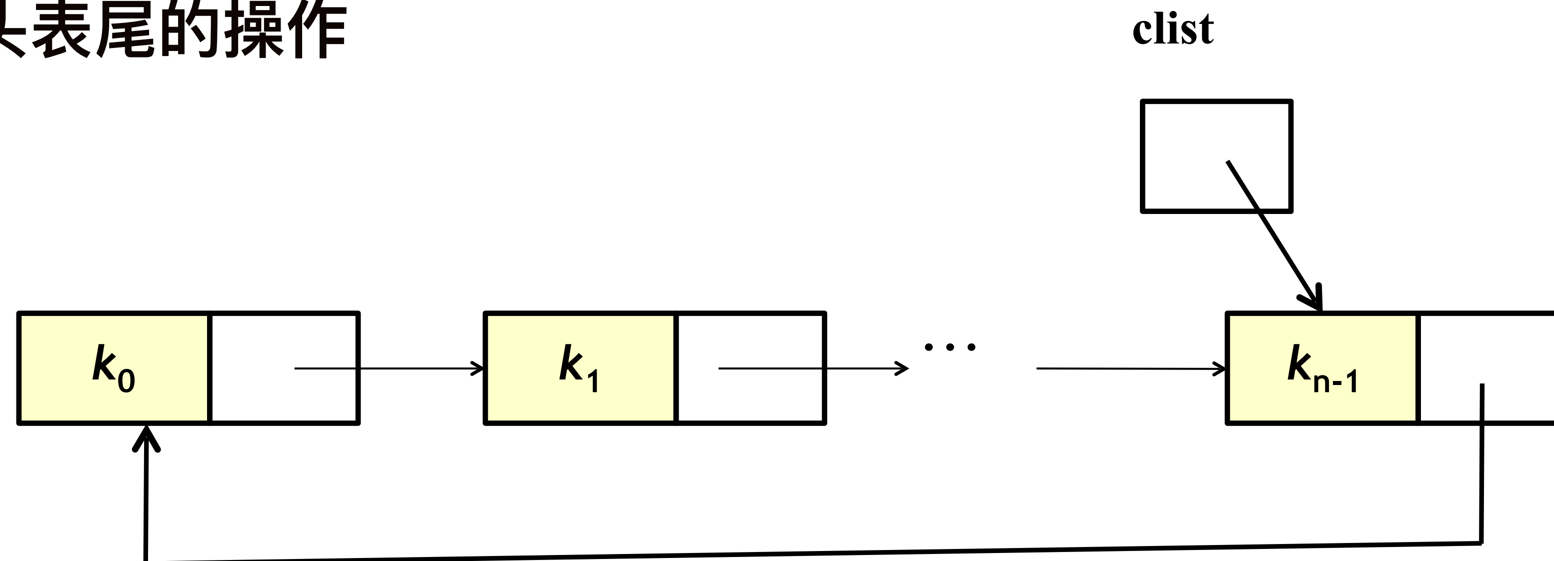
- 注意：新设计的链表与前面单链表结构近似，结构变化应该只影响到表的变动操作，非变动操作不需要修改。
- 有可能重用前面定义吗?

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

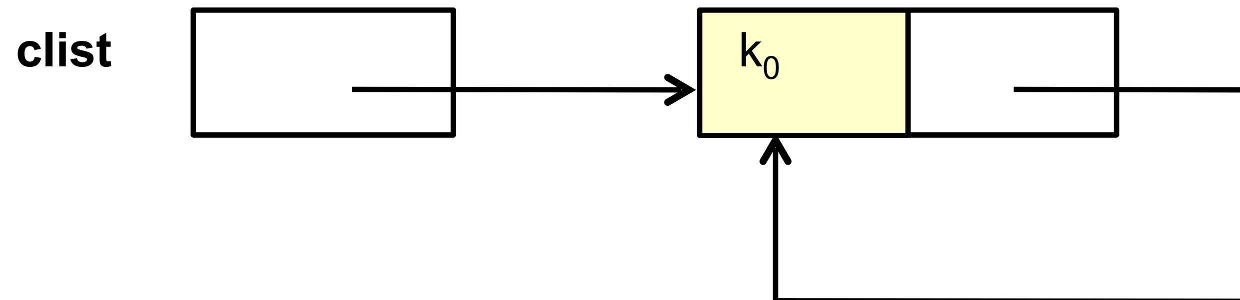


- 操作经常要访问、修改首结点和末结点, 让表头指针指向循环链表末结点, 可以方便对表头表尾的操作

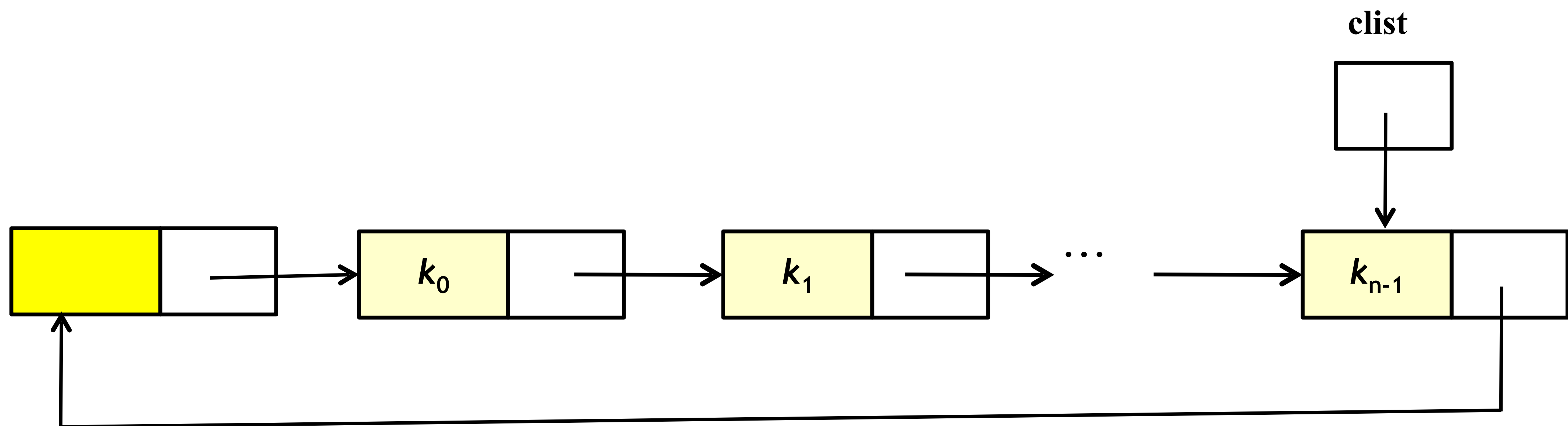


循环链表 (II)

- 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 = (LinkList) malloc( sizeof( Node ) );
if ( llist != NULL ) llist->link = llist;
return llist;
```

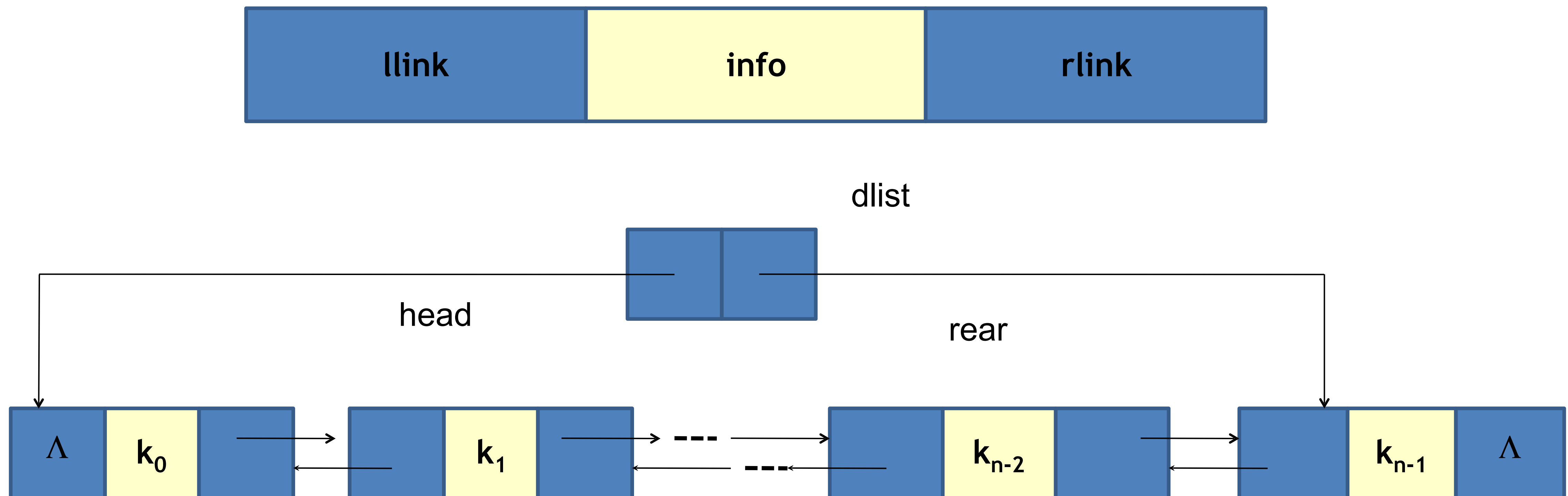
- 无头结点的表插入第一个结点时，也需建立循环链接
- 判断空表的操作（有头结点的表）：

```
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



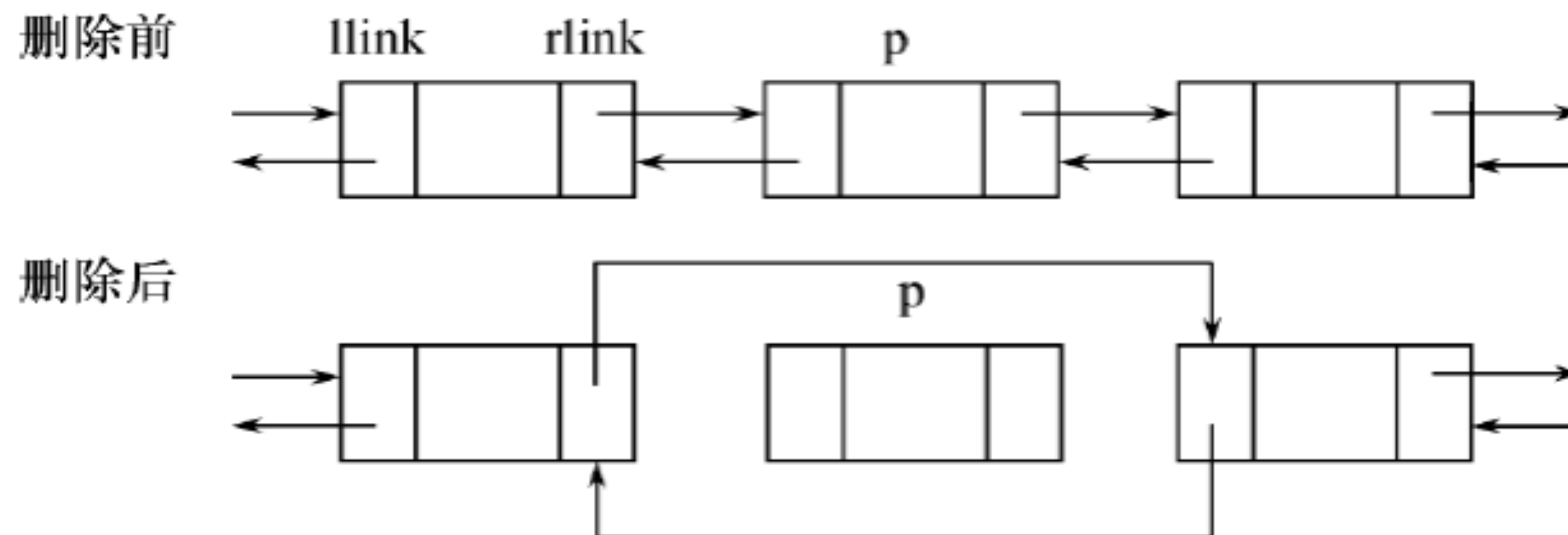
双链表的实现

```
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

- 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;
```



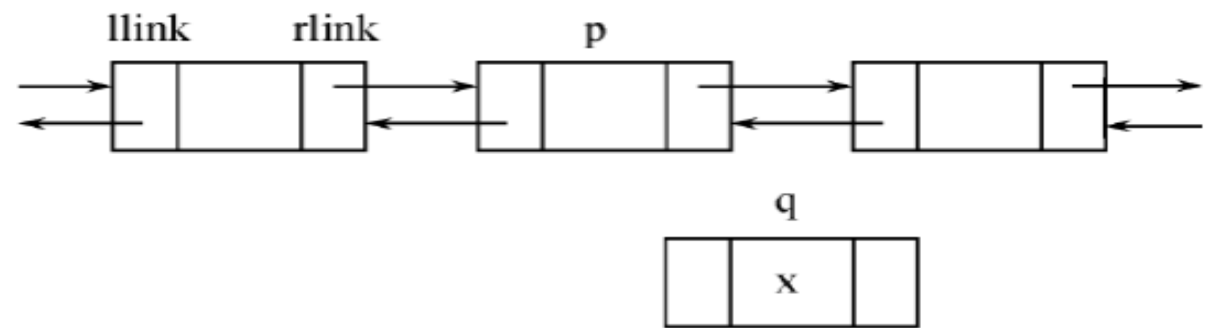
双链表的插入

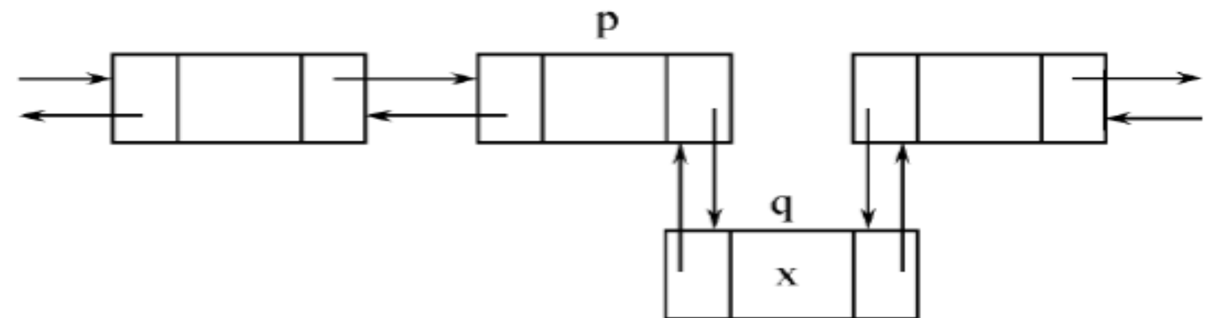
- **在p所指的结点后插入一个新结点，首先申请一个新结点**

  q=(PDoubleNode)**malloc**(**sizeof**(**struct** DoubleNode));

- **修改p所指结点的rlink和原后继结点的llink：**

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

- 链接存储结构的优点:
  - 不用预先按最大需要分配连续空间;
  - 线性表的插入和删除只需修改指针域，而不需移动其它元素。
- 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;
  - 链接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。

讨论

- 本讲讨论的线性表和随后几讲要讨论的串、栈与队列都属于线性结构。它们之间的区别，主要是操作的不同，所以分别属于几种不同抽象数据类型的实现 。