第 2 章习题

2.1 若将顺序表中记录其长度的分量 listlen 改为指向最后一个元素的位置 last，在实现各基本运算时需要做那些修改？

【解】

```
//用线性表最后一个元素的下标 last 代替 listLen 实现顺序表
#define MAXLEN 100
typedef int elementType;

typedef struct sllLast
{
    elementType data[MAXLEN];
    int last;
}seqList;

//初始化
void initialList(seqList &S)
{
    S.last=-1;
}

//求表长度
int listLength(seqList S)
{
    return S.last+1;
}

//按序号取元素
bool getElement(seqList S,int i,elementType &x)
{
    if(i<1 || i>S.last+1)      //i 为元素编号，有效范围在 1--S.last+1 之间
        return false;
    else
    {
        x=S.data[i-1];
        return true;
    }
```

```
}

//查找元素 x，成功：返回元素编号；失败：返回 0
int listLocate(seqList S,elementType x)
{
    int i;
    for(i=0;i<=S.last;i++)
    {
        if(S.data[i]==x)
            return i+1;    //找到，转换为元素编号输出
    }
    return 0;
}

//插入元素
int listInsert(seqList &S,elementType x, int i)
{
    int k;
    if(S.last>MAXLEN-1)
        return 0;  //表满，返回 0
    else if(i<1 || i>S.last+2)
        return 1;  //插入位置查处范围，返回 1
    else
    {
        for(k=S.last;k>=i-1;k--)
            S.data[k+1]=S.data[k];
        S.data[i-1]=x;
        S.last++;
        return 2;
    }
}

//删除元素
int listDelete(seqList &S,int i)
{
    int k;
    if(S.last==-1)
        return 0;    //空表，返回 0
    else if(i<1 || i>S.last+1)
        return 1;    //删除元素编号超出范围，返回 1
    else
```

```
        {
            for(k=i;k<=S.last;k++)
                S.data[k-1]=S.data[k];
            S.last--;
            return 2;
        }
}

//7. 打印表中所有元素
void printList(seqList L)
{
    int i;
    for(i=0;i<=L.last;i++)
        cout<<L.data[i]<<"\t"; //元素之间以制表符分割
    cout<<endl;
}

//8. 交互输入数据元素--特殊输入结束
void listInputC(seqList &L)
{
    if(L.last>=0)
    {
        cout<<"顺序表已经存在，请先初始化，再输入元素。"<<endl;
        return;
    }

    elementType x;

    cout<<"请输入数据元素(整数，-9999 退出):"<<endl;
    cout<<"x=";
    cin>>x;

    while(x!=-9999)
    {
        L.last++;
        L.data[L.last]=x;

        cout<<"x=";
        cin>>x;
    }
}
```

```
//随机数创建顺序表
void rndCList(seqList &L)
{
    int i;
    int n,m;

    L.last=-1;
    cout<<"请输入要产生的随机数个数，n=";
    cin>>n;

    if(n>MAXLEN-1)
    {
        cout<<"您要求产生的随机数个数超出了查找表长度"<<MAXLEN-1<<"，创建顺序表失败。"<<endl;
        return;

    }
    cout<<"请输入控制随机数大小参数，比如 100 以内数，请输入 100，m=";
    cin>>m;

    srand((unsigned)time(NULL));    //产生随机数种子
    //srand((unsigned)GetTickCount());    //产生随机数种子
    for(i=0;i<n;i++)                    //随机数写入排序表 A[]
        L.data[i]=rand()%m;
    L.last=n-1;                         //表长度为 n
    cout<<endl;
}
```

2.2 试用顺序表表示较多位数的大整数，以便于这类数据的存储。请选择合适的存放次序，并分别写出这类大数的比较、加、减、乘、除等运算，并分析算法的时间性能。

【解】顺序表 0 单元存放操作数的符号，区分操作数是正数还是负数；为方便处理运算时进位和借位，数据的低位存放数组低位，高位存放数组高位。

【时间性能】

加减法：O(max(m,n))

乘除法：O(m×n)

2.3 试用顺序表表示集合，并确定合适的约定，在此基础上编写算法以实现集合的交、并、差等运算，并分析各算法的时间性能。

【解】

```
//求 C=A∩B
```

依次读取 A 的元素，检查次元素是否在 B 中，若在 B 中，则为交集元素，插入 C 中。

```
void interSet(seqList A, seqList B, seqList &C)
{
    int i;
    for(i=0;i<A.listLen;i++)
    {
        if(listLocate(B,A.data[i])!=0)     //A.data[i]在 B 中出现，是交集元素，插入 C 中
            listInsert(&C,A.data[i],C.listLen+1);
    }
}
```

//求 C=A∪B

现将 A 中元素全部插入 C 中。依次读取 B 中元素，检查是否出现在 A 中，若不在 A 中，则为并集元素，插入 C 中。

```
void mergeSet(seqList A, seqList B, seqList &C)
{
    int i;
    for(i=0;i<A.listLen;i++)     //A 中元素全部插入 C 中
    {
        listInsert(&C,A.data[i],C.listLen+1);
    }

    for(i=0;i<B.listLen;i++)
    {
        if(listLocate(A,B.data[i])==0)       //B.data[i]不在 A 中，插入 C
            listInsert(&C,B.data[i],C.listLen+1);
    }
}
```

//求 C=A-B

依次读取 A 中元素，检查是否在 B 中出现，若不在 B 中，则为差集元素，插入 C 中。

```
void differenceSet(seqList A,seqList B,seqList &C)
{
    int i;
    for(i=0;i<A.listLen;i++)
    {
        if(listLocate(B,A.data[i])==0)
            listInsert(&C,A.data[i],C.listLen+1);   //A.data[i]不在 B 中，插入 C
    }
}
```

【算法分析】

时间复杂度：O(|A| × |B|)

2.4 假设顺序表 L 中的元素递增有序，设计算法在顺序表中插入元素 x，要求插入后仍保持其递增有序特性，并要求时间尽可能少。

【解】

如果表空间满，插入失败，返回-1；否则，从 L 最后一个元素开始，与 x 比较，若大于 x，元素后移，直到 L 中元素小于或等于 x，这个元素的后面的单元即为 x 的插入位置，插入成功返回插入位置。

```
//空间满：返回值-1；正确插入：返回表中的插入位置
int incInsert(seqList &L,elementType x)
{
    int i=L.listLen-1;
    if(L.listLen==MAXLEN)
        return -1;   //表空间已满，不能插入新的元素
    else
    {
        while(i>=0 && L.data[i]>x)
        {
            L.data[i+1]=L.data[i];
            i--;
        }
    }
    L.data[i+1]=x;   //插入 x
    L.listLen++;      //修改表长度

    return i+2;     //成功插入，返回 x 在顺序表中的插入位置（元素编号）
}
```

【算法分析】

时间复杂度：O(n)

2.5 假设顺序表 L 中的元素递增有序，设计算法在顺序表中插入元素 x，并要求在插入后也没有相同的元素，即若表中存在相同的元素，则不执行插入操作。

【解】

与上题相似，只是在移动插入元素之前，检查 L 中是否已经存在值 x，若存在，插入失败，返回-2。

```
//空间满：返回值-1；x 已经存在返回-2；正确插入：返回表中的插入位置
int incInsert(seqList &L,elementType x)
{
    int i;
```

```
    if(L.listLen==MAXLEN)
        return -1;                          //表空间已满，不能插入新的元素
    else
    {
        for(i=0;i<L.listLen;i++)
            if(L.data[i]==x)
                return -2;                  //元素 x 已经存在，插入失败，返回-2
        i=L.listLen-1;
        while(i>=0 && L.data[i]>x)     //后移元素
        {
            L.data[i+1]=L.data[i];
            i--;
        }
    }
    L.data[i+1]=x;   //插入 x
    L.listLen++;      //修改表长度

    return i+2;       //成功插入，返回 x 在顺序表中的插入位置（元素编号）
}
```

2.6 设计算法以删除顺序表中重复的元素，并分析算法的时间性能。

【解】

【分析】

三重循环实现。第一层循环，从左往右依次取出 L 元素，用 i 指示；第二层循环，对 i 元素在 L 中循环查重，用下标 j 指示；第三重循环，删除重复元素。

查重和删除从 j=L.listLen-1 开始，效率稍微好一点，因为这样重复元素本身不需重复移动。如果从 i+1 开始查重、删除，则 j+1 以后的重复元素会被移动。

【算法描述】

```
void DeleteRepeatData(seqList & L)
{
    int i,j;
    if(L.listLen==0)
    {
        cout<<"当前顺序表空！"<<endl;
        return;
    }
    if(L.listLen==1)
    {
        cout<<"当前顺序表只有一个元素！"<<endl;
        return;
    }
```

```
    i=0;
    while(i<L.listLen-1)
    {
        for(j=L.listLen-1; j>i; j--)  //从后往前删除，效率较高
        {
            if(L.data[i]==L.data[j])  //元素重复，调用删除
            {
                listDelete(&L,j+1); //调用删除函数，下标差 1，所以+1
                //以下部分代码是直接删除，没有调用删除函数
                //int k;
                //for(k=j; k<L.listLen-1; k++)
                //    L.data[k]=L.data[k+1];
                //L.listLen--;
            }
        }
        i++;
    }
}
```

【算法分析】时间性能：$O(n^3)$

2.7 假设顺序表 L 中的元素按从小到大的次序排列，设计算法以删除表中重复的元素，并要求时间尽可能少。要求：

（1）对顺序表（1,1,2,2,2,3,4,5,5,5,6,6,7,7,8,8,8,9）模拟执行本算法，并统计移动元素的次数。

（2）分析算法的时间性能。

【解】

【分析】

将元素分成两个部分：已经处理元素和待处理元素。已经处理部分返回 L 中，用下标 i 指示最后一个元素，初始化 i=0。待处理部分用下标 j 指示第一个元素，初始化 j=1.

左边下标小于 i 的元素已经处理好重复，等于 i 是当前正在处理的元素，将 data[i]与 data[j]进行比较，会出现下列情况：

① data[i]==data[j]，说明 j 指示的是 i 的重复元素，继续处理 j 的下一个元素，即执行 j++。

② data[i]<data[j]， 说明 j 指示元素与 i 的元素不同，如果 i+1!=j，将 j 元素复制到 i+1，即：L.data[i+1]=L.data[j]，再执行 j++，i++；若 i+1==j，说明 j 是 i 的直接后继，无需复制，直接执行 i++，j++。

循环执行上述操作，直到表尾。

修改 L 的长度为 i+1。

【算法描述】

```
void DeleteRepeatData(seqList & L)
{
    int i,j;   //分别指向已处理部分最后元素和未处理部分第一个元素，皆为数组下标

    if(L.listLen<2)
        return;    //少于 2 个元素，直接退出
    i=0;   //初始化 i 指向第一个元素
    j=1;   //j 指向第二个元素
    while(j<L.listLen)
    {
        if(L.data[i]==L.data[j])   //j 为重复元素，j 后移
            j++;
        else   //因为 L 递增，所以剩下情况即 L.data[i]<L.data[j]，j 为目标元素
        {
                //如果 j==i+1，说明 j 紧随 i，无需移动元素，直接 i++、j++即可
            if((i+1)!=j)
                L.data[i+1]=L.data[j];   //j 元素复制到 i+1

            i++;   //无论那种情况，都需要同时后移 i、j
            j++;
        }
    }
    L.listLen=i+1;   //修改表的实际长度
}
```

【算法分析】时间复杂度 O(n)。上例中只需移动 8 个元素。如果有 n 个不同元素，则最多移动元素 n-1 次。

2.8 若递增有序顺序表 A、B 分别表示一个集合，设计算法求解 A=A∩B，并分析其时间性能。

【解】

【分析】

审题：A、B 为集合，说明两个表中都没有重复元素

A=A∩B，即要求交集元素就放在 A 表中，而不是创建一个新表来存放。

设置两个指针 ia、ib 分别指向 A、B 表当前处理的元素；

设置一个指针 i 指示已经求取的交集元素在 A 的表中的最后元素位置；

比较 A、B 表当前元素，会出现以下三种情况

(1) A.data[ia]==B.data[ib]，则 ia 或 ib 是交集元素，如果 ia!=i+1，将 ia 元素复制到 i+1，即：A.data[i+1]=A.data[ia]；否则，若 ia==i+1，说明 ia 就在 i 的后面，无需复制元素。最后，无论那种情况，修改指针：ia++，ib++，i++

(2) A.data[ia]<B.data[ib]，当前元素为非交集元素，只需移动 ia，即 ia++

(3) A.data[ia]>B.data[ib]，当前元素为非交集元素，只需移动 ib，即 ib++

重复以上过程，直至 A、B 中至少一个表结束。
修改 A 表长度为 i+1。
【算法描述】

```
void InterSet(seqList &A, seqList &B)
{
    int i=-1;  //为了最后更新交集元素表长度操作一致，初始化为-1
    int ia=0, ib=0;  //A、B 表当前元素的数组下标
    while(ia<A.listLen && ib<B.listLen)
    {
        if(A.data[ia]==B.data[ib])  //ia 和 ib 指示的是交集元素
        {
            if(ia!=i+1)  //ia 元素复制到 i+1，否则 ia 位置即目标位置，不需复制元素
                A.data[i+1]=A.data[ia];
            i++;
            ia++;
            ib++;
        }
        else if(A.data[ia]<B.data[ib])  //以下为非交集元素处理
            ia++;
        else
            ib++;
    }
    A.listLen=i+1;  //更新 A 表长度，使等于交集元素个数。
}
```

2.9 递增有序顺序表 A、B 分别表示一个集合，设计算法求解 A=A－B，并分析其时间性能。

【解】【分析】

审题：A=A-B，即要求要利用 A 表的空间保存差集元素，而不是创建一个新表。

设置两个指针 ia、ib 分别指向 A、B 表当前处理的元素；

设置一个指针 i 指示已经求取的差集元素在 A 的表中的最后元素位置；

比较 A、B 表当前元素，会出现以下三种情况

(1) A.data[ia]==B.data[ib]，则 ia 或 ib 是交集元素，不是 A-B 中元素，直接跳过，修改指针：ia++，ib++。

(2) A.data[ia]>B.data[ib]，ia 指示的元素可能在 B 表 ib 指示的元素后面，ia 不动，移动 ib，即 ib++。

(3) A.data[ia]<B.data[ib]，ia 指示的元素不可能在 B 中出现，故比为 A-B 中元素。需要的话迁移到目标位置，即 (i+1)!=ia 时，执行 A.data[i+1]=A.data[ia]。若(i+1)==ia，ia 即为目标位置，无需复制迁移。迁移完成，移动指示器：i++，ia++。

重复以上过程，直至 A、B 中至少一个表结束。

还有一种情况：B 表已结束，但 A 表尚未结束，说明 A 剩下元素不在 B 中，全为 A-B

中元素，全部迁移到目标位置。

最后，修改 A 表长度为 i+1。

【算法描述】

```
void SetSubtraction(seqList & A, seqList & B)
{
    int i=-1;   //指示 A 中已经处理的最后元素
    int ia=0, ib=0;   //指示 A、B 中，当前待处理的元素，初始指向第一个元素
    while( ia<A.listLen && ib<B.listLen )
    {
        if(A.data[ia]==B.data[ib])     //非 A-B 中元素，ia、ib 同时后移
        {
            ia++;
            ib++;
        }
        else if(A.data[ia]>B.data[ib])
            ib++;     //此时，ia 指示元素可能在 B 中 ib 指示的元素后面，移动 ib。
        else   //此为，A.data[ia]<B.data[ib]，因为递增性，ia 指示的元素不可能在 B 中。
        {      //所以 ia 指示元素必在 A-B 中。
            //如果(i+1)==ia，说明 ia 元素不需迁移位置，直接为 A-B 中元素
            if(i+1!=ia)   //(i+1!=ia)，需要将 ia 指示元素迁移到目标位置 i+1
                A.data[i+1]=A.data[ia];

            i++;   //A-B 集合最后元素指示器后移
            ia++; //A 的指示器后移
        }
    }
        //处理 B 已经，A 尚未结束情况，A 中剩下部分元素全部为 A-B 元素
    while(ia<A.listLen)
    {
        if(i+1!=ia)
            A.data[i+1]=A.data[ia];

        i++;   //A-B 集合最后元素指示器后移
        ia++; //A 的指示器后移
    }
    A.listLen=i+1;   //更新表 A 的长度，使等于|A-B|
}
```

【算法分析】时间性能 O(|A| + |B|)

【思考问题】A、B 两表谁先结束？

下面是本题的另一种解法，因为用到 A 表中交集元素的删除，所以效率较差。

【算法描述-1】

```
void SetSubtraction1(seqList & A, seqList & B)
{
    int ia=0, ib=0;   //ia、ib 指示 A、B 表中当前元素。初始指向第一个元素。
    while(ia<A.listLen && ib<B.listLen)
    {
        if(A.data[ia]<B.data[ib])   //元素不在 B 中，移动 A 到下一个元素
            ia++;
        else if(A.data[ia]>B.data[ib])   //A 元素可能在 B 中，移动 B 到下一个元素
        {
            ib++;
            if(ib>=B.listLen)
                break;
        }
        else   //A.data[ia]==B.data[ib]，删除 A 中元素，同时移动指针
        {
            listDelete(&A, ia+1);
            //ia++;   不能增加 ia，因为 A 已经往前移动一个元素
            ib++;
            if(ib>=B.listLen)
                break;
        }
    }
}
```

【算法分析】时间性能 O(|A| + |B|)

2.10 假设带头结点的单链表是递增有序的，设计算法在其中插入一个值为 x 的结点，并保持其递增特性。

【解】

首先需要找到插入点前一个结点的指针 p，为此初始化时 p 指向头结点，循环比较 p->next->data 与 x，如果 p->next->data<x，后移 p 指针，循环结束时，p 即指向插入点前一个结点。申请新结点，新结点接入链表。

```
void incListInsert(linkedList &L,elementType x)
{
  node* u;
  node* p=L;   //p 指向头结点（头指针）
  while(p->next!=NULL && p->next->data<x)   //搜索插入位置
  {
    p=p->next;   //P 后移一个结点
  }
        //循环结束 p 指向插入位置的前一个结点
```

```
    u=new node;   //产生新结点
    u->data=x;
    u->next=p->next;
    p->next=u;
  }
```

2.11 设计算法以删除链表中值为 x 的元素结点。

【解】【分析】

与基本运算基本相同，只是这里要根据结点的元素值来搜索待删除目标结点的指针。与基本删除类似，我们用一个指针 p 指向待删除结点的前驱，则 p->next 指向目标结点。初始化时让 p=L，即指向头结点。循环将 p->next->data 与 x 进行比较，若相等，p->next 指示的即是待删除结点。

此外，要处理 x 不在表中情况。

本题也可以调用单链表的基本运算 listLocate()和 listDelete()两个函数来完成。

【删除第一个 x 结点算法描述】

```
bool listDeleteX(node* L, elementType x)
{
    node* u;
    node* p=L;     //指向头结点
    int succ=0;   //是否删除成功标记，成功删除 succ=1，失败 succ=0。
    while(p->next)
    {
        if(p->next->data==x)
        {       //找到目标结点，删除此结点
                //p 指向目标结点的前驱，p->next 指向待删除目标结点，并以 u 保存。
            u=p->next;
            p->next=u->next;   //ai-1 的 next 指向 ai+1 节点，或为空（ai-1 为最后节点）
            delete u;       //释放目标结点占据的空间
            succ=1;         //标记结点成功删除

            break;          //退出循环
        }
        p=p->next;
    }

    if(succ==1)
        return true;     //成功删除，返回 true
    else
        return false;   //删除失败，返回 false，x 不在链表中。
}
```

【算法分析】时间复杂度 O(n)。

【调用基本函数删除第一个 x 结点算法描述】

```
bool listDeleteX(node* L, elementType x)
{
    node* u;
    int i;

    listLocate( L, u, i );

    if( i!=0 )
    {
        listDelete( L, i );
        return 1;
    }
    else
        return 0;
}
```

【删除所有 x 结点算法描述】

```
bool listDeleteX1(node* L, elementType x)
{
 node* u;
 node* p=L;     //p 指向待删除结点的前驱，初始化指向头结点
 int succ=0;   //是否删除成功标记，成功删除 succ=1，失败 succ=0。
 while(p->next)
 {
     if(p->next->data==x)
     {          //找到目标结点，删除此结点
         u=p->next;   //p 指向目标结点的前驱，p->next 指向待删除目标结点，并以 u 保存。
         p->next=u->next;   //ai-1 的 next 指向 ai+1 节点，或为空（ai-1 为最后节点）
         delete u;     //释放目标结点占据的空间
         succ=1;        //标记结点成功删除
     }                    //删除结点后 p->next 为一个新的结点，可能值也为 x，故 p 不移动
     else
         p=p->next;
 }

 if(succ==1)
     return true;     //成功删除，返回 true
```

```
    else
        return false;   //删除失败，返回 false，x 不在链表中。
    }
```

2.12 设计算法将两个带头结点的单循环链表 A，B 首尾相接为一个单循环链表 A。

【解】

分为两种情况：

1.不带尾指针的单循环链表合并

合并后保留 A 链表的头结点，A 的尾结点接 B 的首元素结点。B 的尾结点的 next 指向 A。销毁 B 的头结点。

```
void scMergeAB(node *&A, node *&B)
{
 node *u,*p;
      p=B;
 if(p->next==B)                  //B 表为空表，删除 B 表头，其它不变
 {
       delete B;
       return;
 }
          //以下为 B 表不空情况
 p=A;
 while(p->next!=A)               //p 指向 A 的尾结点
 {
       p=p->next;
 }
 p->next=B->next;                //A 的尾结点接 B 的首元素结点

 p=B;
 while(p->next!=B)               //p 指向 B 的尾结点
 {
       p=p->next;
 }
 p->next=A;                       //形成循环
 delete B;                       //释放 B 的头结点
}
```

2.带尾指针的单循环链表合并

保存 A 表头指针 A->next 到 u；将 B 表首元素结点接到 A 表尾；释放 B 表头结点；B 的尾结点的 next 指向 A；形成大循环；B=A。

```
void scrMergeAB(linkedList &A, linkedList &B)
```

```
{
 node* u;
 u=A->next;                    //存放 A 的头指针
 A->next=B->next->next;        //B 表头链接到 A 表尾
 delete B->next;               //释放 B 的头结点
 B->next=u;                    //B 表尾的 next 指向 A 的头结点，形成大循环
 A=B;                          //A、B 同时指向新的尾结点，成为尾指针
}
```

2.13 假设链表 A、B 分别表示一个集合，试设计算法以判断集合 A 是否是集合 B 的子集，若是，则返回 1，否则返回 0，并分析算法的时间复杂度。

【解】

设置指针 pa、pb 分别指向 A、B 表结点。依次取 A 的结点，从 B 表首元素结点开始检查，如果 A 元素不在 B 中，返回 0；如果 A 的每个元素都在 B 表中，返回 1.

```
int llSubset( node* A, node* B )
{
 node *pa, *pb;
 int suc=0;                //设置 A 元素是否在 B 中标志
 pa=A->next;               //pa 指向 A 的首元素结点
 while(pa!=NULL)
 {
      pb=B->next;          //pb 指向 B 表的首元素结点
      suc=0;
      while(pb!=NULL)
      {
           if(pa->data==pb->data)
           {
                suc=1;
                break;
           }
           else
                pb=pb->next;
      }
      if(suc==0)           //pa->data 不在 B 中，A 不是 B 的子集，返回 0
           break;

      pa=pa->next;
 }
 return suc;
}
```

2.14 假设递增有序的带头结点的链表 A、B 分别表示一个集合，试设计算法以判断集合 A 是否是集合 B 的子集，若是，则返回 1，否则返回 0，并分析算法的时间复杂度。

【解题分析】

审题：A、B 是集合，即没有重复元素。

简单的做法是使用通用判定子集方法：依次取 A 的每一个元素，判定是在 B 中，全在 B 中，A 是 B 子集；若一个不在 B 中，则 A 不是 B 的子集。用二层循环实现，时间复杂度为 O(|A|+|B|)。这个方法没有使用“递增有序”条件，对本题时间性能不是最好。

用两个指针 pa 和 pb 分别指向 A、B 的结点，初始化指向首元素结点。通过一层循环，比较当前 pa 和 pb 指向结点的元素值大小，分为以下三中情况：

① pa->data==pb->data，说明 pa 指示元素在 B 中，后移指针 pa，有因是集合，可同时后移 pb，即执行：pa=pa->next，pb=pb->next。

② pa->data>pb->data，说明 pa 指示元素可能在 B 中 pb 指示的元素后面，pa 不动，后移 pb 指针，即执行：pb=pb->next。

③ pa->data<pb->data，说明 pa 指示元素不可能在 B 中，因 B 中后面的元素值越来越大。直接返回 0，即 A 不是 B 的子集。

循环执行上述操作，直到其中一个表到达表尾。

结束循环后，要根据 A、B 中谁到达表尾作如下判断：

如果 A 表到达表尾，不管 B 表如何，说明 A 的所有元素都在 B 中，A 是 B 的子集。返回 1。为什么呢？

否则，B 到达表尾，而 A 表中尚有元素，且不在 B 中，A 不是 B 的子集，返回 0。

【算法描述】

```
int SubSetDecision( node* A, node* B )
{
    node *pa, *pb;
    pa=A->next;   //pa 和 pb 分别指向 A 和 B 表的首元素结点
    pb=B->next;
    while(pa!=NULL && pb!=NULL)
    {
        if(pa->data==pb->data)
        {       //pa 指示元素在 B 中，因是集合，pa、pb 同时后移一个结点
            pa=pa->next;
            pb=pb->next;
        }
        else if(pa->data>pb->data)
            pb=pb->next;   //pa 指示元素可能在 B 中 pb 指示的元素后面，后移 pb
        else
            return 0;   //此时，pa->data<pb->data，pa 指示元素不可能在 B 中，A 非 B 子集
    }
        //下面根据 A、B 的结束情况，判定 A 是否 B 的自己
    if(pa==NULL)
```

```
            return 1;   //A 到表尾，不管 B 表如何，A 全部元素在 B 中，是 B 的子集
        else
            return 0;   //B 到表尾，A 未到表尾，A 后面元素不在 B 中，A 非 B 的子集

    }
```

【算法分析】时间复杂度 O(|A|+|B|)。

2.15 假设链表 A、B 分别表示一个集合，设计算法以求解 C= A∩B，并分析算法的时间复杂度。

【解】算法思想：设置指针 pa、pb 分别指向 A、B 的元素结点，用两层循环求 A、B 的交集元素。第一层循环依次取 A 的结点，第二层循环检查 A 当前元素知否在 B 中，若在 B 中则是交集元素，申请新结点，元素值等于交集元素值，尾插到表 C。

【算法描述】

```
//求 C=A∩B
void interSet(linkedList &A,linkedList &B,linkedList &C)
{
 node* pa, *pb, *Rc, *u;
 Rc=C;                  //C 表尾指针，空表时头尾指针相同
 pa=A->next;          //Pa 指向 A 的第一个元素结点
 while(pa!=NULL)   //循环取 A 的元素（结点）
 {
      pb=B->next;   //Pb 指向 B 的第一个元素结点
      while(pb)
      {
          if(pa->data==pb->data)   //交集元素，插入 C 中
          {
              u=new node;
              u->data=pa->data;   //或 u->data=Pb->data，新结点赋值。
              Rc->next=u;           //尾插法在 C 中插入 u，
              Rc=u;                  //修改 C 的尾指针 Rc，指向 u
              break;                 //退出 B 表循环
          }
          else
              pb=pb->next;          //否则，pb 后移一个结点
      }
      pa=pa->next;                  //Pa 后移，取 A 的下一个元素
 }
 Rc->next=NULL; //表 C 结束
}
```

【算法分析】时间复杂度 O(|A|×|B|)。

2.16 假设递增有序的带头结点的链表 A、B 分别表示一个集合，设计算法以求解 C= A∩B，并分析算法的时间复杂度。

【解】

设置指针 pa、pb 分别指向 A、B 链表结点。比较 pa->data 和 pb->data，如果 pa->data 大，pb 后移；如果 pb->datada，则 pa 后移；如果 pa->data==pb->data，则为交集元素，申请新结点，赋值 p->data（或 pb->data），尾插法插入 C 表。重复上述操作，直到 A 或 B 表有一个结束。

```
//求递增有序集合的交集
void InterSet(linkedList &A,linkedList &B,linkedList &C)
{
 node* Pa, *Pb, *Rc, *u;
 Rc=C; //C 表尾指针，空表时头尾指针相同
 Pa=A->next;
 Pb=B->next;   //Pa 和 Pb 分别指向 A 和 B 表的第一个元素节点
 while(Pa!=NULL && Pb!=NULL)   //A 和 B 只要一个没有元素，即没有交集元素，退出
 {
      if(Pa->data<Pb->data)   //B 中没有 A 的当前元素，即 Pa->data
           Pa=Pa->next;          //取 A 的下一个元素，回去循环
      else if (Pa->data>Pb->data)   //A 当前元素值大于 B 当前元素值，移动 Pb 继续搜索 B
           Pb=Pb->next;
      else   //Pa->data=Pb->data，即为交集元素，在 C 中产生新节点，Pa,Pb 同时后移，回去循环
      {
           u=new node;
           u->data=Pa->data;   //或 u->data=Pb->data，新结点赋值。
           Rc->next=u;            //尾插法在 C 中插入 u，
           Rc=u;                    //修改 C 的尾指针 Rc，指向 u
           Pa=Pa->next;          //Pa 和 Pb 同时后移，分别取 A 和 B 的下一个元素
           Pb=Pb->next;
      }
      Rc->next=NULL; //表 C 结束
 }
}
```

2.17 假设递增有序的带头结点的链表 A、B 分别表示一个集合，设计算法以求解 A= A∩B，并分析算法的时间复杂度。

【解】算法思想：将 A 中非交集结点删除，B 表不变。用指针 pa 指 A 表结点，pb 指 B 表结点，为了便于 A 表结点删除，pa 初始化为 A，用 pa->next->data 来搜索交集元素，pb 初始化为 B->next，指向 B 表的第一个元素结点。当 A、B 表都有结点时，循环比较 pa->next->data 与 pb->data，会出现如下三种情况：

(1)pa->next->data<pb->data

说明 pa->next 结点不是交集元素，删除 pa->next 结点，pa 和 pb 保持不变。

(2)pa->next->data>pb->data

pa->next 结点可能在 pb 的后面，后移 pb，即：pb=pb->next，pa 不动。

(3)pa->next->data==pb->data

pa->next 是交集结点，同时移动 pa 和 pb，即：pa=pa->next，pb=pb-next。

当 A 表结束时，A 即为交集链表。如果 B 表结束，A 表没有结束，则 p->next 至表尾的结点都不是交集结点，全部删除。

【算法描述】

```
//求 A=A∩B
void interSet(linkedList &A,linkedList &B)
{
    node* pa, *pb, *u, *p;
    pa=A;              //Pa 指向 A
    pb=B->next;    //pb 指向 B 的首元素结点
    while(pa->next && pb)   //A、B 都有结点，循环处理
    {
        if(pa->next->data<pb->data)
        {                  //pa->next 不是交集结点，删除 pa->next 结点，pa 和 pb 不变
            u=pa->next;
            pa->next=u->next;
            delete u;
        }
        else if(pa->next->data>pb->data)   //pa->next 可能在 pb 后面，pa 不变，pb 后移
            pb=pb->next;
        else       //pa->next->data==pb->data 情况，pa->next 是交集结点，同时后移 pa、pb
        {
            pa=pa->next;
            pb=pb->next;
        }
    }

      //如果 A 表未结束，pa->next 及后面所有结点都是非交集结点，删除
    if(pa->next)    //置 A 表结束
    {
        u=pa->next;
        pa->next=NULL;
        p=u;
      //删除 A 表剩下的非交集结点
```

```
        while(p)
        {
            u=p;
            p=p->next;
            delete u;
        }
    }
}
```

【算法分析】时间复杂度 O(|A|+|B|)。

2.18 假设递增有序的带头结点的单循环链表 A、B 分别表示两个集合，设计算法以求解 A= A ∪B，并分析算法的时间复杂度。

【解】算法思想：本题有不同的理解，一是 B 表的处理，简单的处理 B 表中的并集元素，申请新结点插入 A 中，B 表保持不变。并集链表是否要求递增有序，简单的处理是不要求并集链表递增有序。以下算法即按此实现。

用两个指针 pa、pb 分别指向 A 和 B 的结点，初始化指向两表的首元素结点。当 A、B 表都有元素时，循环比较 pa->data 和 pb->data，按以下情况分别处理：

(1)pa->data<pb->data

pa 是并集结点，pa 后移，pb 不变。

(2)pa->data>pb->data

pb 是并集结点，申请新结点，值为 pb->data，插入 pa->next 位置，pb 后移，因为 pa 可能出现在 pb 后面，pa 不变。

(3)pa->data==pb->data

pa、pb 同时后移。

为了 A 表的插入处理，pa 初始化指向 A 表的头结点，即 pa=A。比较时用 pa->next->data 与 pb->data 进行比较。

【算法描述】

```
void unionSet(node *A, node *B)
{
 node*pa,*pb,*u,*R;
 pa=A;                    //pa 初始化指向 A 的头结点，为了 A 表的插入处理
 pb=B->next;              //pb 初始化指向第一个元素结点
 while(pa->next!=A && pb!=B)
 {
     if(pa->next->data<pb->data)          //pa 为并集结点
         pa=pa->next;
     else  if(pa->next->data>pb->data)    //pa 有可能在 pb 的后面，pb 是交集结点，申请新结点插入 pa 之后
     {                                    //pa 不动
         u=new node;
```

```
            u->data=pb->data;
            u->next=pa->next;
            pa->next=u;
            pb=pb->next;
        }
        else      //pa->data==pb->data，pa、pb 同时后移
        {
            pa=pa->next;
            pb=pb->next;
        }
    }
        //如果 B 没有结束，pb 及其后结点皆为并集结点，插入 A 表
    while(pb!=B)
    {
        u=new node;
        u->data=pb->data;
        u->next=pa->next;
        pa->next=u;
        pa=u;                  //pa 指尾结点
        pb=pb->next;
    }
}
```

【算法分析】时间复杂度：O(|A|+|B|)。

2.19 假设链表 A、B 分别表示两个集合，设计算法以求解 C= A∪B，并分析算法的时间复杂度。

【解】算法思想：第一步将 A 的所有元素，尾插到新表 C。再用 2 层循环搜索 B 中不在 A 中的元素，尾插到表 C。

【算法描述】

```
//求 C=A∪B
void unionSet(linkedList &A,linkedList &B,linkedList &C)
{
 node* pa, *pb, *Rc, *u;
 bool succ=1;        //A 的元素是否在 B 中的标记
 Rc=C;                 //C 表尾指针，空表时头尾指针相同
 pa=A->next;         //Pa 指向 A 的第一个元素结点
          //A 表元素尾插入 C 中
 while(pa)
 {
     u=new node;
     u->data=pa->data;
```

```
        Rc->next=u;
        Rc=u;
        pa=pa->next;
    }

    pb=B->next;         //pb 指向 B 的第一个元素
    while(pb)           //B 中不在 A 中的元素尾插到 C
    {
        succ=1;
        pa=A->next;   //pa 指向 A 的第一个元素结点
        while(pa)
        {
            if(pb->data==pa->data)      //B 中当前元素在 A 中，不是并集元素
            {
                succ=0;
                break;          //退出 A 的循环
            }
            else
                pa=pa->next;   //A 后移一个结点
        }

        if(succ==1)             //B 当前元素是并集元素，尾插到 C
        {
            u=new node;
            u->data=pb->data;
            Rc->next=u;
            Rc=u;
        }
        pb=pb->next;            //取 B 的下一个结点

    }
    Rc->next=NULL; //表 C 结束
}
```

【算法分析】时间复杂度 O(|A|×|B|)。

2.20 设计算法将两个递增有序的带头结点的单链表 A、B 合并为一个递增有序的带头结点的单链表，并要求算法的时间复杂度为两个表长之和的数量级。

【解】算法如下

```
void Merge_LinkList1(node* La,node *Lb)
{
 //合并后，以 La 为头结点指针，Lb 指示的头结点删除
```

```
node*pa,*pb,*R;   //pa 和 pb 为 La 和 Lb 当前结点指针
                  //R 为已经部分的尾指针
pa=La->next;
pb=Lb->next;
R=La;
while(pa!=NULL && pb!=NULL)
{
    if(pa->data<=pb->data)
    {
        R->next=pa;   //pa 指示结点，接到合并表尾
        R=pa;   //移动尾指针
        pa=pa->next;   //pa 指向下一个结点
    }
    else
    {
        R->next=pb; //pb 结点接到合并表尾
        R=pb;   //移动尾指针
        pb=pb->next;   //pb 移到下一个结点
    }
}
   //以下处理其中一个链表已经结束，另一个表尚未结束，
   //元素值都不小于已经接入部分，直接接入即可
if(pa!=NULL)   //Lb 已结束，La 未结束
    R->next=pa ;
else    //La 已结束，Lb 未结束
    R->next=pb ;
delete(Lb);   //删除  Lb  头结点
}
```

2.21 设计算法将链表 L 就地逆置，即利用原表各结点的空间实现逆置。

解：算法如下

```
void ListReverse_L(node *L)
{
    //单链表的就地逆置
    node *p,*q;    //p 指向当前待逆置结点，q 指向 p 的下一个结点
    p=L->next;    //p 指向第 1 个元素结点
    L->next=NULL;
    while(p!=NULL)
    {
        q=p->next;    //指向待逆置结点的下一个结点
        p->next=L->next;    //p->next 指向已经逆置部分的第一个元素结点
```

```
            L->next=p;     //p 成为第 1 个元素结点，到此结点 p 逆置完成
            p=q;     //p  和 q 都指向未逆置部分的第一个结点
        }
    }
```

2.22 设计算法将两个递增有序的带头结点的单链表 A、B 合并为一个递减有序的带头结点的单链表，并要求算法的时间复杂度为两个表长之和的数量级。

【解】算法如下

```
node* ListJoinAndReverse_L(node *A, node *B)
{
 node *pa,*pb,*p,*L;
        //pa、pb 分别指向 A 和 B 待逆置的结点
 L=A;  //以 L 为合并后的头结点指针，即 A 的头结点指针
 pa=A->next;
 pb=B->next;
 L->next=NULL;
 while(pa!=NULL && pb!=NULL)
 {
      if(pa->data<pb->data)
      {
           p=pa->next; //p 指向 A 下一个未逆置结点
           pa->next=L->next;
           L->next=pa;  //pa 作为已经合并部分的第一个结点，至此逆置完成
           pa=p;  //pa 指向 A 第一个未逆置结点
      }
      else
      {
           p=pb->next;  //p 指向 B 下一个未逆置结点
           pb->next=L->next;  //pb->next 指向已经合并部分的第一个数据结点
           L->next=pb;  //pb 作为已经合并部分的第一个结点，至此逆置完成
           pb=p; //pb 指向 A 第一个未逆置结点
      }
 }
 //以下处理一个表结束，另一个表未结束情况
 while(pa!=NULL && pb==NULL)  //A 未结束，B 已经结束
 {
          p=pa->next;
      pa->next=L->next;
      L->next=pa;
      pa=p;
 }
```

```
while(pa==NULL && pb!=NULL)   //A 已结束，B 未结束
{
        p=pb->next;
    pb->next=L->next;
    L->next=pb;
    pb=p;
}
return(L);
}
```

2.23 设计算法以判断带头结点的双循环链表 L 是否是对称的，即从前往后和从后往前的输出序列是相同的。若对称，返回 1，否则返回 0。

【解】算法思想：用两个指针 p、r 分别指向链表的首元素结点和尾元素结点，循环比较 p->data 和 r->data 是否相等，若不相等直接返回 false；若相等，则 p 后移一个结点，r 前移一个结点，即执行 p=p->next 和 r=r->prior，继续比较。循环结束，则可判定链表对称。那么循环结束的条件呢？分为两种情况，一种情况是链表对称且有奇数个元素结点，p 和 r 最终将移到中间的结点上，循环的条件为 p!=r。另一种情况是链表对称且有偶数个元素结点，最后一次比较时，p、r 指在相邻 2 个结点，即 p->next==r，移动指针后将出现 r->next==p，或 p->prior==r，此时要退出循环，所以循环条件为 r->next!=p，或 p->prior!=r。

【算法描述】

```
bool SymmetricalDLList(dnode *L)
{
 dnode *p, *r;
 p=L->next;       //p 指向首元素结点
 r=L->prior;      //r 指向尾元素结点
 while(p!=r && r->next!=p)
 {
     if(p->data!=r->data)
         return false;   //不对称，返回 false
     else
     {
         p=p->next;     //p、r 当前指向结点相等，p 后移一个结点，r 前移一个结点
         r=r->prior;
     }
 }
 return true;   //到此，双循环链表对称，返回 true
}
```

【时间复杂度】O(n)。

2.24 设计算法将带头结点的双循环链表 L 就地逆置，即利用原表各结点的空间实现逆置。

【解】

方法一：重接

方法二：用两根指针 p、r 分别指向首元素和尾元素结点。在 p 和 r 不重合情况下，循环处理：交换所指结点的元素值，然后执行 p=p->next，r=r->prior。注意偶数结点情况下，p 和 r 永远不会相等，要处理循环退出的特殊处理，这种情况下交换最后元素的两个结点的指针关系是 p->next==r，要退出循环。

```
//双循环链表就地逆置
void DLListReverse(node *L)
{
    node *P, *u;      //P 指向当前待逆置的结点，u 指向未逆置的下一个结点
    P=L->next;        //P 指向原链表的首元素结点，此结点逆置后为尾结点，需要单独处理

    if(P!=L)          //非空链表，处理第一个结点
    {
        u=P;
        p=p->next;    //p 指向链表下一个结点

        u->next=L;    //建立 next 向循环
        L->prior=u;   //建立 prior 向循环
    }
        //循环逆置剩余结点
    while(P!=L)
    {
        u=P;
        p=p->next;    //p 指未逆置部分下一个结点

        u->prior=L;           //u 头插成为首元素结点，前向指针指向头结点
        u->next=L->next;      //u->next 指向已逆置部分原来的首元素结点
        L->next->prior=u;     //已逆置部分原来首元素结点的 prior 指针指向 u
        L->next=u;            //L->next 指向 u，u 插入成为首元素结点
    }
}
```