数据结构与算法设计

2021-09

课程内容简介

第1章 绪论	第8章 排序与分治	串与串匹配算法
第2章 线性表	第9章 外部排序	红黑树
第3章 栈和队列	第10章 动态规划算法	k-d树
第4章 数组和广义表	第11章 有限自动机	复杂图算法
第5章 树、二叉树、回溯法	第12章 图灵机	文本检索技术
第6章 图与贪心算法	第13章可判定性	分支限界算法
第7章 查找	第14章 时间复杂性	随机化算法
		上下文无关文法

第2章 线性表

2.1 线性表的类型定义
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 线性表应用实例

2.1 线性表的基本概念

线性表特点：

在数据元素的非空有限集中

- 存在唯一的一个被称作“第一个”的数据元素
- 存在唯一的一个被称作“最后一个”的数据元素
- 除第一个外，集合中的每个数据元素均只有一个前驱
- 除最后一个外，集合中的每个数据元素均只有一个后继

线性表是类型相同的元素有限序列，记作：

$$(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

2.1 线性表的基本概念

设 $A=(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$ 是一线性表

1. 线性表的数据元素可以是各种各样的，但同一线性表中的元素必须是同一类型的；
2. 在表中 a_{i-1} 领先于 a_i，a_i 领先于 a_{i+1}，称 a_{i-1} 是 a_i 的直接前趋，a_{i+1} 是 a_i 的直接后继；
3. 在线性表中，除第一个元素和最后一个元素之外，其他元素都有且仅有一个直接前趋，有且仅有一个直接后继。线性表是一种线性数据结构；

2.1 线性表的基本概念

设 A=（a_1, a_2, … , a_{i-1}, a_i , a_{i+1}, …, a_n）是一线性表

……

4. 元素的个数 n 称为表的长度，n=0时称为空表；
5. a_i 是表的第 i 个元素，称 i 为数据元素 a_i 的序号，每个元素在线性表中的位置，仅取决于它的序号。
6. 可在表的任意位置进行插入和删除操作。

2.1 线性表的基本概念

对线性表的ADT描述

ADT List {

数据对象：

D = { a_i | a_i ∈ElemSet, i=1,…,n， n≥0 }

数据关系：

R1 = { <a_{i-1}, a_i> | a_{i-1},a_i ∈D, i=2, …,n }

基本操作：

InitList (&L)

操作结果：构造一个空的线性表L。

DetroyList (&L)

初始条件：线性表L已经存在。

操作结果：销毁线性表L。

……

} ADT List

2.1 线性表的基本概念

对线性表的基本操作

1 初始化操作 InitList (&L)
功能：建立空的线性表L；

2 销毁操作 DetroyList (&L)
功能：回收为线性表L动态分配的存储空间；

3 置空操作 ClearList (&L)
功能：L中已存在，重新将其置成空表；

4 判空操作 ListEmpty (L)
功能：判断线性表L是否为空表，若为空表返回TRUE，否则返回FALSE；

5 求表长操作 ListLength (L)
功能：返回线性表L的表长；

2.1 线性表的基本概念

对线性表的基本操作

6 取元素操作：GetElem（ L, i, &e）

功能：将线性表L中第 i 个元素赋值给 e；

7 查找操作 LocateElem（ L, e, compare() ）

功能：在线性表 L 中查找与元素e 满足compare()的第 1 个元素，返回该元素在表中的序号（或位置），若表中不存在这样的元素，则返回 0；

8 查找前驱 PriorElem（ L, cur_e, &pre_e ）

功能：若 cur_e 是 L 中的数据元素且不是第一个，则用 pre_e 返回它的前驱，否则失败，pre_e无定义。

2.1 线性表的基本概念

对线性表的基本操作

9 查找后继 NextElem（ L, cur_e, &next_e ）

功能：若 cur_e 是L中的数据元素且不是最后一个，则用next_e返回它的后继，否则失败，next_e无定义。

10 插入操作 ListInsert（ &L, i, e ）

功能：在线性表L的第i个元素之前插入1个新元素e;

2.1 线性表的基本概念

对线性表的基本操作

11 删除操作 ListDelete（ &L, i, &e ）

功能：删除线性表L的第i个元素，并用 e 返回；

12 遍历操作 ListTraverse（ &L,visit( ) ）

功能：依次对线性表L的每个元素调用函数visit()。若visit()失败，则返回 ERROR，否则返回 OK；

2.1 线性表的基本概念

- 说明

1、基本操作是一种数据结构中最常见的操作，它具有明确的逻辑意义。

2、可以将基本操作看成一个整体，使用基本操作来解决新的问题，设计新的算法。使我们可以在更高的层次上进行抽象，在更高的基础上进行设计。

3、在算法的程序实现的过程中，基本操作需要通过编制公共函数进行具体实现。可以为基本操作建立公共函数库。

2.1 线性表的基本概念

例2-1：若有两个集合 A 和 B 分别用两个线性表 LA 和 LB 表示，即：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∪B

问题分析

上述问题可以分解为：（1）扩大线性表 LA；（2）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。

2.1 线性表的基本概念

利用基本操作进行算法设计

1．从线性表LB中依次察看每个数据元素；

GetElem（ LB, i ）→ e

2．依值在线性表LA中进行查访；

LocateElem（ LA, e, equal( ) ）

3．若不存在，则插入之。

ListInsert（ LA, n+1, e ）

```
void union ( List &La,  List Lb )  {
   La_len = ListLength (La);       // 求线性表的长度
   Lb_len = ListLength (Lb);
   for ( i = 1;  i <= Lb_len;  i++ )  {
         GetElem(Lb, i, &e);  // 取Lb中第i个数据元素赋给e
         if ( ! LocateElem ( La, e, equal( ) ) )
           ListInsert ( La, ++La_len, e );
          // La中不存在和 e 相同的数据元素，则插入到La列表
   }
} // union
```

2.1 线性表的基本概念

例2-1-A：已知一个非空集合B，试构造一个集合 A，使 A中只包含 B 中所有值各不相同的数据元素。

采用线性表表示集合

策略：从集合 B 取出物件放入集合 A。

要求集合 A 中同样的物件不能有两件以上。

```
void union ( List &La,  List Lb ) {
   InitList (La);   // 构造(空的)线性表LA
   La_len=ListLength (La);  Lb_len=ListLength (Lb);
   for ( i = 1;  i <= Lb_len;  i++ ) {
       GetElem(Lb, i, &e);  // 取Lb中第i个数据元素赋给e
       if ( ! LocateElem ( La, e, equal( ) ) )
         ListInsert ( La, ++La_len, e );
        //遍历 La，若不存在和 e 相同的数据元素，则插入到La表中
   }
} // union
```

2.1 线性表的基本概念

利用基本操作进行算法设计

例2-2-B：已知一个非空集合B，试构造一个集合 A，使 A 中只包含 B 中所有值各不相同的数据元素。

采用有序表表示集合

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：

$$a_i \geqslant a_{i-1} \text{ 或 } a_i \leqslant a_{i-1} \quad (i=2,3,\cdots,n)$$

则称该线性表为有序表(Ordered List)。

2.1 线性表的基本概念

对集合 B 而言，值相同的数据元素必定相邻；

例如：

（2，3，3，5，6，6，6，8，12）

对集合 A 而言，

数据元素依值从小至大的顺序插入。

因此，数据结构改变了，

解决问题的策略也要相应进行调整。

```
void purge ( List &La, List Lb ) {
  InitList ( La );     La_len = ListLength ( La );
  Lb_len = ListLength ( Lb );   // 求线性表的长度
  for ( i = 1;  i <= Lb_len;  i++ ) {
        GetElem ( Lb, i, e ); // 取Lb中第i个数据元素赋给 e
        if ( ListEmpty(La) || ! equal (en, e) ) {
              ListInsert ( La, ++La_len, e );
              en = e;
        } // 只比较La最后一个元素，若与 e 不相同，则插入
  }
} // purge
```

2.1 线性表的基本概念

小结：

线性表是逻辑的数据结构，要求元素之间的前驱和后继保持唯一性。

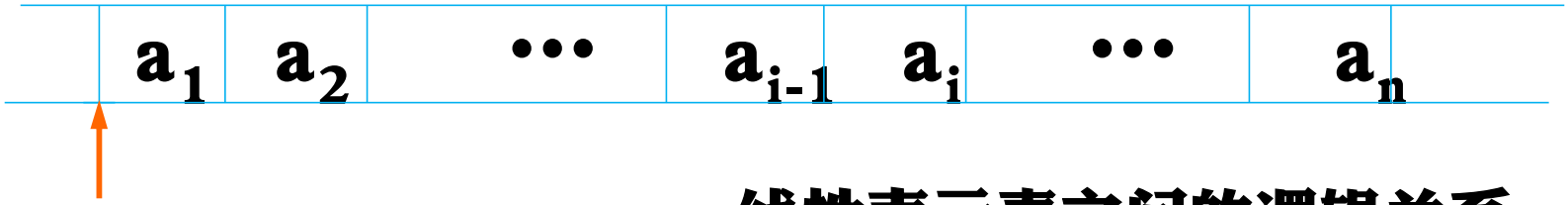
在实现线性表的物理结构上，可以有两种方法：

顺序表和链表两大类

2.2 线性表的顺序表示与实现

2.2 线性表的顺序表示与实现

- 线性结构的顺序表示是指使用一组**物理内存地址连续**的存储单元依次存储线性表的数据元素。



线性表的起始地址，称作线性表的**基地址**

线性表元素之间的逻辑关系，通过元素的存储顺序反映（表示）出来；

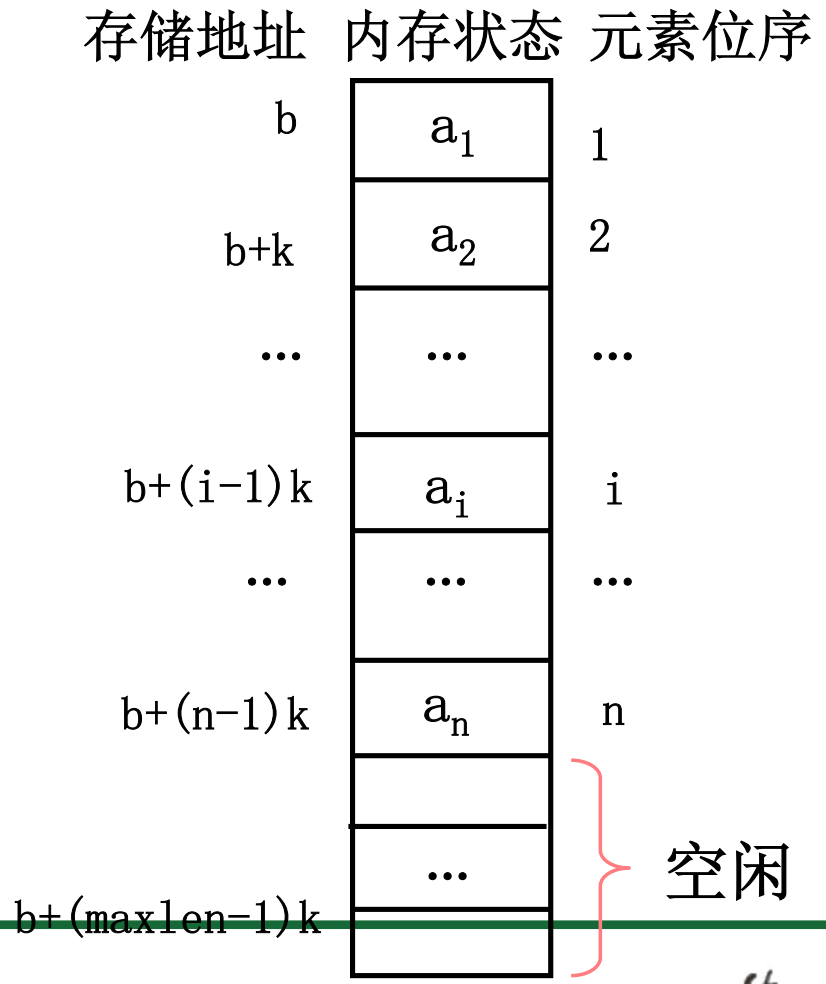
假设：线性表中每个数据元素占用 k 个存储单元，那么，在顺序存储结构中，线性表的第 i 个元素的存储位置与第 1 个元素的存储位置的关系：

$$Loc(a_i) = Loc(a_1) + (i-1) \times k$$

2.2 线性表的顺序表示与实现

顺序表的特点

- **用连续的存储单元存放线性表的元素(采用一维数组存放)。**
- **元素存储顺序与元素的逻辑顺序一致。**
- **读写元素方便，通过下标即可指定位置。**



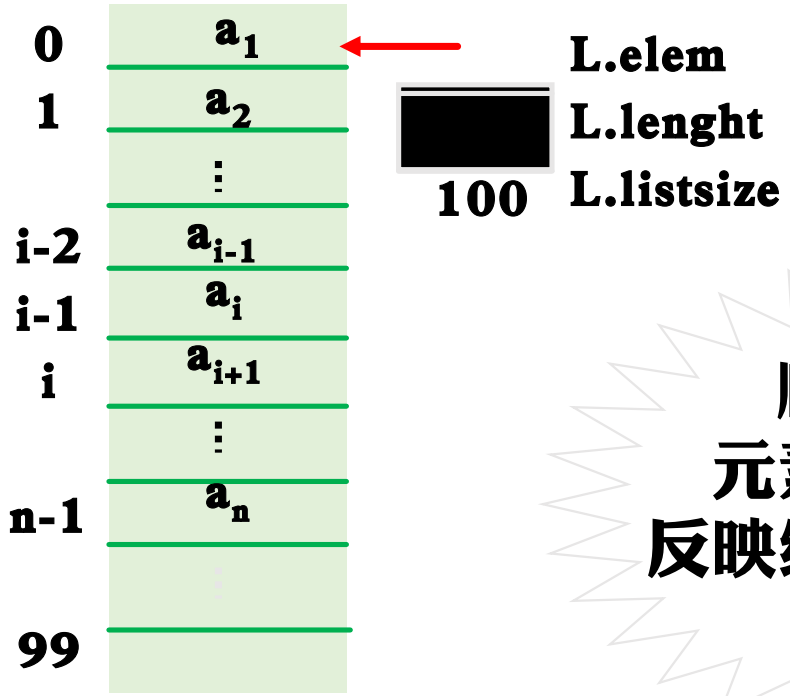
2.2 线性表的顺序表示与实现

- 顺序表的类型定义

```
#define   LIST_INIT_SIZE   100
                                          //  线性表存储空间的初始分配量
#define   LISTINCREMENT    10
                                            // 线性表存储空间的分配增量
typedef struct  {
  ElemType   * elem;    //线性表存储空间基址
  int     length;         //当前线性表长度
  int     listsize;       //当前分配的表空间大小
                 //（以sizeof(ElemType)为单位）
}SqList;
```

2.2 线性表的顺序表示与实现

设 A = (a_1, a_2, a_3, …, a_n) 是一线性表，L是 SqList 类型的结构变量，用于存放线性表 A：

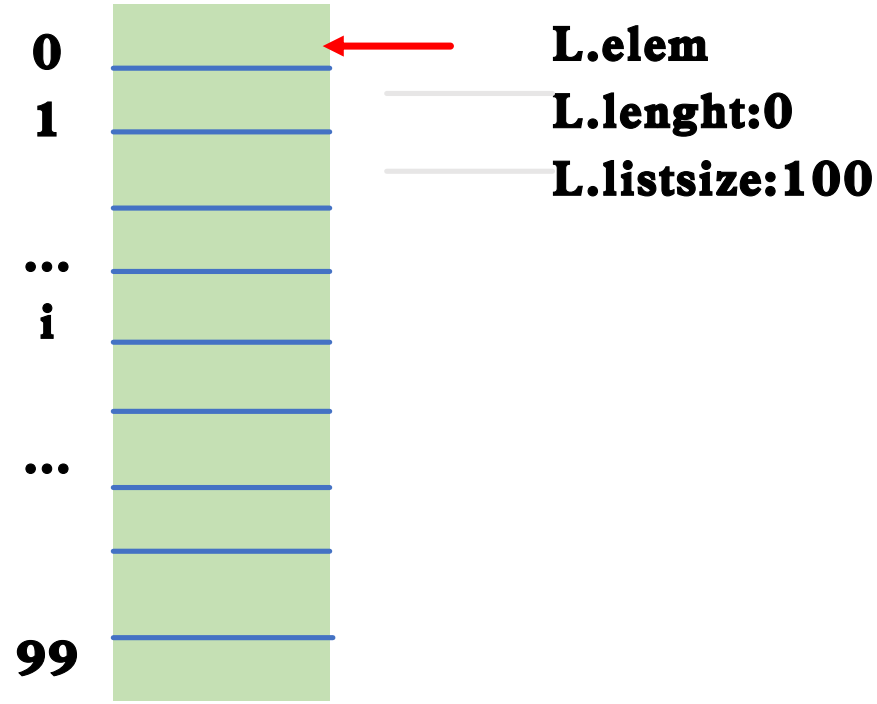


存放线性表元素的一维数组

顺序表通过元素的存储顺序反映线性表元素间的逻辑关系

2.2 线性表的顺序表示与实现

- 初始化操作 InitList_Sq（ SqList &L ）
  功能：建立空的顺序表L
  参数：L: 顺序表



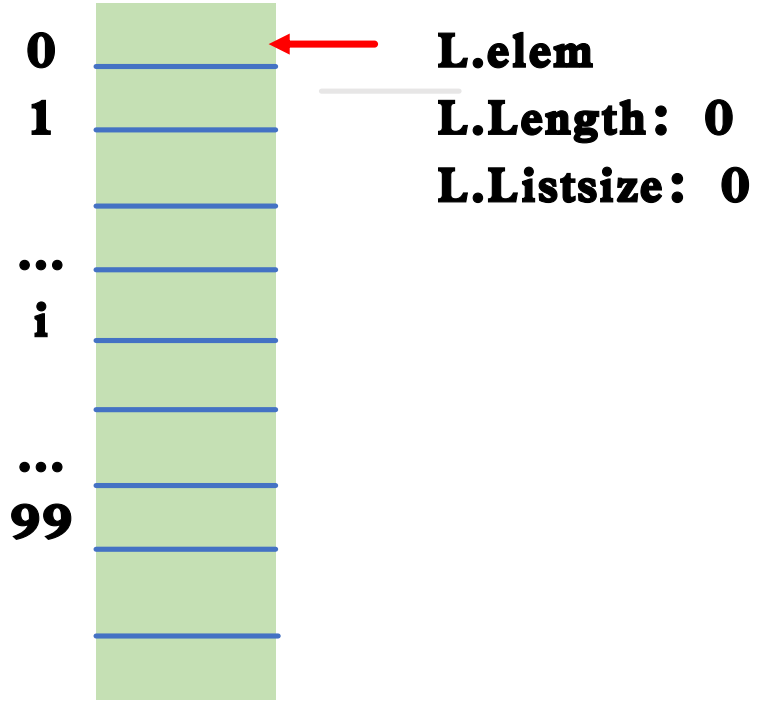
2.2 线性表的顺序表示与实现

初始化操作InitList_Sq(SqList &L)

```
Status InitList_Sq ( SqList &L )
{ //构造一个空的顺序表L
  L.elem = ( ElemType * )
                              malloc(LIST_INIT_SIZE *sizeof(ElemType));
  if ( !L.elem )
      exit ( OVERFLOW ); //如果分配空间失败
  L.length = 0;   //空表长度为0
  L.listsize = LIST_INIT_SIZE;
  return OK;
} //InitList_Sq
```

2.2 线性表的顺序表示与实现

- 销毁操作 DetroyList_Sq（ SqList &L ）
  功能：销毁顺序表L



2.2 线性表的顺序表示与实现

• 销毁操作 DetroyList_Sq（ SqList &L）

```
Status DetroyList_Sq（ SqList &L）
{   free (L.elem);
    L.elem = NULL;
    L.length = 0;
    L.Listsize = 0;
    return OK;
}  // DetroyList_Sq
```

2.2 线性表的顺序表示与实现

- 插入操作

  定义：线性表的插入是指在第 i（$1 \le i \le n+1$）个元素之前插入一个新的数据元素 x，使长度为 n 的线性表：

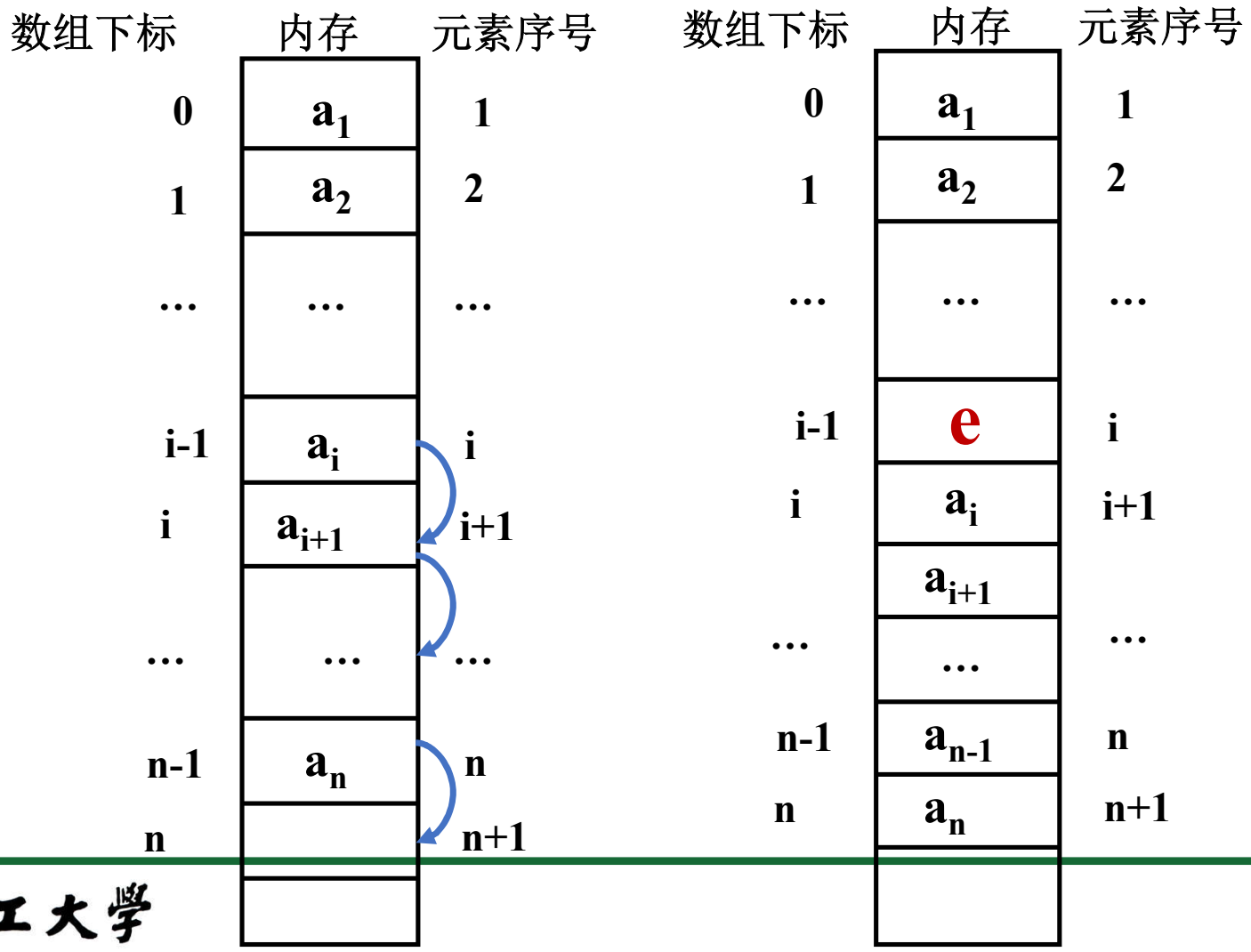
$$A = (a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

  变成长度为n+1的线性表：

$$A = (a_1, a_2, \cdots, a_{i-1}, x, a_i, a_{i+1}, \cdots, a_n)$$

  需将第 i 至第 n 共（n-i+1）个元素后移。

2.2 线性表的顺序表示与实现



2.2 线性表的顺序表示与实现

· 插入操作：基本算法

```
Status ListInsert_Sq (SqList &L, int i, ElemType e)
 {  //在顺序线性表L中第 i 个位置之前插入新的元素e,
        if ( i<1 || i>L.length+1 )
      return ERROR;                  // i 超出表长则不合法
  q = & ( L.elem[i-1] );             // q 指向插入位置
        for ( p=&(L.elem[L.length-1]); p>=q; --p ) *(p+1) = *p;
        // 初始化时p指向最后一个数据元素
  *q = e;                            // 插入e
  ++L.length;          // 表长增1
        return OK;
} // ListInsert_Sq
```

2.2 线性表的顺序表示与实现

插入操作的时间复杂度分析

插入操作的主要代价体现在表中元素的移动。

在位置 i 插入元素，需移动 n-i+1 个元素，

元素总个数为 k，假设各个位置插入的概率相等，为p＝1/n，

则，平均移动元素次数为：

$$\sum_{i=1}^{n} \frac{1}{n} \times (n - i + 1) \approx \frac{n}{2}$$

等概情况，算法的时间复杂度为 O(n)。

2.2 线性表的顺序表示与实现

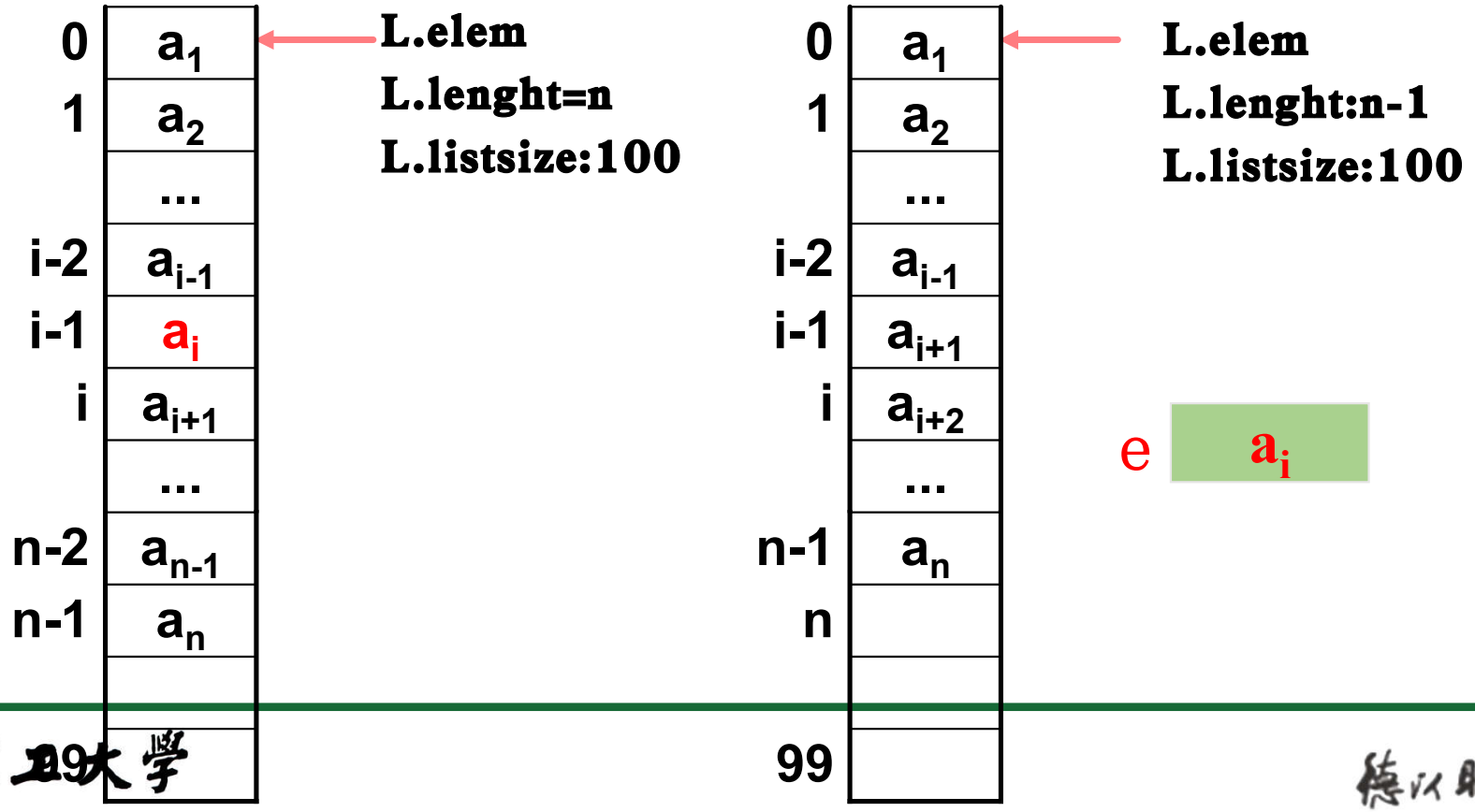
```
Status ListInsert_Sq(SqList &L, int i, ElemType e)
 { //在顺序线性表L中第 i 个位置之前插入新的元素e,
        if ( i<1 ‖ i>L.length+1 )    return ERROR;  //i不合法
        if ( L.length >= L.listsize )    //空间已满，重新分配空间
  {  newbase = (ElemType*) realloc (L.elem,
                                    (L.listsize+L.incresize) * sizeof(ElemType) );
       if ( !newbase ) exit(OVERFLOW);  //存储分配失败
       L.elem = newbase;                //新基址
       L.listsize+= L.incresize;        //增加存储容量
   }
  q = & ( L.elem[i-1] );        // q为插入位置
  for ( p=&(L.elem[L.length-1]); p>=q; --p )  // p指向尾元
        * (p+1) = *p;
  *q=e;   ++L.length;           // 插入e ，表长增1
  return OK;
} //ListInsert_Sq
```

2.2 线性表的顺序表示与实现

- 删除操作

ListDelete_Sq (SqList &L, int i, ElemType &e)

在顺序线性表L中删除第i个元素，并用e返回.



2.2 线性表的顺序表示与实现

```
Status ListDelete_Sq(SqList &L,int i,ElemType &e)
{      // 在顺序线性表L中删除第 i 个元素，并用 e 返回其值。i 的
  合法值为 1≤i≤L.length。
  if ( (i<1) || (i>L.Length) )     return ERROR;
       p = & ( L.elem[i-1] );              //p为被删除元素的位置
  e = *p;                                  // 被删除元素的值赋给e
  q = L.elem[L.length-1];                  // 表尾元素的位置
  for ( ++p; p<=q; ++p )
      *(p-1) = *p;              //被删除元素之后的元素前移
  -- L.length;                            //表长减1
  return OK;
} //ListDelete_Sq
```

2.2 线性表的顺序表示与实现

- 顺序表的算法分析

  插入和删除操作的主要代价体现在表中元素的**移动**

  - 插入：移动 n-i+1 个
  - 删除：移动 n-i 个

2.2 线性表的顺序表示与实现

小结：

- 顺序表的优缺点
  - **优点**
    - 不需要附加空间
    - 随机存取任一个元素（根据下标）
  - **缺点**
    - 很难估计所需空间的大小
    - 开始就要分配足够大的一片连续的内存空间
    - 更新操作代价大

2.3 线性表的链式存储与实现

2.3 线性表的链式存储与实现

- 线性表的链式存储结构特点

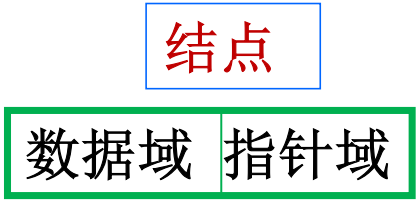
  1、用一组任意的存储单元存储线性表的数据元素。

  2、利用指针实现了用物理上不相邻的存储单元存放逻辑上相邻的一组元素。

  3、每个数据元素 a_i，除存储本身信息外，还需存储其直接后继的信息（指针）。

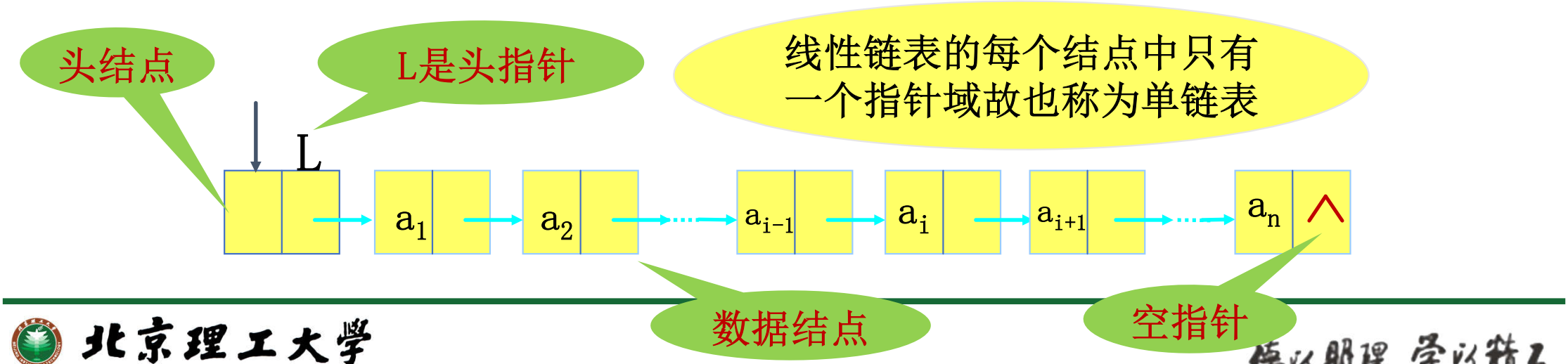
2.3 线性表的链式存储与实现

- **结点**
  - **数据域：数据元素本身的信息**
  - **指针域：指示直接后继的存储位置**
    - 根据链接方式和指针多寡
      - 单链表
      - 双链表
      - 循环链表



2.3 线性表的链式存储与实现

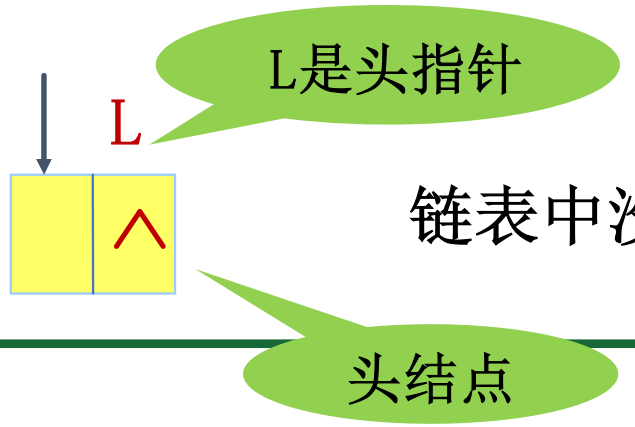
- **头指针：** 存放线性链表中第一个结点的存储地址；
- **空指针：** 不指向任何结点，线性链表最后一个结点的指针通常是指针；
- **头结点：** 线性链表的第一数据元素结点前面的一个附加结点，称为头结点。头结点不保存数据。
- **带头结点的线性链表：** 第一元素结点前面增加一个附加结点的线性链表称为带头结点的线性链表。



2.3 线性表的链式存储与实现

- 结点的形式定义

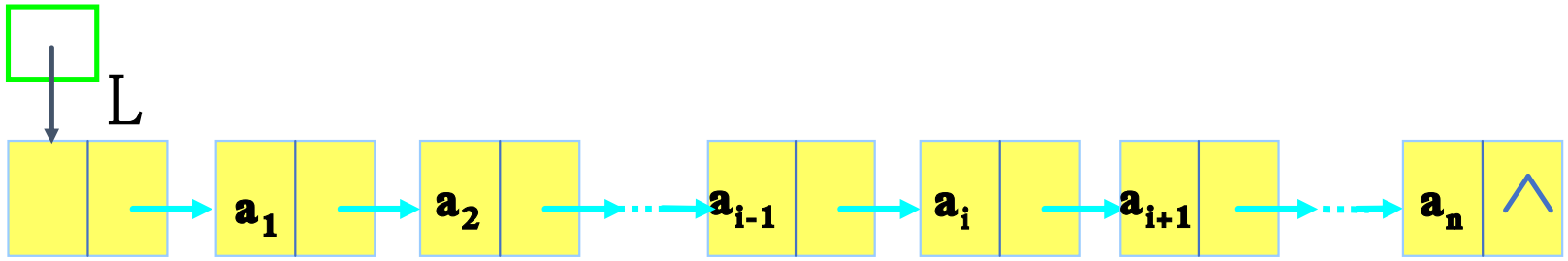
```
Typedef  struct   LNode {
    ElemType        data;          // 数据域
    struct Lnode    *next;    // 指针域
} LNode，* LinkList;
LinkList  L;
// L 为单链表的头指针，是 LinkList 类型的变量
```

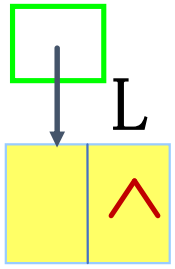


链表中没有数据结点，称为：空表

2.3 线性表的链式存储与实现

· 相关术语：带头结点的单向链表





空表

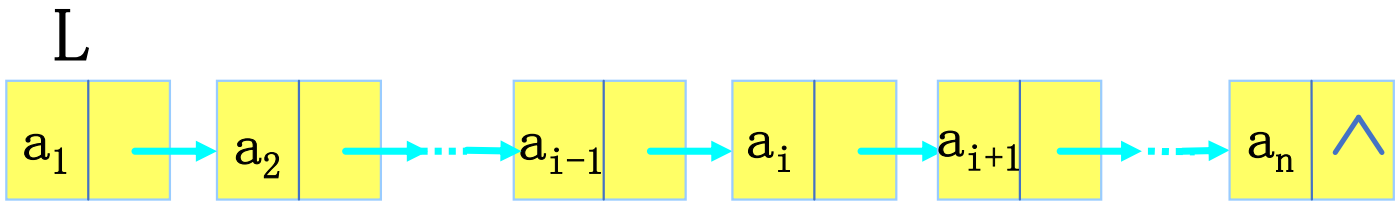
怎样判断带头结点的单向链表是否为空表?

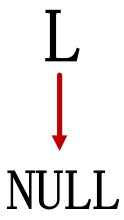
如果：L->next == NULL 成立，则 L 为空表。

如果：L->next != NULL 成立，则 L 不是空表。

2.3 线性表的链式存储与实现

· 相关术语：不带头结点的单向链表





空表：L=NULL

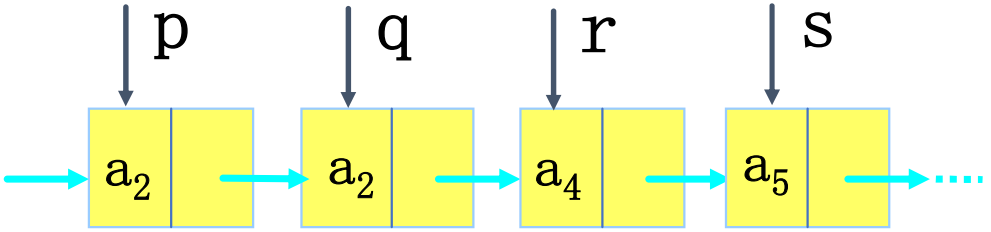
怎样判断不带头结点的单向链表是否为空表?

如果：L == NULL 成立，则 L 为空表。

如果：L != NULL 成立，则 L 不是空表。

2.3 线性表的链式存储与实现

· 结点之间的基本关系



则下列关系成立：

p->next == q

r->next == s == q->next->next

== p-> next->next->next

p指向当前结点 p

指向下一个结点的指针为：p->next

访问下一个结点的下一个结点：

p->next->next

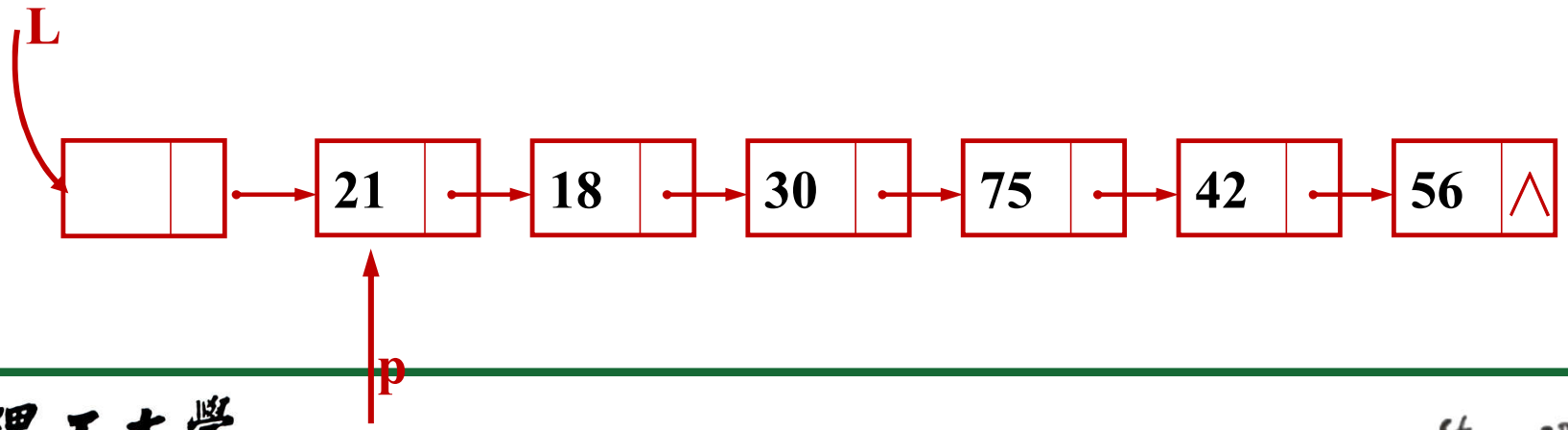
2.3 线性表的链式存储与实现

- 链表操作基本运算
  - 检索
    - 在链表中查找满足某种条件的元素
  - 插入
    - 在链表的适当位置插入一个元素
  - 删除
    - 从链表中删除一个指定元素

2.3 线性表的链式存储与实现

取数据元素操作 GetElem（ L, i, &e ）

- 功能：取带有头结点的线性链表 L 中第 i 个元素结点，其值存入 e。
- 主要步骤
  - 查找到第 i-1 个元素结点
  - 取值存入 e



2.3 线性表的链式存储与实现

- 单链表操作的特点

单链表是一种顺序存取的结构，**为找第 i 个数据元素，必须先找到第 i-1 个数据元素。**

因此，查找第 i 个数据元素的基本操作为：移动指针，比较 j 和 i 。

令指针 p 始终指向线性表中第 j 个数据元素。

2.3 线性表的链式存储与实现

```
Status GetElem_L (LinkList L,int i, ElemType &e)
{ // L是带头结点的链表的头指针，以e返回第i个元素
    p = L->next;  j = 1; // p 指向第1个结点，j为计数器
    while  ( p!=NULL && j<i ) // 沿指针向后查找
    {  p = p->next;  ++j;
    }
    // 退出循环时，p 指向第i个元素 或 p 为空
    if ( p ==NULL || j>i )
       return ERROR;          // 第 i 个元素不存在
    e = p->data;              // 取得第 i 个元素
    return OK;
} // GetElem_L
```

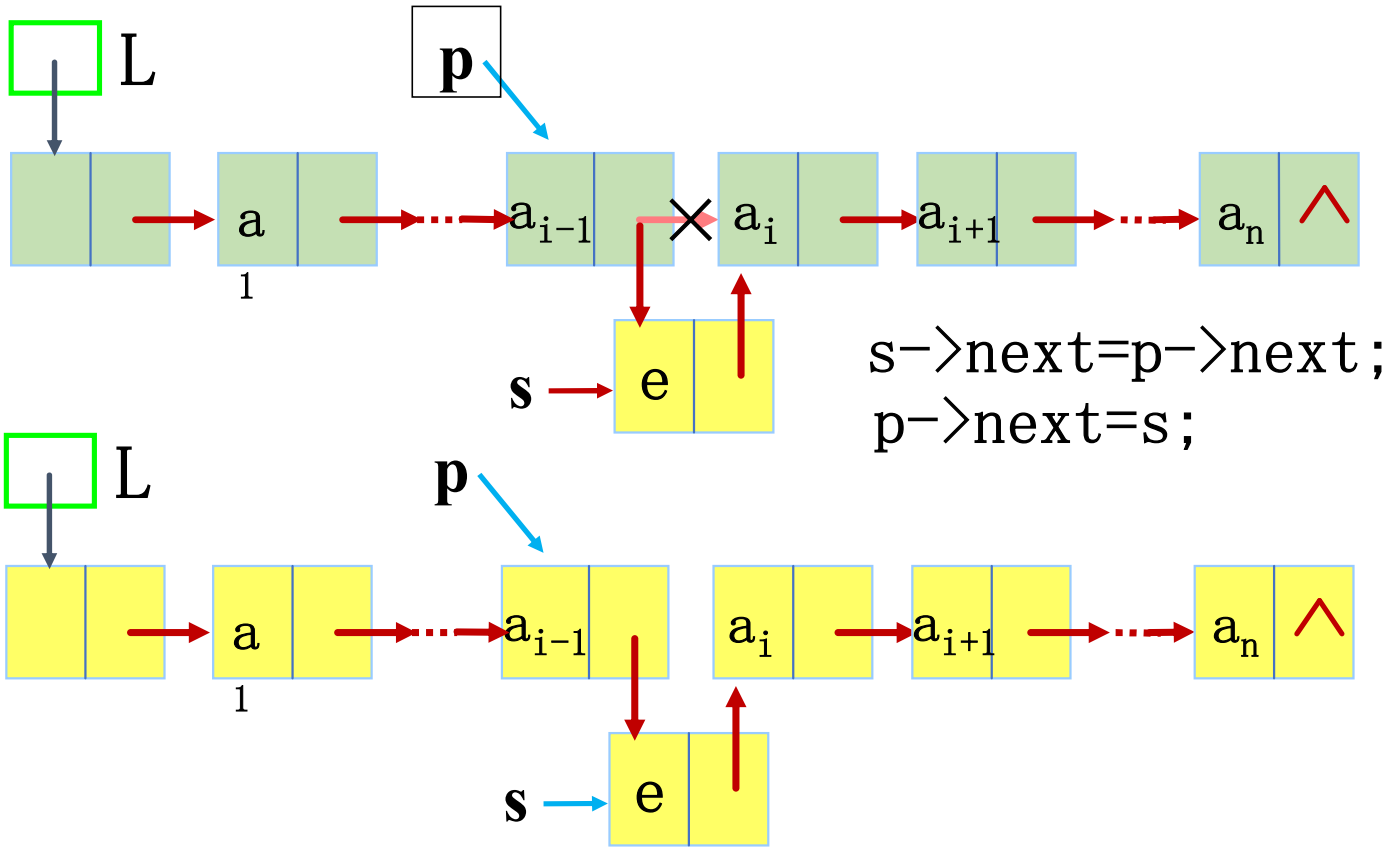
2.3 线性表的链式存储与实现

插入操作 ListInsert_L（ &L, i, e ）

- 功能：在带有头结点的线性链表 L 的第 i 个元素结点之前插入一个新元素结点e。
- 主要步骤
  - 查找第 i-1 个元素结点
  - 建立新结点 e
  - 修改指针插入新结点

2.3 线性表的链式存储与实现

- 主要步骤
  - p 指向第i-1个元素结点
  - s 指向新结点 e
  - 修改指针插入新结点



2.3 线性表的链式存储与实现

```
Status ListInsert_L(LinkList &L, int i, ElemType e)
{  //在带头结点的链表第i结点之前插入e，1≤i≤表长+1
   p=L;  j=0;                                  // p 指向头结点
   while ( p!=NULL && j<i-1 ) // 寻找第 i-1 个数据结点
   {  p=p->next; ++j;
   }        // 退出循环时 p 为NULL  或  指向第 i-1个结点
   if ( p==NULL || j>i-1 )
        return ERROR;                          // i<1 或 i>表长+1
   s = (LinkList) malloc(sizeof(LNode));  //建新结点
   s->data = e;
   s->next = p->next;
   p->next = s;        // 插入新结点
   return OK;
} //ListInsert_L
```

问题：对于不带头结点的链表应该如何进行插入操作?

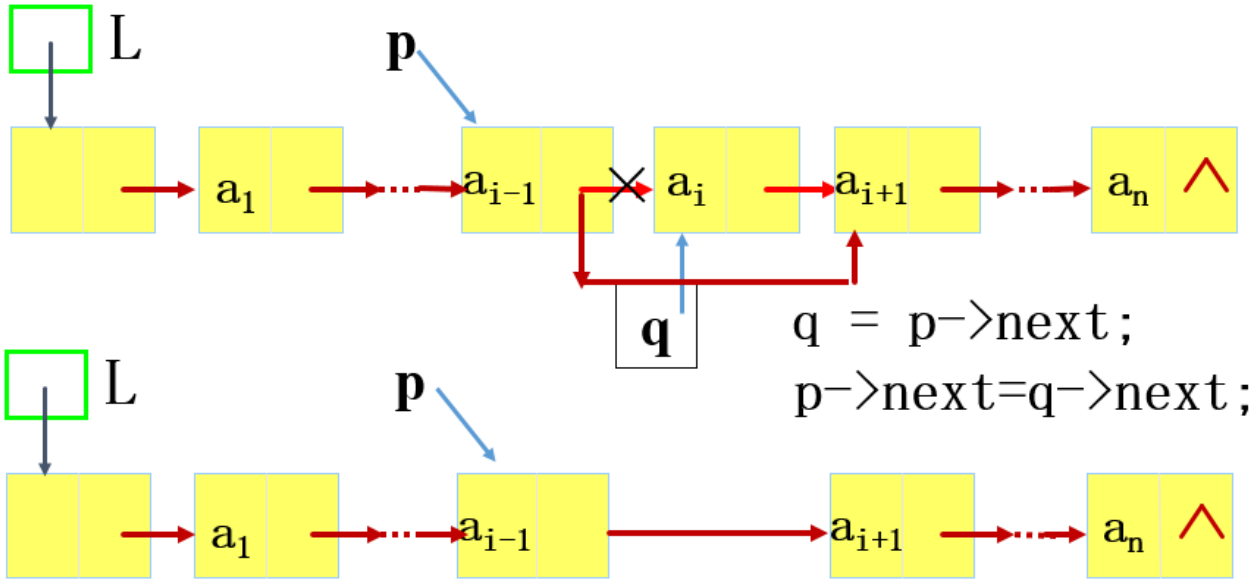
2.3 线性表的链式存储与实现

删除操作 ListDelete_L（ &L, i, e ）

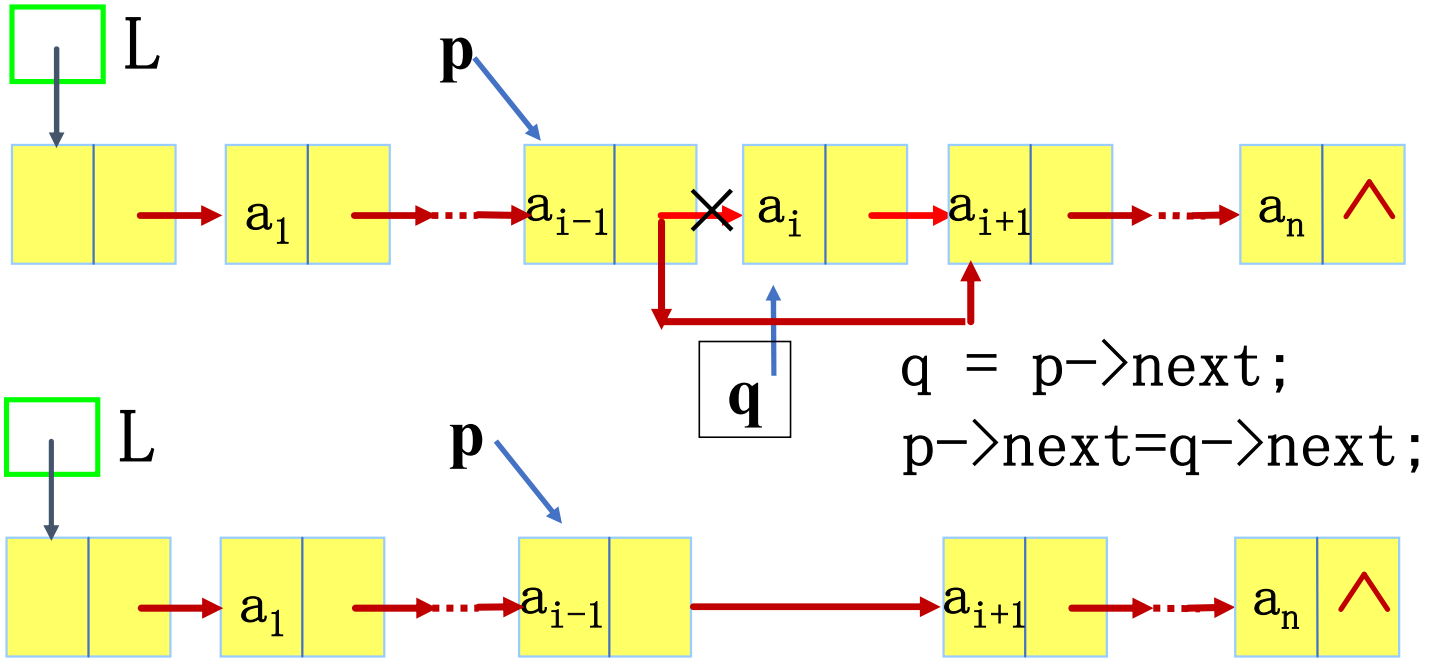
- **功能：删除线性链表 L 中的第 i 个元素结点，并由 e 返回。**
- **主要步骤**
  - **查找第 i-1 个元素结点**
  - **修改指针删除第 i 个元素结点**
  - **回收被删除结点**

2.3 线性表的链式存储与实现

- 主要步骤
  - p 指向第 i-1 个元素结点，q指向被删除结点
  - 修改指针删除第 i 个元素结点
  - 回收被删除结点



2.3 线性表的链式存储与实现



2.3 线性表的链式存储与实现

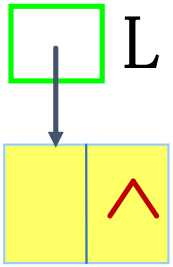
```
Status ListDelete_L(LinkList &L, int i, ElemType &e)
{  // 在带头结点的链表L中，删除第 i 个元素，由 e 返回其值。1≤i≤表长。
   p=L;    j=0;
   while( p->next !=NULL && j<i-1 ) //寻找第i-1个结点
   { p=p->next;  ++j;
   } // 退出循环时，p为NULL 或 指向第 i-1 个结点
         if ( !p->next || j>i-1 )   return ERROR;
   q=p->next;     p->next=q->next;   // 删除结点
   e=q->data;
   free(q);              // 回收结点空间
   return OK;
}  //ListDelete_L
```

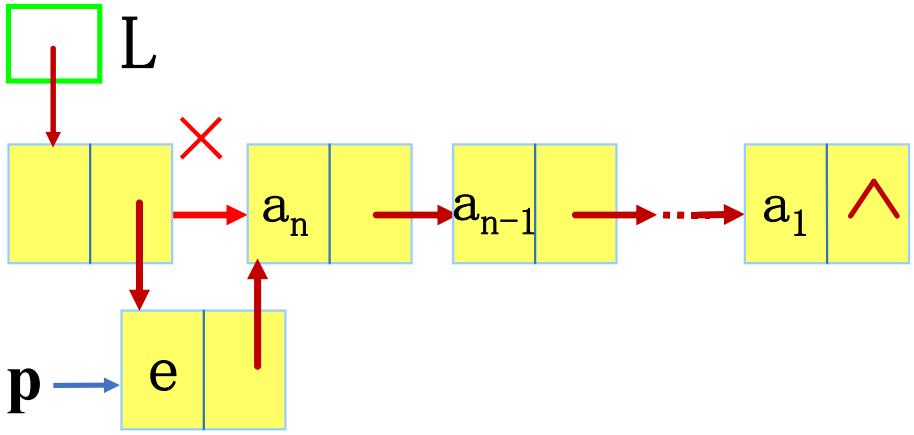
问题：对于不带头结点的链表应该如何进行删除操作?

2.3 线性表的链式存储与实现

· 建立带头结点的线性链表

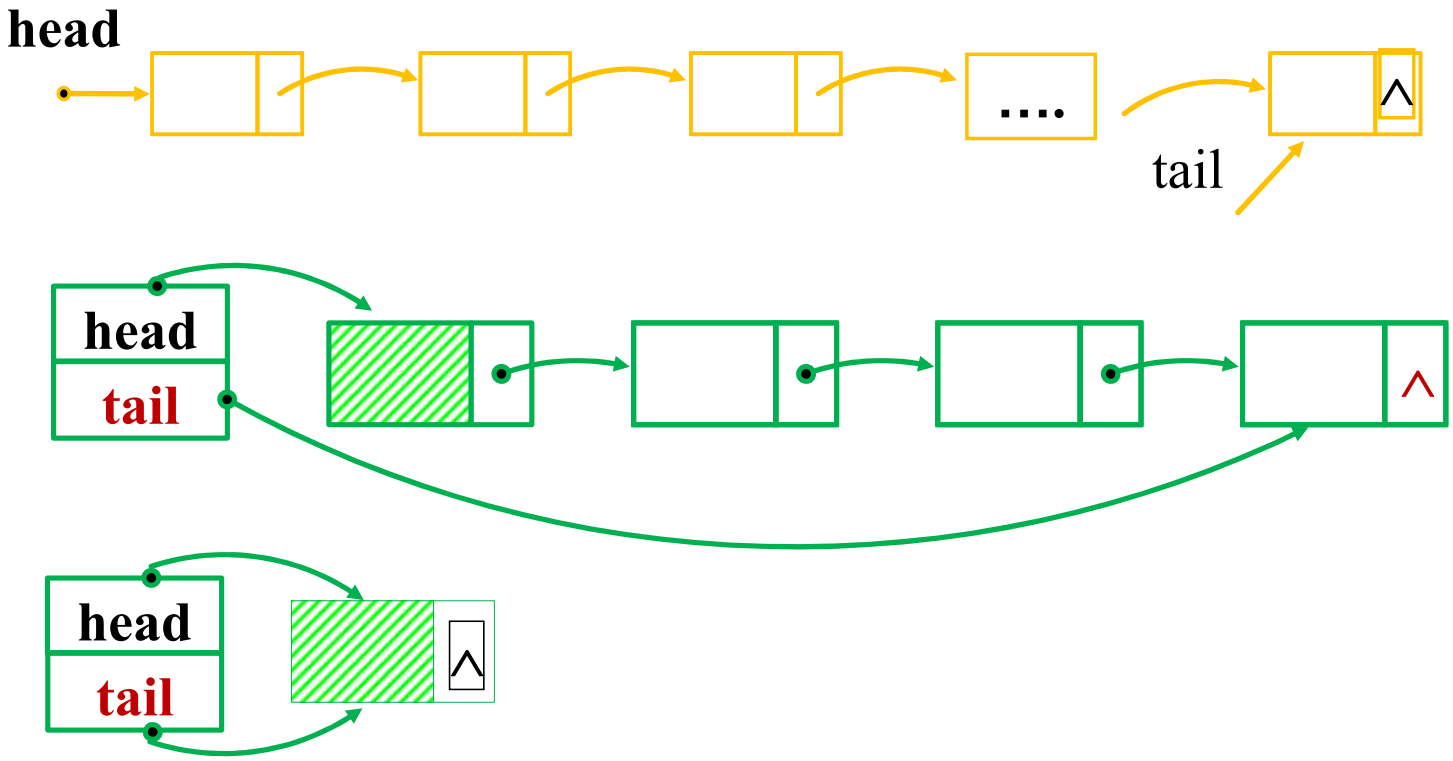
```
void CreateList_L ( LinkList &L, int n )
{  L = ( LinkList ) malloc ( sizeof (LNode) ); //建空表
    L->next = NULL;
    for ( i=n; i>0; --i ) // 逆序输入n个元素
    { read(e);   // 读入元素
       p = ( LinkList ) malloc ( sizeof(LNode) );
       p->data = e;
       p->next = L->next;
       L->next=p; // 插入到表头
    }
} //CreateList_L
```





2.3 线性表的链式存储与实现

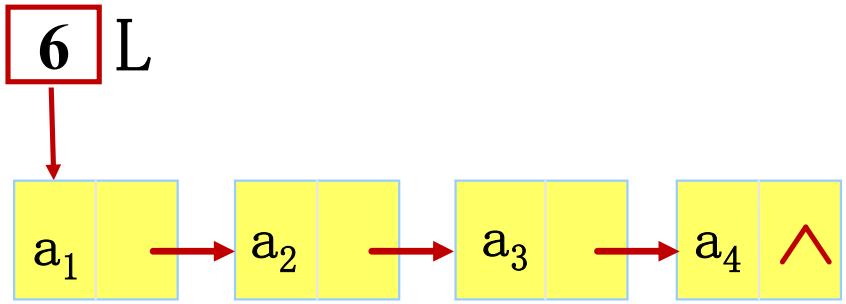
- 单链表的其他形态



2.3 线性表的链式存储与实现

- **静态链表**

  用**数组**实现的链式结构，称为**静态链表**。



0		
1	a4	0
2		
3	a3	1
4		
5		
6	a1	8
7		
8	a2	3
9		
10		

2.3 线性表的链式存储与实现

- 线性链表的特点
  1. 用一组**任意的存储单元**存储线性表中数据元素；
  2. 通过**指针保存直接后继元素的存储地址**来表示数据元素之间的逻辑关系；
  3. 通过**头指针（或首结点）**给出线性链表；
  4. 链表中结点空间是**动态分配**的；
  5. 插入、删除操作通过**修改结点的指针**实现；
  6. **只能顺序存取**元素，不能直接存取元素。

2.3 线性表的链式存储与实现

- 单链表操作分析
  1. 对一个结点操作，必先找到它，即用一个指针指向它；
  2. 找单链表中任一结点，都必须从第一个点开始：

```
p = head;
while (没有到达结尾)
    p = p->next;
```

  3. 单链表的时间复杂度　O(n)

     定位：　O(n)

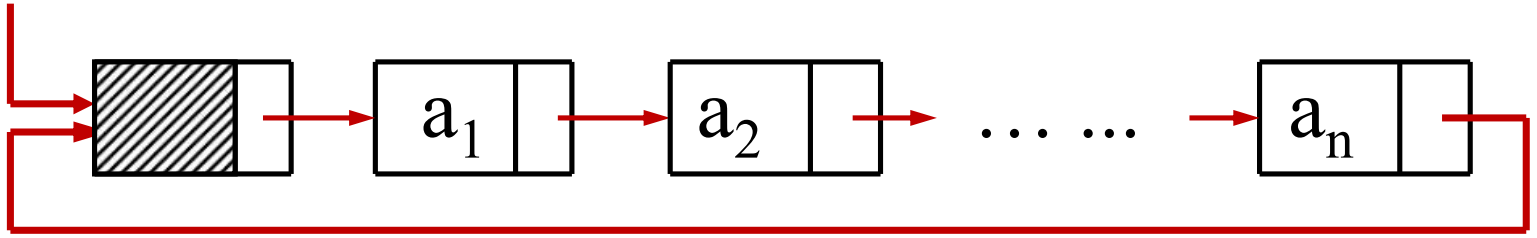
     插入：　O(n) + O(1)

     删除：　O(n) + O(1)

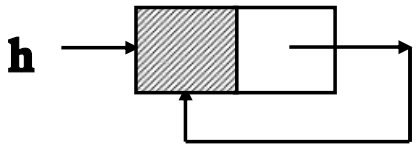
2.3 线性表的链式存储与实现

· 循环链表（带有头结点的单向环表）

最后一个结点的指针域的指针又指回第一个结点（头结点）的链表。



与单向链表的差别仅在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。



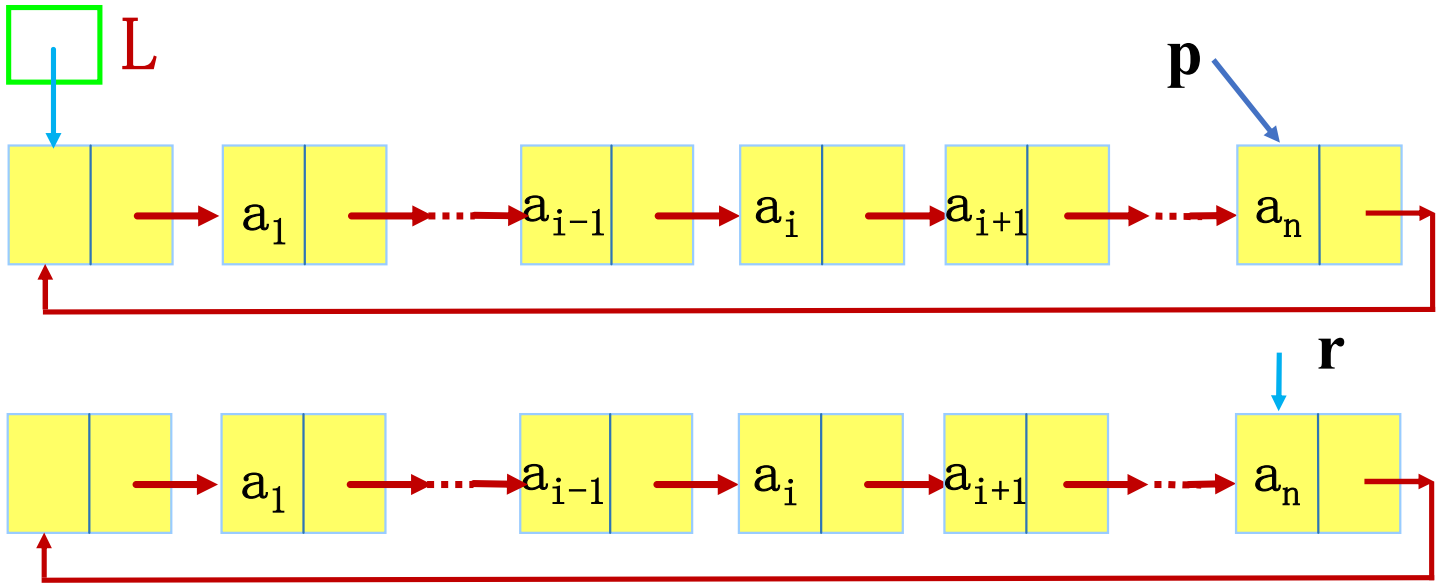
空表

判断为是否空表的条件：

h->next == h

2.3 线性表的链式存储与实现

- 循环链表的特点
  1. 从一个结点可找到链表中的任意一个结点；
  2. 判断是否为表尾结点的条件：p->next == L。
  3. 有时，用表尾指针表示循环链表。

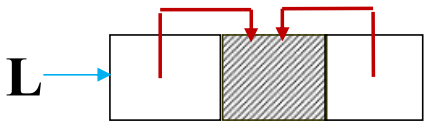


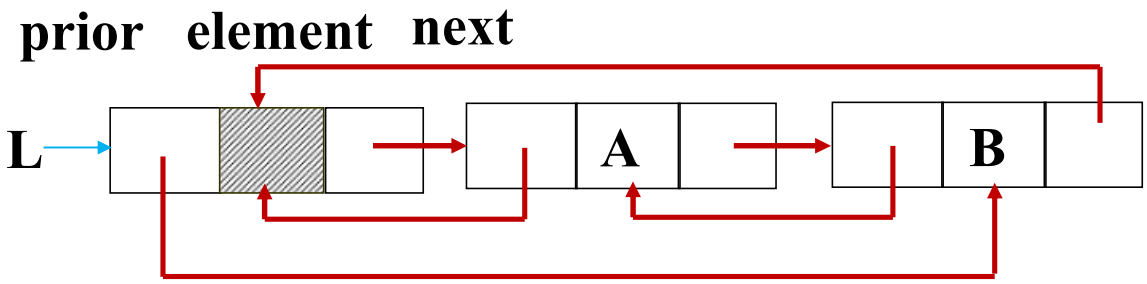
2.3 线性表的链式存储与实现

- 双向链表

```
typedef struct  DuLNode {
    ElemType          data;      // 数据域
    struct DuLNode   *prior;     // 指向前驱的指针域
    struct DuLNode   *next;      // 指向后继的指针域
} DuLNode, *DuLinkList;
```

空双向循环链表：

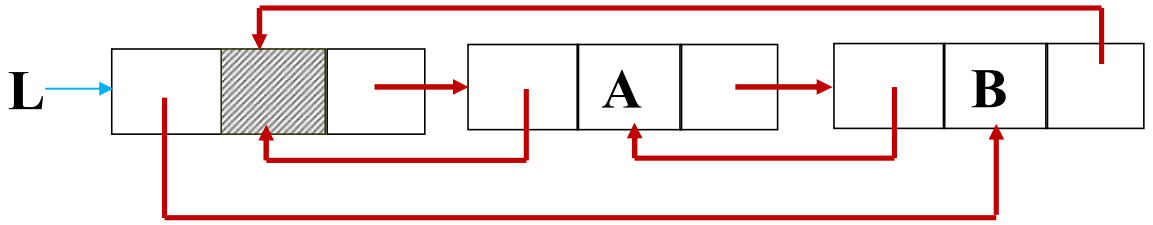




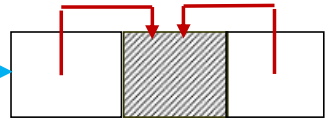
非空双向循环链表：

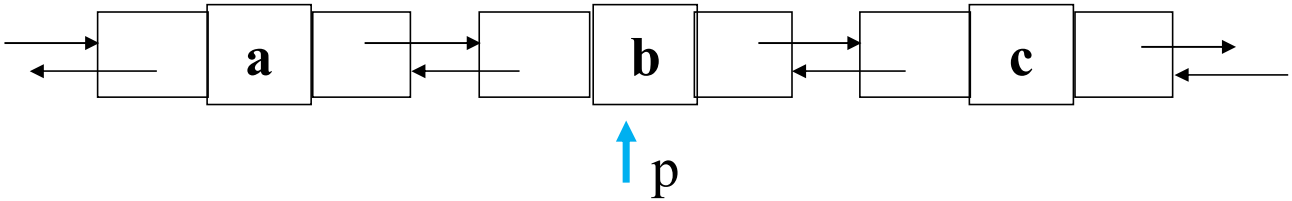
2.3 线性表的链式存储与实现

非空双向循环链表：



空双向循环链表：






```
p->prior->next ⇔  p ⇔ p->next->proir;
```

2.3 线性表的链式存储与实现

小结：

- 链表的优缺点
  - 优点
    - 需要附加空间
    - 不需要提前分配空间
  - 缺点
    - 查找需要循环
    - 插入删除简单

2.4 线性表的应用实例

2.4 线性表的应用实例

一元多项式的表示及相加

$$P_n(x) = P_0 + P_1x + P_2x^2 + \cdots\cdots + P_nx^n$$

可用线性表P表示 $P = (P_0, P_1, P_2, \cdots\cdots, P_n)$

但对S(x)多项式浪费空间 $S(x) = 1 + 3x^{1000} + 2x^{20000}$

一般 $P_n(x) = P_1x^{e1} + P_2x^{e2} + \cdots\cdots + P_mx^{em}$

其中 $0 \le e1 \le e2 \cdots\cdots \le em$（$P_i$为非零系数）

用数据域含两个数据项的线性表表示 $((P_1, e1),(P_2, e2),\cdots\cdots(P_m, em))$

其存储结构可以用顺序存储结构，也可以用单链表

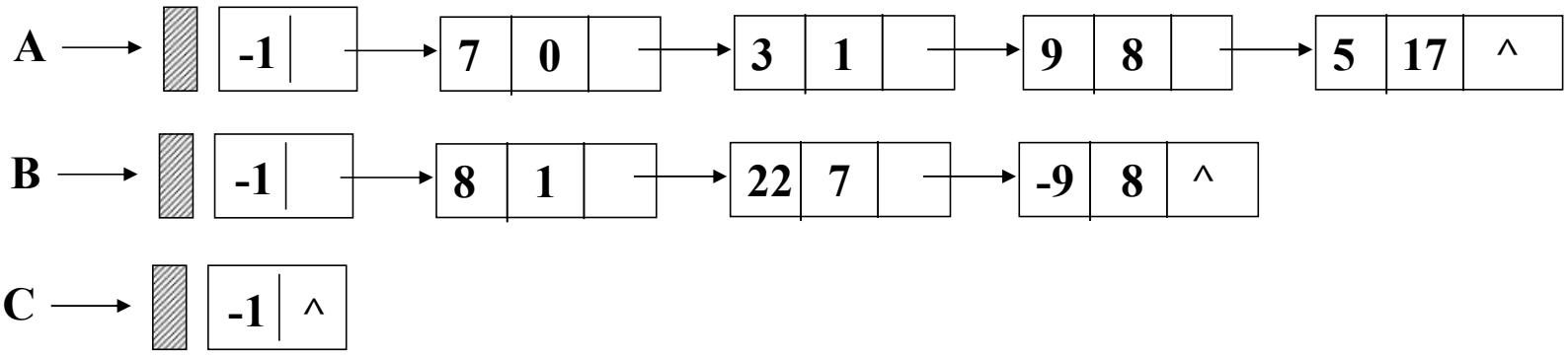
2.4 线性表的应用实例

单链表的结点定义

```
typedef    struct node
{    int       coef,exp;
     struct node   *next;
}JD;
```

$$A(x) = 7 + 3x + 9x^2 + 5x^{17}$$
$$B(x) = 8x + 22x^7 - 9x^8$$
$$C(x) = A(x) + B(x)$$

coef:用来记录系数
exp：用来记录指数
next：用来记录下一项元素的地址



2.4 线性表的应用实例

运算规则

假设：p,q分别指向A,B中某一结点，p,q初值是第一结点

比较 p->exp与 q->exp

- p->exp < q->exp: p结点是和多项式中的一项
  p后移，q不动
- p->exp > q->exp: q结点是和多项式中的一项
  将q插在p之前，q后移，p不动
- p->exp = q->exp: 系数相加
  - =0：从A表中删去p，
    释放p,q，p,q后移
  - ≠0：修改p系数域，
    释放q，p,q后移

直到p或q为NULL

- 若q==NULL，结束
- 若p==NULL，将B中剩余部分连到A上

本章 小结

线性表实现方法的比较

- 顺序表的主要优点
  - 没有使用指针，不用花费额外开销
  - 线性表元素的读访问非常简洁便利
- 链表的主要优点
  - 无需事先了解线性表的长度
  - 允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况
- 结论
  - 顺序表是存储静态数据的不二选择
  - 链表是存储动态变化数据的良方

线性表实现方法的比较

- 顺序表
  - 插入、删除运算时间代价O(n)，查找则O(1)
  - 预先申请固定长度的数组
  - 如果整个数组元素很满，则没有结构性存储开销
- 链表
  - 插入、删除运算时间代价O(1)，但找第 i 个元素运算时间代价O(n)
  - 存储利用指针，动态地按照需要为表中新的元素分配存储空间
  - 每个元素都有结构性存储开销

线性表应用场合的选择

- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
  - 线性表的最大长度也是一个重要限制因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

线性表存储密度

n表示线性表中当前元素的数目，
P表示指针的存储单元大小（通常为4bytes）
E表示数据元素的存储单元大小
D表示可以在数组中存储的线性表元素的最大数目

- 空间需求
  - 顺序表的空间需求为DE
  - 链表的空间需求为n(P+E)
- n的临界值，即n > DE/(P+E)
  - n越大，顺序表的空间效率就更高
  - 如果P = E，则临界值为n = D/2

线性表应用场合的选择

- 顺序表
  - 结点总数目大概可以估计
  - 线性表中结点比较稳定（插入删除少）
  - n > DE/(P+E)
- 链表
  - 结点数目无法预知
  - 线性表中结点动态变化（插入删除多）
  - n < DE/(P+E)

线性表应用习题1

- 假定一个线性表采用顺序表表示，要求删除线性表中所有值为x的元素，要求时间复杂度为O(n)，空间复杂度为O(1);

线性表应用习题2

- 假定一个线性表采用链表表示，要求访问离最后一个元素相隔为r的结点，要求时间复杂度为O(n)。

线性表应用习题3

- 假定一个线性表采用顺序表表示，要求以第一个元素为基准，比它小的元素全部移到它的前面，比它大的元素全部移到后面，要求时间复杂度为O(n)，空间复杂度为O(1);

线性表应用习题4

- 假定一个线性表采用顺序表表示，要求线性表中所有的奇数全部排在偶数的前面，要求时间复杂度为O(n)，空间复杂度为O(1);

线性表应用习题5

- 假定一个线性表采用链表L表示，要求创建两个链表，其中：L中的奇数结点放在L1链表里，偶数结点放在L2链表里。

线性表应用习题6

- 假定一个线性表采用链表L表示，要求删除结点数最大的结点，要求时间复杂度为O(n)，空间复杂度为O(1)，只能遍历一次链表;

线性表应用习题7

- 创建一个链表，要求结点的数值以增序排列。

线性表应用习题8

- 设有一个双链表，每个结点除了有prior，data，next三个域外，还有一个访问频度域freq，在链表启用之前，所有结点的频度域的值均为0。
- 每当在链表进行一次locateNode（L，x）操作的时候，令元素x的访问频度+1，并调整表中结点的顺序，按照频度的非递增顺序排列。
- 试给出locateNode（L，x）算法的实现。

线性表应用习题9

- 设有ABC三个带头结点的线性链表，它们的结点均按照元素值从小到大非递减排列（可能存在两个以上值相同的结点）；

- 编写出算法，对链表A进行处理，使得处理之后的链表A中仅留下3个表中均包含的数据元素的结点，且没有值相同的结点，并释放所有无用的结点空间。

- 要求：算法的复杂度为O(m+n+p)，其中m，n，p分别代表ABC三个链表的长度。

线性表应用习题10

- 设有一个不带头结点的单链表list，每个结点有data，next两个个域。
- 已知初始情况下，链表list无序排列。
- 请你给出一个算法，将该链表按结点数据域的值的大小，将其从大到小进行重新链接。
- 要求：在链接过程中不得使用该链表以外的任何链接结点空间；

本章学习要点

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和链式存储结构。用前者表示的线性表简称为顺序表，用后者表示的线性表简称为链表。
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。