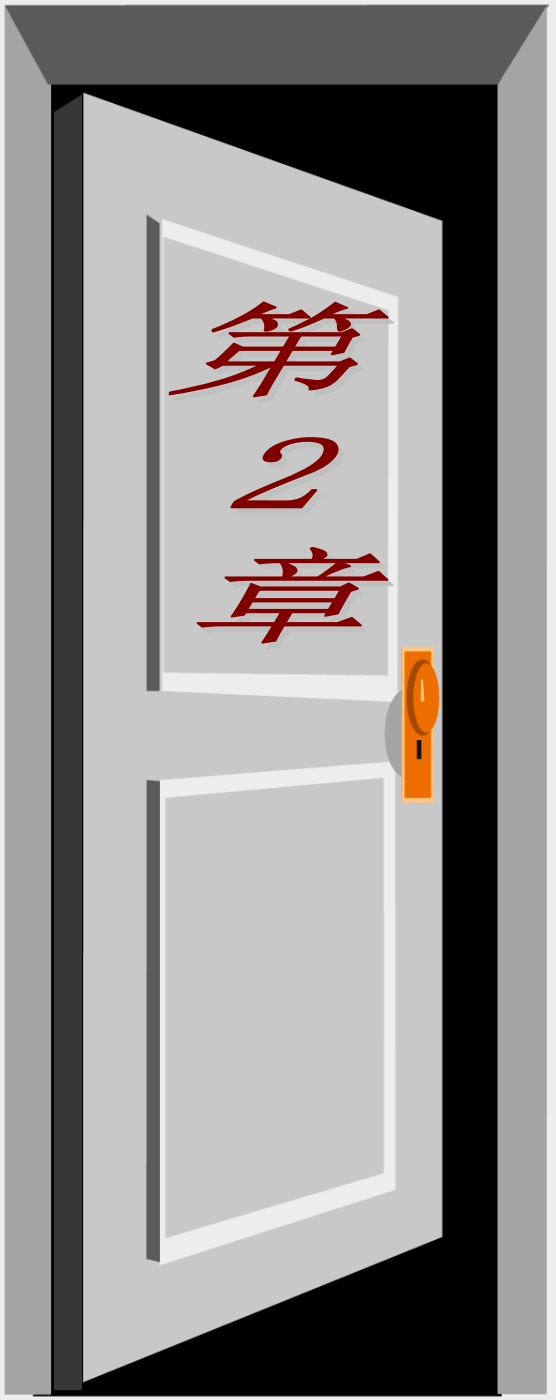
尚德机构

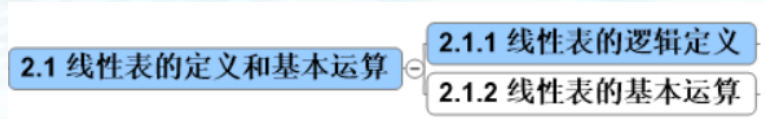
数据结构

主讲：王老师

学习是一种信仰！ IN LEARING WE TRUST



线性表



线性表（Linear List）是最简单和最常用的一种数据结构，它是由**n**个数据元素（结点）a_1，a_2，…a_n如组成的**有限序列**。其中，数据元素的个数n为表的长度。当**n为零时称为空表**，非空的线性表通常记为

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

这里的元素$a_i(a \le i \le n)$是一个抽象的符号，它可以是一个数或者一个符号，还可以是较复杂的记录。如一个学生、一本书等信息就是一个**数据元素**，它可以由若干个数据项组成。

例如，有学生基本情况如表2.1所示。

表2.1　学生情况

学号	姓名	性别	年龄	籍贯
Pb0812008	赵学民	男	21	内蒙古
Pb0805021	王一品	男	19	上海
Pb0801103	陈达晴	女	20	天津
Pb0823096	杨洋	男	18	广东
⋮	⋮	⋮	⋮	⋮

真题演练

下列选项中，不属于线性结构特征的是（　　）

A: 数据元素之间存在线性关系

B: 结构中只有一个开始结点

C: 结构中只有一个终端结点

D: 每个结点都仅有一个直接前趋

真题演练

下列选项中，不属于线性结构特征的是（　　）

A: 数据元素之间存在线性关系

B: 结构中只有一个开始结点

C: 结构中只有一个终端结点

D: 每个结点都仅有一个直接前趋

答案：D

线性表的逻辑特征	①有且仅有一个称为**开始元素**的a1，它**没有前趋**，仅有一个直接后继a2；
	②有且仅有一个称为**终端元素**的an，它**没有后继**，仅有一个直接前趋；
	③其余元素ai(2≤i≤n-1)称为**内部元素**，它们都有且**仅有一个直接前趋**ai-1和**一个直接后继**ai+1。

对于线性表，常见的基本运算有以下几种：

（1）置空表InitList (L)，构造一个空的线性表L。

（2）求表长ListLength (L)，返回线性表L中元素个数，即表长。

（3）取表中第i个元素GetNode (L, i)，若1≤i≤ListLength (L)，则返回第i个元素a_i。

（4）按值查找LocateNode (L, x)，在表L中查找第一个值为x的元素，并返回该元素在表L中的位置，若表中没有元素的值为x，则返回0值。

（5）插入InsertList (L, i, X)，在表L的第i元素之前插入一个值为x的新元素，表L的长度加1。

（6）删除DeleteList (L, i)，删除表L的第i个元素，表L的长度减1。

【例2.1】假设有两个线性表LA和LB分别表示两个集合A和B，现要求一个新集合(集合的并)。

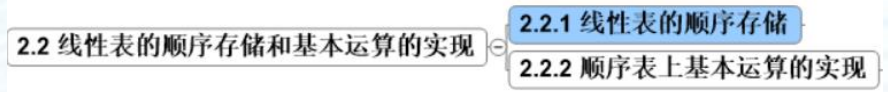
其算法思想：扩大线性表LA，将表LB中不在LA中出现的元素插入到LA中。只要从线性表LB中依次取出每个元素，按值在线性表LA中查找，若没查到则插入之。其算法描述如下：

```
void union ( Linear_List LA, Linear_List LB )
{                                                    //假定Linear_List是线性表类型
   n=ListLength ( LA );                              //求LA的表长
   for ( i=1; i<=ListLength ( LB ); i++) {
      x=GetNode ( LB, i );                           //取LB中第i个元素赋给x
      if ( LocateNode ( LA, x ) ==0 )
         InsertList ( LA, ++n, x );
   }
}
```

【例2.2】删除线性表L中重复的元素。

实现该功能的算法思想是：从表L的第一个元素（i=1）开始，逐个检查i位置以后的任一位置j，若两元素值相同，则从表中删除第j个元素，……，直到i移到当前表L的最后一个位置为止。其算法描述如下：

```
void purge ( Linear_List L )
{   i=1;    .
    while ( i<=ListLength ( L ) ) {
        x=GetNode ( L, i ); j=i+1;
        while ( j<=ListLength ( L ) ) {
            y=GetNode ( L, j );
            if(x==y)                          //结点x和结点y内容相同
                DeleteList ( L, j ) ;
            else
                j++;   }
        i++;
    }
}
```



线性表的顺序存储指的是将线性表的数据元素按其逻辑次序依次存入一组地址连续的存储单元里，用这种方法存储的线性表称为顺序表。

假设线性表中所有元素的类型是相同的，且每个元素需占用d个存储单元，其中第一个单元的存储位置（地址）就是该元素的存储位置。那么，线性表中第i+1个元素的存储位置LOC(a_{i+1})和第i个元素的存储位置LOC(a_i)有关系：

$$LOC(a_{i+1})=LOC(a_i)+d$$

一般来说，线性表的第i个元素A的存储位置为

$$LOC(a_i)=LOC(a_1)+(i-1)*d$$

其中，LOC(a_1)是线性表的第一个元素a_1的存储位置，通常称之为基地址。

线性表的这种机内表示称为线性表的顺序存储结构。它的特点是，元素在表中的相邻关系，在计算机内也存在着相邻的关系。

真题演练

将16个数据元素的线性表按顺序存储方式存储在数组中，若第一个元素的存储地址是1000，第6个元素的存储地址是1040，则最后一个元素的存储地址是（　　）

A: 1112

B: 1120

C: 1124

D: 1128

真题演练

将16个数据元素的线性表按顺序存储方式存储在数组中，若第一个元素的存储地址是1000，第6个元素的存储地址是1040，则最后一个元素的存储地址是（　　）

A: 1112

B: 1120

C: 1124

D: 1128

答案：B

真题演练

设顺序表首元素A[0]的存储地址是4000，每个数据元素占5个存储单元，则元素A[20]的起始存储地址是（　）

A: 4005

B: 4020

C: 4100

D: 4105

真题演练

设顺序表首元素A[0]的存储地址是4000，每个数据元素占5个存储单元，则元素A[20]的起始存储地址是（　）

A: 4005

B: 4020

C: 4100

D: 4105

答案：C

真题演练

设n个元素的顺序表中，若将第i（1≤i<n）个元素e移动到第j（1<j≤n，i<j）个位置，不改变除e外其他元素之间的相对次序，则需移动的表中元素个数是（　）

A: j-i-1

B: j-i

C: j-i+1

D: i-j

真题演练

设n个元素的顺序表中，若将第i（1≤i<n）个元素e移动到第j（1<j≤n，i<j）个位置，不改变除e外其他元素之间的相对次序，则需移动的表中元素个数是（　）

A: j-i-1

B: j-i

C: j-i+1

D: i-j

答案：C

真题演练

将12个数据元素保存在顺序表中，若第一个元素的存储地址是100，第二个元素的存储地址是105，则该顺序表最后一个元素的存储地址是（　）

A: 111

B: 144

C: 155

D: 156

真题演练

将12个数据元素保存在顺序表中，若第一个元素的存储地址是100，第二个元素的存储地址是105，则该顺序表最后一个元素的存储地址是（　）

A: 111

B: 144

C: 155

D: 156

答案：C

2.2.1 线性表的顺序存储


```
#define ListSize 100                //表空间的大小应根据实际需要来定义，这里假设为100
typedef int DataType;               //DataType的类型可根据实际情况而定，这里假设为int
typedef struct {
        DataType data [ ListSize ];  //数组data用来存放表结点
        int length;                  //线性表的当前表长（实际存储元素的个数）
} SeqList;
```

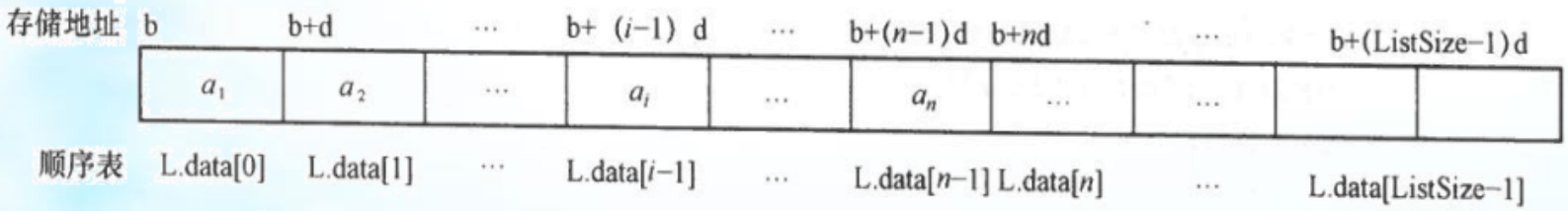


图 2.1　线性表的顺序存储表示

2.2.2二、顺序表上基本运算的实现 1. 插入运算

线性表的插入运算是指在线性表的第i-1个元素和第i个元素之间插入一个新元素x，使长度为n的线性表：

$(a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n)$

变为长度为n+1的线性表：

$(a_1, a_2, \ldots, a_{i-1}, x, a_i, \ldots, a_n)$

由于线性表逻辑上相邻的元素在物理结构上也是相邻的，因此在插入一个新元素之后，线性表的逻辑关系发生了变化，其物理存储关系也要发生相应的变化。除非i=n+1，否则必须将原线性表的第i、i+1、…、n个元素分别向后移动1个位置，空出第i个位置以便插入新元素x。

2.2.2 顺序表上基本运算的实现　1. 插入运算

插入算法描述如下：

```
void InsertList ( SeqList *L, int i, DataType x )
{  //在顺序表L中第i个位置之前插入一个新元素x
    int j;
    if ( i<1 || i>L->length+1 ) {
        printf ( "position error");
        return;
    }
    if ( L->length>=ListSize) {
        printf ( "overflow" );
        return;
    }
    for ( j=L->length-1;j>=i-1; j--)
        L->data[ j+1]=L->data[j];          //从最后一个元素开始逐一后移
    L->data[ i-1]=x;                       //插入新元素x
    L->length++;                           //实际表长加1
}
```

2.2.2 顺序表上基本运算的实现　1. 插入运算

一般情况下，在第i（1≤i≤n）个元素之前插入一个新元素时，需要进行n-i+1次移动。而该算法的执行时间主要花在for循环的元素后移上，因此该算法的时间复杂度不仅依赖于表的长度n，而且还与元素的插入位置i有关。当i=n+1时，for循环一次也不执行，无需移动元素，属于最好情况，其时间复杂度为O(1)；当i=1，循环需要执行n次，即需要移动表中所有元素，属于最坏情况，算法时间复杂度为O(n)。由于插入元素可在表的任何位置上进行，因此需要分析讨论算法的平均移动次数。

在等概率情况下插入，需要移动元素的平均次数为：

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

例如，假定一个有序表A=(23, 31, 46 ,54, 58, 67, 72, 88)，表长n=8。当向其中插入元素56时，此时i等于5，因此应插入到第i-1个位置上，从而需要将第i-1个元素及之后的所有元素都向后移动一位，将第i-1个元素位置空出来，插入新元素56。插入后的有序表为(23, 31, 46 ,54, 56 , 58, 67, 72, 88)。按上述移动次数的计算公式，可知本插入操作需要移动n-i+1=8-5+1=4次。

2. 删除运算

线性表的删除运算指的是将表中第i个元素删除，使长度为n的线性表

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

变为长度为n-1的线性表：

$(a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$

线性表的逻辑结构和存储结构都发生了相应的变化，与插入运算相反，插入是向后移动元素，而删除运算则是向前移动元素，除非i=n时直接删除终端元素，不需移动元素。

2. 删除运算

删除第i个元素的算法：

```
DataType DeleteList ( SeqList *L, int i )
{   //在顺序表L中删除第i个元素，并返回被删除元素
    int j;
    DataType x;              //DataType是一个通用类型标识符，在使用时再定义实际类型
    if ( i<1 || i>L->length ) {
        printf ( "position error" );
        exit ( 0 );                                  //出错退出处理
    }
    x=L->data [ i-1 ];                               //保存被删除元素
    for ( j=i; j<=L->length; j++)
        L->data[j-1]=L->data[ j ];                   //元素前移
    L->length--;                                     //实际表长减1
    return x;                                        //返回被删除的元素
}
```

2. 删除运算

该算法的时间复杂度分析与插入算法类似，删除一个元素也需要移动元素，移动的次数取决于表长n和位置i。当i=1时，则前移n-1次；当i=n时不需要移动，因此算法的时间复杂度为O(n)。由于算法中删除第i个元素是将从第i+1至第n个元素依次向前移动一个位置，共需要移动n-i个元素。同插入类似，假设在顺序表上删除任何位置上元素的机会相等，q_i为删除第i个元素的概率，则删除一个元素的平均移动次数为：

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

在顺序表上做删除运算，平均移动元素次数约为表长的一半，因此该算法的平均时间复杂度为O(n)。

真题演练

长度为n的顺序表，删除位置i上的元素(0≤i≤n-1)，需要移动的元素个数为（　）

A: n-i

B: n-i-1

C: i

D: i+1

真题演练

长度为n的顺序表，删除位置i上的元素(0≤i≤n-1)，需要移动的元素个数为（　）

A: n-i

B: n-i-1

C: i

D: i+1

答案：B

3. 顺序表上的其他运算举例

【例2.3】已知一长度为n顺序存储的线性表，试写一算法将该线性表逆置。

分析：不妨设线性表为($a_1, a_2, \ldots, a_n$)，逆置之后为($a_n, a_{n-1}, \ldots, a_1$)，并且表以顺序存储方式存储。实现其算法的基本思想是：先以表长的一半为循环控制次数，将表中最后一个元素同顺数第一个元素交换，将倒数第二个元素同顺数第二个元素交换，…，依此类推，直至交换完为止。为此可设计算法如下：

```
SeqList Converts ( SeqList L )
{   DataType x;
    int i, k;
    k=L.length/2;
    for ( i=0; i<k; i++) {
        x=L.data[ i ];
        L.data [ i ]=L.data [ L.length-i-1 ];
        L.data [ L.length-i-1 ]=x;
     }
     return L;
}
```

这个算法只需要进行数据元素的交换操作，其主要时间花在for循环上，显然整个算法的时间复杂度为O(n)。

3. 顺序表上的其他运算举例

【例2.4】试写一算法，实现在顺序表中找出最大值和最小值的元素及其所在位置。

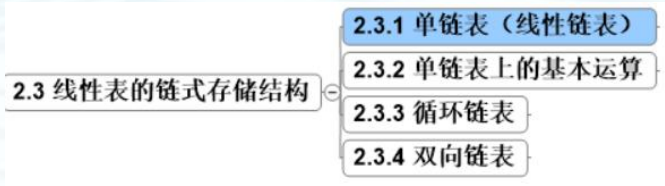
分析：如果在查找最大值和最小值的元素时各扫描一遍所有元素，则至少要比较2n次，可以使用一次扫描找出最大和最小值的元素。另外，在算法中要求带回求得的最大值和最小值元素及其所在位置，可用4个指针变量参数间接得到，也可以用外部变量实现，因为函数本身只可返回一个值。下面是采用指针变量参数来实现的：

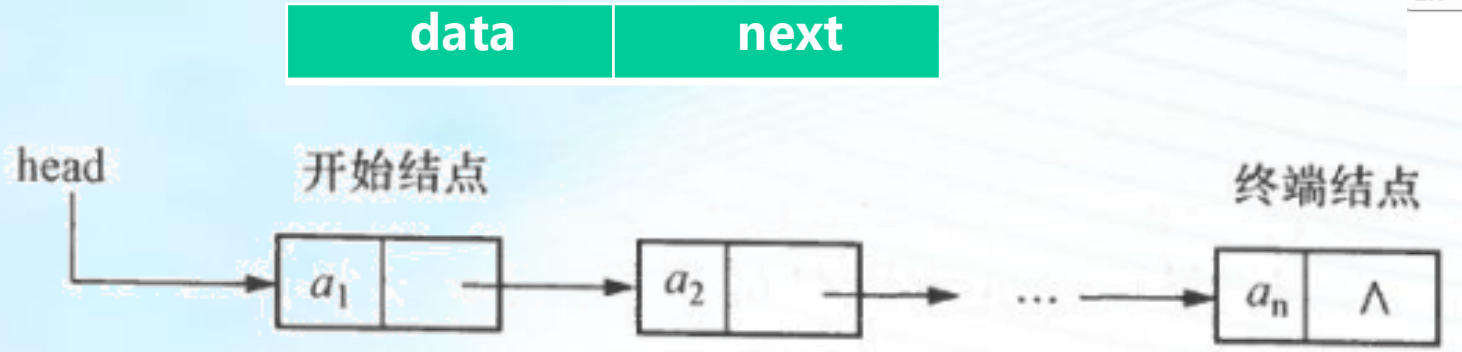
```
void MaxMin ( SeqList L, DataType *max, DataType *min, int *k, int *j )
{   int i;
    *max=L.data[ 0 ];
    *min=L.data [ 0 ];
    *k=*j=i;                    //先假设第一个元素既是最大值，也是最小值
    for ( i=1; i<L.length; i++)
        if ( L.data [ i ]>*max ) {
            *max=L.data [ i ]; *k=i ;
        }
        else if ( L.data [ i ]<*min ) {
            *min=L.data [ i ]; *j=i ;
        }
}
```

该算法的时间复杂度为O(n)。

线性表顺序存储结构的特点是，在逻辑关系上相邻的两个元素在物理位置上也是相邻的，因此可以随机存取表中任一元素。但是，当经常需要做插入和删除操作运算时，则需要移动大量的元素，而采用链式存储结构时就可以避免这些移动。然而，由于链式存储结构存储线性表数据元素的存储空间可能是连续的，也可能是不连续的，因而链表的结点是不可以随机存取的。

2.3.1一、单链表（线性链表）





用C语言中的结构类型描述线性表的链式存储结构如下：

```
typedef struct node {                    //结点类型定义
    DataType data ;                      //结点数据域
    struct node * next ;                 //结点指针域
} ListNode;
typedef ListNode * LinkList ;
ListNode * p ;                           //定义一个指向结点的指针变量
LinkList  head ;                         //定义指向单链表的头指针
```

2.3.2二、单链表上的基本运算　1. 建立单键表

（1）头插法建表

头插法建表是从一个空表开始，重复读入数据，生成新结点，将读入的数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。假设线性表中结点的数据域为字符型，其具体算法如下：

```
LinkList CreateListF ( )
{   LinkList head;
    ListNode *p ;
    char ch ;
    head=NULL;                          //置空单链表
    ch=getchar ( ) ;                    //读入第一个字符
    while ( ch!='\n') {                 //读入字符不是结束标志符时作循环
        p= ( ListNode * ) malloc (sizeof ( ListNode ) ) ;//申请新结点
        p->data=ch;                     //数据域赋值
        p->next=head;                   //指针域赋值
        head=p;                         //头指针指向新结点
        ch=getchar ( ) ;                //读入下一个字符
    }
    return head;                        //返回链表的头指针
}
```

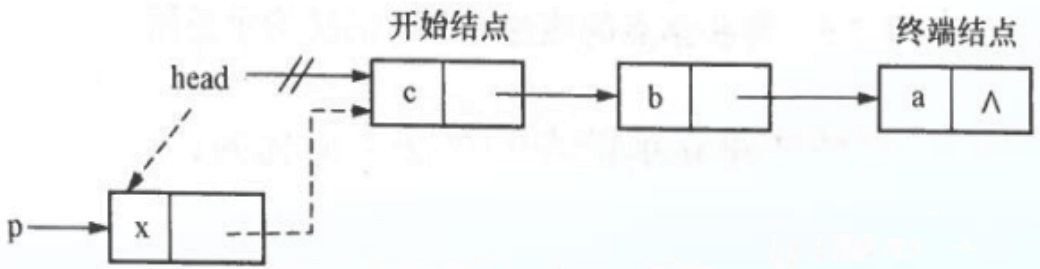
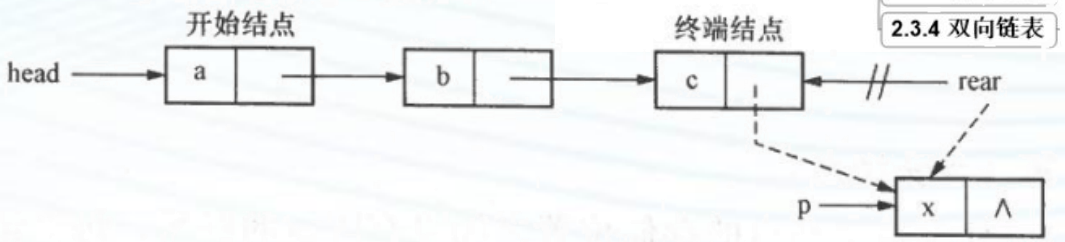


图 2.3　插入新结点*p 到单链表的表头

2.3.2 单链表上的基本运算　1. 建立单键表



（2）尾插法建表

```
LinkList CreateListR ( )
{   LinkList head, rear;
    ListNode *p ;
    char ch ;
    head=NULL; rear=NULL;                    //置空单链表
    ch=getchar ( ) ;                         //读入第一个字符
    while ( ch!= '\n' ) {                    //读入字符不是结束标志符时作循环
        p= ( ListNode * ) malloc ( sizeof ( ListNode ) ) ; //申请新结点
        p->data=ch ;                                       //数据域赋值
        if ( head==NULL ) head=p ;                         //新结点*p插入空表
        else rear->next=p ;                  //新结点*p插入到非空表的表为结点*rear之后
        rear=p;                              //表尾指针指向新的表尾结点
        ch=getchar ( ) ;                     //读入下一个字符
    }
    if ( rear!=NULL ) rear->next=NULL ;      //终端结点指针域置空
    return head ;
}
```

2.3.2 单链表上的基本运算　1. 建立单键表

为了简化算法，方便操作，可在链表的开始结点之前附加一个结点，并称其为头结点。带头结点的单链表结构图2.5所示。

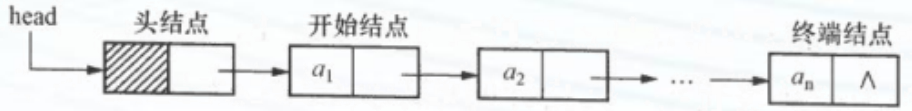
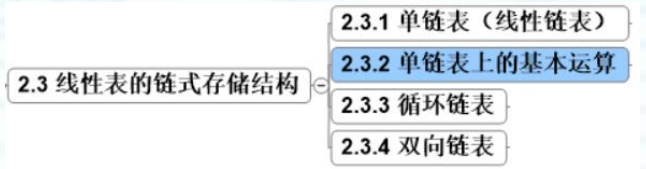


图 2.5　带头结点的线性链表存储结构示意图

```
LinkList CreateListR1 ( )
{   //尾插法建立带头结点的单链表算法
    LinkList head= ( ListNode * ) malloc (sizeof ( ListNode ) ); //申请头结点
    ListNode *p, *r ;
    DataType ch ;
    r=head ;                                          //尾指针初始指向头结点
    while ( ( ch=getchar ( ) ) != '\n' ) {
        p= ( ListNode * ) malloc ( sizeof ( ListNode ) ) ;      //申请新结点
        p->data=ch ;
        r->next=p;                                    //新结点连接到尾结点之后
        r=p ;                                         //尾指针指向新结点
    }
    r->next=NULL;                                     //终端结点指针域置空
    return head;
}
```

2. 查找运算（带头结点）



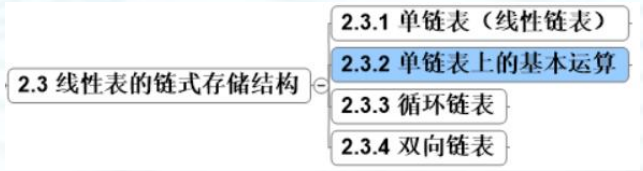
（1）按结点序号查找

在单链表中要查找第i个结点，就必须从链表的第1个结点（开始结点，序号为1）开始，序号为0的是头结点，P指向当前结点，j为计数器，其初始值为1，当p扫描下一个结点时，计数器加1。当时，指针P所指向的结点就是要找的结点。其算法如下：

```
ListNode * GetNodei ( LinkList head, int i )
{  //head为带头结点的单链表的头指针，i为要查找的结点序号
   //若查找成功，则返回查找结点的存储地址（位置），否则返回NULL
    ListNode *p; int j ;
    p=head->next ; j=1 ;                  //使p指向第一个结点，j置1
    while ( p!=NULL && j<i ) {            //顺指针向后查找，直到p指向第i个结点或p为空为止
          p=p->next ; ++j ;
    }
    if ( j==i )
          return p;
    else
          return NULL;
    }
```

在等概率情况下，平均时间复杂度为 $\frac{1}{n+1}\sum_{i=0}^{n} i = \frac{n}{2} = O(n)$

2. 查找运算（带头结点）



（2）按结点值查找

在单链表中按值查找结点，就是从链表的开始结点出发，顺链逐个将结点的值和给定值k进行比较，若遇到相等的值，则返回该结点的存储位置，否则返回NULL。按值查找算法要比按序号查找更为简单，其算法如下：

```
ListNode * LocateNodek ( LinkList head, DataType k )
{  //head为带头结点的单链表的头指针，k为要查找的结点值
   //若查找成功，则返回查找结点的存储地址（位置），否则返回NULL
   ListNode *p=head->next ;    //p指向开始结点
   while ( p && p->data!=k )   //循环直到p等于NULL或p->data等于k为止
      p=p->next ;              //指针指向下一个结点
   return p;          //若找到值为k的结点，则p指向该结点，否则p为 NULL
}
```

该算法的时间主要花在查找操作上，循环最坏情况下执行n次，因此其算法时间复杂度为O(n)。

3. 插入运算


```
void InsertList ( LinkList head, int i, DataType x )
{   //在以head为头指针的带头结点的单链表中第i个结点的位置上
    //插入一个数据域值为x的新结点
    ListNode *p, *s; int j ;
    p=head ; j=0 ;
    while (p!=NULL && j<i-1 ) { //使p指向第i-1个结点
        p=p->next ; ++j;
    }
    if ( p==NULL ) {                    //插入位置错误
        printf ( "ERROR\n") ; return ;
    }
    else {
        s= ( ListNode * ) malloc ( sizeof ( ListNode ) ) ;  //申请新结点
        s->data=x ; s->next=p->next ;
        p->next=s ;
    }
}
```

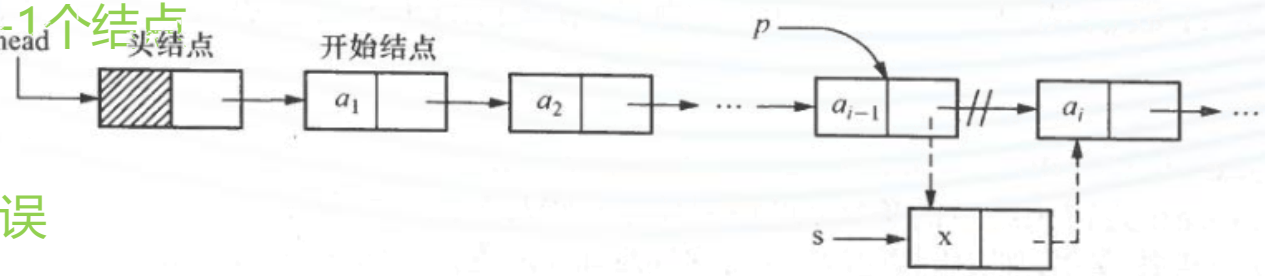


图 2.6　在 p 指向结点之后插入新结点*s 示意图

插入算法不需要移动表结点，但为了使p指向第i-1个结点，仍然需要从表头开始进行查找，因此该插入算法的时间复杂度也是O(n)。

4. 删除运算


```
DataType DeleteList ( LinkList head , int i )
{   //在以head为头指针的带头结点的单链表中删除第i个结点
    ListNode *p , *s ;
    DataType x ; int j ;
    p=head ; j=0 ;
    while ( p!=NULL && j<i-1) {   //使p指向第i-1个结点
          p=p->next ; ++j ;
    }
    if ( p==NULL ) {                    //删除位置错误
          printf ( "位置错误\n" ) ;
          exit ( 0 ) ;                  //出错退出处理
    }
    else {
          s=p->next ;                   //s指向第i个结点
          p->next=s->next ;             //使p->next指向第i+1个结点
          x=s->data ;                   //保存被删除结点的值
          free ( s ) ; return x ;       //删除第i个结点，返回结点值
    }
}
```

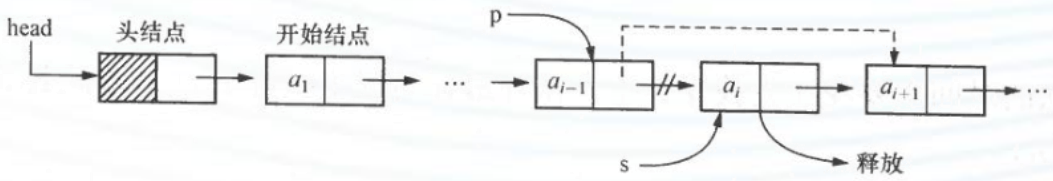
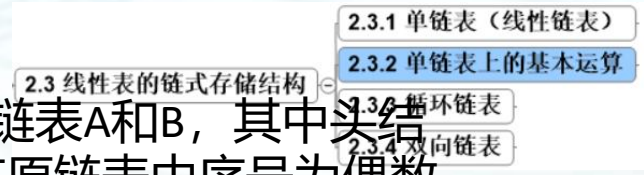


图 2.7　删除第 i 个结点示意图

删除算法与插入算法的时间复杂度一样，都是O(n)。

5. 单链表上运算举例

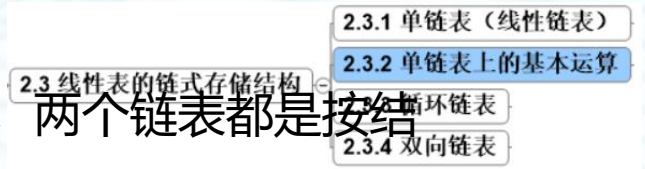


【例2.5】试写一个算法，将一个头结点指针为a的带头结点的单链表A分解成两个单链表A和B，其中头结点指针分别为a和b，使得A链表中含有原链表A中序号为奇数的元素，而B链表中含有原链表中序号为偶数的元素，并保持原来的相对顺序。

```
void split ( LinkList a , LinkList b )
{  //按序号奇偶分解单链表，注意b在调用该算法前是一个带头结点的空链表
   ListNode *p , *r , *s ;
   p=a->next ;                    //p指向表头结点
   r=a ;                          //r指向A表的当前结点
   s=b ;                          //s指向B表的当前结点
   while ( p!=NULL ) {
      r->next=p ;                 //序号为奇数的结点链接到A表上
      r=p ;                       //r总是指向A链表的最后一个结点
      p=p->next ;
      if ( p) {
         s->next=p ;              //序号为偶数的结点链接到B表上
         s=p ;                    //S总是指向B链表的最后一个结点
         p=p->next ;              //P指向原链表A中的偶数序号的结点
      }
   }
   r->next=s->next=NULL ;
}
```

这个算法要从头至尾扫描整个链表，所以它的时间复杂度是O(n)。

5. 单链表上运算举例



【例2.6】假设头指针为La和Lb的单链表（带头结点）分别为线性表A和B的存储结构，两个链表都是按结点数据值递增有序的。试写一个算法，将这两个单链表合并为一个有序链表Lc。

```
LinkList MergeList ( LinkList La , LinkList Lb )
{   //归并两个有序链表La和Lb为有序链表Lc
    ListNode *pa, *pb, *pc ; LinkList Lc ;
    pa=La->next ; pb=Lb->next ;                    //pa和pb分别指向两个链表的开始结点
    Lc=pc=La;                                      //用La的头结点作为的Lc头结点
    while ( pa!=NULL && pb!=NULL ) {
        if ( pa->data<=pb->data ) {
            pc->next=pa ;pc=pa ;pa=pa->next ;
        }
        else {
            pc->next=pb ;pc=pb ;pb=pb->next ;}
    }
    pc->next=pa!=NULL ? pa : pb ;                  //插入链表剩余部分
    free ( Lb ) ;                                  //释放Lb的头结点
    return Lc ;                                    //返回合并后的表
}
```

该算法要从头至尾扫描两个链表，用一个循环，所以它的时间复杂度是O(n)。

真题演练

在一个单链表中，已知q所指结点是p所指结点的后继结点，若在p和q之间插入s所指结点，则正确的操作是（　　）

A: s->next=p->next;p->next=s;

B: s->next=q;p->next=s->next;

C: q->next=s;s->next=p;

D: p->next=s;s->next=p;

真题演练

在一个单链表中，已知q所指结点是p所指结点的后继结点，若在p和q之间插入s所指结点，则正确的操作是（　　）

A: s->next=p->next;p->next=s;

B: s->next=q;p->next=s->next;

C: q->next=s;s->next=p;

D: p->next=s;s->next=p;

答案：A

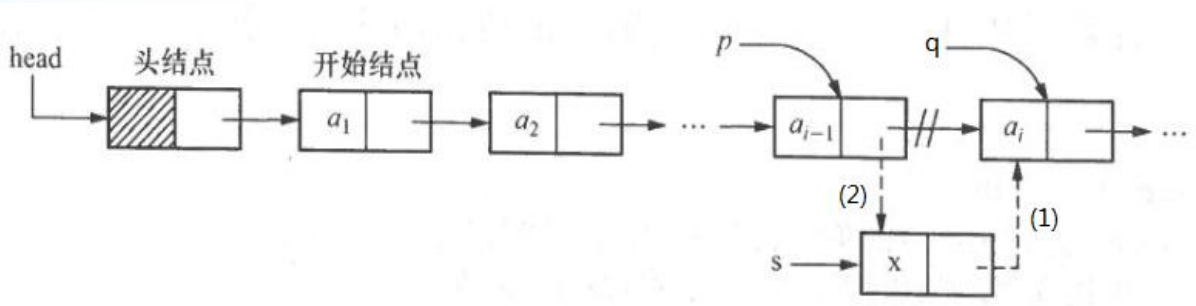


图 2.6　在 p 指向结点之后插入新结点*s 示意图

真题演练

设指针变量P指向非空单链表中的结点，next是结点的指针域，则判断P所指结点为尾结点前一个结点的逻辑表达式中，正确的是（　　）

A: p->next!=NULL&&p->next->next->next ==NULL

B: p->next!=NULL&&p->next->next ==NULL

C: p->next->next==NULL

D: p->next==NULL

真题演练

设指针变量P指向非空单链表中的结点，next是结点的指针域，则判断P所指结点为尾结点前一个结点的逻辑表达式中，正确的是（　　）

A: p->next!=NULL&&p->next->next->next ==NULL

B: p->next!=NULL&&p->next->next ==NULL

C: p->next->next==NULL

D: p->next==NULL

答案：B

2.3.3三、循环链表

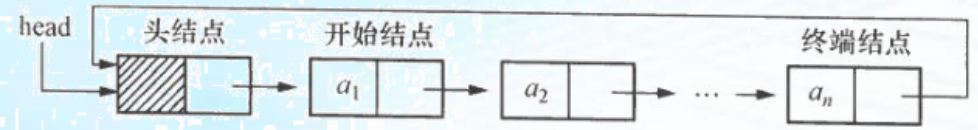


图 2.8　单循环链表示意图

循环链表的结点类型与单链表完全相同，在操作上也与单链表基本一致，差别仅在于算法中循环的结束判断条件不再是p或p->next是否为空，而是它们是否等于头指针。

在用头指针表示的单循环链表中，查找任何结点都必须从开始结点查起，而在实际应用中，表的操作常常会在表尾进行，此时若用尾指针表示单循环链表，可使某些操作简化。仅设尾指针的单循环链表如图2.9所示。

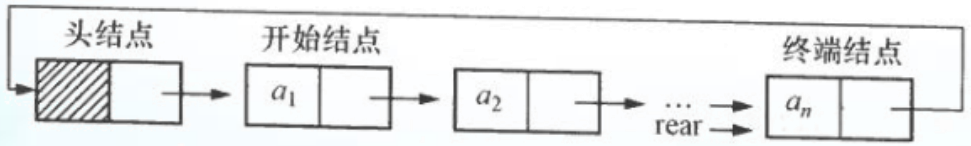
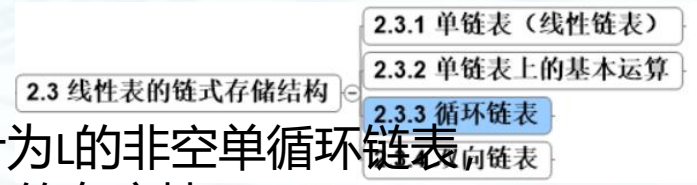


图 2.9　仅设尾指针的单循环链表示意图

2.3.3 循环链表



【例2.7】已知有一个结点数据域为整型的，且按从大到小顺序排列的头结点指针为L的非空单循环链表，试写一算法插入一个结点（其数据域为x）至循环链表的适当位置，使之保持链表的有序性。

```
void InsertList ( LinkList L ,int x )
{  //将值为x的新结点插入到有序循环链表中适当的位置
   ListNode *s , *p , *q ;
   s= ( ListNode *) malloc ( sizeof ( ListNode ) ) ;    //申请结点存储空间
   s->data=x ;p=L ;
   q=p->next ;                                          //q指向开始结点
   while ( q->data>x && q!=L ) {                        //查找插入位置
       p=p->next ;                                      //p指向q的前趋结点
       q=p->next ;                                      //q指向当前结点
      }
   p->next=s ;                                          //插入*s结点
   s->next=q ;
}
```

该算法在最好情况下，也就是插入在第一个结点前面，循环一次也不执行；在最坏情况下，即插在最后一个结点之后，当循环需要执行n次，因此，该算法的时间复杂度为O(n)。

真题演练

某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则下列存储结构中，最节省运算时间的是（　）

A: 单链表

B: 仅有头指针的单循环链表

C: 双向链表

D: 仅有尾指针的单循环链表

真题演练

某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则下列存储结构中，最节省运算时间的是（　）

A: 单链表

B: 仅有头指针的单循环链表

C: 双向链表

D: 仅有尾指针的单循环链表

答案：D

真题演练

在头指针为head的循环链表中，判断指针变量P指向尾结点的条件是（　　）

A: p->next->next==head

B: p->next==head

C: p->next->next==NULL

D: p->next==NULL

真题演练

在头指针为head的循环链表中，判断指针变量P指向尾结点的条件是（　　）

A: p->next->next==head

B: p->next==head

C: p->next->next==NULL

D: p->next==NULL

答案：B

真题演练

头指针head指向带头结点的单循环链表。链表为空时下列选项为真的是（　　）

A: head!=NULL

B: head==NULL

C: head->next==NULL

D: head->next==head

真题演练

头指针head指向带头结点的单循环链表。链表为空时下列选项为真的是（　　）

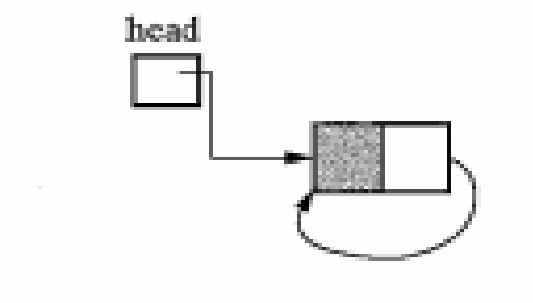
A: head!=NULL

B: head==NULL

C: head->next==NULL

D: head->next==head

答案：D



真题演练

对带头结点的单循环链表从头结点开始遍历（head为头指针，p=head->next)。

若指针p指向当前被遍历结点，则判定遍历过程结束的条件是（　　）。

A: p==NULL

B: head==NULL

C: p==head

D: head!=p

真题演练

对带头结点的单循环链表从头结点开始遍历（head为头指针，p=head->next)。

若指针p指向当前被遍历结点，则判定遍历过程结束的条件是（　　）。

A: p==NULL

B: head==NULL

C: p==head

D: head!=p

答案：C

真题演练

设指针变量head指向非空单循环链表的头结点，指针变量p指向终端结点，next是结点的指针域，则下列逻辑表达式中，值为真的是（　　）

A: p->next->next==head

B: p->next ==head

C: p->next->next==NULL

D: p->next==NULL

真题演练

设指针变量head指向非空单循环链表的头结点，指针变量p指向终端结点，next是结点的指针域，则下列逻辑表达式中，值为真的是（　　）

A: p->next->next==head

B: p->next ==head

C: p->next->next==NULL

D: p->next==NULL

答案：B

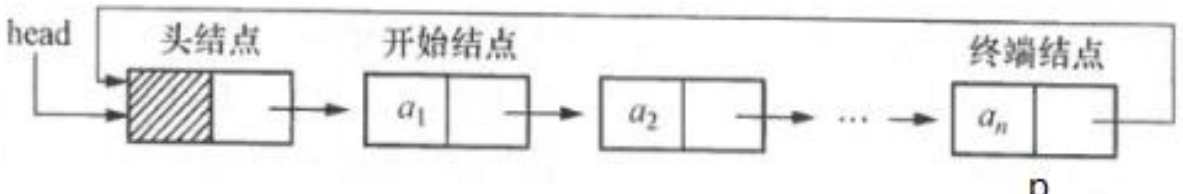
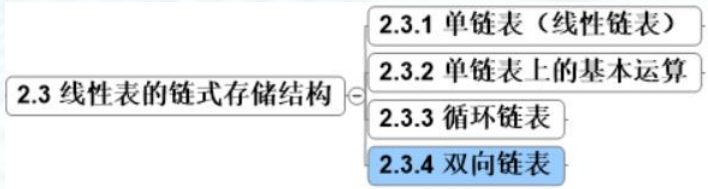


图 2.8　单循环链表示意图

2.3.4四、双向链表



双向链表及其结点类型描述如下：

```
typedef struct dlnode {
    DataType data ;
    struct dlnode *prior , *next ;
} DLNode ;
typedef DLNode * DLinkList ;
DlinkList head ;
```

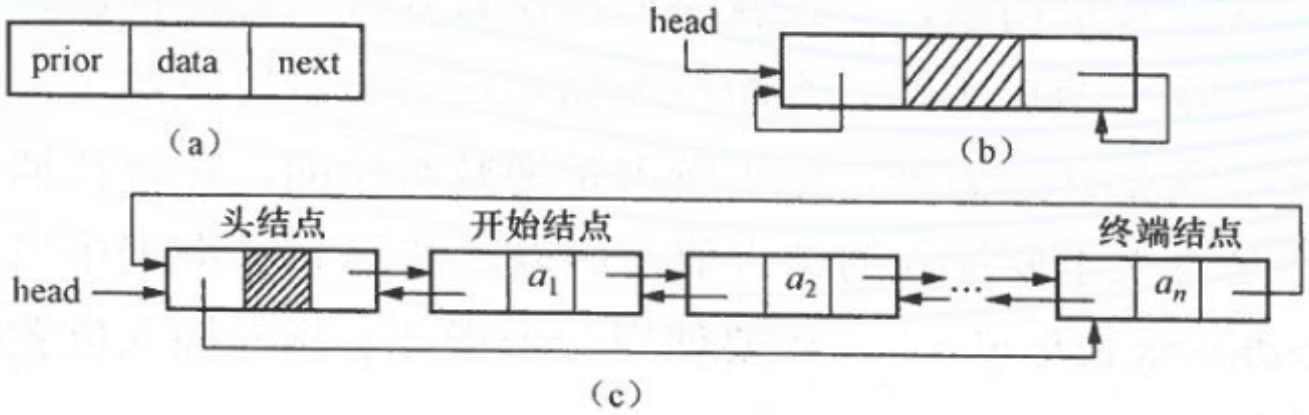


图 2.10　双向循环链表示意图

2.3.4 双向链表

例如，在双向链表的给定结点前插入一结点的操作过程如图2.11所示。设p为给定结点的指针，x为待插入结点的值，其实现算法如下：

```
void DLInsert ( DLNode *p , DataType x )
{  //将值为x的新结点插入到带头结点的双向链表中指定结点*p之前
    DLNode *s= ( DLNode * ) malloc ( sizeof ( DLNode ) ) ;  //申请新结点
    s->data=x ;
    s->prior=p->prior ; s->next=p ;
    p->prior->next=s ; p->prior=s ;
}
```

因为不再需要查找指向删除结点的前趋结点的指针，所以在双向链表上删除指定结点*p的算法更为简单，其删除操作过程如图2.12所示，其删除算法如下：

```
DataType DLDelete ( DLNode *p )
{  //删除带头结点的双向链表中指定结点*p
    p->prior->next=p->next:
    p->next->prior=p->prior ;
    x=p->data ;
    free ( p ) ;
    return x ;
}
```

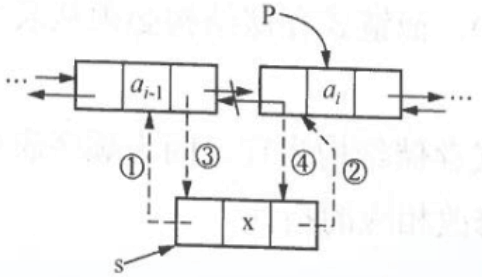


图 2.11　在双向链表上 p 指向结点之前插入新结点*s 示意图

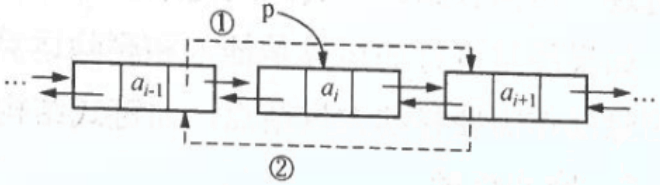
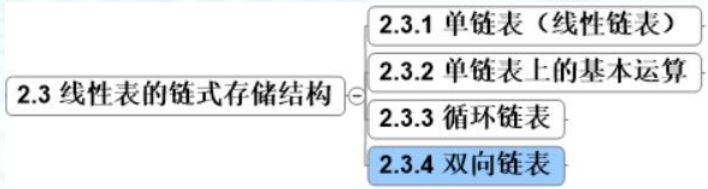


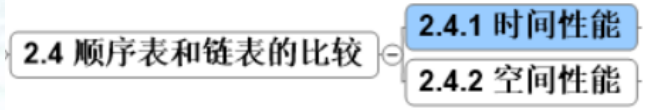
图 2.12　在双向链表上删除 p 指向的结点示意图

2.3.4 双向链表



【例2.8】假设有一个头结点指针为head的循环双向链表，其结点类型结构包括三个域：prior、data和next。其中data为数据域，next为指针域，指向其后继结点，prior也为指针域，其值为空（NULL），因此该双向链表其实是一个单循环链表。试写一算法，将其表修改为真正的双向循环链表。

```
void trans ( DLinkList head )
{
    DLNode *p ;
    p=head ;                          //使p指向头结点
    while ( p->next!=head ) {         //依次从左向右，对每个结点的prior赋值
        p->next->prior=p ;            //p所指结点的直接后继结点的前趋就是p
        p=p->next ;                   //p指针指向下一个结点
    }
    head->prior=p ;                   //head的前趋指向表的终端结点
}
```



1. 时间性能

如果在实际问题中，对线性表的操作是经常性的查找运算，以顺序表形式存储为宜。因为顺序存储是一种随机存取结构，可以随机访问任一结点，访问每个结点的时间代价是一样的，即每个结点的存取时间复杂度均为O(1)。而链式存储结构必须从表头开始沿链逐一访问各结点，其时间复杂度为O(n)。

如果经常进行的运算是插入和删除运算，以链式存储结构为宜。因为顺序表作插入和删除操作需要移动大量结点，而链式结构只需要修改相应的指针。

2. 空间性能

顺序表的存储空间是静态分配的，在应用程序执行之前必须给定空间大小。若线性表的长度变化较大，则其存储空间很难预先确定，设置过大将产生空间浪费，设定过小会使空间溢出，因此对数据量大小能事先知道的应用问题，适合使用顺序存储结构。而链式存储是动态分配存储空间，只要内存有空闲空间，就不会产生溢出，因此对数据量变化较大的动态问题，以链式存储结构为好。

对于线性表结点的存储密度问题，也是选择存储结构的一个重要依据。所谓存储密度就是结点空间的利用率。它的计算公式为

存储密度=（结点数据域所占空间）/（整个结点所占空间）

一般来说，结点存储密度越大，存储空间的利用率就越高。显然，顺序表结点的存储密度是1，而链表结点的存储密度肯定小于1。例如，若单链表结点数据域为整型数，指针所占的存储空间和整型数相同，则其结点的存储密度为50%。因此，若不考虑顺序表的空闲区，则顺序表的存储空间利用率为100%，远高于单链表的结点存储密度。

真题演练

顺序表便于（　　）

A: 插入结点

B: 删除结点

C: 按值查找结点

D: 按序号查找结点

真题演练

顺序表便于（　　）

A: 插入结点

B: 删除结点

C: 按值查找结点

D: 按序号查找结点

答案：D

真题演练

下列选项中，属于顺序存储结构优点的是（　）

A: 插入运算方便

B: 删除运算方便

C: 存储密度大

D: 方便存储各种逻辑结构

真题演练

下列选项中，属于顺序存储结构优点的是（　）

A: 插入运算方便

B: 删除运算方便

C: 存储密度大

D: 方便存储各种逻辑结构

答案：C

线性表 **顺序存储结构**	在逻辑关系上相邻的两个元素在物理位置上也是**相邻**的。	**优点**：可以随机存取表中任一元素。 **缺点**：若经常需要插入和删除操作，需要移动大量元素。
线性表 **链式存储结构**	存储线性表数据元素的存储空间可**能是连续的，也可能是不连续的。**	**优点**：插入和删除操作不需要移动大量元素；可以用来表示各种非线性的数据结构。 **缺点**：结点是不可以随机存取的。

本章总结


第2章 线性表
2.1 线性表的定义和基本运算
2.1.1 线性表的逻辑定义
2.1.2 线性表的基本运算
2.2 线性表的顺序存储和基本运算的实现
2.2.1 线性表的顺序存储
2.2.2 顺序表上基本运算的实现
2.3 线性表的链式存储结构
2.3.1 单链表（线性链表）
2.3.2 单链表上的基本运算
2.3.3 循环链表
2.3.4 双向链表
2.4 顺序表和链表的比较
2.4.1 时间性能
2.4.2 空间性能

祝大家顺利通过考试！