数据结构

北京邮电大学 信息安全中心

武 斌

上次课内容

上次课（线性表<上>）内容：

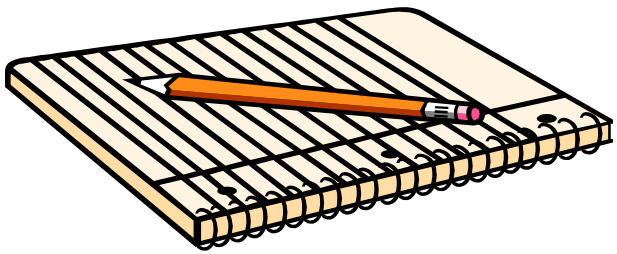
- 了解线性表的概念及其逻辑结构特性
- 理解顺序存储结构的描述方法
- 顺序表的基本操作及算法实现
- 顺序表的优缺点

本次课程学习目标

学习完本次课程，您应该能够：

- 掌握线性链表的存储结构
- 理解链式存储结构的描述方法
- 掌握线性链表的算法实现
- 分析链式存储结构的时间复杂度

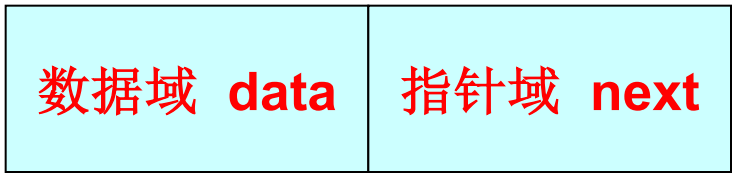
线性表的链式表示和表现



- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和表现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加
- 2.5 有序表

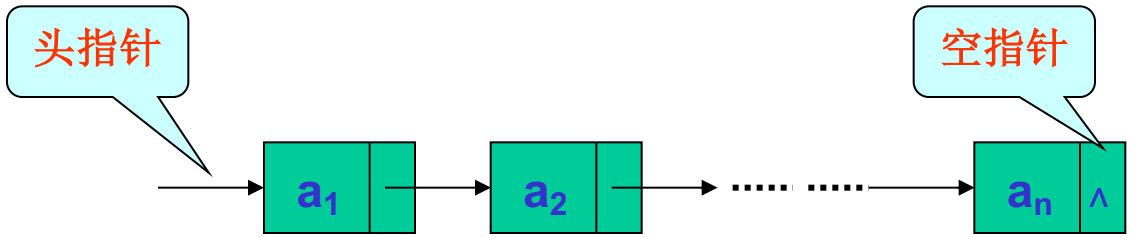
单链表和指针

- 线性表的**链式存储表示**的**特点**是用一组**任意的**存储单元存储线性表的数据元素。
  - ➔ 这组存储单元可以是连续的，也可以是不连续的
- 因此，为了表示每个数据元素 a_i 与其直接后继数据元素 a_{i+1} 之间的逻辑关系，对数据元素 a_i 来说，除了**存储其本身的信息**之外，还需存储一个**指示其直接后继的信息**(即直接后继的存储位置)。
- 这两部分信息组成一个“**结点**”(如下图所示)，表示线性表中一个数据元素。



单链表和指针

- 由分别表示 $a_1, a_2, \ldots, a_n$ 的 N 个结点依次相链构成的链表，称为**线性表的链式存储表示**。
- 由于此类链表的每个结点中**只包含一个指针域**，故又称**单链表**或**线性链表**，如下图所示。

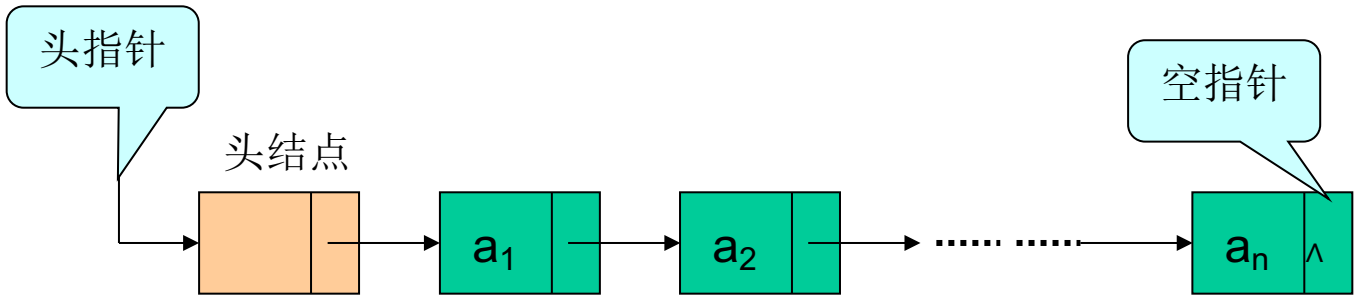


单链表和指针

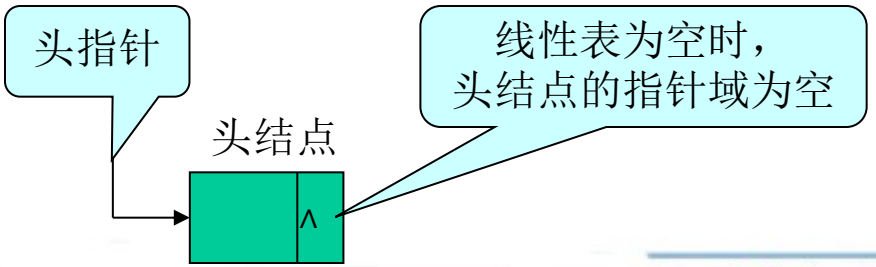
- 和顺序表类似，在链式存储结构中仍**以第一个数据元素的存储地址作为线性表的基地址**，通常称它为**头指针**，线性表中所有数据元素都可以从头指针出发找到。
- 因为线性表的最后一个数据元素没有后继，因此，在单链表中**最后一个结点**中的**“指针”**是一个特殊的值 “NULL”（在图上用∧表示），通常称它为“**空指针**”。

单链表和指针

- 为了便于处理一些特殊情况，在第一个结点之前附加一个“头结点”，令该结点中指针域的指针指向第一个元素结点，并令头指针指向头结点，如下图所示。



❊**注意：**若线性表为空，在不带头结点的情况下，头指针为空(NULL)，但在带头结点的情况下，链表的头指针不为空，而是其头结点中指针域的指针为空，如下图所示。



单链表和指针

- 用 C 语言中的“结构指针”来描述链表结构

```
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
} LNode，*LinkList;
```

若设 **LNode *p，*q;**

LinkList H;

则 p，q 和 H 均为以上定义的**指针型变量**。若 p 的值非空，则表明 p 指向某个结点，p->data 表示 p 所指结点中的**数据域**，p->next 表示 p 所指结点中的**指针域**，若非空，则指向其“后继”结点。

单链表和指针

- 指针型变量只能作同类型的指针赋值与比较操作。并且，指针型变量的“值”除了由同类型的指针变量赋值得到外，都必须用 C 语言中的动态分配函数得到。
  - 例如，p = new LNode; 表示在运行时刻系统动态生成了一个 LNode 类型的结点，并令指针 p “指向”该结点。
  - 反之，当指针 p 所指结点不再使用，可用 delete p， 释放此结点空间。
- 链表是一个进行动态存储管理的结构，因此在初始化时不需要按照线性表实际所需最大容量进行预分配。

单链表中基本操作的实现

- 一、初始化操作
  - 初始化建一个空的链表即为建立一个只有头结点的链表。
- 算法2.12

```
void InitList(LinkList &L )
{     // 创建一个带头结点的空链表，L 为指向头结点的指针
      L = new LNode;
     if (!L) exit(1);       // 存储空间分配失败
      L->next = NULL;
} // InitList
```

✸ 此算法的时间复杂度为*O* (1)。

单链表中基本操作的实现

- 二、销毁结构操作

- 算法2.13

```
void DestroyList(LinkList &L)
{       // 销毁以L为头指针的单链表，释放链表中所有结点空间
    LinkList p = L;
        while(p)
        {   pTemp = p;
            p = p->next;
            delete pTe
         } // while
         L = NULL;
} // DestroyList
```

算法中为什么要加上 L=NULL 的语句？

虽然头结点占有的空间已经释放，但指针变量L中的值没有改变，为安全起见，置L为“空”，以防止对系统空间的访问。

✸ 此算法的时间复杂度为***O* (Listlength(L))**。

单链表中基本操作的实现

●三、存取元素操作

➔单链表是一种“顺序存取”的结构，即：为取第 i 元素，首先必须先找到第 i-1 个元素。因此不论 i 值为多少，都必须从头结点开始起“点数”，直数到第 i 个为止。头结点可看成是第0个结点。

✸演示 2-3-1

单链表中基本操作的实现

❊算法2.14

```
bool GetElem (LinkList L， int pos， ElemType &e )
    {
      // 若1≤pos≤LengthList(L)，则用 e 带回指针L指向头结点的单链表
      // 中第 pos 个元素的值且返回函数值为TRUE,否则返回函数值为FALSE
      p = L->next； j =1；              // 变量初始化，p 指向第一个结点
      while ( p && j< pos )
      {          // 顺结点的指针向后查找，直至 p 指到第pos个结点或 p 为空止
        p = p->next；  ++j；
      } // while
      if ( !p || j>pos ) return FALSE; // 链表中不存在第 pos 个结点
      e = p->data； ；                          // 取到第 pos 个元素
      return TRUE;
    } // GetElem
```

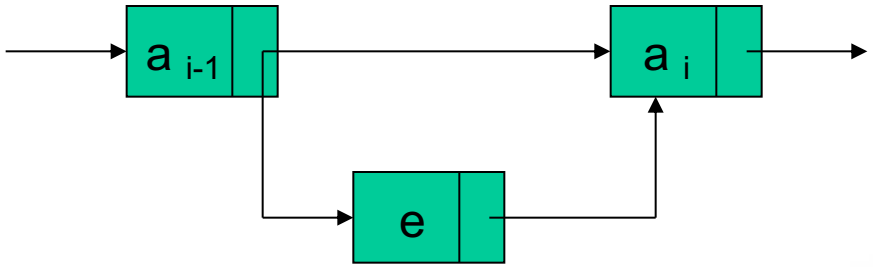
✸ 此算法的时间复杂度为***O* (ListLength(L))** 。

单链表中基本操作的实现

●四、插入元素操作

➔ 在线性表中“插入”一个元素时，使元素之间的关系<a_{i-1},a_i>改变为<a_{i-1},e>和<e,a_i>，只要修改相应数据元素的指针即可。因为新的元素插入在线性表的第i个元素之前，使得a_i不再是a_{i-1}的后继，而是新的元素e的后继，因此需要修改元素e和元素a_{i-1}所在结点的指针。

➔ 由此，**算法的基本思想**就是，首先找到第i-1个结点，然后修改相应指针。



单链表中基本操作的实现

❋算法2.15

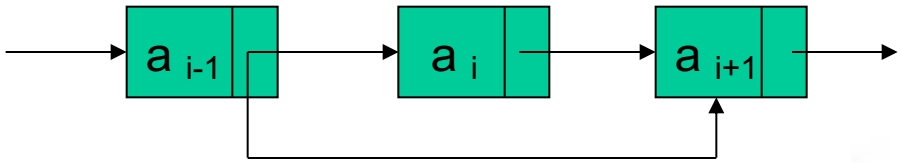
```
bool ListInsert (LinkList &L，int pos，ElemType e )
{       // 若1≤pos≤LengthList(L)+1，则在指针L指向头结点的单链表
        // 的第 pos 个元素之前插入新的元素 e，且返回函数值为 TRUE，
        // 否则不进行插入且返回函数值为 FALSE
        p=L；  j=0；
        while(p && j<pos-1)
        { // 查找第pos-1个结点，并令指针p指向该结点
          p=p->next；++j；
         } // while
        if (!p||j>pos-1)  return FALSE；   // 参数不合法
        s=new LNode；
        if (!s)  exit(1)；                  // 存储空间分配失败
        s->data=e；                         // 创建新元素的结点
        s->next=p-> next；  p->next=s；  // 修改指针
        return TRUE；
} // ListInsert
```

算法**时间复杂度**为***O* (ListLength(L))**。

演示2-3-1.2

单链表中基本操作的实现

- 五、删除元素操作
  - ➔ 和插入类似，由于删除元素 a_i 改变了元素之间的关系，使 a_{i+1} 不再是 a_i 的后继，而是 a_{i-1} 的后继，因此需要修改 a_{i-1} 元素所在结点的指针。
  - ➔ 因此在单链表中删除元素操作的**算法基本思想**和插入相同，也是：首先找到第 i-1 个结点，然后修改相应指针。



单链表中基本操作的实现

❊算法2.16

```
bool ListDelete (LinkList &L， int pos， ElemType &e)
{       // 若1≤pos≤LengthList(L)，则删除指针L指向头结点的单链表
        // 中第 pos 个元素并以 e 带回其值，返回函数值为 TRUE，
        // 否则不进行删除操作且返回函数值为 FALSE
        p = L； j = 0；
        while (p->next && j < i-1)
        {p = p->next； ++j；} // 寻找第pos个结点，并令p指向其前驱
        if (!(p->next) || j > i-1)
           return FALSE；                    // 删除位置不合理
        q = p->next；p->next = q->next；     // 修改指针
        e = q->data； delete(q)；            // 释放结点空间
        return TRUE;
} // ListDelete_L
```

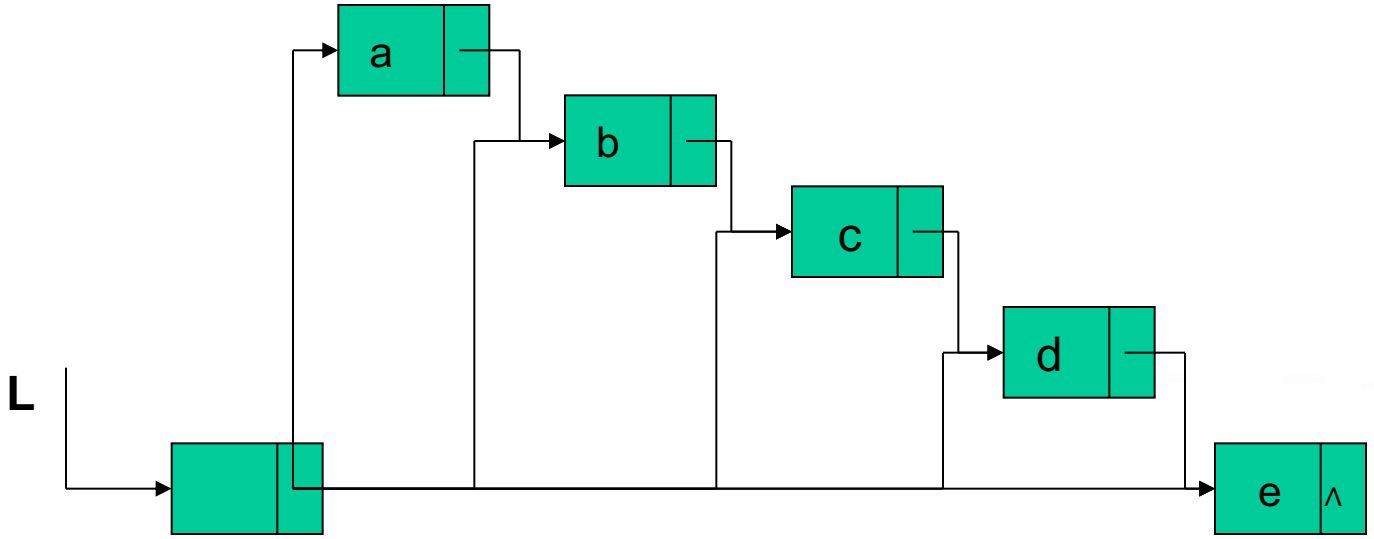
✸ 算法时间复杂度为***O* (ListLength(L))**。

✸演示2-3-1.1

单链表其它算法举例

●**例2-7** 逆序创建链表

假设线性表(a_1, a_2,…, a_n)的数据元素存储在一维数组A[n]中，则**从数组的最后一个分量起，依次生成结点，并逐个插入到一个初始为“空”的链表中**。例如下图演示了线性表(a,b,c,d,e) 的逆序创建的过程。



●演示2-3-2

单链表其它算法举例

❊算法2.17

```
void CreateList_L(LinkList &L，  int n，  ElemType A[ ])
{    // 已知数组 A 中存有线性表的 n 个数据元素，
     // 逆序建立带头结点的单链表。
     L = new LNode；
     if (!L) exit(1)；                  // 存储空间分配失败
     L->next = NULL；                   // 先建立一个带头结点的空的单链表
     for (i = n；  i > 0；   --i)
     {  p = new LNode；
        if (!p) exit(1)；               // 存储空间分配失败
           p->data = A[i-1]；                        // 赋元素值
           p->next = L->next；L->next = p；  // 插入在头结点之后
     } // for
} // CreateList_L
```

✸ 算法**时间复杂度**为***O* (ListLength(L))**。

单链表其它算法举例

- **例2-8** 以链表作存储结构解**例2-5**的问题，即将线性表（$a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n$）改变成（$b_1, b_2, \ldots, b_n, a_1, a_2, \ldots, a_m$）。

➔**解题分析**：

因为对链表来说，“插入”和“删除”仅需修改指针即可完成，并且由于前 m 个元素之间和后 n 个元素之间的链接关系分别都不需要改变，则算法的实际操作为：

(1) 从链表中删除($a_1, a_2, \ldots, a_m$)；

(2) 将($b_1, b_2, \ldots, b_n$)链接到头结点之后；

(3) 将($a_1, a_2, \ldots, a_m$)链接到 b_n 之后。

- 演示2-3-3
- 具体算法如下页**算法2.18**所示。

单链表其它算法举例

算法2.18

```
void exchange_L(LinkList &L，int m )
{    // 本算法实现单链表中前 m 个结点和后 n 个结点的互换
     if ( m && L->next )              // 链表不空且 m!=0
     {   p = L->next；  k = 1；
         while( k< m && p )           // 查找 am 所在结点
         {  p = p->next；  ++k；  } // while
         if (p && p->next)            // n!=0 时才需要修改指针
         {    ha = L->next;                  // 以指针 ha 记 a1 结点的位置
              L->next = p->next;             // 将 b1 结点链接在头结点之后
              p->next = NULL;                  // 设 am 的后继为空
              q = L->next;                   // 令q 指向 b1 结点
              while (q->next)  q = q->next；  // 查找 bn 结点
              q->next = ha；  } // 将 a1 结点链接到 bn 结点之后
     } // if(m)
} // exchange_L
```

✸ 算法**时间复杂度**为***O* (ListLength(L))**。

单链表其它算法举例

- 从以上对链表的各种操作的讨论可知，链式存储结构的优势在于：
  - 能有效利用存储空间；
  - 用“指针”指示数据元素之间的后继关系，便于进行“插入”、“删除”等操作；
- 而其劣势则是不能随机存取数据元素。
  - 同时，它还丢失了一些顺序表有的长处，如线性表的“表长”和数据元素在线性表中的“位序”，在上述的单链表中都看不见了。
  - 又如，不便于在表尾插入元素，需遍历整个表才能找到插入的位置。

单链表其它算法举例

- 例2-9 编写算法删除单链表中"多余"的数据元素，即使操作之后的单链表中所有元素的值都不相同。

解题分析：

可以和顺序表采用同样算法，即设想新建一个链表，然后顺序考察原链表中每一个结点的数据元素，在“新表”中进行查找，如果有相同的则舍弃之，否则就插入到新表中。

- 演示

单链表其它算法举例

```
void purge_L(LinkList &L )
{ // 删除单链表L中的冗余元素，即使操作之后的单链表中只保留
  // 操作之前表中所有值都不相同的元素
  p = L->next;      L->next = NULL;   // 设新表为空表
  while ( p )                         // 顺序考察原表中每个元素
  {   succ = p->next;                 // 记下结点 *p 的后继
    q = L->next;                      // q 指向新表的第一个结点
    while( q && p->data!=q->data )
        q = q->next;  //在新表中查询是否存在与p->data相同的元素
    if ( !q ){                        // 将结点 *p 插入到新的表中
        p->next = L->next;L->next = p;}
    else delete p;                    // 释放结点 *p
    p = succ;
  } // for
} // purge_L
```

- 此算法的时间复杂度为**O (ListLength^2(L))**。

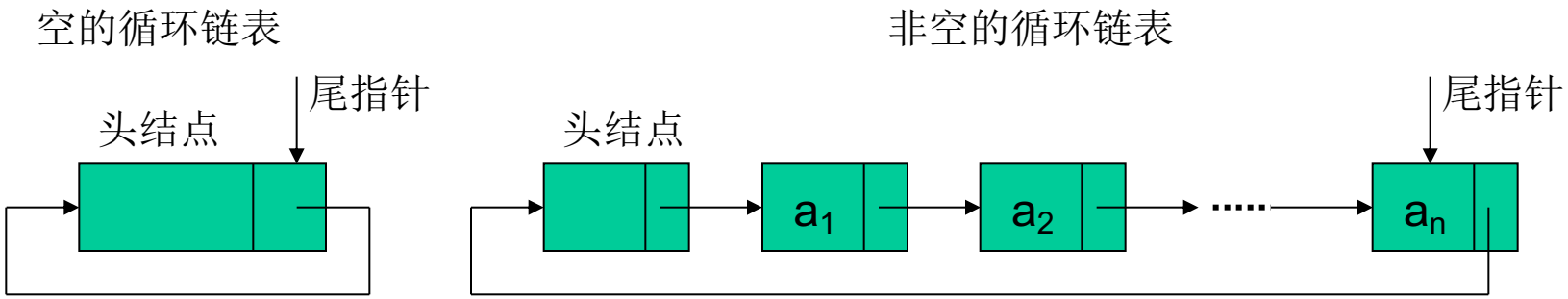
静态链表

- 用数组描述的链表
  - 在不设指针类型的高级程序设计语言中使用链表结构。
  - 数组的第零分量表示头结点，每个结点的指针域表示下一结点在数组中的相对位置。
- 一维数组描述线性链表

```
#define MAXSIZE 1000
    typedef struct {
            ElemType data;
        int             cur;
    } component，SLinkList[MAXSIZE];
```

循环链表*

- **循环链表(Circular Linked List)**是线性表的另一种形式的链式存储表示。它的特点是表中最后一个结点的指针域指向头结点，整个链表成为一个由链指针相链接的环，并且设立尾指针设成指向最后一个结点。空的循环链表由只含一个自成循环的头结点表示。



❊循环链表的操作和单链表基本一致，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"。

双向链表*

- 双向链表(Double Linked List)的特点是其结点结构中含有两个指针域，其一指向数据元素的“直接后继”，另一指向数据元素的“直接前驱”，用 C 语言描述如下：

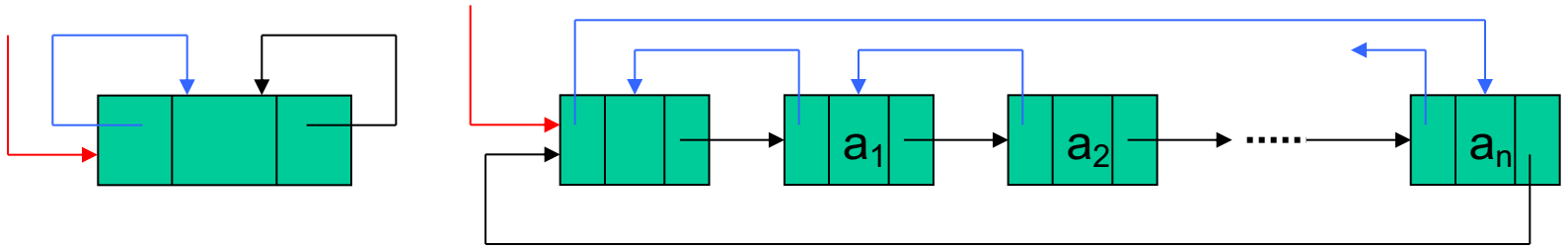
```
typedef struct DuLNode {
        ElemType data;
        struct DuLNode *prior;
        struct DuLNode *next;
} DuLNode,  *DuLink;
```

双向链表*

- 与单链表类似，双向链表也是由指向头结点的头指针唯一确定，若将头尾结点链接起来则构成双向循环链表。空的双向循环链表则由只含一个自成双环的头结点表示。

空的循环链表

非空的循环链表



- 演示

双向链表*

- 在双向链表上进行操作基本上和单向链表相同，例如，查找结点也是要从头指针指示的头结点开始，但插入和删除时必须同时修改两个方向上的指针，它们的算法分别如下所示。

算法2.19

```
void ListInsert_DuL(DuLink &L， DuLNode* p， DuLNode* s )
{
      // 在带头结点的双向循环链表 L 中结点 *p 之后插入结点 *s
       s->next = p->next；  p->next = s；
    s->next->prior = s；  s->prior = p；
}// ListInsert_DuL
```

- 演示

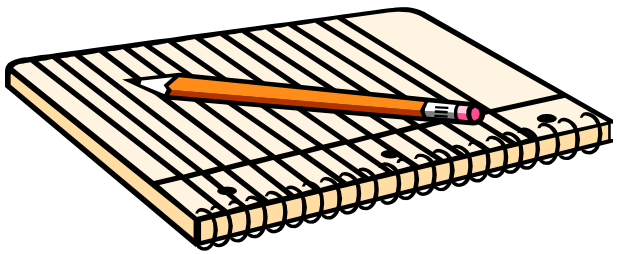
双向链表*

❋算法2.20

```
void ListDelete_DuL(DuLink &L， DuNode* p， ElemType &e)
{
    // 删除带头结点的双向循环链表L中结点 *p 的后继，
    // 并以 e 返回它的数据元素
    q = p->next； e = q->data；
    p->next = q->next；
    p->next->prior = p；
    delete q；
} // ListDelete_DuL
```

● 演示

一元多项式的表示及相加***



- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和表现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加
- 2.5 有序表

一元多项式的表示及相加

一元多项式$P_n(X)$：

$$P_n(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

用线性表表示为：

$$\mathrm{P} = (p_0, p_1, p_2, \cdots, p_n)$$

$Q_n(X)$表示为：

$$Q = (q_0, q_1, q_2, \cdots, q_n)$$

两个多项式的和：

$$R_n(x) = P_n(x) + Q_n(x)$$

用线性表表示为：

$$\mathrm{R} = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \cdots, p_m + p_m, p_{m+1}, \cdots, p_n)$$

可以看出，如果对P，Q和R使用顺序存储结构，
则可以非常简单的表示一元多项式的相加。

一元多项式的表示及相加

- 请注意如下一元多项式：

$$S(x)=1+3x^{2}+2x^{88888}$$

- 该多项式幂次很高，但是项数很少，如果采用顺序存储结构表示，则需要长度为88888的顺序表来表示，从而浪费了大量的存储空间，所以可以考虑使用链式存储结构。

一元多项式的表示及相加

- 一元多项式

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + p_3 x^{e_3} + \cdots p_m x^{e_m}$$

- 满足

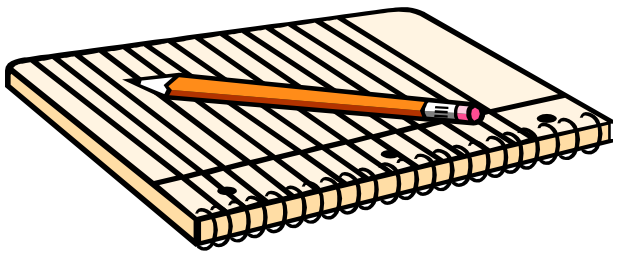
$$0 \le e_1 < e_2 < \cdots e_m = n$$

- 可以表示为：

$$((p_1, e_1), (p_2, e_2), \cdots (p_m, e_m))$$

- 其中每个元素有两个数据项：（系数项和指数项）
- 两个多项式的相加则可表示为指数项相同的数据元素对应的系数项相加。

有序表* * *



- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和表现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加
- 2.5 有序表

有序表的定义

- 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$a_i \geq a_{i-1}$或$a_i \leq a_{i-1}$($i = 2,3,\ldots,n$)，则称该线性表为**有序表(Ordered List)**。

  有序表的“有序”特性可以给某些操作带来很大方便，在某些应用问题中，如果用有序表而不是线性表将使算法的时间复杂度下降。

有序表

- 例2-10 编写算法删除有序顺序表中"多余"的数据元素，即使操作之后的有序顺序表中所有元素的值都不相同。
- 演示
- 算法2.24

```
void purge_OL(SqList &L )
{
  // 删除有序顺序表L中的冗余元素，即使操作之后的有序顺序表中
  // 只保留操作之前表中所有值都不相同的元素
  k = -1;                                  // k 指示新表的表尾
  for (i=0; i<L.length-1; ++i)   // 顺序考察表中每个元素
  {
    if ( k==-1 || L.elem[k]!=L.elem[i])
    L.elem[++k] = L.elem[i]; // 在新表中不存在和L.elem[i]相同的元素
  } // for
  L.length = k+1;                          // 修改表长
} // purge_OL
```

- 显然，此算法的时间复杂度为**O (ListLength(L))**。

有序表

- **例2-11** 已知两个链表（头指针分别为 La 和 Lb）中的数据元素均自小至大有序，编写算法将这两个链表归并为一个链表。
- 通常容易想到的这个题的做法是，将 Lb 表中的各个结点插入到 La 表中的相应位置中去。即按照有序关系首先查找插入位置，然后修改相应指针。另一种做法是，受例2-9中分析的启发，试设想新建一个空的链表，然后将已知两个链表中的结点依从小到大的次序逐个插入到这个新的链表中。
- 演示

有序表

- 算法2.25

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)
{
  // 已知单链线性表 La 和 Lb 的元素按值非递减排列。本算法
  // 归并 La 和 Lb 得到新的单链线性表 Lc，Lc 的元素也按
  // 值非递减排列。操作之后 La 和 Lb 消失
  pa = La->next; pb = Lb->next;
  Lc = rc = new LNode;              // 建空表，Lc 为头指针
  while (pa && pb)
  {
    if (pa->data <= pb->data)
    {                                // 将 *pa 插入Lc表，指针 pa 后移
      rc->next = pa; rc = pa; pa = pa->next;
    } // if
    else
    {                                // 将 *pb 插入Lc表，指针 pb 后移
      rc->next = pb; rc = pb; pb = pb->next;
    } // else
  } // while
```

有序表的基本操作

●有序表的基本操作

有序表类型的基本操作和线性表类型中定义的基本操作基本相同，但由于 LocateElem 中函数参数 compare 的类型不同，（在有序表中，元素相互比较的结果将产生三种结果："小于"、"等于"和"大于"），则该函数的定义和实现的算法也就和线性表中的不同。

LocateElem(L, e, &q, int(*compare)(ElemType,ElemType))

初始条件：有序表L已存在，compare为有序判定函数。

操作结果：若有序表L中存在元素 e，则 q 指示L中第一个值为 e 的元素的位置，并返回函数值TRUE；否则 q 指示第一个大于 e 的元素的前驱的位置，并返回函数值 FALSE。

此外，在有序表中进行"插入"操作时必须保持表的有序性。也就是说，在有序表中进行插入时首先要查找插入位置，显然，上述函数返回的位置即为插入位置，即应插入在 q 指示的元素之后。

有序链表类型

- 有序链表类型

```
//   结构定义
     typedef struct LNode { // 结点结构
       ElemType data;
       struct LNode *next;
     } *SLink;

typedef struct {              // 链表结构
   SLink head,                    // 指向有序链表中的头结点
         tail,                    // 指向有序链表中最后一个结点
         curPtr;                  // 指向操作的当前结点，称为"当前指针"
   int length,                    // 指示有序链表的长度
       curPos;                    // 指示当前指针所指结点的位序
} OrderedLinkList;
```

基本操作接口

- // 基本操作接口(函数声明)
  - bool MakeNode(SLink &p, ElemType e);
    - // 生成一个数据元素和 e 相同的新结点 *p，并返回TRUE,若存储分配失败，
    - // 则返回 FALSE。
  - bool InitList(OrderedLinkList &L);
    - // 构造一个空的有序链表L，若存储分配失败，令L.head为空并返回FALSE，
    - // 否则返回 TRUE。
  - void DestroyList(OrderedLinkList &L);
    - // 销毁有序链表 L，L 不再存在。
  - bool ListEmpty (OrderedLinkList L);
    - // 若有序链表 L 为空表，则返回TRUE，否则返回 FALSE。
  - int ListLength(OrderedLinkList L);
    - // 返回有序链表L中元素个数。

基本操作接口

- SLink PriorPos(OrderedLinkList L);
  - // 移动有序链表L中当前指针到它当前所指结点的直接前驱并返回。
- SLink NextPos (OrderedLinkList L);
  - // 移动有序链表L中当前指针到它当前所指结点的直接后继并返回。
- bool GetPos (OrderedLinkList L, int pos) ;
  - // 若1≤pos≤LengthList(L)，则移动当前指针指向第 pos 个结点
  - // 且返回函数值为TRUE，否则不移动当前指针且返回函数值为FALSE。
- void GetCurElem(OrderedLinkList L, ElemType& e);
  - // 以 e 带回当前指针所指结点中的数据元素。
- bool LocateElem (OrderedLinkList L, ElemType e,
- int (*compare)(ElemType, ElemType));
  - // 若有序链表L中存在和e相同的数据元素,则当前指针指向第1个和e相同的结点，并返回TRUE，否则当前指针指向第一个大于e的元素的前驱，并返回FALSE。

基本操作接口

- void ListTraverse(OrderedLinkList L, void (*visit)());
  - // 依次对L的每个元素调用函数 visit()。一旦 visit() 失败，则操作失败。
- void ClearList(OrderedLinkList &L);
  - // 将有序链表L重置为空表，并释放原链表的结点空间。
- void SetcurElem(OrderedLinkList L , ElemType e);
  - // 将有序链表L中当前指针所指结点中的数据元素修改为和 e 相同。
- void InsAfter (OrderedLinkList &L, SLink s);
  - // 在有序链表L中当前指针所指结点之后插入一个新的结点 *s
  - // 并移动当前指针指向新插入的结点。
- bool DelAfter(OrderedLinkList &L, ElemType& e);
  - // 若当前指针所指非单链表L中最后一个结点，则删除当前指针所指结点之后的结点,以e带回它的数据元素并返回TRUE,否则不进行删除操作且返回FALSE。

部分操作的伪码

- bool MakeNode(SLink &p, ElemType e)

```
{
   // 生成一个数据元素和 e 相同的新结点 *p，并返回TRUE，
   // 若存储分配失败，则返回 FALSE。
   p = new LNode;
   if (!p) return FALSE;
   p->data = e; p->next = NULL;
   return TRUE;
}
```

部分操作的伪码

```
bool InitList( OrderedLinkList &L )
{
   // 构造一个空的有序链表 L，若存储分配失败，
   // L.head = NULL 并返回 FALSE，否则返回 TRUE。
   if ( MakeNode( L.head, 0 ) )
   { L.tail = L.curPtr = L.head;
     L.length= L.curPos = 0;
     return TRUE; }
   else
   { L.head = NULL;
     return FALSE; }
} // InitList
```

部分操作的伪码

```
bool GetPos (OrderedLinkList L, int pos )
{
   // 若1≤pos≤LengthList(L)，则移动当前指针指向第pos个结点,
   // 且返回函数值为TRUE,否则不移动当前指针且返回函数值为FALSE。
   if ( pos < 1 || pos > L.len )
      return FALSE;
   if ( L.curPos > pos )
   { L.curPtr = L.head -> next; L.curPos = 1; }
   while ( L.curPos < pos )
   { L.curPtr = L.curPtr -> next; ++L.curPos; }
   return TRUE;
}
```

部分操作的伪码

- bool LocateElem (OrderedLinkList L, ElemType e, int (*compare)(ElemType, ElemType))

```
{// 若有序链表L中存在和e相同的数据元素，则当前指针指向第1个和e相同的结点，
 //并返回 TRUE，否则当前指针指向第一个大于e 的元素的前驱，并返回FALSE。
   L.current = L.head; L.curPos = 0;
   while ( L.current -> next &&
                        compare( e,L.current -> next -> data )> 0 )
   {
      L.current = L.current -> next;       // 指针后移，继续查询
      L.curPos ++;
   }
   if ( L.current -> next &&
                        compare( e,L.current -> next -> data ) == 0 )
   {                                          // 查到和 e 相同元素，当前指针后移
      L.current = L.current -> next; L.curPos ++;
      return TRUE;
   }
   else return FALSE;                         // 当前指针所指后继元素大于 e
} // LocateElem
```

部分操作的伪码

```
void InsAfter (OrderedLinkList &L, SLink s )
{
   // 在有序链表L中当前指针所指结点之后插入一个新的结点 *s，
   // 并移动当前指针指向新插入的结点。
    s->next = L.cruPtr->next;
   L.curPtr -> next = s;
   if ( L.tail == L.curPtr )
      L.tail = s;              // 若新结点插入在尾结点之后，则修改尾指针
   L.curPtr = s;               // 移动当前指针
   ++L.curPos;                 // 当前指针所指结点的位序增1
   ++L.length;                 // 表长增 1
}
```

部分操作的伪码

```
bool DelAfter(OrderedLinkList &L, ElemType& e )
{
// 若当前指针所指非单链表L中最后一个结点，
// 则删除当前指针所指结点之后的结点，以 e 带回它的数据元素
// 并返回 TRUE，否则不进行删除操作且返回 FALSE。
   if ( L.curPtr == L.tail )
      return FALSE;
   p = L.curPtr -> next; e = p -> data;
   L.curPtr -> next = p -> next;                  // 修改当前结点的指针
   if ( L.tail == p )
      L.tail = L.curPtr;                          // 删除尾结点时修改尾指针
   delete p;                                      // 释放被删结点
   --L.length;                                    // 表长减1
   return TRUE;
} // DelAfter
```

有序表的应用例子

- 例2-12 假设以两个有序表分别表示集合A和B，试求集合C=A∪B。
  - 算法的基本思想为：顺序考察有序表A和B，比较当前考察的元素a_i和b_j，将两者之中值“较小”者插入到C表中。

解题分析：

假设在解题过程中，已由有序表

$A = (a_1, \cdots, a_{i-1}, a_i, \cdots, a_m)$ 和

$B = (b_1, \cdots, b_{j-1}, b_j, \cdots, b_n)$

求得有序表

$C = (c_1, \cdots, c_{k-1})$

当前C中所含元素是A的子集$\{a_1, \cdots, a_{i-1}\}$和B的子集$\{b_1, \cdots, b_{j-1}\}$的并集，那么下一个插入到有序表C中的元素应该是哪一个呢？

显然，

$$C_k = \begin{cases} a_i & \text{若 } a_i \le b_j \\ b_j & \text{若 } a_i > b_j \end{cases}$$

例如：假设

A = {3, 5, 6, 8, 9}

B = {2, 4, 5, 8, 10}

则 C = {2, 3, 4, 5, 6, 8, 9, 10}

有序表的应用例子

- 算法2.26 算法时间复杂度为：O (Listlength(A)+ListLength(B))

```
void union ( OrderLinkList A, OrderLinkList B, OrderLinkList &C )
{// 已知有序链表 A 和 B 分别表示两个集合， 本算法求得有序链表 C 中所含元素是 A 和 B 的并集
  if ( InitList(C) )    {                                  // 初始化建空表
    m = ListLength(A); n = Listlength(B);        // 分别求得表长
    i = 1; j = 1;
    while ( i <= m || j <= n )    {                  // 顺序考察表中元素
    if ( GetPos(A,i) && GetPos(B,j) )    {        // 两个表中都还有元素未曾考察到
        GetCurElem(A,ea); GetCurElem(B,eb );
         if ( ea <= eb ) {                                          // 插入和  相同的元素
           if ( !MakeNode( s,ea ) ) exit(1);   ++i;
             if ( ea == eb )   ++j;                                  // 舍弃B表中相同元素
           }          //GetPos(A,i) 和 GetPos(B,j) 都为"真"说明i和j都没有超出表长的范围。
        else {                                              // 插入和  相同的元素
           if ( !MakeNode( s,eb ) ) exit(1);                ++j;       }
     }//if    else if ( GetPos(A,i) )     {       // A表中尚有元素未曾插入
        GetCurElem( A,ea );
        if ( !MakeNode( s,ea ) ) exit(1);    ++i;  }//else
      else {                                                 // B表中尚有元素未曾插入
        GetCurElem( B,eb );
        if ( !MakeNode( s,eb ) ) exit(1);   ++j; }//else
      InsAfter(C,s);                                        // 插入到C表    }
} // union
```

有序表的应用例子

- 例2-13 假设以两个有序表分别表示集合A和B，试求集合C=A∩B。

解题分析：

集合求"交"和集合求"并"的解法几乎是完全相同的，只是在求交时只需要将A和B中相同的元素插入C表即可。

例如：假设

A = {3, 5, 6, 8, 9}

B = {2, 4, 5, 8, 10}

则 C = { 5, 8 }

有序表的应用例子

- 算法2.27 时间复杂度为：O (Listlength(A)+ListLength(B))

```
void Intersection (OrderLinkList A, OrderLinkList B, OrderLinkList &C)
{ // 已知有序链表 A 和 B 分别表示两个集合，本算法求得有序链表 C 中所含元素是 A 和 B 的交集
  if ( InitList(C) )  {      // 初始化建空表
    m = ListLength(A); n = Listlength(B); // 分别求得表长
    i = 1; j = 1;
    while ( i <= m && j <= n )  {   // 顺序考察表中元素
      if ( GetPos(A,i) && GetPos(B,j) ) {   // 两个表中都还有元素未曾考察
        GetCurElem( A,ea ); GetCurElem( B,eb );
        if ( ea < eb ) ++i;
        else if ( ea > eb ) ++j;
        else {   // 插入和a_i相同的元素
          if ( !MakeNode( s,ea ) ) exit(1);
          ++i;++j;
          InsAfter(C,s);
        } // else          } // if      } // while    } // if
} // Intersection
```

本章小结

- 顺序表是线性表的顺序存储结构的一种别称。它的特点是以“存储位置相邻”表示两个元素之间的前驱后继关系。因此，顺序表的优点是可以随机存取表中任意一个元素，其缺点是每作一次插入或删除操作时，平均来说必须移动表中一半元素。常应用于主要是为查询而很少作插入和删除操作，表长变化不大的线性表。 (上次课)
- 链表是线性表的链式存储结构的别称。它的特点是以“指针”指示后继元素，因此线性表的元素可以存储在存储器中任意一组存储单元中。它的优点是便于进行插入和删除操作，但不能进行随机存取，每个元素的存储位置都存放在其前驱元素的指针域中，为取得表中任意一个数据元素都必须从第一个数据元素起查询。由于它是一种动态分配的结构，结点的存储空间可以随用随取，并在删除结点时随时释放，以便系统资源更有效地被利用。这对编制大型软件非常重要，作为一个程序员在编制程序时必须养成这种习惯。
- 一元多项式的表示及相加
- 有序表

本章小结（续）

- 由于线性表是一种应用很广的数据结构，链表的操作又很灵活，因此，在C++等面向对象的程序设计语言中都已为程序员提供了链表类，读者在使用时应该首先充分了解它的操作接口。
- 在自己实现链表类时，正如课程中所述，应该为链表结构设置合适的数据成员和恰当的操作接口，以便使每个基本操作的时间复杂度在尽可能低的级别上。

本次课知识点与重点

- **知识点**

  链表、一元多项式的表示、有序表

- **重点和难点**

  - **单链表及其算法实现是本次课的重点和难点**。
  - 扎实的指针操作和内存动态分配的编程技术是学好本章的基本要求，
  - 分清链表中指针 p 和结点 *p 之间的对应关系，
  - 区分链表中的头结点、头指针和首元结点的不同
  - 循环链表、双向链表的特点等。