线性表

901专业课考纲： 线性表、顺序表、链表的定义、特点、存储结构及相关的基本算法

（Ps：本章节出题花样多，很灵活，在理解顺序表和链表特点的基础上熟练掌握相关的插入删除操作，熟练适应各种线性表的变种题型，由于901不考填空和选择，所以这章以练习熟悉链表的结构和各种变体以及写算法为主，毕竟这章的算法比较基础，适合简单入门）

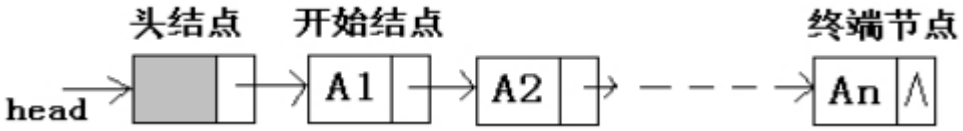
线性表：零个（**设计算法注意临界情况即表元素个数为0的情况**）或多个数据元素的有限序列。

单链表（带头结点）

1）带头结点的单链表中头指针 head 指向头结点，头结点的值域不含任何信息，从头结点的后继结点开始存储信息。头指针 head 始终不等于 NULL，head->next 等于 NULL 的时候链表为空。

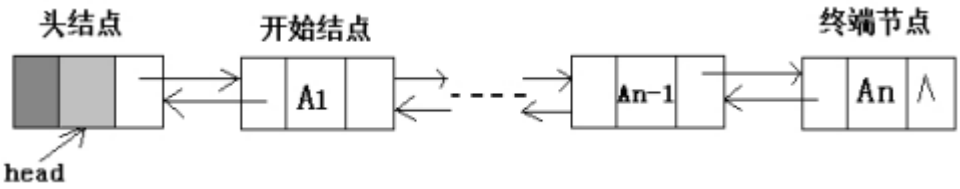
2）不带头结点的单链表其中的头指针 head 直接指向开始结点，即图中的结点 A1，当 head 等于 NULL 的时侯链表为空。

总之，两者最明显的区别是，带头结点的单链表有一个结点不存储信息，只是作为标记，而不带头结点的单链表所有结点都存储信息。



注意：在题目中要区分头结点和头指针，不论是带头结点的链表还是不带头结点的链表，头指针都指向链表中第一个结点，即图 2.2 中的 head 指针。而头结点是带头结点的链表中的第一个结点，只作为链表存在的标志，结点内不存信息。

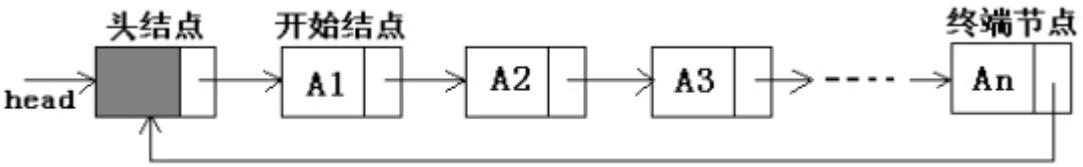
双链表（带头结点）



单链表只能由开始结点走到终端结点，而不能由终端结点反向走到开始结点。如果要求输出从终端结点到开始结点的数据序列，则对于单链表来说操作就非常麻烦。为了解决这类问题我们构造了双链表。如图所示，即为带头结点的双链表。双链表就是在单链表结点上增添了一个指针域，指向当前结点的前驱。这样就可以方便的由其后继来找到其前驱，而实现输出终端结点到开始结点的数据序列。

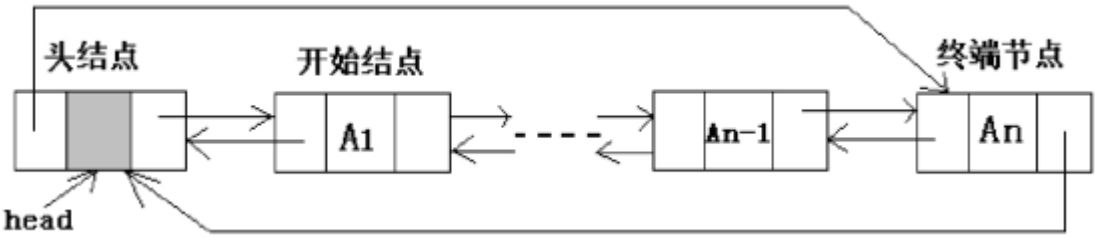
同样，双链表也分为带头结点的双链表和不带头结点的双链表，情况类似于单链表。带头结点的双链表 `head->next` 为 `NULL` 的时候链表为空。不带头结点的双链表 `head` 为 `NULL` 时链表为空。

循环单链表（带头结点）



只要将单链表的最后一个指针域（空指针）指向链表中第一个结点即可（这里之所以说第一个结点而不说是头结点是因为，如果循环单链表是带头结点的则最后一个结点的指针域要指向头结点；如果循环单链表不带头结点，则最后一个指针域要指向开始结点）

循环双链表（带头结点）



构造源自双链表，即将终 端结点的 next 指针指向链表中第一个结点，将链表中第一个结点的 `prior` 指针指向终端结点，如图所示。循环双链 表同样有带头结点和 不带头结点之分。带头结点的循环双链表当 `head->next` 和 `head->prior` 两个指针都等于 `head` 时链表为空，不带头结点 的循环双链表当 `head` 等于 `NULL` 的时候为空。

静态链表（非重点，简单理解是通过结构体数组表达了链表的结构）

线性表上定义的主要基本运算：

·构造空表：`Initlist(&L)`

·求表长：`Listlength(L)`

·取结点元素值：`GetElem(L, p, &e)`

·插入：`Insert(&L, p, e)`

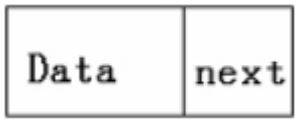
·删除：`Delete(&L, p, &e)`

```
typedef struct {
    int data[maxSize];  //maxSize是使用宏定义的常量，用define定义它的初始值
    int length;
} Sqlist;
```

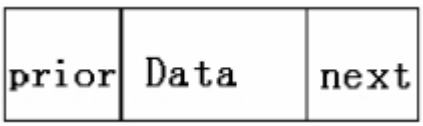
简单形式：

```
int A[maxSize];
int n;
```

```
typedef struct LNode {
    int data;           //data 中存放结点数据域（默认是 int 型）
    struct LNode *next;  //指向后继结点的指针
}LNode;                 //单链表的结构定义，需要牢固记忆，考链表算法需要在结构体的基础上进行运用
```




```
typedef struct DLNode {
    int data;
    struct DLNode *prior;
    struct DLNode *next;
}DLNode;                    //双链表结点形式，多一个前驱指针，和单链表对比记忆
```



顺序表的相关操作很基础，供参考入门：

```
int LocateElem(Sqlist L, int e) {   // 若找到元素值为e的元素，再返回它在顺序表的位置，否则返回0（默认位置从1开始）
    int i;
    for(i = 1; i <= L.length; ++i)
        if(e == L.data[i])
            return i;
    return 0;
}
```

```
int insert(Sqlist &L, int p, int e) {  // 插入成功返回1，插入失败返回0
                                       // 同时通过指针修改L
    int i;
    if(p < 1 || p > L.length + 1 || L.length == maxSize - 1)
        return 0;                      // 一些非法操作，如p传入的值不合法
    for(i = L.length; i >= p; --i)
        L.data[i+1] = L.data[i];       // 顺序表插入需要移动插入点之后的所有元素
    L.data[p] = e;                     // 将要插入的值赋给这个位置
    ++(L.length);                      // 相关的完善操作，插入后表的长度要加1
    return 1;
}
```

```
int delete(Sqlist &L, int p, int &e) {  //删除成功返回1，失败返回0
    int i ;
    if(p < 1 || p > L.length || L.length = 0)
        return 0;    //一些非法操作，如p传入的值不合法
    e = L.data[p]; //e来保存要删除的值
    for(i = p; i < L.length; ++i)
         L.data[i] = L.data[i+1];//顺序表删除需要移动删除点之后的所有元素
    --(L.length);
    return 1;
}
```

> Note: 顺序表插入需要从前往后移动插入点之后的所有元素
>
> 顺序表删除需要从后往前移动删除点之后的所有元素

单链表的尾插法

```
void CreatelistR(LNode *&C, int a[], int n) {
    LNode *s, *r;
    int i;
    C = (LNode*) malloc (sizeof(LNode));
    C -> next = NULL;
    r = C;
    for(i = 1; i <= n; ++i) {   //循环申请 n 个结点来接受数组 a 中的元素
        s = (LNode *)malloc(sizeof(LNode));
        s->data = a[i];//用新申请的结点来接受 a 中的一个元素 。
        r->next = s;   //用 r 来接纳新结点。
        r = r->next;//r 指向终端结，点以便于接纳下一个到来的结点。关键代码
    }
    r->next = NULL; //数组 a 中所有的元素都已经装入链表 C 中，C 的终端结点//的指针域置为 NULL，C 建立
完成。
}
```

单链表尾插法

```
void CreatelistF(LNode *&C, int a[], int n) { //单链表的头插法建立法
    LNode *s; int i;
    C = (LNode *)malloc(sizeof(LNode));
    C->next = NULL;
    for(i = 1; i <= n; ++i) {
        s = (LNode *)malloc(sizeof(LNode));
        s->data = a[i];
        /*下边两句是头插法的关键步骤。*/
        s->next=C->next;//s 所指新结点的指针域 next 指向 C 中的开始结点
        C->next=s;//头结点的指针域 next 指向 s 结点，使得 s 成为了新的开始结点。
    }
}
```

在这两个算法中，有一个语句需要学弟妹掌握并牢记：

```
C = (LNode*) malloc (sizeof(LNode));
```

这是分配结点空间的函数语句，其中调用了`malloc`内存分配函数，前面的括号表示强制抓换类型，后面调用的`sizeof`运算符可以测量结点所占空间大小

例题：

两个链表的合并算法（熟练掌握）

例：A 和 B 是两个单链表（带表头结点），其中元素递增有序。设计一个算法将 A 和 B 归并成一个按元素递增有序的链表 C，C 由 A 和 B 中的结点组成。

A：1 3 5

B：2 4 6

->

C:1 2 3 4 5 6

分析：已知 A，B 中的元素递增有序，怎样使归并后的 C 中元素依然有序呢。我们可以从 A， B 中挑出最小的元素插入 C 的尾部，这样当 A，B 中所有元素都插入 C 中的时候，C 一定是递增有序的。哪一个元素是 A，B 中最小的元素呢？很明显，由于 A，B 是递增的，所以 A 中的最小元素是其开始结点中的元素，B 也一样。我们只需从 A，B 的开始结点中选出一个较小的来插入 C 的尾部即可。这里还需注意，A 与 B 中的元素有可能一个已经全部被插入到 C 中，另一个还没有插完，比如 A 中所有元素已经全部被插入到 C 中而 B 还没有插完，这说明 B 中所有元素都大于 C 中元素，因此只要将 B 链接到 C 的尾部即可，如果 A 没有插完则用类似的方法来解决。

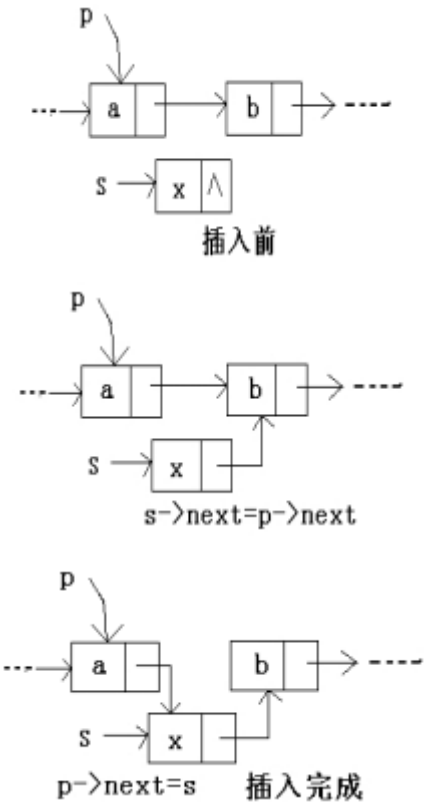
```
void merge(LNode *&A,LNode *&B,LNode *&C) {
    LNode *p=A->next; //p 来跟踪 A 的最小值结点。
    LNode *q=B->next; //q 来跟踪 B 的最小值结点。
    while(p!=NULL&&q!=NULL)  { //当 p 与 q 都不空时选取 p 与 q 所指结点中的较小者插入 C 的尾部。
    /*以下的 if else 语句中，r 始终指向当前链表的终端节点，作为接纳新结点的一个媒介，通过它新结点被链接入
C 并且重新指向新的终端结点以便于接受下一个新结点，这里体现了建立链表的尾插法思想。*/
        if(p->data<=q->data) {
            r->next=p;
            p=p->next;
            r=r->next;
        } else {
            r->next=q;
            q=q->next;
            r=r->next;
        }
    }
    r->next=NULL;
    /*以下两个 if 语句将还有剩余结点的链表链接在 C 的尾部*/
    if(p!=NULL)
        r->next=p;
    if(q!=NULL)
        r->next=q;
}    //尾插法
```

在上述算法中不断的将新结点插入链表的前端，因此新建立的链表中元素的次序和数组 a 中的元素的次序是相反的。将归并成一个递增的链表 C 改为归并成一个递减的链表 C。怎么来解决呢？答案是显然的，将插入过程改成头插法即可解决。代码如下，这里不需要 r 追踪 C 的终端结点，用 s 来接受新的结点插入链表 C 的前端。

```
void merge(LNode *&A,LNode *&B,LNode *&C) {
    LNode *p=A->next;
    LNode *q=B->next;
    LNode *s;
    C=A;
    C->next=NULL;
    free(B);
    while(p!=NULL&&q!=NULL) {
    /*下边这个 if else 语句体现了链表的头插法*/
        if(p->data<=q->data) {
            s=p;
            p=p->next;
            s->next=C->next;
            C->next=s;
        } else {
            s=q;
            q=q->next;
            s->next=C->next;
            C->next=s;
        }
    }
    /*下边这两个循环是和求递增归并序列不同的地方，必须将剩余元素逐个插入 C 的头部才能得到最终的递减序列。*/
    while(p!=NULL) {
        s=p;
        p=p->next;
        s->next=C->next;
        C->next=s;
    }
    while(q!=NULL) {
        s=q;
        q=q->next;
        s->next=C->next;
        C->next=s;
    }
}
```

上边头插法的程序中提到了单链表的结点插入操作，**但有一点需要注意。假设 p 指向一个结点，要将 s 所指结点插入 p 所指结点之后的操作如下：**

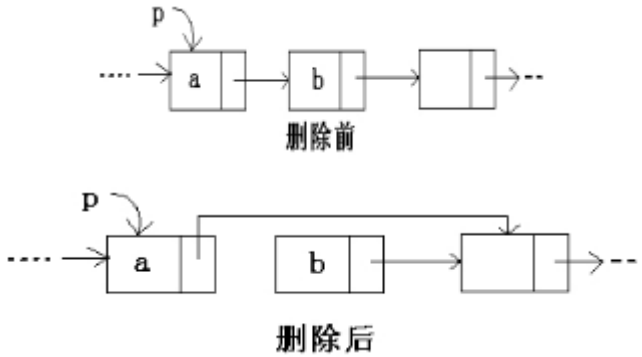
```
s -> next = p ->next;
p -> next = s;
```



注意：以上插入操作语句能不能颠倒一下顺序写成 `p->next=s;s->next=p->next;` 呢？显然是不可以的，因为第一句 `p->next=s;` 虽然将s 链接在 `p` 之后，但是同时也丢失了 `p` 直接后继结点的地址（`p->next` 指针原本所存储的 p 直接后继结点的地址在没有被转存到其他地方的情况下被 `s` 所覆盖，而正确的写法中，`p->next` 中的值在被覆盖前被转存在了 `s->next` 中，因而 `p` 后继结点的地址依然可以找到），这样链表断成了两截，没有满足将 `s` 插入链表的要求。

与插入结点对应的是删除结点，要将单链表的第 i 个结点删去，必须先在单链表中找到第 i-1 个结点，再删除其后继结点。如下图所示，若要删除结点 b，为了实现这一逻辑关系的变化，仅需要修改结点 a 中的指针域。假设 p 为指向 a 的指针。则只需将 p 的指针域 next 指向原来 p 的下一个结点的下一个结点即可。即

```
p -> next = p -> next -> next;
```



删除操作要释放所删除结点的内存空间，即完整的删除操作应该是这样：

```
q=p->next;
p->next=p->next->next;
free (q) ;//调用 free 函数来释放 q 所指结点的内存空间。
```

掌握了单链表中结点删除的算法后，下边再看一个例题。

例题2:

查找链表 C（带头结点）中是否存在一个值为 x 的结点，存在就删除之并返回1，否则返回 0。

分析:

对于本题需要解决两个问题，一是要找到值为 x 的结点；二是将找到的结点删除。问题一引出了本章要讲的单链表中最后一个重要操作，链表中结点的查找。为了实现查找，我们定义一个结点指针变量 p，让他沿着链表一直走到表尾，每来到一个新结点就检测其值是否为 x，是则证明找到，不是则继续检测下一个结点。当找到值为 x 的结点后就是删除操作的内容，如何删除上面已经讲过。

```
int SearchAndDelete(LNode *&C,int x) {
    LNode *p,*q; p=C;
    /*查找部分开始*/
    while(p->next!=NULL) {            //都是对p->next进行操作
        if(p->next->data==x) break;
        p=p->next;
    }
    /*查找部分结束*/
    if(p->next==NULL)
        return 0;
    else {
    /*删除部分开始*/
        q=p->next;
        p->next=p->next->next;
        free(q);
    /*删除部分结束*/
        return 1;
    }
}
```

说明：以上程序中之所以要使 p 指向所要删除结点的前驱结点而不是直接指向所要删除结点本身，是因为要删除一个结点必须知道其前驱结点的位置，这在之前删除操作的讲解中已经体现。

双链表的算法操作

(1)采用尾插法建立双链表

```
void CreateDlistR(DLNode *&L,int a[],int n) {
    DLNode *s,*r;
    int i;
    L=(DLNode*)malloc(sizeof(DLNode));
    L->next=NULL;
    r=L; //和单链表一样 r 始终指向终端结点，开始头结点也是尾结点
    for(i=1;i<=n;i++) {
        s=(DLNode*)malloc(sizeof(DLNode)); //创建新结点 s->data=a[i];
        /*下边 3 句将 s 插入在 L 的尾部并且 r 指向 s，s->prior=r; 这一句是和建立单链表不同的地方。*/
        r->next=s;
        s->prior=r;               //表示的是前驱指针
        r=s;
    }
```

```
    r->next=NULL;
}
```

(2) 查找结点的算法

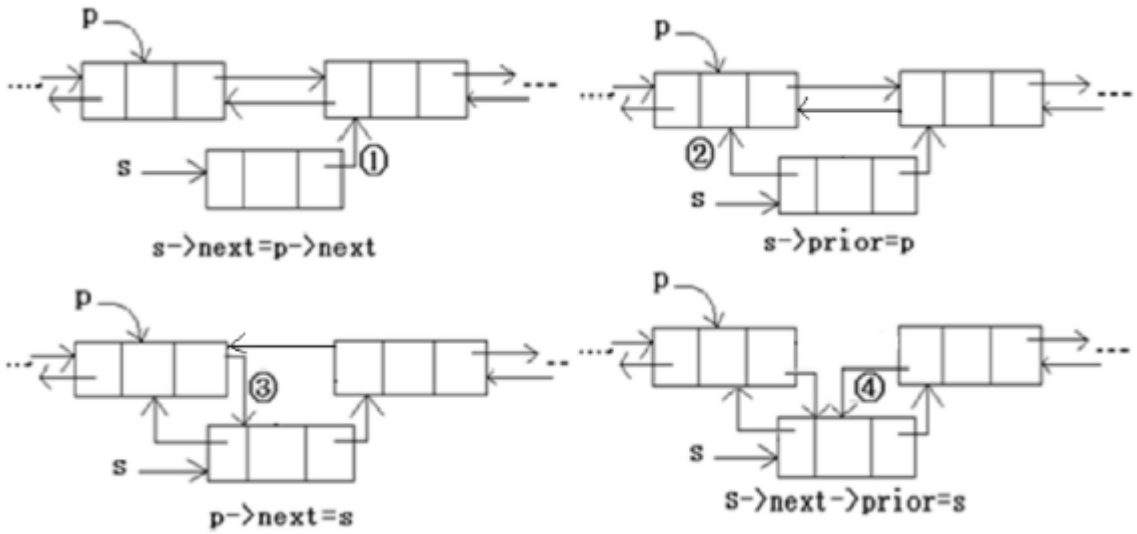
在双链表中查找第一个结点值为 x 的结点。从第一个结点开始，边扫描边比较，若找到这样的结点，则返回结点指针，否则返回 `NULL`。算法代码如下：

```
Void Finfnode(DLNode *C,int x) {
    DLNode *p=C->next;
    while(p!=NULL) {
        if(p->data==x) break;
        p=p->next;
    }
    return p; //如果找到则 p 中内容是结点地址（循环因 break 结束），没找到 p 中内容是 NULL（循环因 p 等于 NULL 而结束）因此这一句可以将题干中要求的两种返回值的种情况统一。
}
```

(3) **插入结点的算法（牢牢理解）**

假设在双链表中 p 所指的结点之后插入一个结点 s，其操作语句描述为：

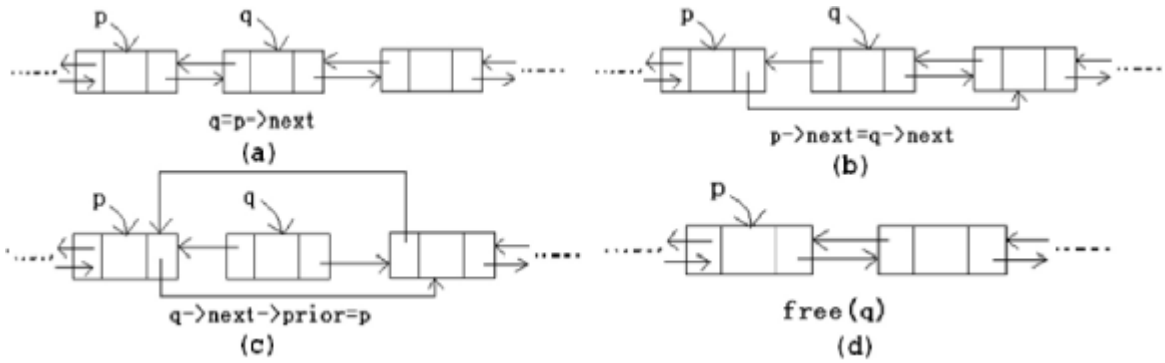
```
s->next=p->next;
s->prior=p;
p->next=s;
s->next->prior=s;
```



说明：按照上面的顺序来插入，可以看成是一个万能的插入方式。不管怎样，先将要插入的结点两边链接好，可以保证不会发生链断之后找不到结点的情况。所以请一定要记住这种万能插入结点的方式。

删除结点的算法，设要删除双链表中 p 结点的后继结点，其操作的语句为：

```
q=p->next;
p->next=q->next;
q->next->prior=p;
free(q);
```



循环链表的算法操作

循环单链表和循环双链表由对应的单链表和双链表改造而来，只需在终端结点和头结点间建立联系即可。循环单链表终端结点的 `next` 结点指针指向表头结点；循环双链表终端结点的 `next` 指针指向表头结点，头结点的 `prior` 指针指向表尾结点。需要注意的是如果 p 指针沿着循环链表行走，判断 p 走到表尾结点的条件是 `p->next==head`。循环链表的各种操作均与非循环链表类似，这里不再细说。

习题练习：

1.设顺序表用数组 A[]表示，表中元素存储在数组下标 1~m+n 的范围内，前 m 个元素递增有序，后 n 个元素递增有序，设计一个算法，使得整个顺序表有序。

2.已知递增有序的单链表 A,B（A,B 中元素个数分别为 m,n 且 A,B 都带有头结点）分别存储了一个集合，请设计算法以求出两个集合 A 和 B 的差集 A-B（即仅由在 A 中出现而不在 B 中出现的元素所构成的集合）。将差集保存在单链表 A 中，并保持元素的递增有序性。

3.设计一个算法，将顺序表中的所有元素逆置。

4.设计一个算法，从一给定的顺序表 L 中删除下标 i 到 j（i≤j，包括 i，j）之间的所有元素，假定 i,j 都是合法的。

5.有一个顺序表 L，其元素为整型数据，设计一个算法，将 L 中所有小于表头元素的整数放在前半部分，大于的整数放在后半部分，数组从下表 1 开始存储。

6.设计一个算法删除单链表 L（有头结点）中的一个最小值结点。

7.有一个线性表，采用带头结点的单链表 L 来存储。设计一个算法将其逆置。要求不能建立新结点，只能通过表中已有结点的重新组合来完成。

8.**设计一个算法将一个头结点为A 的单链表（其数据域为整数）分解成两个单链表 A和 B，使得 A链表只含有原来链表中data 域为奇数的结点，而B 链表只含有原链表中data域为偶数的结点，且保持原来相对顺序。**

参考答案：

1.算法基本设计思想：将数组 `A[]` 中的 `m+n` 个元素（假设元素为 `int` 型）看成两个顺序表，表 `L` 和表 `R`。将数组当前状态看做起始状态，即此时表 `L` 由 `A[]` 中前 `m` 个元素构成，表 `R` 由 `A[]` 中后 `n` 个元素构成。要使 `A[]` 中 `m+n` 个元素整体有序只需将表 `R` 中的元素逐个插入表 `L` 中的合适位置即可。

插入过程：取表 `R` 中的第一个元素 `A[m+1]` 存入辅助变量 `temp` 中，让 `temp` 逐个与 `A[m],A[m-1],…,A[1]` 进行比较，当 `temp<A[j]（1≤j≤m）` 时，将 `A[j]` 后移一位；否则将 `temp` 存入 `A[j+1]` 中。重复上述过程继续插入 `A[m+2],A[m+3],……,A[m+n]`，最终 `A[]` 中元素整体有序。

```
void Insert(int A[],int m,int n) {
    int i,j;
    int temp;   //辅助变量，用来暂存待插入元素。
    for(i=m+1; i<=m+n; i++) {  //将 A[m+1…m+n]插入到 A[1…m]中。
        temp=A[i];
        for(j=i-1; j>=1&&temp<A[j]; j--) {
            A[j+1]=A[j];    //元素后移，以便腾出一个位置插入 temp。
            A[j+1]=temp;
        }   //在 j+1 位置插入 temp。
    }
}
```

2.算法基本设计思想：只需从 A 中删去 A 与 B 中共有的元素即可。由于两个链表中元素是递增有序的所以可以这么做：设置两个指针 p,q 开始时分别指向 A 和 B 的开始结点。循环进行以下判断和操作，如果 p 所指结点的值小于 q 所指结点值，则 p 后移一位；如果 q 所指结点的值小于 p 所指结点的值，则 q 后移一位；如果两者所指结点的值相同，则删除 p 所指结点。最后 p 与 q 任一指针为 NULL 的时候算法结束。

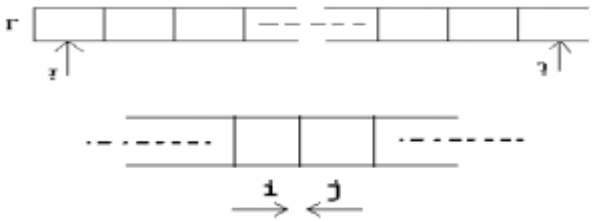
```
void    Difference(LNode *&A,LNode *B) {
    LNode *p=A->next,*q=B->next;    //p 和 q 分别是链表 A 和 B 的工作指针。
    LNode *pre=A;   //pre 为 A 中 p 所指结点的前驱结点的指针。
    LNode *r;
    while(p!=NULL&&q!=NULL) {
        if(p->data<q->data) {
            pre=p;
            p=p->next;  //A 链表中当前结点指针后移。
        } else if(p->data>q->data)
            q=q->next; //B 链表中当前结点指针后移。
        else {
            pre->next=p->next; //处理 A，B 中元素值相同的结点，应删除。 \
            r=p;
            p=pre->next;
            free(r);    //删除结点。
        }

    }
}
```

3.算法基本设计思想：两个变量 i，j 指示顺序表的第一个元素和最后一个元素，交换 i，j 所指元素，然后 i 向后移动一个位置，j 向前移动一个位置，如此循环，直到 i 与 j 相遇时结束，此时顺序表 L 中的元素已经逆置。

```
void reverse(Sqlist &L) {   //L 要改变，用引用型
    int i,j;
    int temp; //辅助变量，用于交换
    for(i=1,j=L.length;i<j;i++,j--) { //当 i 与 j 相遇时循环结束
        temp=L.data[i];
        L.data[i]=L.data[j];
        L.data[j]=temp;
    }
}
```

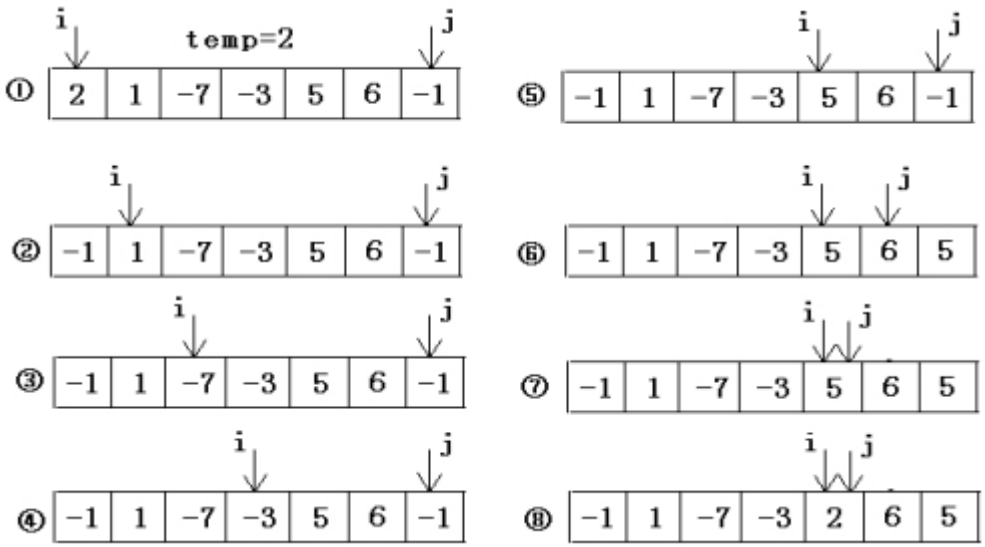
注意：本题中 `for` 循环的执行条件要写成 i<j 而不要写成 `i != j`。如果数组中元素有偶数个则 i 与 j 会出现下图所示状态，此时 i 继续往右走，j 继续往左走，会互相跨越对方，循环不会结束。



4.算法基本设计思想：本题是顺序表删除算法的扩展，可以采用如下方法解决，从第 `j+1` 个元素开始到最后一个元素为止，用这之间的每个元素去覆盖从这个元素开始往前数第 `j-i+1` 个元素，即可完成删除 `i~j` 之间的所有元素。

```
void Delete(Sqlist &L,int i,int j) {  //L 要改变，用引用型。
    int k,l;
    l=j-i+1; //元素要移动的距离。
    for(k=j+1;k<=L.length;k++) {
        L.data[k-l]=L.data[k]; //用第 k 个元素去覆盖它前边的第 l 个元素。
    }
    L.length-=l;    //表长改变。
}
```

5.算法基本设计思想：本题可以这样解决，先将 L 的第一个元素存于变量 `temp` 中，然后定义两个整型变量 `i,j`。i 从左往右扫描，j 从右往左扫描。边扫描边交换。具体执行过程如下：



各步的解释如下：

① 开始状态，`temp=2,i=1;j=L.length`

② j 先移动，从右往左，边移动边检查 j 所指元素是否比 2 小，此时发现-1 比 2 小，则执行 `L.data[i]=L.data[j]; i++;`（i 中元素已经被存入 temp 所以可以直接覆盖并且 i 后移一位，准备开始 i 的扫描）

③ i 开始移动，从左往右，边移动边检测，看是否 i 所指元素比 2 大，此时发现-7 比 2 小，因此 i 在此位置是什么都不做。

④ i 继续往右移动，此时 i 所指元素为-3 也比 2 小，此时什么都不做。

⑤ i 继续往右移动，此时 i 所指元素为 5，比 2 大，因此执行 `L.data[j]=L.data[i]; j--;` （j 中元素已被保存，j 前移一位，准备开始 j 的扫描）

⑥ j 往左运动，此时 j 所指元素为 6，比 2 大，j 在此位置时，什么都不做。

⑦ j 继续往左移动，此时 `j==i`，说明扫描结束。

⑧ 执行 `L.data[i]=temp;` 此时整个过程结束，所有比 2 小的元素被移到了 2 前边，所有比 2 大的元素被移到了 2 后边。

在这里面有两点是需要注意的：

① i 和 j 是轮流移动的，即当 i 找到比 2 大的元素时，将 i 所指元素放入 j 所指位置，i停在当前位置不动，j 开始移动。j 找到比 2 小的元素，将 j 所指元素放在 i 所指位置，j 停在当前位置不动，i 开始移动，如此交替直到 `i==j`。

②每次元素覆盖（比如执行 `L.data[i]=L.data[j];` ）不会造成元素丢失，因为在这之前被覆盖位置的元素已经存入其他位置

```
void move(Sqlist &L) { //L 要改变所以用引用型
    int temp;
    int i=1,j=L.length;
    temp=L.data[i];
    while(i<j) {
        /*关键步骤开始*/
        while(i<j&&L.data[j]>temp) j--; //j 从左往右扫描，当来到第一个比 temp 小的元素时停止,并且每走一步都要判断 i 是否小于 j,这个判断容易遗漏。
        if(i<j) { //检测看是否已仍满足 i<j，这一步同样很重要
            L.data[i]=L.data[j]; //移动元素。
            i++;    //i 右移一位。
        }
        while(i<j&&L.data[i]<temp) i++;  //与上边的处理类似。
        if(i<j) { //与上边的处理类似。
            L.data[j]=L.data[i];
            j--;    //与上边的处理类似。
        }
        /*关键步骤结束*/
        L.data[i]=temp;//将表首元素放在最终位置。
    }
}
```

6.算法基本设计思想：用 p 从头至尾扫描链表，`pre` 指向 `*p` 结点的前驱，用 `minp` 保存值最小的结点指针，`minpre` 指向 `minp` 的前驱。一边扫描，一边比较，将最小值结点放到 `minp` 中。

```
void delminnode(LNode *&L) {
    LNode *pre=L,*p=pre->next,*minp=p,*minpre=pre;
    while(p!=NULL) { //查找最小值结点 minp 以及前驱结点 minpre
        if(p->data<minp->data) {
            minp=p;
            minpre=pre;
        }
        pre=p;
        p=p->next;
    }
    minpre->next=minp->next; //删除*minp 结点。
    free(minp);
}
```

7.算法基本设计思想：在前边提到过关与逆序的问题，那就是链表建立的头插法。头插法完成后，链表中的元素顺序和原数组中元素的顺序相反。这里我们可以将 L 中的元素作为逆转后 L 的元素来源，即将 `L->next` 设置为空，然后将头结点后的一串结点用头插法逐个插入 L 中，这样新的 L 中的元素顺序正好是逆序的。

```
void Reversel(LNode *&L) {
    LNode *p=L->next,*q;
    L->next=NULL;
    while(p!=NULL) { //p 结点始终指向旧的链表的开始结点。q 结点作为辅助结点来记录 p 的直接后继结点的位置。
        q=p->next;
        p->next=L->next;  //将 p 所指结点插入新的链表中。
        L->next=p;  //因为后继结点已经存入 q 中，因此 p 仍然可以找到后继
        p=q;
    }
}
```

8.算法基本设计思想：用指针 p 从头至尾扫描 A 链表，当发现结点 data 域为偶数的结点则取下，插入链表 B 中。要用头插法还是用尾插法呢，因为题目要求保持原来数据元素的相对顺序，所以要用尾插法来建立 B 链表。

```
void split2(LNode *&A,LNode *&B) {
    LNode *p,*q,*r;
    B=(LNode*)malloc(sizeof(LNode));    //申请链表 B 的头结点
    B->next=NULL;  //每申请一个新结点的时候，将其指针域 next 设置为 NULL 是个好习惯，这样可以避免很多因链表的终端结点忘记置 NULL 而产生的错误。
    r=B;
    p=A;
    while(p->next!=NULL) {  //p 始终指向当前被判断结点的前驱结点，和删除结点类似，因为取下一个结点，就是删除一个结点，只是不释放这个结点的内存空间而已。
        if(p->next->data%2==0) { //判断结点的 data 域是否为偶数，是则从链表中取下
            q=p->next;  //q 指向要从链表中取下的结点
            p->next=q->next;    //从链表中取下这个结点
            q->next=NULL;
            r->next=q;
            r=q;
        }
        p=p->next;  //p 后移一个位置，即开始检查下一个结点。
    }
```

```
}
```

小结:

顺序表是按线性表的逻辑结构次序依次存放在一组地址连续的存储单元中。在存储单元中的各元素的物理位置和逻辑结构中各结点相邻关系是一致的。地址计算：`LOCa(i)=LOCa(1)+(i-1)*d`;(假设首地址为1)

在顺序表中实现的基本运算：

·插入：平均移动结点次数为n/2;平均时间复杂度均为O(n)。

·删除：平均移动结点次数为(n-1)/2;平均时间复杂度均为O(n)。

线性表的链式存储结构中结点的逻辑次序和物理次序不一定相同，为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还存储了其后继结点的地址信息(即指针或链)。这两部分信息组成链表中的结点结构。 一个单链表由头指针的名字来命名。

单链表的运算要熟练掌握头插法和尾插法。

·查找

·按序号：与查找位置有关，平均时间复杂度均为O(n)。

单循环链表是一种首尾相接的单链表，终端结点的指针域指向开始结点或头结点。链表终止条件是以指针等于头指针或尾指针。

采用单循环链表在实用中多采用尾指针表示单循环链表。优点是查找头指针和尾指针的时间都是O(1)，不用遍历整个链表。

双链表就是双向链表，就是在单链表的每个结点里再增加一个指向其直接前趋的指针域prior，形成两条不同方向的链。由头指针head惟一确定。

双链表也可以头尾相链接构成双(向)循环链表。

双链表上的插入和删除时间复杂度均为O (1)。

顺序表和链表的比较：

·基于空间：

·顺序表的存储空间是静态分配，存储密度为1；适于线性表事先确定其大小时采用。

·链表的存储空间是动态分配，存储密度<1；适于线性表长度变化大时采用。

·基于时间：

·顺序表是随机存储结构，当线性表的操作主要是查找时，宜采用。

·以插入和删除操作为主的线性表宜采用链表做存储结构。

·若插入和删除主要发生在表的首尾两端，则宜采用尾指针表示的单循环链表。