2012 人大考研

2012年 考研 计算机科学 专业基础综合考试教程

阳光考研命题研究中心/编写

权威专家联手

紧扣大纲
囊括全部知识点
从容应对全国联考

中国人民大学出版社

2012年
考研计算机科学专业基础综合考试教程

阳光考研命题研究中心　编写

中国人民大学出版社
·北京·

图书在版编目（CIP）数据

2012年考研计算机科学专业基础综合考试教程/阳光考研命题研究中心编写
北京：中国人民大学出版社，2010
ISBN 978-7-300-09799-2

Ⅰ. ①2…
Ⅱ. ①阳…
Ⅲ. ①计算机科学-研究生-入学考试-自学参考资料
Ⅳ. ①TP3

中国版本图书馆CIP数据核字（2010）第250498号

2012年考研计算机科学专业基础综合考试教程

阳光考研命题研究中心　编写

2012 Nian Kaoyan Jisuanji Kexue Zhuanye Jichu Zonghe Kaoshi Jiaocheng

出版发行	中国人民大学出版社		
社　　址	北京中关村大街31号	**邮政编码**	100080
电　　话	010-62511242（总编室）	010-62511398（质管部）	
	010-82501766（邮购部）	010-62514148（门市部）	
	010-62515195（发行公司）	010-62515275（盗版举报）	
网　　址	http://www.crup.com.cn		
	http://www.1kao.com.cn（中国1考网）		
经　　销	新华书店		
印　　刷	北京市媛明印刷厂		
规　　格	210 mm×285 mm　16开本	**版　　次**	2011年1月第1版
印　　张	23.5	**印　　次**	2011年1月第1次印刷
字　　数	706 000	**定　　价**	49.00元

前 言

Foreword

专业课统考是教育部的一项重要工作，其目的就是为了规范各招生单位的招生行为，对考生进行统一评价。在这项工作中，教育部只出版考试大纲，但考试大纲只规定了考试的命题范围和试卷形式，对知识点没有进行详细论述，广大考生只能在众多的大学教材中选择。大学教材无论是从体例结构设计、知识点描述方式，还是练习题的设计都与考研要求相距甚远，有的甚至还会有副作用。考生急需一套针对考研本身量身定制的复习资料，该资料能够针对大纲要求对知识点进行全面、正确、有用、适度的描述，能够对试题进行正确的分析，能够针对复习需要提供较为强大的试题演练，最好还能够给出一定量的模拟试题以供考生测试水平、感悟考场使用。

本教程完全依据大纲的要求，在深刻领会命题精神的基础上，对大纲要求的知识点进行实用的描述，并将考过的一些试题串联在该知识点下面，并配备相当数量的提高练习题，方便考生复习使用。

本书的编写得到了陈小文、李小生、马忠斌、朱鸿飞、刘冬梅、姜淑琴、文婷、郭丽梅、祁丹丹、张海军、孙立建、李晓峰、赵翠翠、文亚、孙美红、段鹏阳、朱福建、车丹丹、高晓玲、孙砚铭、朱陶丽、车志富、陈娟、王静、陈梅、徐彩华、谭爽、刘志英、李子谦、张艳芳、王亚珍、赵波文、乔基庆、杜军龙、康怡、于青华、张俊红、王晓娜、郑弘、季雨、王元元、曹晶等同志的帮助，特此感谢。

编 者

2011年1月

第一部分

数据结构

【考查目标】

1. 理解数据结构的基本概念；掌握数据的逻辑结构、存储结构及其异同，以及各种基本操作的实现。

2. 在掌握基本的数据处理原理和方法的基础上，能够对算法进行设计和分析。

3. 能够选择合适的数据结构和方法进行问题求解。

一、线性表

（一）线性表的定义和基本操作

线性表是具有相同特性的数据元素的一个有限序列。线性表的长度是线性表中所含元素的个数，用 n 表示，$n \geqslant 0$。当 $n=0$ 时，表示线性表是一个空表。

设序列中第 i 个元素为 a_i（$1 \leqslant i \leqslant n$），则线性表的一般表示形式为

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

线性表的基本操作包括以下几种。

第一，初始化线性表 InitList(&L)：构造一个空的线性表 L。

第二，销毁线性表 DestroyList(&L)：释放线性表 L 占用的内存空间。

第三，判定线性表是否为空表 ListEmpty(L)：若 L 为空表，则返回真，否则返回假。

第四，求线性表的长度 ListLength(L)：返回 L 中元素个数。

第五，输出线性表 DispList(L)：当线性表 L 不为空时，顺序显示 L 中各结点的值域。

第六，求线性表 L 中指定位置的某个数据元素 GetElem(L，i，&e)：用 e 返回 L 中第 i（$1 \leqslant i \leqslant$ ListLength(L)）个元素的值。

第七，定位查找 LocateElem(L，e)：返回 L 中第 1 个值域与 e 相等的位序。若这样的元素不存在，则返回值为 0。

第八，插入数据元素 ListInsert(&L，i，e)：在 L 的第 i（$1 \leqslant i \leqslant$ ListLength(L) $+1$）个元素之前插入新的元素 e，L 的长度增 1。

第九，删除数据元素 ListDelete(&L，i，&e)：删除 L 的第 i（$1 \leqslant i \leqslant$ ListLength(L)）个元素，并用 e 返回其值，L 的长度减 1。

(二) 线性表的实现

1. 顺序存储

(1) 线性表的顺序存储

线性表的顺序存储称为顺序表。顺序表就是把线性表中的所有元素按照其逻辑顺序，依次存储到从计算机存储器中指定存储位置开始的一块连续的存储空间中。

这样，线性表中第一个元素的存储位置就是指定的存储位置，第 $i+1$ 个元素（$1 \leqslant i \leqslant n-1$）的存储位置紧接在第 i 个元素的存储位置的后面。

在定义线性表的顺序存储类型时，需要定义数组存储线性表中的所有元素和整型变量存储线性表的长度。假定数组用 data [MaxSize] 表示，长度整型变量用 length 表示，并采用结构体类型表示，则元素类型为通用类型标识符 ElemType 的线性表的顺序存储类型可描述如下：

```
typedef struct
{
ElemType data [MaxSize];
int length;
} SqList;        /*顺序表类型*/
```

其中，MaxSize 为一个整型常数，data 成员存放元素，length 成员存放线性表的实际长度。

顺序表的示意图如图 1.1 所示。

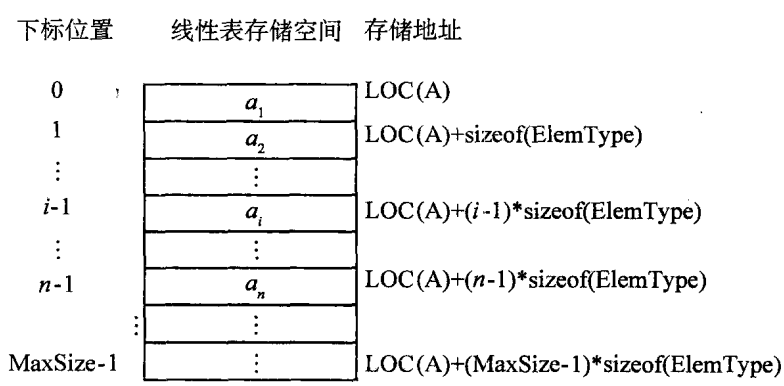

图 1.1 顺序表示意图

(2) 顺序表的建立

顺序表的建立是将给定的含有 n 个元素的数组的每个元素依次放入顺序表中，并将 n 赋给顺序表的长度成员。

建立顺序表算法如下：

```
void CreateList (SqList *&L, ElemType a [], int n)
/*建立顺序表*/
{
int i;
L= (SqList*) malloc (sizeof (SqList));
for (i=0; i<n; i++)
L->data [i] =a [i];
L->length=n;
}
```

(3) 线性表的基本操作在顺序表中的实现

1) 初始化线性表 InitList(L)：

只需将 length 成员设置为 0 即可。

```
void InitList (SqList *&L)    //引用型指针
{
L= (SqList *) malloc (sizeof (SqList));
/*分配存放线性表的空间*/
L->length=0;
}
```

本算法的时间复杂度为 O(1)。

2）销毁线性表 DestroyList(L)：

释放线性表 L 占用的内存空间。

```
void DestroyList (SqList *&L)
{
free (L);
}
```

本算法的时间复杂度为 O(1)。

3）判定是否为空表 ListEmpty(L)：

若 L 为空表，则返回 1，否则返回 0。

```
int ListEmpty (SqList *L)
{
return (L->length==0);
}
```

本算法的时间复杂度为 O(1)。

4）求线性表的长度 ListLength(L)：

只需返回 length 成员的值即可。

```
int ListLength (SqList *L)
{
return (L->length);
}
```

本算法的时间复杂度为 O(1)。

5）输出线性表 DispList(L)：

该运算当线性表 L 不为空时，顺序显示 L 中各元素的值。

```
void DispList (SqList *L)
{
int i;
if (ListEmpty (L)) return;
for (i=0; i<L->length; i++)
  printf ("%c", L->data [i]);
printf ("\n");
}
```

本算法的时间复杂度为：O(L->length) 或 O(n)

6）求某个数据元素值 GetElem(L，i，e)：

该运算返回 L 中第 i（$1 \leqslant i \leqslant$ ListLength(L)）个元素的值，存放在 e 中。

```
int GetElem (SqList *L, int i, ElemType &e)
{
if (i<1 || i>L->length) return 0;
```

```
e=L->data [i-1];
return 1;
}
```

本算法的时间复杂度为O(1)。

7）按元素值查找 LocateElem(L，e)：

该运算顺序查找第1个值域与 e 相等的元素的位序。若这样的元素不存在，则返回值为0。

```
int LocateElem (SqList *L, ElemType e)
{
  int i=0;
  while (i<L->length && L->data [i]! =e) i++;
  if (i>=L->length)      return 0;
  else return i+1;
}
```

本算法的时间复杂度为：O(L->length) 或 O(n)。

8）插入数据元素 ListInsert(L，i，e)：

该运算在顺序表 L 的第 i 个位置（$1 \leqslant i \leqslant$ ListLength(L) $+1$）上插入新的元素 e。

如果 i 值不正确，则显示相应错误信息；否则将顺序表原来第 i 个元素及以后元素均后移一个位置，腾出一个空位置插入新元素，顺序表长度增1。

```
int ListInsert (SqList *&L, int i, ElemType e)
{
int j;
if (i<1 || i>L->length+1) return 0;
i--;
/*将顺序表位序转化为 elem 下标*/
for (j=L->length; j>i; j--) L->elem [j] =L->elem [j-1];
/*将 elem [i] 及后面元素后移一个位置*/
L->elem [i] =e; L->length++;
/*顺序表长度增1*/
return 1;
}
```

元素移动的次数与下面的两个因素有关。

①表长 L->length(n)；

②插入位置 i（有 $n+1$ 个可能的插入位置）。

假设 $p_i\left(\frac{1}{n+1}\right)$ 是在第 i 个位置上插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需移动元素的平均次数为：

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

插入算法的平均时间复杂度为O(n)。

9）删除数据元素 ListDelete(L，i，e)：

如果 i 值不正确，则显示相应错误信息；否则，将线性表第 i 个元素以后元素均向前移动一个位置，这样覆盖原来的第 i 个元素，达到删除该元素的目的，最后顺序表长度减1。

```
int ListDelete (SqList *&L, int i, ElemType &e)
{
int  j;
if (i<1 || i>L->length) return 0;
i--;    /*将顺序表位序转化为 elem 下标*/
```

```
e=L->data [i];
for (j=i; j<L->length-1; j++)   L->data [j] =L->data [j+1];
/*将 data [i] 之后的元素前移一个位置*/
L->length--; /*顺序表长度减 1*/
return 1;
}
```

元素移动的次数也与表长 n 和删除元素的位置 i 有关。

①当 $i=n$ 时，移动次数为 0；

②当 $i=1$ 时，移动次数为 $n-1$。

在线性表 sq 中共有 n 个元素可以被删除。假设 $p_i\left(=\frac{1}{n}\right)$是删除第 i 个位置上元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素的平均次数为：

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

删除算法的平均时间复杂度为 O(n)。

2. 链式存储

(1) 链式存储结构的定义

线性表的链式存储结构称为链表。在链式存储结构中，每个存储结点不仅包含所存元素本身的信息(数据域)，而且包含元素之间逻辑关系的信息，我们可以通过前驱结点的指针域方便地找到后继结点的位置。一般每个结点有一个或多个这样的指针域。若一个结点中的某个指针域不需要任何结点，则它的值为空，用常量 NULL 表示。

在线性表的链式存储结构中，为了便于插入和删除算法的实现，每个链表带有一个头结点，并通过头结点的指针唯一标识该链表。

图 1.2 是链表的示意图，如下：

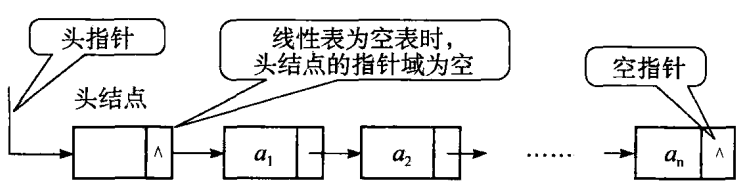

图 1.2 链表示意图

(2) 链表的形式

链表包含四种形式：单链表（线性链表）、双链表、循环链表、静态链表。

1) 单链表。

单链表的每个结点中除包含有数据域外，只设置一个指针域，用以指向其后继结点。

单链表中的结点类型用 LinkList 表示，包括存储元素的数据域，用 data 表示，其类型用通用类型标识符 ElemType 表示，用 next 表示存储后继元素位置的指针域。

LinkList 类型的定义如下：

```
typedef struct LNode          /*定义单链表结点类型*/
{
      ElemType data;
      struct LNode *next;   /*指向后继结点*/
} LinkList;
```

2) 双链表。

双链表的每个结点中除包含有数值域外，还可以设置两个指针域，分别用以指向其前驱结点和后继

结点。

对于双链表，采用类似于单链表的类型定义，其 DLinkList 类型的定义如下：

```
typedef struct DNode
/*定义双链表结点类型*/
{  ElemType data;
   struct DNode *prior;     /*指向前驱结点*/
   struct DNode *next;      /*指向后继结点*/
} DLinkList;
```

3）循环链表。

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域不再是空，而是指向表头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到链表中其他结点。

带头结点的循环单链表和循环双链表如图 1.3、图 1.4 所示。

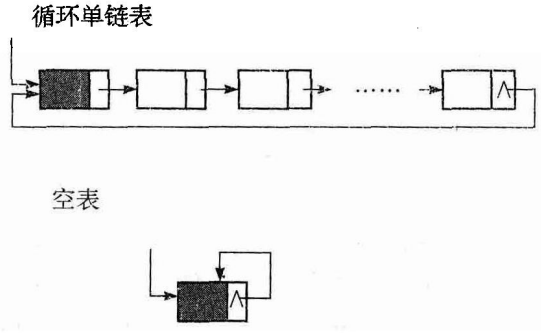

图 1.3　带头结点的循环单链表

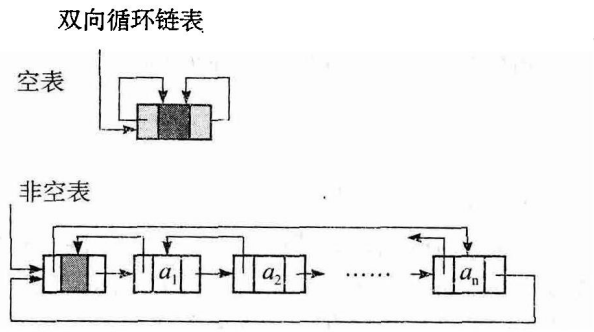

图 1.4　带头结点的循环双链表

4）静态链表。

静态链表借用一维数组来描述线性链表。数组中的一个分量表示一个结点，同时使用游标（指示器 cur）代替指针以指示结点在数组中的相对位置。数组中的第 0 个分量可以看成头结点，其指针域指示静态链表的第一个结点。这种存储结构仍然需要预先分配一个较大空间，但是在进行线性表的插入和删除操作时不需要移动元素，仅需要修改“指针”，因此仍然具有链式存储结构的主要优点。

（3）单链表的建立

假设通过一个含有 n 个数据的数组来建立单链表。建立单链表的常用方法有如下两种。

1）头插法建表：栈式建表法。

使用该方法建表在一个空表中开始，首先读取字符数组 a 中的字符，生成新结点，将读取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，直到结束为止。采用头插法建表的算法如下。

```
void CreateListF (LinkList *&L, ElemType a [ ], int n)
{
LinkList *s; int i;
    L= (LinkList *) malloc (sizeof (LinkList));
    /*建头结点*/
    L->next = NULL;
    for (i=0; i<n; i++)
     {    s= (LinkList *) malloc (sizeof (LinkList));
                /*创建新结点*/
          s->data=a [i]; s->next=L->next;
                /*将*s插在原开始结点之前，头结点之后*/
          L->next=s;
                /* L->next 始终指向链表的当前最新插入结点*/
     }
     }
```

2）尾插法建表：队列建表法。

头插法建立链表虽然算法简单，但生成的链表中结点的次序和原数组元素的顺序相反。若希望两者次序一致，可采用尾插法建立。尾插法建表法是将新结点插到当前链表的表尾上，为此必须增加一个尾指针 r，使其始终指向当前链表的尾结点。采用尾插法建表的算法如下：

```
void CreateListR (LinkList *&L, ElemType a [], int n)
{
LinkList *s, *r; int i;
    L= (LinkList *) malloc (sizeof (LinkList));
            /*创建头结点*/
    L->next = NULL;
    r = L;    /*r 始终指向尾结点，开始时指向了头结点*/
    for (i = 0; i<n; i++)
     {    s= (LinkList *) malloc (sizeof (LinkList));
            /*创建新结点*/
          s->data = a [i]; r->next = s;
            /*将*s 插入*r 之后*/
          r = s;
     }
     r->next = NULL; /*尾结点 next 域置为 NULL*/
}
```

(4) 线性表的基本操作在单链表中的实现

1）初始化线性表 InitList(L)：

建立一个空的单链表，即创建一个头结点。

```
void InitList (LinkList *&L)
{
L= (LinkList *) malloc (sizeof (LinkList));
                    /*创建头结点*/
L->next = NULL;
}
```

2）销毁线性表 DestroyList(L)：

释放单链表 L 占用的内存空间。即逐一释放全部结点的空间。

```
void DestroyList (LinkList *&L)
{  LinkList   *p = L,  *q = p->next;
while (q! =NULL)
{  free (p);
   p = q; q = p->next;
}
free (p);
}
```

3）判断线性表是否为空表 ListEmpty(L)：

若链表 L 没有结点，则返回真，否则返回假。

```
int ListEmpty (LinkList *L)
{
return (L->next == NULL);
}
```

4）求线性表的长度 ListLength(L)：

返回单链表 L 中数据结点的个数。

```
int ListLength (LinkList  *L)
{  LinkList  *p=L;   int i=0;
while (p->next ! =NULL)
{  i++;
   p=p->next;
}
return (i);
}
```

5）输出线性表 DispList(L)：

逐一扫描单链表 L 的每个数据结点，并显示各结点的 data 域的值。

```
void DispList (LinkList *L)
{  LinkList  *p=L->next;
while (p! = NULL)
{  printf ( "%c", p->data);
   p=p->next;
}
printf ("\n");
}
```

6）求线性表 L 中指定位置的某个数据元素 GetElem(L，i，&e)：

在单链表 L 中从头开始找到第 i 个结点，若存在第 i 个数据结点，则将其 data 域的值赋给变量 e。

```
int GetElem (LinkList *L, int i, ElemType &e)
{  int j=0;
   LinkList *p=L;

   while (j<i && p! =NULL)
   {    j++;
        p=p->next;
   }
   if (p==NULL)   return 0; /*不存在第 i 个数据结点*/
   else   /*存在第 i 个数据结点*/
   {    e=p->data;
        return 1;
   }
}
```

7）按元素值查找 LocateElem(L，e)：

在单链表 L 中从头开始找第 1 个值域中与 e 相等的结点，若存在，则返回其所在位置，否则返回 0。

```
int LocateElem (LinkList *L, ElemType e)
{  LinkList  *p=L->next; int n=1;
   while   (p! =NULL && p->data! =e)
   {    p=p->next; n++;   }
   if   (p==NULL)   return (0);
   else return (n);
}
```

8）插入数据元素 ListInsert(&L，i，e)：

先在单链表 L 中找到第 $i-1$ 个结点 *p，若存在这样的结点，将值为 e 的结点 *s 插入到其后，插入过程如图 1.5 所示。

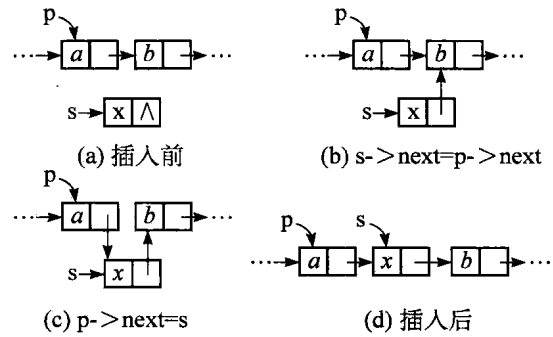

图 1.5 插入数据元素的过程

插入数据元素算法：

```
int ListInsert (LinkList *&L, int i, ElemType e)
{    int j = 0;
       LinkList *p = L, *s;
       while   (j<i-1 && p! =NULL)    /*查找第 i-1 个结点*/
       {    j++;
       p = p->next;
       }
   if (p == NULL)    return 0;              /*未找到位序为 i-1 的结点*/
   else /*找到位序为 i-1 的结点*p*/
      {  s= (LinkList *) malloc (sizeof (LinkList));
     /*创建新结点*s*/
     s->data = e;
     s->next = p->next;                    /*将*s 插入到*p 之后*/
     p->next = s;
     return 1;
   }
}
```

9）删除数据元素 ListDelete(&L, i, &e)：

思路：先在单链表 L 中找到第 $i-1$ 个结点 *p，若存在这样的结点，且也存在后继结点，则删除该后继结点，删除过程如图 1.6 所示。

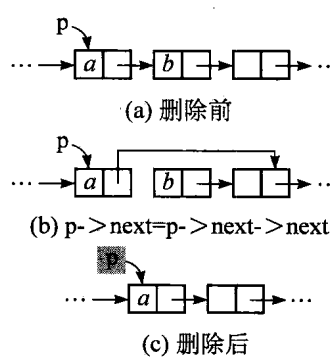

图 1.6 删除数据元素的过程

删除数据元素算法：

```
int ListDelete (LinkList *&L, int i, ElemType &e)
{  int j = 0;
   LinkList *p = L, *q;
   while (j<i-1 && p! =NULL)      /*查找第 i-1 个结点*/
   {  j++;
```