2.3.1

> 设计一个递归算法，删除不带头结点的单链表 L 中所有值为 x 的结点。

```
LinkList Delete_Same(LinkList L,ElemType e)//以当前结点作为L的开始
{
    LNode *p;
    if(L==NULL)//递归终止条件
        return NULL;
    //cout << L->data << endl;
    if(L->data==e)//当前元素等于e
    {
        p=L;
        L=L->next;
        free(p);
        return Delete_Same(L, e);//L已后移
    }
    else
    {
        L->next=Delete_Same(L->next, e);//将后面修改的结果接在当前位置
        return L;
    }
}
```

2.3.2

> 在带头结点的单链表工中，删除所有值为 x 的结点，并释放其空间，假设值为 x 的结点不唯一,试编写算法以实现上述操作.

- **方法一**:从头开始遍历,把值为 x 的结点删去,即改变其前驱结点的 $next$ 指针.

```
void Delete_Same(LinkList &L,ElemType e)
{
    LNode *p=L->next,*pre=L;
    LNode *q;//临时变量用来释放

    while(p!=NULL)//从头遍历
    {
        if(p->data==e)
        {
            q=p;//临时存储当前结点
            pre->next=p->next;//将q结点与链表断开
            p=p->next;//p指针后移
            //pre不变
            free(q);
        }
        else
        {
            pre=p;//同步后移
            p=p->next;
        }
```

```
    }
}
```

- 方法二:将值不为 x的结点**尾插法**插入该链表中,反之,释放空间.

```
void Delete_Same(LinkList &L,ElemType e)
{
    LNode *p=L->next,*r=L;//此处r是尾指针
    LNode *q;//临时变量用来释放

    while(p!=NULL)//从头遍历
    {
        if(p->data==e)//相等释放
        {
            q=p;
            p=p->next;
            free(q);
        }
        else
        {//将其重新链接到链表中
            r->next=p;
            r=p;
            p=p->next;
        }
    }
}
```

2.3.3

> 设 L为带头结点的单链表，编写算法实现从尾到头反向输出每个结点的值。

递归可实现反向输出

```
void ReversePrint(LNode *p)
{
    if(p==NULL)//终止条件
        return ;

    ReversePrint(p->next);
    cout << p->data << " ";
}

ReversePrint(L->next);//调用时使用第一个结点(而不是头结点)
```

2.3.4

> 试编写在带头结点的单链表 L中删除一个最小值结点的高效算法（假设最小值结点是唯一的).

```
LinkList Delete_Min_Elem(LinkList L,ElemType &e)
{
    LNode *p=L->next,*pre=L;
```

```
    LNode *p_min=L->next;//记录最小值的指针
    LNode *pre_min=L;//记录最小值的前驱指针(方便后续更改)

    while(p!=NULL)
    {
        if(p->data<p_min->data)
        {
            p_min=p;
            pre_min=pre;
        }
        pre=p;
        p=p->next;
    }

    pre_min->next=p_min->next;
    e=p_min->data;
    free(p_min);

    return L;
}
```

2.3.5

> 试编写算法将带头结点的单链表就地逆置，所谓"就地"是指辅助空间复杂度为 $O(1)$.

- **方法一**: 利用**头插法**,从头到尾重新插入到单链表 L中.

```
LinkList ReverseList(LinkList L)
{
    LNode *p=L->next;
    LNode *r;//r为p的后继,防止中间处理时断链
    L->next=NULL;//让L与结点断开
    while(p!=NULL)
    {
        r=p->next;//保存p的后继结点

        //头插法插入结点
        p->next=L->next;
        L->next=p;

        p=r;//p指针后移
    }

    return L;
}
```

- **方法二**:**指针反指**,使用三个指针 pre(当前结点的前驱), p(当前结点), r(当前结点的后继)记录对应的执行,每次先将 p 和 r断开,再将三个指针后移,让当前结点 p的 $next$ 指向 pre(即 `p->next=pre`),当 r到达链表的尾部(即 `r==NULL`)此时让 L指向当前的最后一个结点 p.

```
LinkList ReverseList(LinkList L)
```

```
{
    LNode *pre;
    LNode *p=L->next;
    LNode *r=p->next;//r为p的后继,防止中间处理时断链

    p->next=NULL;//处理第一个结点
    while(r!=NULL)
    {
        pre=p;
        p=r;
        r=r->next;
        p->next=pre;//指针反指

    }

    L->next=p;//处理最后一个结点

    return L;
}

```

两种方法的时间复杂度均为 $O(n)$,空间复杂度均为 $O(1)$.

2.3.6

> 有一个带头结点的单链表 L，设计一个算法使其元素递增有序。

- **方法一**:**直接插入排序**,时间复杂度为 $O(n^2)$.

```
void SortList(LinkList &L)
{
    LNode *p=L->next;//p指向第一个结点
    LNode *r=p->next;//r记录p的后继
    LNode *temp,*pre;//temp用于遍历,pre记录前驱

    p->next=NULL;//记得断链
    p=r;//从第二个结点开始枚举

    while(p!=NULL)
    {
        //cout << p->data << endl;
        r=p->next;//更新r

        temp=L->next;//重头遍历(找到合适的位置)
        pre=L;
        while(temp!=NULL&&temp->data<p->data)
        //跳过小于p的所有结点,在temp(此时temp的值大于p的值)的前面插入p结点
        {
            pre=temp;
            temp=temp->next;
        }
        //前插p结点
```

```
        p->next=temp;
        pre->next=p;

        p=r;//p后移
        //PrintList(L);
    }
}
```

- **方法二**:通过`链表->数组->排序->链表`,以空间换取时间,时间复杂度可降低到 $O(nlog_2n)$,此处不过多展示该方法的代码.

2.3.7

> 设在一个带表头结点的单链表中所有元素结点的数据值无序，试编写一个函数，删除表中所有介于给定的两个值（作为函数参数给出)之间的元素的元素（若存在).

```
LinkList Delete_Range_Elem(LinkList L,int l,int r)
{
    LNode *p=L->next,*pre=L;
    while(p!=NULL)
    {
        if(p->data>l&&p->data<r)
        {
            //此处可不用新开一个结点指针,由于pre的next更新后就已知指针p的下一个next
            pre->next=p->next;//前驱结点的next更新
            free(p);//释放
            p=pre->next;//p指针为pre的next
        }
        else
        {
            pre=p;
            p=p->next;
        }
    }
    return L;
}
```

2.3.8

> 给定两个单链表,编写算法找出两个链表的公共结点。

- **方法一**:设两个单链表分别是 L_1, L_2,设它们的长度分别是 $len1$, $len2$,在第一个链表上顺序遍历每个结点,每遍历一个结点,在第二个链表上顺序遍历所有结点,若找到两个相同的结点(此处是**地址相同**,而不是 $data$**域相同**),则找到它们的公共结点。时间复杂度为 $O(len1 \times len2)$.

```
void Search_Common_Node(LinkList L1,LinkList L2,LinkList &L)
{
    LNode *p1=L1,*p2=L2;

    while(p1!=NULL)
    {
```

```
        p2=L2;
        while(p2!=NULL)
        {
            //cout << p1->data << " " << p2->data << endl;
            if(p1==p2)
            {
                L->next=p1;
                return ;
            }
            p2=p2->next;
        }
        p1=p1->next;
        //cout << endl;
    }
}

LinkList L;
InitList(L);
Search_Common_Node(L1, L2,L);
```

- **方法二**:在长的链表上先遍历长度之差个结点之后,再同步遍历两个链表,直到找到相同的结点或者一直到链表结束.时间复杂度为 $O(len1+len2)$(或 `O(max { len1,len2 })`).

```
int Length(LinkList L)//求表的长度
{
    int len=0;
    LNode *p=L->next;
    while(p!=NULL)
    {
        p=p->next;
        len++;
    }
    return len;
}

void Search_Common_Node(LinkList L1,LinkList L2,LinkList &L)
{
    int len1=Length(L1),len2=Length(L2);
    int delta=abs(len1-len2);//长度差值
    LNode *p1=L1->next,*p2=L2->next;

    for(int i=1;i<=delta;i++)
    {
        if(len1>len2)//p1后移(l1长度大于L2)
            p1=p1->next;
        else//p2后移
            p2=p2->next;
    }

    while(p1!=NULL)//公共长度部分
    {
```

```
        if(p1==p2)
        {
            L->next=p1;
            return ;
        }
        p1=p1->next;
        p2=p2->next;
    }
}

LinkList L;
InitList(L);
Search_Common_Node(L1, L2,L);
```

2.3.9

> 给定一个带表头结点的单链表，设 $head$为头指针，结点结构为 $(data, next)$，$data$为整型元素，$next$为指针，试写出算法:按递增次序输出单链表中各结点的数据元素，并释放结点所占的存储空间（要求:不允许使用数组作为辅助空间).

对链表进行遍历,在每次遍历中找出整个链表的**最小值元素**,输出并释放结点所占空间;再查找次小值元素,输出并释放空间,直至链表为空.时间复杂度为 $O(n^2)$.由于题目限制,时间复杂度无法再降到更低.

```
void Sort_And_Delete(LinkList &L)
{
    while(L->next!=NULL)
    {
        LNode *p=L->next,*pre=L;
        LNode *p_min=L->next;//记录最小值的指针
        LNode *pre_min=L;//记录最小值的前驱指针(方便后续更改)
        ElemType num;

        while(p!=NULL)
        {
            if(p->data<p_min->data)
            {
                p_min=p;
                pre_min=pre;
            }
            pre=p;
            p=p->next;
        }

        pre_min->next=p_min->next;
        num=p_min->data;
        cout << num << " ";
        free(p_min);
    }
    cout << endl;
}
```

此处可借鉴 `2.3.4` 的查找最小值再释放的函数,如下:

```
void Sort_And_Delete_short(LinkList &L)
{
    while(L->next!=NULL)
    {
        ElemType num;
        Delete_Min_Elem(L, num);//使用2.3.4的函数
        cout << num << " ";
    }
    cout << endl;
}
```

2.3.10

> 将一个带头结点的单链表 A分解为两个带头结点的单链表 A和 B，使得 A表中含有原表中序号为**奇数**的元素,而 B表中含有原表中序号为**偶数**的元素，且保持其相对顺序不变。

```
void Divide_List(LinkList &L1,LinkList &L2)
{
    int tot=1;

    LNode *p1=L1;
    LNode *p2=L2;
    LNode *p=L1->next;
    p1->next=NULL;//让L1置为空表

    while(p!=NULL)
    {
        if(tot%2==1)//奇数
        {
            p1->next=p;
            p1=p;
        }
        else//偶数
        {
            p2->next=p;
            p2=p;
        }
        p=p->next;
        tot++;
    }
    p1->next=NULL;
    p2->next=NULL;
}
```

2.3.11

> 设 C={ $a_1,b_1,a_2,b_2,\cdots,a_n,b_n$ }为线性表，采用带头结点的单链表存放，设计一个就地算法，将其拆分为两个线性表，使得 A={ $a_1,a_2,\cdots,a_n$ }，B={ $b_n,\cdots,b_2,b_1$ }.

```
void Divide_TwoList(LinkList L,LinkList &L1,LinkList &L2)
{
    int tot=1;

    LNode *p=L->next;
    LNode *p1=L1;//尾插法
    LNode *temp;//L2头插法,temp临时变量

    while(p!=NULL)
    {
        if(tot%2==1)//尾插法
        {
            p1->next=p;
            p1=p;
            p=p->next;
        }
        else
        {
            temp=p->next;
            p->next=L2->next;
            L2->next=p;
            p=temp;
        }
        tot++;
    }
    p1->next=NULL;//尾插法需置空
}

//可发现计数变量tot可不用
void Divide_TwoList(LinkList L,LinkList &L1,LinkList &L2)
{
    LNode *p=L->next;
    LNode *p1=L1;//尾插法
    LNode *temp;//L2头插法,temp临时变量

    while(p!=NULL)
    {
        p1->next=p;//L1尾插
        p1=p;
        p=p->next;

        temp=p->next;//L2头插
        p->next=L2->next;
        L2->next=p;
        p=temp;
    }
```

```
    p1->next=NULL;//尾插法需置空
}
```

2.3.12

> 在一个递增有序的线性表中，有数值相同的元素存在．若存储方式为单链表，设计算法去掉数值相同的元素,使表中不再有重复的元素,例如 $(7,10,10,21,30,42,42,42,51,70)$将变为 $(7,10,21,30,42,51,70)$.

- **方法一**:扫描链表**删除相连相同** $data$**域的后一个结点**,若当前结点的 $data$域与后继结点的 $data$域相同,则删除后继结点,反之后移.时间复杂度为 $O(n)$.

```
void DeleteSameElem(LinkList &L)
{
    LNode *p=L->next;

    if(p==NULL)//空表不用处理
        return ;

    while(p->next!=NULL)
    {
        LNode *q=p->next;
        if(p->data==q->data)//删除后继相等结点
        {
            p->next=q->next;
            free(q);
        }
        else p=p->next;
    }
}
```

- **方法二**: 使用**尾插法**,将当前结点与已经插入结点的链表的最后一个结点比较,若不等则直接插入,反之删除当前结点并处理下一个结点,直到全部扫描完.

```
void DeleteSameElem(LinkList &L)
{
    LNode *p=L->next,*r=L,*q;//r为尾指针
    while(p!=NULL)
    {
        if(p->data!=r->data)//判断当前结点的值是否和尾结点的值相等
        {
            r->next=p;
            r=p;

            p=p->next;
        }
        else
        {
            q=p;//临时存储,以便释放
            p=p->next;
            free(q);
```

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

2.3.13

> 假设有两个按元素值递增次序排列的线性表，均以单链表形式存储。请编写算法将这两个单链表归并为一个按元素值递减次序排列的单链表，并要求利用原来两个单链表的结点存放归并后的单链表。

利用**归并排序的思想**(将两序列中的最小值插入)和**头插法**(逆序插入)实现即可.

```
void MergeList(LinkList &L1,LinkList &L2)
{
    LNode *p1=L1->next,*p2=L2->next;
    LNode *r;//存储后继结点(头插法会导致中间过程断开)

    L1->next=NULL;//置L1为空表

    while(p1!=NULL&&p2!=NULL)
    {
        if(p1->data<=p2->data)//将p1的当前结点头插进 L1中
        {
            r=p1->next;//存储p1的后继结点
            p1->next=L1->next;
            L1->next=p1;

            p1=r;//p1指向其原来的后继结点
        }
        else//将p2的当前结点头插进 L1中
        {
            r=p2->next;//存储p2的后继结点
            p2->next=L1->next;
            L1->next=p2;

            p2=r;//p2指向其原来的后继结点
        }
    }

    while(p1!=NULL)//将p1剩余部分头插进L1中
    {
        r=p1->next;
        p1->next=L1->next;
        L1->next=p1;

        p1=r;
    }

    while(p2!=NULL)//将p2剩余部分头插进L1中
    {
        r=p2->next;
        p2->next=L1->next;
```

```
        L1->next=p2;

        p2=r;
    }
    L2->next=NULL;
}
```

2.3.14

> 设 A和 B是两个单链表（带头结点），其中元素递增有序。设计一个算法从 A和 B中的公共元素产生单链表 C，要求不破坏 A、 B的结点。

两个链表从第一个结点开始遍历,若遇到两个结点的值不相等,则让小的链表指针后移,反之,创建一个值等于两结的元素值的新结点,使用**尾插法**插入(需要使用尾指针),直到某一个链表遍历完即结束.

```
void MergeList(LinkList &L1,LinkList &L2,LinkList &L)//此处L1,L2可引用也可不引用
{
    LNode *p1=L1->next,*p2=L2->next;
    LNode *r=L;//尾指针
    while(p1!=NULL&&p2!=NULL)
    {
        if(p1->data==p2->data)//相等,分配空间
        {
            LNode *s=(LNode *)malloc(sizeof(LNode));
            s->data=p1->data;
            r->next=s;
            r=s;
            p1=p1->next;
            p2=p2->next;
        }
        else if(p1->data<p2->data)//p1后移
            p1=p1->next;
        else p2=p2->next;//p2后移
    }
    r->next=NULL;
}
```

2.3.15

> 已知两个链表 A和 B分别表示两个集合，其元素递增排列。编制函数，求 A与 B的交集，并存放于 A链表中。

思想同**2.3.13,2.3.14**,借鉴课本,采用**归并**的思想来释放非交集结点.

```
void Search_Intersection(LinkList &L1,LinkList &L2)
{
    LNode *p1=L1->next;
    LNode *p2=L2->next;
    LNode *r=L1,*q;//r为尾指针,q为临时变量

    while(p1!=NULL&&p2!=NULL)
```

```
    {
        if(p1->data==p2->data)
        {
            r->next=p1;//后插
            r=p1;

            p1=p1->next;//p1后移

            q=p2;
            p2=p2->next;//p2后移

            free(q);//释放
        }
        else if(p1->data<p2->data)
        {
            q=p1;
            p1=p1->next;
            free(q);
        }
        else
        {
            q=p2;
            p2=p2->next;
            free(q);
        }
    }

    while(p1!=NULL)//释放p1剩下的
    {
        q=p1;
        p1=p1->next;
        free(q);
    }

    while(p2!=NULL)
    {
        q=p2;
        p2=p2->next;
        free(q);
    }

    r->next=NULL;

    L2->next=NULL;
}
```

2.3.16

> 两个整数序列 $A=a_1,a_2,a_3,\cdots,a_m$和 $B=b_1,b_2,b_3,\cdots,b_n$已经存入两个单链表中,设计一个算法，判断序列 B是否是序列 A的连续子序列.

从头开始扫描两个序列,若当前两个结点的值相同,两个序列后移,反之,让 A后移同时 B回到开头,最后检查 B是否为 `NULL`,若是,则 B是 A的连续子序列;若不是,则 B不是 A的连续子序列.

```
bool Judge_List(LinkList L1,LinkList L2)
{
    LNode *p1=L1->next,*p2=L2->next;

    while(p1!=NULL&&p2!=NULL)
    {
        if(p1->data==p2->data)//相等,两链表均后移
        {
            p1=p1->next;
            p2=p2->next;
        }
        else//L1后移,L2重新匹配
        {
            p1=p1->next;
            p2=L2->next;
        }
    }

    if(p2==NULL)
        return true;
    else return false;
}
```

2.3.17

> 设计一个算法用于判断带头结点的循环双链表是否对称。

由于是循环双链表 L,可直接判断,令 p为 L的后继, q为 L的前驱,循环判断是否所有 p 和 q的 $data$域是否相同, p后移, q前移.(注意**奇数,偶数中间判断**)

```
bool Judge_Symmetry(LinkList L)
{
    LNode *p=L->next;
    LNode *q=L->prior;

    //偶数的时候q在前面,p在后面,此时q->next==p
    //奇数的时候,两者相遇,p==q
    while(p!=q&&q->next!=p)
    {
        if(p->data==q->data)//相等
        {
            p=p->next;
            q=q->prior;
```

```
        }
        else return false;//不相等
    }
    return true;
}
```

2.3.18

> 有两个循环单链表，链表头指针分别为 $h1$和 $h2$，编写一个函数将链表 $h2$链接到链表 $h1$之后，要求链接后的链表仍保持循环链表形式。

让两者的头尾相接即可(让一个链表的头结点与另一个链表尾结点相连,让另一个链表的头结点与这个链表的尾结点相连).

```
void Link_List(LinkList &L1,LinkList &L2)
{
    LNode *p=L1;
    LNode *q=L2;

    while(p->next!=L1)//找到L1的尾
        p=p->next;

    while(q->next!=L2)//找到L2的尾
        q=q->next;

    p->next=L2->next;
    //L1尾接L2头(由于带头结点,故此处是L2->next即第一个结点)
    q->next=L1;//L2尾接L1头
}
```

2.3.19

> 设有一个带头结点的循环单链表，其结点值均为正整数。设计一个算法，反复找出单链表中结点值最小的结点并输出，然后将该结点从中删除，直到单链表空为止，再删除表头结点。

思想同**2.3.9**,主要修改判断条件即可.

```
//方法一
void Sort_And_Delete(LinkList &L)
{
    while(L->next!=L)
    {
        LNode *p=L->next,*pre=L;
        LNode *p_min=L->next;//记录最小值的指针
        LNode *pre_min=L;//记录最小值的前驱指针(方便后续更改)
        ElemType num;

        while(p!=L)
        {
            if(p->data<p_min->data)
            {
```

```
                p_min=p;
                pre_min=pre;
            }
            pre=p;
            p=p->next;
        }

        pre_min->next=p_min->next;
        num=p_min->data;
        cout << num << " ";
        free(p_min);
    }
    cout << endl;
}


//方法二
LinkList Delete_Min_Elem(LinkList L,ElemType &e)//辅助函数
{
    LNode *p=L->next,*pre=L;
    LNode *p_min=L->next;//记录最小值的指针
    LNode *pre_min=L;//记录最小值的前驱指针(方便后续更改)

    while(p!=L)
    {
        if(p->data<p_min->data)
        {
            p_min=p;
            pre_min=pre;
        }
        pre=p;
        p=p->next;
    }

    pre_min->next=p_min->next;
    e=p_min->data;
    free(p_min);

    return L;
}

void Sort_And_Delete_short(LinkList &L)
{
    while(L->next!=L)
    {
        ElemType num;
        Delete_Min_Elem(L, num);
        cout << num << " ";
    }
    cout << endl;
    free(L);
}
```

2.3.20

> 设头指针 L为工的带有表头结点的非循环双向链表，其每个结点中除有 pre(前驱指针)、 $data$（数据）和 $next$（后继指针）域外，还有一个访问频度域 $freq$。在链表被启用前，其值均初始化为零. 每当在链表中进行一次 $Locate(L,x)$运算时，令元素值为 x的结点中 $freq$域的值增 1，并使此链表中结点保持按访问频度非增（递减）的顺序排列，同时最近访问的结点排在频度相同的结点前面，以便使频繁访问的结点总是靠近表头。试编写符合上述要求的 $Locate(L,x)$运算的算法，该运算为函数过程，返回找到结点的地址,类型为指针型.

先查找数据值为 x的结点,查到后,将结点从链表中删除,然后顺着结点的前驱链找到该结点的插入位置(频度域递减,且排在同频度的第一个),并插入到该位置.

```
LinkList Locate(LinkList &L,ElemType e)//此处前驱用prior,此处的pre与题目的不一样
{
    LNode *p=L->next,*pre;//当前结点,前驱
    while(p!=NULL&&p->data!=e)
        p=p->next;

    if(p==NULL)//未找到该值
        exit(0);
    else
    {
        p->freq++;//频率域增加
        if(p->prior==L||p->prior->freq>p->freq)//p是第一个结点或者前驱的值大于当前结点
            return p;

        //先将p移除链表
        if(p->next!=NULL)
            p->next->prior=p->prior;
        p->prior->next=p->next;

        //从当前的前驱往前找合适的位置
        pre=p->prior;

        while(pre!=L&&pre->freq<=p->freq)
            pre=pre->prior;

        //将p插入链表中

        p->next=pre->next;
        if(pre->next!=NULL)
            pre->next->prior=p;

        p->prior=pre;
        pre->next=p;

    }
    return p;
}
```

2.3.21

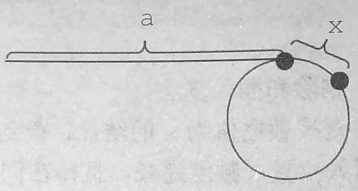
> 单链表有环,是指单链表的最后一个结点的指针指向了链表中的某个结点(通常单链表的最后一个结点的指针域是空的).试编写算法判断单链表是否存在环.
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

> **解释**:`快慢指针`
>
> 首先创建两个指针 1和 2，同时指向这个链表的头节点。然后开始一个大循环，在循环体中，让指针 1每次向下移动一个节点，让指针 2每次向下移动两个节点，然后**比较两个指针指向的节点是否相同**。如果相同，则判断出链表有环，如果不同，则继续下一次循环。
>
> 例如链表`A->B->C->D->B->C->D`，两个指针最初都指向节点 A，进入第一轮循环，指针 1移动到了节点 B，指针 2移动到了 C。第二轮循环，指针 1移动到了节点 C，指针 2移动到了节点 B。第三轮循环，指针 1移动到了节点 D，指针 2移动到了节点 D，此时两指针指向同一节点，判断出链表有环。
>
> 此方法也可以用一个更生动的例子来形容：在一个环形跑道上，两个运动员在同一地点起跑，一个运动员速度快，一个运动员速度慢。当两人跑了一段时间，速度快的运动员必然会从速度慢的运动员身后再次追上并超过，原因很简单，因为跑道是环形的。

- 判断是否成环,使用一个慢指针 $slow$走一步,快指针 $fast$走两步,若相遇则说明存在环.

```
bool Judge_Loop(LinkList L)
{
    LNode *slow=L->next,*fast=L->next;//指向第一个结点

    while(fast!=NULL&&fast->next!=NULL)
    {
        slow=slow->next;//slow走一步
        fast=fast->next->next;//fast走两步

        if(slow==fast)//相遇
            break;
    }

    if(fast==NULL||fast->next==NULL)
        return false;
    else return true;
}
```

如下图所示，设头结点到环的入口点的距离为 a，环的入口点沿着环的方向到相遇点的距离为 x，环长为 r，相遇时 fast 绕过了 n 圈。



则有 2(a+x)=a+n*r+x，即 a=n*r-x。显然从头结点到环的入口点的距离等于 n 倍的环长减去环的入口点到相遇点的距离。因此可设置两个指针，一个指向 head，一个指向相遇点，两个指针同步移动（均为一次走一步），相遇点即为环的入口点。

- 若要找到**环的入口**,代码如下:

```
LNode* Judge_Loop(LinkList L)
{
    LNode *slow=L,*fast=L;

    while(fast!=NULL&&fast->next!=NULL)
    {
        slow=slow->next;//slow走一步
        fast=fast->next->next;//fast走两步

        if(slow==fast)//相遇
            break;
    }

    if(fast==NULL||fast->next==NULL)
        return NULL;

    LNode *p1=L,*p2=fast;//头,fast为相交的地方,当两者相遇时,即为环的入口
    while(p1!=p2)
    {
        p1=p1->next;
        p2=p2->next;
    }
    return p1;
}
```

时间复杂度为 $O(n)$,空间复杂度为 $O(1)$.

2.3.22

> **2009统考真题**：已知一个带有表头结点的单链表，结点结构为

$data$	$link$

> 假设该链表只给出了头指针 $list$。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第 k个位置上的结点（ k为正整数)。若查找成功，算法输出该结点的 $data$域的值,并返回 1;否则，只返回 0。要求:
>
> 1）描述算法的基本设计思想.
>
> 2）描述算法的详细实现步骤。
>
> 3）根据设计思想和实现步骤，采用程序设计语言描述算法（使用`C、C++或Java语言`实现)，关键之处请给出简要注释.

- **方法一**:先查询链表的长度 len,再从头跑 $len-k+1$次即可找到,注意 k可能大于长度 len.时间复杂度为 $O(n)$

```
int Search_k(LinkList list,int k,ElemType &e)
{
    int len=Length(list);

    if(k>len)//超过长度
        return 0;

    LNode *p=list;

    for(int i=1;i<=len-k+1;i++)
        p=p->link;
    e=p->data;
    return 1;
}
```

- **方法二**:设置两个指针 p_1和 p_2,让两者的距离始终为 k,当后面的指针到达尾部,此时前面的指针即为所求.

1）算法的基本设计思想如下：

问题的关键是设计一个尽可能高效的算法，通过链表的一次遍历，找到倒数第 k 个结点的位置。算法的基本设计思想是：定义两个指针变量 p 和 q，初始时均指向头结点的下一个结点（链表的第一个结点），p 指针沿链表移动；当 p 指针移动到第 k 个结点时，q 指针开始与 p 指针同步移动；当 p 指针移动到最后一个结点时，q 指针所指示结点为倒数第 k 个结点。以上过程对链表仅进行一遍扫描。

2）算法的详细实现步骤如下：

① count=0，p 和 q 指向链表表头结点的下一个结点。

② 若 p 为空，转⑤。

③ 若 count 等于 k，则 q 指向下一个结点；否则，count=count+1。

④ p 指向下一个结点，转②。

⑤ 若 count 等于 k，则查找成功，输出该结点的 data 域的值，返回 1；否则，说明 k 值超过了线性表的长度，查找失败，返回 0。

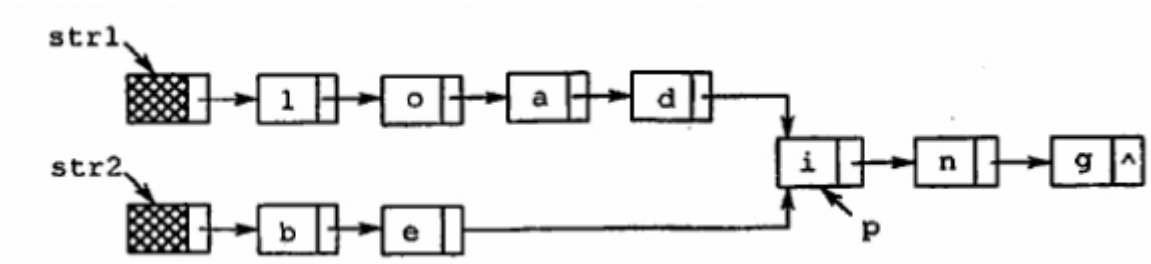
⑥ 算法结束。

```
int Search_k(LinkList list,int k,ElemType &e)
{
    if(k<=0)
        return 0;
```

```
    LNode *p1=list->link,*p2=list->link;
    int cnt=0;

    while(p2!=NULL&&cnt<k)
    {
        p2=p2->link;
        cnt++;
    }

    //cout << p2->data << endl;
    //cout << cnt << endl;

    if(cnt<k)//未找到p2
        return 0;

    while(p2!=NULL)//同步后移
    {
        p1=p1->link;
        p2=p2->link;
    }

    e=p1->data;

    return 1;
}
```

2.3.23

> **2012统考真题**：假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，可共享相同的后缀存储空间，例如，`loading`和`being`的存储映像如下图所示。



设 $str1$和 $str2$分别指向两个单词所在单链表的头结点，链表结点结构为

$data$	$next$

请设计一个时间上尽可能高效的算法，找出由 $str1$和 $str2$所指向两个链表共同后缀的起始位置（如图中字符 i所在结点的位置 p).要求:

1)给出算法的基本设计思想。

2)根据设计思想，采用`C或C++或Java语言`描述算法，关键之处给出注释。

3)说明你所设计算法的时间复杂度。

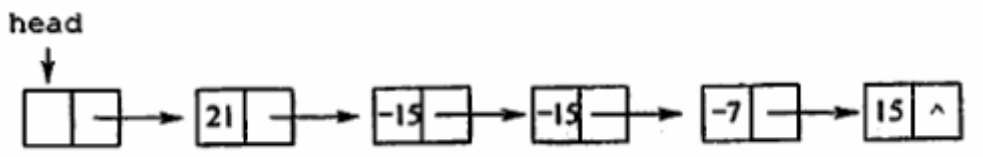
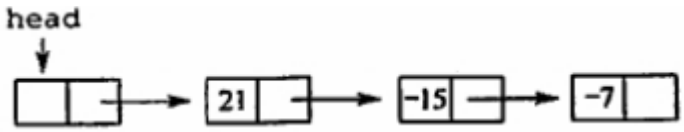
可参考**2.3.8**的代码思想.

声明两个节点指针 p_1、 p_2，计算出两个链表的长度，并计算它们的差，然后让节点指针指向长度较长的链表并使该指针后移，直到与另一个较短的链表等长，最后让两个指针同时后移，当两个指针指向同一个地址时，该地址即为所要寻找的两个链表共同后缀的起始位置。

```
LNode* Search_Common_Node(LinkList L1,LinkList L2)
{
    int len1=Length(L1),len2=Length(L2);
    int delta=abs(len1-len2);//长度差值
    LNode *p1=L1->next,*p2=L2->next;

    for(int i=1;i<=delta;i++)
    {
        if(len1>len2)//p1后移(l1长度大于L2)
            p1=p1->next;
        else//p2后移
            p2=p2->next;
    }

    while(p1!=NULL)//公共长度部分
    {
        if(p1==p2)//找到(假设后面均相同)
            return p1;
        p1=p1->next;
        p2=p2->next;
    }

    return NULL;
}
```

时间复杂度为 $O(len1+len2)$(或 `O(max { len1,len2 })`).

2.3.24

> **2015统考真题**：用单链表保存 m个整数，结点的结构为$[data][link]$，且 $|data| \leq n$ (n为正整数)。现要求设计一个时间复杂度尽可能高效的算法，对于链表中 $data$的绝对值相等的结点，仅保留第一次出现的结点而删除其余绝对值相等的结点。例如，若给定的单链表 $head$如下:
>
> 
>
>
> 则删除结点后的 $head$为
>
> 
>
>
> 要求 :

> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

用一个计数数组保存$|data|$的出现次数(大小为$n+1$),扫描整个链表,若第一次出现,则保留,并把计数数组更新;反之,将其从链表中删除.

```
void Delete_Abs_Same(LinkList &head,int n)//n为data域的最大范围
{
    int *cnt=(int *)malloc((n+1)*sizeof(int));//记录个数

    for(int i=0;i<=n;i++)//初始化计数数组
        cnt[i]=0;

    LNode *p=head->link,*pre=head;

    while(p!=NULL)
    {
        int num=abs(p->data);
        cnt[num]++;
        if(cnt[num]==1)//第一次出现,后移
        {
            pre=p;
            p=p->link;
        }
        else//重复出现,删除
        {
            LNode *q=p;
            pre->link=p->link;
            p=p->link;
            free(q);
        }
    }

    free(cnt);
}
```

时间复杂度为$O(m)$,空间复杂度为$O(n)$(以**空间换取时间**).

2.3.25

> **2019统考真题**：设线性表$L=(a_1,a_2,a_3,\cdots,a_{n-2},a_{n-1},a_n)$采用带头结点的单链表保存，链表中的结点定义如下:
>
> ```
> typedef struct node{
>  int data;
>  struct node *next;
> }NODE;
> ```

> 请设计一个空间复杂度为$O(1)$且时间上尽可能高效的算法，重新排列L中的各结点，得到线性表$L'=(a_1,a_n,a_2,a_{n-1},a_3,a_{n-2})$。要求:
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用C或C++语言描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度。

利用**快慢指针**的思想(可参考**2.3.21**)找到中间结点,以此作为分届处,将后半部分进行逆置,然后再重新插入前半段中.

先观察 $L=(a_1,a_2,a_3,\cdots,a_{n-2},a_{n-1},a_n)$ 和 $L'=(a_1,a_n,a_2,a_{n-1},a_3,a_{n-2},\cdots)$，发现 L' 是由 L 摘取第一个元素，再摘取倒数第一个元素……依次合并而成的。为了方便链表后半段取元素，需要先将 L 后半段原地逆置［题目要求空间复杂度为 $O(1)$，不能借助栈］，否则每取最后一个结点都需要遍历一次链表。①先找出链表 L 的中间结点，为此设置两个指针 p 和 q，指针 p 每次走一步，指针 q 每次走两步，当指针 q 到达链尾时，指针 p 正好在链表的中间结点；②然后将 L 的后半段结点原地逆置。③从单链表前后两段中依次各取一个结点，按要求重排。

```
void ChangeList(LinkList &L)
{
    NODE *p=L,*q=L;//p走一步,q走两步

    //由于p,q同步走,此时若q到达尾结点,说明p已到达中间节点
    while(q->next!=NULL)//寻找中间节点
    {
        p=p->next;
        q=q->next;
        if(q->next!=NULL)//q未到头
            q=q->next;
    }


    //将中间结点的后半部分结点逆置

    q=p->next;//保存p开始的下一个结点
    p->next=NULL;


    while(q!=NULL)
    {
        NODE *r=q->next;//防止断链

        q->next=p->next;//头插法
        p->next=q;

        q=r;//q后移
    }

    NODE *p1=L->next;//指向第一个结点
    NODE *p2=p->next;//指向后一半逆置后的第一个结点

```

```
    p->next=NULL;//将前后断开

    while(p2!=NULL)//将后一半的结点插入前面中
    {
        NODE *r=p2->next;

        p2->next=p1->next;
        p1->next=p2;

        p1=p2->next;//由于已经更新好p2->next

        p2=r;//后半段后移
    }

}
```

查找中间结点的时间复杂度为 $O(n)$,逆置的时间复杂度为 $O(n)$,将后半段插入前半段的时间复杂度为 $O(n)$,故总的时间复杂度为 $O(n)$.