第 2 章 线性表

教材中练习题及参考答案

1. 简述线性表两种存储结构各自的主要特点。

答：线性表的两种存储结构分别是顺序存储结构和链式存储结构。顺序存储结构的主要特点如下：

① 数据元素中只有自身的数据域，没有关联指针域。因此，顺序存储结构的存储密度较大。

② 顺序存储结构需要分配一整块比较大存储空间，所以存储空间利用率较低。

③ 逻辑上相邻的两个元素在物理上也是相邻的，通过元素的逻辑序号可以直接其元素值，即具有随机存取特性。

④ 插入和删除操作会引起大量元素的移动。

链式存储结构的主要特点如下：

① 数据结点中除自身的数据域，还有表示逻辑关系的指针域。因此，链式存储结构比顺序存储结构的存储密度小。

② 链式存储结构的每个结点是单独分配的，每个结点的存储空间相对较小，所以存储空间利用率较高。

③ 在逻辑上相邻的结点在物理上不一定相邻，因此不具有随机存取特性。

④ 插入和删除操作方便灵活，不必移动结点，只需修改结点中的指针域即可。

2. 简述单链表设置头结点的主要作用。

答：对单链表设置头结点的主要作用如下：

① 对于带头结点的单链表，在单链表的任何结点之前插入结点或删除结点，所要做的都是修改前一个结点的指针域，因为任何结点都有前驱结点（若单链表没有头结点，则首结点没有前驱结点，在其前插入结点和删除该结点时操作复杂些），所以算法设计方便。

② 对于带头结点的单链表，在表空时也存在一个头结点，因此空表与非空表的处理是一样的。

3. 假设某个含有n个元素的线性表有如下运算：

Ⅰ. 查找序号为i（$1 \leqslant i \leqslant n$）的元素

Ⅱ. 查找第一个值为x的元素

Ⅲ. 插入新元素作为第一个元素

Ⅳ. 插入新元素作为最后一个元素

Ⅴ. 插入第i（$2 \leqslant i \leqslant n$）个元素

Ⅵ. 删除第一个元素

Ⅶ. 删除最后一个元素

Ⅷ. 删除第i（$2 \leqslant i \leqslant n$）个元素

现设计该线性表的如下存储结构：

① 顺序表

② 带头结点的单链表

③ 带头结点的循环单链表

④ 不带头结点仅有尾结点指针标识的循环单链表

⑤ 带头结点的双链表

⑥ 带头结点的循环双链表

指出各种存储结构中对应运算算法的时间复杂度。

答：各种存储结构对应运算的时间复杂度如表2.1所示。

表 2.1 各种存储结构对应运算的时间复杂度

	Ⅰ	Ⅱ	Ⅲ	Ⅳ	Ⅴ	Ⅵ	Ⅶ	Ⅷ
①	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)
②	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
③	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
④	O(n)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
⑤	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
⑥	O(n)	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)

4. 对于顺序表L，指出以下算法的功能。

```
void fun(SqList *&L)
{   int i,j=0;
    for (i=1;i<L->length;i++)
        if (L->data[i]>L->data[j])
            j=i;
    for (i=j;i<L->length-1;i++)
        L->data[i]=L->data[i+1];
    L->length--;
}
```

答：该算法的功能是在顺序表L中查找第一个值最大的元素，并删除该元素。

5. 对于顺序表L，指出以下算法的功能。

```
void fun(SqList *&L,ElemType x)
{   int i,j=0;
    for (i=1;i<L->length;i++)
        if (L->data[i]<=L->data[j])
            j=i;
    for (i=L->length;i>j;i--)
        L->data[i]=L->data[i-1];
    L->data[j]=x;
    L->length++;
```

```
}
```

答：在顺序表 *L* 中查找最后一个值最小的元素，在该位置上插入一个值为 *x* 的元素。

6. 有人设计如下算法用于删除整数顺序表 *L* 中所有值在[*x*，*y*]范围内的元素，该算法显然不是高效的，请设计一个同样功能的高效算法。

```
void fun(SqList *&L,ElemType x)
{   int i,j;
    for (i=0;i<L->length;i++)
        if (L->data[i]>=x && L->data[i]<=y)
        {   for (j=i;j<L->length-1;j++)
                L->data[j]=L->data[j+1];
            L->length--;
        }
}
```

答：该算法在每次查找到 *x* 元素时，都通过移动来删除它，时间复杂度为 O(n^2)，显然不是高效的算法。实现同样功能的算法如下：

```
void fun(SqList *&L,ElemType x,ElemType y)
{   int i,k=0;
    for (i=0;i<L->length;i++)
        if (!(L->data[i]>=x && L->data[i]<=y))
        {   L->data[k]=L->data[i];
            k++;
        }
    L->length=k;
}
```

该算法（思路参见《教程》例 2.3 的解法一）的时间复杂度为 O(*n*)，是一种高效的算法。

7. 设计一个算法，将元素*x*插入到一个有序（从小到大排序）顺序表的适当位置上，并保持有序性。

解：通过比较在顺序表 *L* 中找到插入 *x* 的位置 *i*，将该位置及后面的元素均后移一个位置，将 *x* 插入到位置 *i* 中，最后将 *L* 的长度增 1。对应的算法如下：

```
void Insert(SqList *&L,ElemType x)
{   int i=0,j;
    while (i<L->length && L->data[i]<x) i++;
    for (j=L->length-1;j>=i;j--)
        L->data[j+1]=L->data[j];
    L->data[i]=x;
    L->length++;
}
```

8. 假设一个顺序表 *L* 中所有元素为整数，设计一个算法调整该顺序表，使其中所有小于零的元素放在所有大于等于零的元素的前面。

解：先让 *i*、*j* 分别指向顺序表 *L* 的第一个元素和最后一个元素。当 *i*<*j* 时循环：*i* 从前向后扫描顺序表 *L*，找大于等于 0 的元素，*j* 从后向前扫描顺序表 *L*，找小于 0 的元素，当 *i*<*j* 时将两元素交换（思路参见《教程》例 2.4 的解法一）。对应的算法如下：

```
void fun(SqList *&L)
{   int i=0,j=L->length-1;
    while (i<j)
    {   while (L->data[i]<0) i++;
        while (L->data[j]>=0) j--;
        if (i<j)      //L->data[i]与L->data[j]交换
            swap(L->data[i], L->data[j]);
    }
}
```

9. 对于不带头结点的单链表 $L1$，其结点类在为 LinkNode，指出以下算法的功能。

```
void fun1(LinkNode *&L1,LinkNode *&L2)
{   int n=0,i;
    LinkNode *p=L1;
    while (p!=NULL)
    {   n++;
        p=p->next;
    }
    p=L1;
    for (i=1;i<n/2;i++)
        p=p->next;
    L2=p->next;
    p->next=NULL;
}
```

答：对于含有 n 个结点的单链表 $L1$，将 $L1$ 拆分成两个不带头结点的单链表 $L1$ 和 $L2$，其中 $L1$ 含有原来的前 $n/2$ 个结点，$L2$ 含有余下的结点。

10. 在结点类型为 DLinkNode 的双链表中，给出将 p 所指结点（非尾结点）与其后继结点交换的操作。

答：将 p 所指结点（非尾结点）与其后继结点交换的操作如下：

```
q=p->next;              //q 指向结点 p 的后继结点
if (q->next!=NULL)      //从链表中删除结点 p
    q->next->prior=p;
p->next=q->next;
p->prior->next=q;       //将结点 q 插入到结点 p 的前面
q->prior=p->prior;
q->next=p;
p->prior=q;
```

11. 有一个线性表(a_1，a_2，…，a_n)，其中 $n\geq 2$，采用带头结点的单链表存储，头指针为 L，每个结点存放线性表中一个元素，结点类型为（*data*，*next*），现查找某个元素值等于 x 的结点指针，若不存在这样的结点返回 NULL。分别写出下面 3 种情况的查找语句。要求时间尽量少。

（1）线性表中元素无序。

（2）线性表中元素按递增有序。

（3）线性表中元素按递减有序。

答：（1）元素无序时的查找语句如下：

```
p=L->next;
while (p!=NULL && p->data!=x)
    p=p->next;
if (p==NULL) return NULL;
else return p;
```

（2）元素按递增有序时的查找语句如下：

```
p=L->next;
while (p!=NULL && p->data<x )
    p=p->next;
if (p==NULL || p->data>x) return NULL;
else return p;
```

（3）元素按递减有序时的查找语句如下：

```
p=L->next;
while (p!=NULL && p->data>x)
    p=p->next;
if (p==NULL || p->data<x) return NULL;
else return p;
```

12. 设计一个算法，将一个带头结点的数据域依次为 a_1、a_2、…、a_n（n≥3）的单链表的所有结点逆置，即第一个结点的数据域变为 a_n，第 2 个结点的数据域变为 a_{n-1}，…，尾结点的数据域为 a_1。

解：首先让 *p* 指针指向首结点，将头结点的 *next* 域设置为空，表示新建的单链表为空表。用 *p* 扫描单链表的所有数据结点，将结点 *p* 采用头插法插入到新建的单链表中。对应的算法如下：

```
void Reverse(LinkNode *&L)
{   LinkNode *p=L->next,*q;
    L->next=NULL;
    while (p!=NULL)              //扫描所有的结点
    {   q=p->next;               //q 临时保存 p 结点的后继结点
        p->next=L->next;         //总是将 p 结点作为首结点插入
        L->next=p;
        p=q;                     //让 p 指向下一个结点
    }
}
```

13. 一个线性表（a_1，a_2，…，a_n）（n>3）采用带头结点的单链表 *L* 存储。设计一个高效算法求中间位置的元素（即序号为⌊n/2⌋的元素）。

解：让 *p*、*q* 首先指向首结点，然后在 *p* 结点后面存在两个结点时循环：*p* 后移两个结点，*q* 后移一个结点。当循环结束后，*q* 指向的就是中间位置的结点，对应的算法如下：

```
ElemType Midnode(LinkNode *L)
{   LinkNode *p=L->next,*q=p;
    while (p->next!=NULL && p->next->next!=NULL)
    {   p=p->next->next;
        q=q->next;
    }
    return q->data;
```

```
}
```

14. 设计一个算法在带头结点的非空单链表 *L* 中第一个最大值结点（最大值结点可能有多个）之前插入一个值为 *x* 的结点。

解：先在单链表 *L* 中查找第一个最大值结点的前驱结点 *maxpre*，然后在其后面插入值为 *x* 的结点。对应的算法如下：

```
void Insertbeforex(LinkNode *&L,ElemType x)
{   LinkNode *p=L->next,*pre=L;
    LinkNode *maxp=p,*maxpre=L,*s;
    while (p!=NULL)
    {   if (maxp->data<p->data)
        {   maxp=p;
            maxpre=pre;
        }
        pre=p; p=p->next;
    }
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=x;
    s->next=maxpre->next;
    maxpre->next=s;
}
```

15. 设有一个带头结点的单链表 *L*，结点的结构为（*data*，*next*），其中 *data* 为整数元素，*next* 为后继结点的指针。设计一个算法，首先按递减次序输出该单链表中各结点的数据元素，然后释放所有结点占用的存储空间，并要求算法的空间复杂度为 O(1)。

解：先对单链表 *L* 的所有结点递减排序（思路参见《教程》例 2.8），再输出所有结点值，最后释放所有结点的空间。对应的算法如下：

```
void Sort(LinkNode *&L)             //对单链表 L 递减排序
{   LinkNode *p,*q,*pre;
    p=L->next->next;                //p 指向第 2 个数据结点
    L->next->next=NULL;
    while (p!=NULL)
    {   q=p->next;
        pre=L;
        while (pre->next!=NULL && pre->next->data>p->data)
            pre=pre->next;
        p->next=pre->next;          //在结点 pre 之后插入 p 结点
        pre->next=p;
        p=q;
    }
}
void fun(LinkNode *&L)              //完成本题的算法
{   printf("排序前单链表 L:");
    DispList(L);                    //调用基本运算算法
    Sort(L);
    printf("排序后单链表 L:");
    DispList(L);                    //调用基本运算算法
    printf("释放单链表 L\n");
```

```
    DestroyList(L);              //调用基本运算算法
}
```

16. 设有一个双链表 *h*，每个结点中除有 *prior*、*data* 和 *next* 三个域外，还有一个访问频度域 *freq*，在链表被起用之前，其值均初始化为零。每当进行 LocateNode(*h*，*x*)运算时，令元素值为 *x* 的结点中 *freq* 域的值加 1，并调整表中结点的次序，使其按访问频度的递减序排列，以便使频繁访问的结点总是靠近表头。试写一符合上述要求的 LocateNode 运算的算法。

解：在 DLinkNode 类型的定义中添加整型 *freq* 域，给该域初始化为 0。在每次查找到一个结点 *p* 时，将其 *freq* 域增 1，再与它前面的一个结点 *pre* 进行比较，若 *p* 结点的 *freq* 域值较大，则两者交换，如此找一个合适的位置。对应的算法如下：

```
bool LocateNode(DLinkNode *h,ElemType x)
{   DLinkNode *p=h->next,*pre;
    while (p!=NULL && p->data!=x)
        p=p->next;                          //找 data 域值为 x 的结点 p
    if (p==NULL)                            //未找到的情况
        return false;
    else                                    //找到的情况
    {   p->freq++;                          //频度增 1
        pre=p->prior;                       //结点 pre 为结点 p 的前驱结点
        while (pre!=h && pre->freq<p->freq)
        {   p->prior=pre->prior;
            p->prior->next=p;               //交换结点 p 和结点 pre 的位置
            pre->next=p->next;
            if (pre->next!=NULL)            //若 p 结点不是尾结点时
                pre->next->prior=pre;
            p->next=pre;pre->prior=p;
            pre=p->prior;                   //q 重指向结点 p 的前驱结点
        }
        return true;
    }
}
```

17. 设 $ha=(a_1, a_2, \cdots, a_n)$和 $hb=(b_1, b_2, \cdots, b_m)$ 是两个带头结点的循环单链表。设计一个算法将这两个表合并为带头结点的循环单链表 *hc*。

解：先找到 *ha* 的尾结点 *p*，将结点 *p* 的 *next* 指向 *hb* 的首结点，再找到 *hb* 的尾结点 *p*，将其构成循环单链表。对应的算法如下：

```
void Merge(LinkNode *ha, LinkNode *hb, LinkNode *&hc)
{   LinkNode *p=ha->next;
    hc=ha;
    while (p->next!=ha)     //找到 ha 的尾结点 p
        p=p->next;
    p->next=hb->next;       //将结点 p 的 next 指向 hb 的首结点
    while (p->next!=hb)
        p=p->next;          //找到 hb 的尾结点 p
    p->next=hc;             //构成循环单链表
    free(hb);               //释放 hb 单链表的头结点
```

```
}
```

18. 设两个非空线性表分别用带头结点的循环双链表 *ha* 和 *hb* 表示。设计一个算法Insert(*ha*，*hb*，*i*)。其功能是：*i*=0 时，将 *hb* 插入到 *ha* 的前面；当 *i*>0 时，将 *hb* 插入到 *ha* 中第 *i* 个结点的后面；当 *i* 大于等于 *ha* 的长度时，将 *hb* 插入到 *ha* 的后面。

解：利用带头结点的循环双链表的特点设计的算法如下：

```
void Insert(DLinkNode *&ha, DLinkNode *&hb,int i)
{   DLinkNode *p=ha->next,*post;
    int lena=1,j;
    while (p->next!=ha)              //求出 ha 的长度 lena
    {   lena++;
        p=p->next;
    }
    if (i==0)                        //将 hb 插入到 ha 的前面
    {   p=hb->prior;                 //p 指向 hb 的尾结点
        p->next=ha->next;            //将结点 p 链到 ha 的首结点前面
        ha->next->prior=p;
        ha->next=hb->next;
        hb->next->prior=ha;          //将 ha 头结点与 hb 的首结点链起来
    }
    else if (i<lena)                 //将 hb 插入到 ha 中间
    {   j=1;
        p=ha->next;
        while (j<i)                  //在 ha 中查找第 i 个结点 p
        {   p=p->next;
            j++;
        }
        post=p->next;                //post 指向 p 结点的后继结点
        p->next=hb->next;            //将 hb 的首结点作为 p 结点的后继结点
        hb->next->prior=p;
        hb->prior->next=post;        //将 post 结点作为 hb 尾结点的后继结点
        post->prior=hb->prior;
    }
    else                             //将 hb 链到 ha 之后
    {   ha->prior->next=hb->next;    //ha->prior 指向 ha 的尾结点
        hb->next->prior=ha->prior;
        hb->prior->next=ha;
        ha->prior=hb->prior;
    }
    free(hb);                        //释放 hb 头结点
}
```

19. 用带头结点的单链表表示整数集合，完成以下算法并分析时间复杂度：

（1）设计一个算法求两个集合*A*和*B*的并集运算即*C*=*A*∪*B*。要求不破坏原有的单链表*A*和*B*。

（2）假设集合中的元素按递增排列，设计一个高效算法求两个集合*A*和*B*的并集运算即*C*=*A*∪*B*。要求不破坏原有的单链表*A*和*B*。

解：（1）集合 *A*、*B*、*C* 分别用单链表 *ha*、*hb*、*hc* 存储。采用尾插法创建单链表 *hc*，先将 *ha* 单链表中所有结点复制到 *hc* 中，然后扫描单链表 *hb*，将其中所有不属于 *ha* 的结点复制到 *hc* 中。对应的算法如下：

```
void Union1(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *pa=ha->next,*pb=hb->next,*pc,*rc;
    hc=(LinkNode *)malloc(sizeof(LinkNode));
    rc=hc;
    while (pa!=NULL)        //将 A 复制到 C 中
    {   pc=(LinkNode *)malloc(sizeof(LinkNode));
        pc->data=pa->data;
        rc->next=pc;
        rc=pc;
        pa=pa->next;
    }
    while (pb!=NULL)        //将 B 中不属于 A 的元素复制到 C 中
    {   pa=ha->next;
        while (pa!=NULL && pa->data!=pb->data)
            pa=pa->next;
        if (pa==NULL)       //pb->data 不在 A 中
        {   pc=(LinkNode *)malloc(sizeof(LinkNode));
            pc->data=pb->data;
            rc->next=pc;
            rc=pc;
        }
        pb=pb->next;
    }
    rc->next=NULL;          //尾结点 next 域置为空
}
```

本算法的时间复杂度为 O(*m*×*n*)，其中 *m*、*n* 为单链表 ha 和 hb 中的数据结点个数。

（2）同样采用尾插法创建单链表 *hc*，并利用单链表的有序性，采用二路归并方法来提高算法效率。对应的算法如下：

```
void Union2(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *pa=ha->next,*pb=hb->next,*pc,*rc;
    hc=(LinkNode *)malloc(sizeof(LinkNode));
    rc=hc;
    while (pa!=NULL && pb!=NULL)
    {   if (pa->data<pb->data)      //将较小的结点 pa 复制到 hc 中
        {   pc=(LinkNode *)malloc(sizeof(LinkNode));
            pc->data=pa->data;
            rc->next=pc;
            rc=pc;
            pa=pa->next;
        }
        else if (pa->data>pb->data) //将较小的结点 pb 复制到 hc 中
        {   pc=(LinkNode *)malloc(sizeof(LinkNode));
            pc->data=pb->data;
            rc->next=pc;
```

```
            rc=pc;
            pb=pb->next;
        }
        else                        //相等的结点只复制一个到 hc 中
        {   pc=(LinkNode *)malloc(sizeof(LinkNode));
            pc->data=pa->data;
            rc->next=pc;
            rc=pc;
            pa=pa->next;
            pb=pb->next;
        }
    }
    if (pb!=NULL) pa=pb;            //让 pa 指向没有扫描完的单链表结点
    while (pa!=NULL)
    {   pc=(LinkNode *)malloc(sizeof(LinkNode));
        pc->data=pa->data;
        rc->next=pc;
        rc=pc;
        pa=pa->next;
    }
    rc->next=NULL;                  //尾结点 next 域置为空
}
```

本算法的时间复杂度为 O(*m*+*n*)，其中 *m*、*n* 为单链表 *ha* 和 *hb* 中的数据结点个数。

20. 用带头结点的单链表表示整数集合，完成以下算法并分析时间复杂度：

（1）设计一个算法求两个集合*A*和*B*的差集运算即*C*=*A*−*B*。要求算法的空间复杂度为O(1)，并释放单链表*A*和*B*中不需要的结点。

（2）并假设集合中的元素按递增排列，设计一个高效算法求两个集合*A*和*B*的差集运算即*C*=*A*−*B*。要求算法的空间复杂度为O(1)，并释放单链表*A*和*B*中不需要的结点。

解：集合*A*、*B*、*C*分别用单链表*ha*、*hb*、*hc*存储。由于要求空间复杂度为O(1)，不能采用复制方法，只能利用原来单链表中结点重组产生结果单链表。

（1）将 *ha* 单链表中所有在 *hb* 中出现的结点删除，最后将 *hb* 中所有结点删除。对应的算法如下：

```
void Sub1(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *prea=ha,*pa=ha->next,*pb,*p,*post;
    hc=ha;                          //将 ha 的头结点作为 hc 的头结点
    while (pa!=NULL)                //删除 A 中属于 B 的结点
    {   pb=hb->next;
        while (pb!=NULL && pb->data!=pa->data)
            pb=pb->next;
        if (pb!=NULL)               //pa->data 在 B 中,从 A 中删除结点 pa
        {   prea->next=pa->next;
            free(pa);
            pa=prea->next;
        }
        else
        {   prea=pa;                //prea 和 pa 同步后移
```

```
            pa=pa->next;
        }
    }
    p=hb; post=hb->next;          //释放 B 中所有结点
    while (post!=NULL)
    {   free(p);
        p=post;
        post=post->next;
    }
    free(p);
}
```

本算法的时间复杂度为 O($m\times n$)，其中 m、n 为单链表 *ha* 和 *hb* 中的数据结点个数。

（2）同样采用尾插法创建单链表 *hc*，并利用单链表的有序性，采用二路归并方法来提高算法效率，一边比较一边将不需要的结点删除。对应的算法如下：

```
void Sub2(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *prea=ha,*pa=ha->next;  //pa 扫描 ha,prea 是 pa 结点的前驱结点指针
    LinkNode *preb=hb,*pb=hb->next;  //pb 扫描 hb,preb 是 pb 结点的前驱结点指针
    LinkNode *rc;                    //hc 的尾结点指针
    hc=ha;                           //ha 的头结点作为 hc 的头结点
    rc=hc;
    while (pa!=NULL && pb!=NULL)
    {   if (pa->data<pb->data)       //将较小的结点 pa 链到 hc 之后
        {   rc->next=pa;
            rc=pa;
            prea=pa;                 //prea 和 p 同步后移
            pa=pa->next;
        }
        else if (pa->data>pb->data) //删除较大的结点 pb
        {   preb->next=pb->next;
            free(pb);
            pb=preb->next;
        }
        else                         //删除相等的 pa 结点和 pb 结点
        {   prea->next=pa->next;
            free(pa);
            pa=prea->next;
            preb->next=pb->next;
            free(pb);
            pb=preb->next;
        }
    }
    while (pb!=NULL)                 //删除 pb 余下的结点
    {   preb->next=pb->next;
        free(pb);
        pb=preb->next;
    }
    free(hb);                        //释放 hb 的头结点
    rc->next=NULL;                   //尾结点 next 域置为空
}
```

本算法的时间复杂度为 O(*m*+*n*)，其中 *m*、*n* 为单链表 *ha* 和 *hb* 中的数据结点个数。