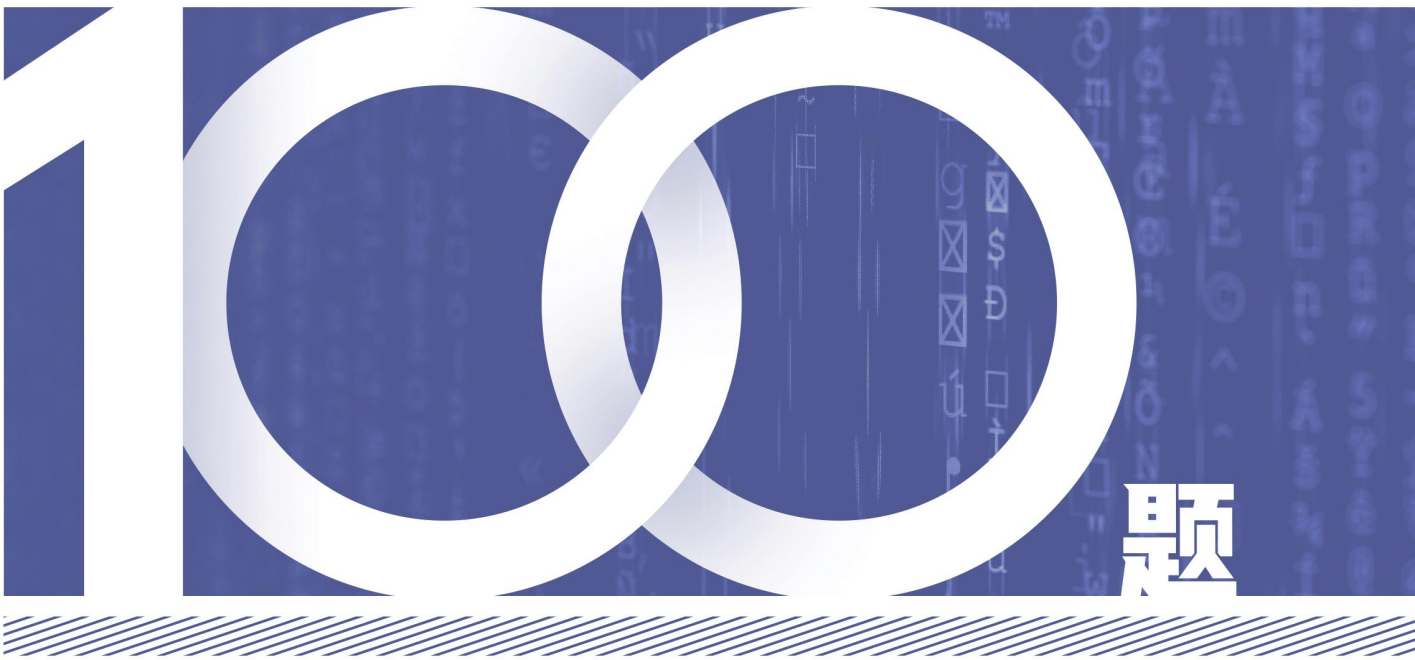
抓机遇，码未来

抓码 DRAMA·计算机考研

抓码计算机考研

必会代码

100 QUESTIONS OF MUST KNOW THE CODE



冲刺必会代码 100 题

顺序表

1、已知线性表（$a_1,a_2,\ldots,a_n$）按顺序结构存储且每个元素为不相等的整数。设计把所有奇数移动到所有偶数前边的算法（**要求时间最少，辅助空间最少**）。

【算法思想】：对于顺序表 L，从左向右找到偶数 L.data[i]，从右向左找到奇数 L.data[j]，将两者交换。循环这个过程直到 i 大于 j 为止。对应的算法如下：

```
void move(SqList &L)
{
    int i = 0, j = L.length-1, k;
    ElemType temp;
    while(i <= j)
    {
        while(L.data[i]%2 == 1)
            i++;        //i 指向一个偶数
        while(L.data[j]%2 == 0)
            j--;        //j 指向一个偶数
        if(i < j)
        {
            temp = L.data[i];          //交换 L.data[i]和 L.data[j]
            L.data[i] = L.data[j];
            L.data[j] = temp;
        }
    }
}
```

本算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

2、设计一个高效算法，将顺序表 L 中所有元素逆置，要求算法的空间复杂度为 O(1)。

【算法思想】：扫描顺序表 L 的前半部分元素，对于元素 L.datai,将其与后半部分对应元素 L.data[L. length-i-1]进行交换。对应的算法如下:

```
void reverse(SqList &L)
{
    int i;
    ElemType x;
    for(i = 0; i<L.length/2; i++)
    {
        x = L.data[i];
        L.data[i] = L.data[L.length-i-1];    //L.data[i]与 L.data[L.length-i-1]交换
        L.data[L.length-i-1] = x;
    }
}
```

3、将两个有序表合并为一个新的有序顺序表，并由函数返回结果顺序表。

【算法思想】
首先，按照顺序不断取下两个顺序表表头较小的结点存入新的顺序表中。然后，看看哪个表还有剩余，将剩下的部分加到新的顺序表后面。

```
bool Merge(SqList A, SqList B, SqList &C)
{
    if(A.length + B.length > C.length)   //表长超过
       return false;
    int i = 0, j = 0, k = 0;
    while(i < A.length && j<B.length)
    {    //循环，两两比较，小者存入结果表
        if(A.data[i] <= B.data[j])
           C.data[k++] = A.data[i++];
        else
           C.data[k++] = B.data[j++];
    }
    while(i < A.length)
       C.data[k++] = A.data[i++];
    while(j < B.length)
       C.data[k++] = B.data[j++];
    C.length = k;
    return true;
}
```

4、从顺序表中删除具有最小值的元素（假设唯一）并由函数返回被删除元素的值。空出的位置由最后一个元素填补。

【算法思想】
搜索整个顺序表，查找最小值元素并记在其位置，搜索结束后用最后一个元素填补空出的原最小值元素的位置。

```
bool Delete_Min(SqList &L ,ElemType &value)
{
    //删除顺序表 L 中最小值元素结点，并通过引用型参数 value 返回其值
    if(L.length == 0)
        return false; //表空，终止操作
    value = L.data[0];
    int pos = 0; //假设 0 号元素的值最小
    for(int i = 1; i<L.length; i++) //循环遍历，寻找具有最小值的元素
    {
        if(L.data[i] < value) //让 value 记忆当前具有最小值的元素
        {
            value = L.data[i];
            pos = i;
        }
    }
    L.data[pos] = L.data[L.length-1]; //空出的位置由最后一个元素填补
    L.length--;
    return true;
```

```
}
```

5、已知长度为 n 的线性表 L 采用顺序存储结构,编写一个时间复杂度为 O(n)、空间复杂度为 O(1)的算法,该算法删除线性表中所有值为 x 的数据元素。

【算法思想】

解法 1：用 k 记录顺序表 L 中等于 x 的元素个数,边扫描 L 边统计 k,并将不为 x 的元素前移 k 位,最后修改 L 的长度。对应的算法如下:

```
void del_x(SqList &L, ElemType x)
{
    int k, i = 0;        //k 用于记录 L 中值等于 x 的元素个数，初试值为 0
    while(i<L.length)
    {
        if(L.data[i] == x)
            k++;          //k 自增
        else
            L.data[i-k] = L.data[i];   //将不为 x 的元素元素前移 k 个位置
        i++;
    }
    L.length = L.length-k;       //顺序表长度较少 k
}
```

解法 2：用 i 从头开始遍历顺序表 L，k 置初始 0。若 L.data[i]不等于 x，则将其存放在 L.data[k]中，k 增 1;若 L.data[i]等于 x，则跳过继续判断下一个元素。最后顺序表长度置为 k。对应算法如下。

```
void del_x(SqList &L, ElemType x)
{
    int i; k = 0;       //k 用于记录 L 中值等于 x 的元素个数，初试值为 0
    for(i = 0; i<L.length; i++)
        if(L.data[i] != x)
        {
            L.data[k] = L.data[i];
            k++;
        }
    L.length = k;         //表长置为 k
}
```

6、设计一个算法，从一给定的顺序表 L 中删除元素值在 x 到 y(x≤y)之间的所有元素，要求以较高的效率来实现，空间复杂度为 O(1)。

【算法思想】

解法一：本题是上诉题目的变形。可以采用上诉解法一的方法，只是将 L.data[i] == x 的条件改成 L.data[i] >= x && L.data[i] <= y。

```
void del_xy(SqList &L, ElemType x, ElemType y)
{
    int i,k = 0;
    for(i = 0; j<L.length; i++)
        if(!(L.data[i] >= x && L.data[i] <= y))
        {
            L.data[k] = L.data[i];
            k++;
        }
    L.length = k;       //表长置为 k
}
```

解法二：

```
void del_xy(SqList &L, ElemType x, ElemType y)
{
    int i = 0, k = 0;
    while(i<L.length)
    {
        if(L.data[i] >= x && L.data[i] <= y)
            k++;          //k 记录被删除记录的个数
        else
            L.data[i-k] = L.data[]
    }
```

7、【2010 年统考】设将 n(n>1)个整数存放在一维数组 R 中。试着设计一个在时间复杂度和空间复杂度都尽可能高效的算法，将 R 中保存的序列循环左移 p（0<p<n）个位置，即将 R 中的数据由（$x_0,x_1,\ldots,x_{n-1}$）变换为（$x_p, x_{p+1},\ldots,x_{n-1},x_0,x_1,\ldots,x_{p-1}$）。要求：

（1）给出算法的基本设计思想；

（2）根据算法设计思想，采用 C 或 C++语言描述，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。

【算法思想】先将 n 个数据 $x_0,x_1,\ldots,x_{n-2},x_{n-1}$ 原地逆置，得到 $x_{n-1},x_{n-2},\cdots,x,x_0$,然后再将 n-p 个数据和后 p 个数据分别原地逆置，得到最终结果：$x_p, x_{p+1},\ldots,x_{n-1},x_0,x_1,\ldots,x_{p-1}$

```
void Reverse(int R[], int left, int right)
{
    //将数组原地逆置
    i = left, j = right;
    while(i < j)
    {
        int tmp = r[i];
        r[i] = r[j];
        r[j] = tmp;
        i++;              //i 右移动一个位置
        j--;              //j 左移一个位置
    }
}
void LeftShift(int R[], int n, int p)
{   //将长度为 n 的数组 R 中的数据循环左移 p 个位置
    if(p>0 && p<n)
    {
        Reverse(r,0,n-1);      //将数组全部逆置
        Reverse(r,0,n-p-1);    //将前 n-p 个数据逆置
        Reverse(r,n-p,n-1);    //将后 p 个数据逆置
    }
}
```

链表

1．将两个递增的有序链表合并为一个递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中不允许有重复的数据。

```
void MergeList(LinkList &La, LinkList &Lb, LinkList &Lc)
{   //将两个递增的有序链表 La 和 Lb 合并为一个递增的有序链表 Lc
    pa = La->next;                  //pa 是链表 La 的工作指针，初始化为首元结点
    pb = Lb->next;                  //pb 是链表 Lb 的工作指针，初始化为首元结点
    Lc = pc = La;                   //La 的头结点作为 Lc 的头结点
    while(pa && pb)
    {
        if(pa->data < pb->data)
        {   //取较小者 Lb 中的元素，将 pb 链接在 pc 的后面，pb 指针后移
            pc->next = pa;
            pc = pa;
            pa = pa->next;
         }//if
        else if(pa->data > pb->data)
        {
            //取较小者 Lb 中的元素，将 pb 链接在 pc 的后面，pb 指针后移
            pc->next = pb;
            pc = pb;
            pb = pb->next;
        }//else if
        else
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
            q = pa->next;
            free(pb);
            pb = q;
         }
     }
     pc->next = pa?pa:pb;     //将非空的剩余元素直接链接在 Lc 表的最好
    free(Lb);                  //释放 Lb 的头结点
}
```

2、将两个非递减的有序表合并为一个非递增的有序表。要求结果链表仍然使用原来两个链表的存储空间，不占用另外的存储空间。表中允许有重复的数据。

【算法思想】与上述题目类似，从原有两个链表中依次摘取结点，通过更改结点的指针域重新建立新的元素之间的线性关系，得到一个新链表。但与上面题目不同的点有两个：①为保证新表与原表顺序相反，需要利用前插法建立单链表，而不能利用后插法；②当一个表到达表尾结点为空时，另一个非空表的剩余元素应该依次摘取，依次链接在 Lc 表的表头结点之后，而不能全部直接链接在 Lc 表的最后。

算法思想是:假设待合并的链表为 La 和 Lb，合并后的新表使用头指针 Lc(Lc 的表头结点设为 La 的表头结点)指向。pa 和 pb 分别是链表 La 和 Lb 的工作指针，初始化为相应链表的首元结点。从首元结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，依次摘取其中较小者重新链接在 Lc 表的表头结点之后，如果两个表中的元素相等，只摘取 La 表中的元素，保留 Lb 表中的元素。当一个表到达表尾结点为空时,将非空表的剩余元素依次摘取，链接在 Lc 表的表头结点之后。最后释放链表 Lb 的头结点。

```
void MergeList(LinkList &La, LinkList &Lb, LinkList &Lc)
{    //将两个非递减的有序表 La 和 Lb 合并为一个非递增的有序链表 Lc
     pa = La->next;        //pa 是链表的 La 的工作指针，初始化为首元结点
     pb = Lb->next;        //pb 是链表 Lb 的工作指针，初始化为首元结点
     Lc = pc = La;         //用 La 的头结点作为 Lc 的头结点
     Lc->next = NULL;
     while(pa || pb)      //只要有一个表未到达表尾指针，用 q 指向待摘取的元素
     {
          if(!pa)          //La 表为空，用 q 指向 pb,pb 指针后移
          {
               q = pb;
               pb = pb->next;
          }
          else if(!pb)      //Lb 表为空，用 q 指向 pa,pa 指针后移
          {
               q = pa;
               pa = pa->next;
          }
          else if(pa->data <= pb->data)     //去较小者 La 中的元素，用 q 指向 pa，
pa 指针后移
          {
               q = pa;
               pa = pa->next;
          }
          else                    //去较小者 Lb 中的元素，用 q 指向 pb，pb 指针后
移
```

```
            {
                q = pb;
                pb = pb->next;
            }
            q->next = Lc->next;
            Lc->next = q;
        }
        free(Lb)
}
```

3、已知两个链表 A 和 B 分别为两个集合，其元素递增排列。请设计一个算法，用于求出 A 与 B 的交集，并存放在 A 链表中。

【算法思想】

A 与 B 的交集是指同时出现在两个集合中的元素，因此，此题的关键点在于:依次摘取两个表中相等的元素重新进行链接,删除其他不等的元素。

算法思想是:假设待合并的链表为 La 和 Lb，合并后的新表使用头指针 Lc(Lc 的表头结点设为 La 的表头结点)指向。pa 和 pb 分别是链表 La 和 Lb 的工作指针，初始化为相应链表的首元结点。从首元结点开始进行比较，当两个链表 La 和 Lb 均为到达表尾结点时，如果两个表中的元素相等，摘取 La 表中的元素，删除 Lb 表中的元素;如果其中一个表中的元素较小，删除此表中较小的元素，此表的工作指针后移。当链表 La 和 Lb 有一个到达表尾结点为空时，依次删除另一个非空表中的所有元素。最后释放链表 Lb 的头结点。

```
void Intersection(LinkList &La, LinkList &Lb, LinkList &Lc)
{
    pa = La->next;        //pa 是链表 La 的工作指针，初始化为首元结点
    pb = Lb->next;        //pb 是链表 Lb 的工作指针，初始化为首元结点
    Lc = pc = La;         //用 La 的头结点作为 Lc 的头结点
    while(pa && pb)
    {
        if(pa->data == pb->data)      //相等，交集并入结果表中
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
            u = pb;
            pb = pb->next;
            free(u);
        }
        else if(pa->data < pb->data)     //删除较小者 La 中的元素
        {
            u = pa;
```

```
                pa = pa->next;
                free(u);
            }
            else                                    //删除较小者 La 中的元素
            {
                u = pb;
                pb = pb->next;
                free(u);
            }
        }
        while(pa)                                   //Lb 为空，删除非空表 La 中的所有元素
        {
            u = pa;
            pa = pa->next;
            free(u);
        }
        while(pb)                                   //La 为空，删除非空表 Lb 中的所有元素
        {
            u = pb;
            pb = pb->next;
            free(u);
        }
        pc->next = NULL;
        free(Lb)
}
```

4、已知两个链表 A 和 B 分别表示两个集合，其元素递增排列。请设计两个算法求出两个集合 A 和 B 的差集（即仅由在 A 中出现而不在 B 中出现的元素所构成的集合），并且以同样的形式存储，同时返回该集合的元素个数。

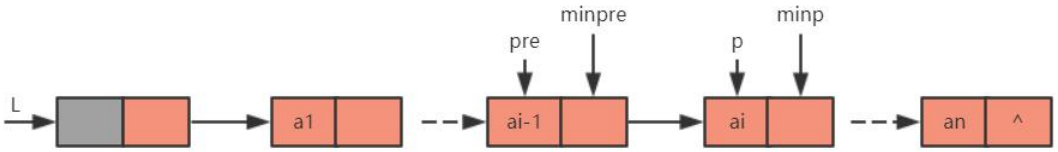
【算法思想】
求两个集合 A 和 B 的差集是指在 A 中删除 A 和 B 中共有的元素，即删除链表中的数据域相等的结点。由于要删除结点，此题的关键点在于:在遍历链表时，需要保存待删除结点的前驱。

算法思想是:假设待求的链表为 La 和 Lb，pa 和 pb 分别是链表 La 和 Lb 的工作指针，初始化为相应链表的首元结点。pre 为 La 中 pa 所指结点的前驱结点的指针，初始化为 La。从首元结点开始进行比较，当两个链表 La 和 Lb 均未到达表尾结点时，如果 La 表中的元素小于 Lb 表中的元素，La 表中的元素即为待求差集中的元素，差集元素个数增 1，pre 置为 La 表的工作指针 pa,pa 后移;如果 Lb 表中的元素小于 La 表中的元素,pb 后移;如果 La 表中的元素等于 Lb 表中的元素，则在 La 表删除该元素值对应的结点。

```
void Difference(LinkList &La, LinkList &Lb, int &n)
{
    pa = La->next;                  //pa 是链表的 La 的工作指针，初始化为首元结点
    pa = Lb->next;                  //pb 是链表 Lb 的工作指针，初始化为首元结点
    pre = La;                       //pre 为 La 中 pa 所指结点的前驱结点的指针
    while(pa && pb)
    {
        if(pa->data < pb->data)             //A 链表当前的结点指针后移
        {
            n++;
            pre = pa;
            pa = pa->next;
        }
        else if(pb->data > pb->data)
            pb = pb->next;                  //A 链表当前的结点指针后移
        else                                //在 La 中删除 La 和 Lb 中元素值相同的结点
        {
            pre->next = pa->next;
            u = pa;
            pa = pa->next;
            free(u);
        }
    }
}
```

5、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率算法（假设最小值结点是唯一的）。

算法思想：用 p 从头至尾扫描单链表，pre 指向*p 结点的前驱，用 minp 保存值最小的结点指针（初值为 p）,minpre 指向*minp 结点的前驱（初值为 pre）。一遍扫描，一边比较，若 p->data 小于 minp->data，则将 p、pre 分别赋值给 minp，minpre，如下图所示。当 p 扫描完毕，minp 指向最小值结点，minpre 指向最小值结点的前驱结点，再将 minp 所指结点删除即可。




```
LinkList Delete_Min(LinkList &L)
{
    //L 是带头结点的单链表，本算法是删除其最小值结点
    LNode *pre = L, *p = pre->next;   //p 为工作指针，pre 指向其前驱
```

```
    LNode *minpre = pre, *minp = p;   //保存最小值结点及其前驱
    while(p != NULL)
    {
        if(p->data < minp->data)
        {
            minp = p;
            minpre = pre;
        }
        pre = p;                          //继续扫描下一节点
        p = p->next;
    }
    minpre->next = minp->next;          //删除最小值结点
    free(minp);
    return L;
}
```

6、在带头结点的单链表 L 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，试编写算法实现上述操作。

算法思想：

解法一：用 p 从头至尾扫描单链表，pre 指向*p 结点的前驱。若 p 所指结点的值为 x，则删除，并让 p 移向下一个结点，否则让 pre、p 同步后移一个结点。

提示：本算法其实也算的上是一个**删除某结点相关题型的代码模板**。此题是在无序单链表中删除满足某种条件的所有结点，这里的条件是结点的值为 x，实际上，这个条件是可以任意改写的，只需要修改 if 条件即可。例如，我们要求删除结点值介于 mink 和 maxk 之间的所有结点，则只需要将 if 语句修改为 if(p->data >mink && p->data <maxk)。

```
void Delete_x(LinkList &L, ElemType x)
{
    //L 为带头结点的单链表，本算法删除 L 中所有值为 x 的结点
    LNode *p = L->next, *pre = L, *q; //置 p 和 pre 的初值
    while(p != NULL)
    {
        if(p->data == x)
        {
            q = p;                    //q 指向该结点
            p = p->next;
            pre->next = p;          //删除*q 结点
            free(q);                  //释放*q 结点的空间
        }
        else                              //否则，pre 和 p 同步后移
```

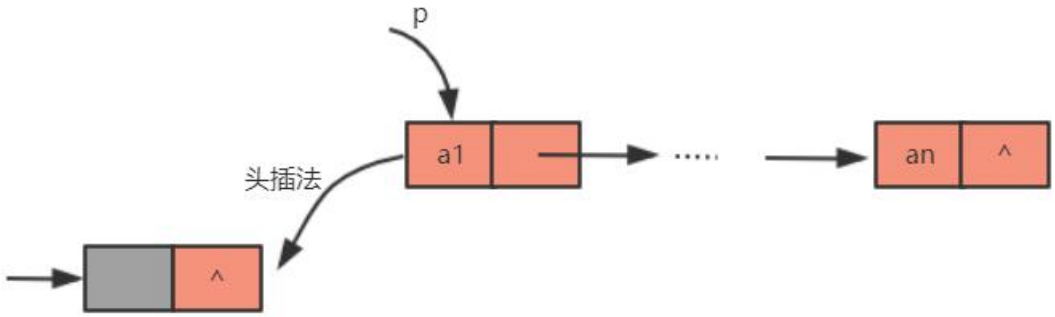
```
        {
            pre = p;
            p = p->next;
        }//else
    } //while
}
```

解法二：采用尾插法建立单链表。用 p 指针扫描 L 的所有结点，当其值不为 x 时将其链接到 L 之后，否则将其释放。

```
void Delet_x(Linklist &L, ElemType x)
{
    //L 为带头结点的单链表，本算法删除 L 中所有值为 x 的结点
    LNode *p = L->next, *r = L, *q;   //r 指向尾节点，其初值为头结点
    while(p != NULL)
    {
        if(p->data != x)        //*p 结点值不为 x 时将其链接到 L 的尾部
        {
            r->next = p;
            r = p;
            p = p->next;        //继续扫描
        }
        else
        {
            q = p;
            p = p->next;
            free(q);
        }
    }//while
    r->next = NULL;             //插入结束后置尾节点指针为 NULL
}
```

7、试编写算法将带头结点的单链表就地逆置，所谓“就地”是辅助空间复杂度为 O(1)。

算法思想：将头结点摘下，然后从第一结点开始，一次插入到头结点的后面（头插法建立单链表），直到最后一个结点为止，这样就实现了链表的逆置，如下图所示。




```
LinkList Reverse(LinkList L)
{
    //L 是带头结点的单链表，本算法将 L 逆置
    LNode *p, *r;              //p 为工作指针，r 为 p 的后继，以防断链
    p = L->next;               //从第一个元素结点开始
    L->next = NULL;            //先将头结点 L 的 next 域值为 NULL
    while(p != NULL)           //依次将元素结点摘下
    {
        r = p->next;           //暂存 p 的后继
        p->next = L->next;//将 p 结点插入到头结点之后
        L->next = p;
        p = r;
     }
    return L;
}
```

8、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 表中中含有原标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。

```
LinkList DisCreat(LinkList &A)
{
    //将 A 表中结点按照序号的奇偶性分解到表 A 或者表 B 中
    i = 0;          //i 记录表 A 中结点的序号
    B = (LinkList)malloc(sizeof(LNode)); //创建 B 表表头
    B->next = NULL;             //B 表初始化
    LNode *ra = A, *rb = B;     //ra 和 rb 将分别指向将创建的 A 表和 B 表的尾节点
    p = A->next;
    A->next = NULL;     //将 A 表置空
    while(p != NULL)
    {
        i++;                    //序号+1
        if(i % 2 == 0)        //处理序号为偶数的链表结点
        {
```

```
            rb->next = p;    //在表尾插入新结点
            rb = p;          //rb 指向新尾结点
        }
        else
        {
            ra->next = p;    //处理原序号为奇数的结点
            ra = p;          //在 A 表尾插入新的结点
        }
        p = p->next;         //将 p 恢复为指向新的待处理结点
    }
    ra->next = NULL;
    rb->next = NULL;
    return B;
}
```

9、设 C = {a_1，b_1，a_2，b_2，…，a_n，b_n}为线性表，采用头结点的 hc 单链表存放，设计一个就地算法，将其拆分为两个单链表，使得 A = {a_1，a_2，…，a_n}，B = {b_n，…，b_2，b_1}。

算法思想：**本题的思路和上题基本一样，不设置序号变量。**二者的差别仅在于对 B 表的建立不采用尾插法，而是采用头插法。

```
LinkList DisCreat(LinkList &A)
{
    LinkList B = (LinkList)malloc(sizeof(LNde)); //创建 B 表表头
    B->next = NULL;                  //B 表的初始化
     LNode *p = A->next, *q;         //p 为工作指针
    LNode *ra = A;                   //ra 始终指向 A 的尾节点
    while(p != NULL)
    {
        ra->next = p; ra = p;     //将*p 链接到 A 的表尾
        p = p->next;
        q = p->next;
        p->next = B->next;        //头插后，*p 将断链，因此 q 记忆*p 的后继
        B->next = p;              //*p 插入到 B 的前端
        p = q;
     }
    ra->next = NULL;              //A 的尾节点的 next 域置空
    return B;
}
```

10、设计算法将一个带头结点的单链表 A 分解为两个具有相同结构的链表 B 和 C，其中 B 表的结点为 A 表中小于零的结点，而 C 表中的结点为 A 表中值大于

零的结点（链表 A 中的元素为非零整数，要求 B、C 表利用 A 表的结点）。

【算法思想】
首先将 B 表的头结点初始化 A 表的头结点，为 C 申请一个头结点，初始化为空表。从 A 表的首元结点开始，一次对 A 表进行遍历。p 为工作指针，r 为 p 的后继指针。当 p->data<0 时，将 p 指向的结点适用前插法插入到 B 表中；当 pa->data>0 时，将 p 指向的结点适用前插法插入到 C 表，然后 p 指向新的待插入的结点

```
void Decompose(LinkList &A, LinkList &B, LinkList &C)
{
    //单链表 A 分解为两个具有相同结构的链表 B 和 C
    B = A;
    B->next = NULL;
    C = (LinkList)malloc(sizeof(LinkList));
    C->next = NULL;
    p = A->next;
    while(p != NULL)
    {
        r = p->next;                    //暂存 p 的后继
        if(p->data < 0)                 //将小于 0 的结点插入 B 中，前插法
        {
            p->next = B->next;
            B->next = p;
        }
        else
        {
            p->next = C->next;
            C->next = p;
        }
        p = r;                          //p 指向新的待处理结点
    }
}
```

11、设计一个算法，通过一趟遍历确定长度为 n 的单链表中值最大的结点，返回该结点的数据域。

【算法思想】
此题的关键在于：在遍历的时候利用指针 pmax 记录值最大的结点的 位置。
算法思想：初值时候指针 pmax 指向首元结点，然后在遍历过程中，用 pmax 依次和后面的结点进行比较，发现大者则用 pmax 指向该结点。这样将链表从头到尾遍历一遍时，pmax 所指向的结点中的数据即为最大值。

```
ElemType Max(LinkList L)
{
```

```
    //确定单链表中值最大的结点
    if(L->next == NULL)
        return NULL;
    pmax = L->next;              //pmax 指向首元结点
    p = L->next->next;          //p 指向第二个结点
    while(p != NULL)             //遍历链表，如果下一个结点存在
    {
        if(p->data > pmax->data)
            pmax = p;            //pmax 指向数值大的结点
        p = p->max;              //p 指向下一个结点，继续遍历
    }
    return pmax->data;
}
```

12、设计一个算法，删除递增有序表中值大于 mink 且小于 maxk（mink 和 maxk 是给定的两个参数，其值可以和表中的元素相同，也可以不同）的所有元素

```
void Delete_Min_Max(LinkList &L, int mink, int maxk)
{
    //删除递增有序表 L 中值大于 mink 且小于 maxk 的所有元素
    p = L->next;
    while(p && p->data <= mink)   //查找第一个值大于 mink 的结点
    {
        pre = p;
        p = p->next;
    }
    while(p && p->data < maxk)     //查找第一个值大于等于 maxk 的结点
        p = p->next;
    q = pre->next;
    pre->next = p;                 //修改待删除结点的指针
    while(q != p)                  //依次释放待删除结点的空间
    {
        s = q->next;
        free(q);
        q = s;
    }
}
```

13、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，使表中不再具有重复的元素。

【算法思想】

注意到题目是**有序表，**说明所有相同值域的结点都是相邻的。用 p 扫描递增单链

表 L，若*p 结点的值域等于其后继结点的值域，则删除后者，否则 p 移向下一个结点。

```
void DeleteSame(LinkList &L)
{
    //L 是递增有序的单链表，本算法删除表中数值相同的元素
    LNode *p = L->next, *q;   //p 为扫描工作指针
    if(p == NULL)
       return ;
    while(p->next != NULL)
    {
        q = p->next;                    //q 指向*p 的后继结点
        if(p->data == q->data)     //找到重复值的结点
        {
            p->next = q->next;     //释放 q 结点
            free(q);
         }
        else
             p = p->next;
     }
}
```

14、已知由单链表表示的线性表中,含有 3 类字符的数据元素(如:字母字符、数字字符和其他字符),试编写算法构造 3 个以循环链表表示的线性表,使每个表中只含同一类的字符,且利用原表中的节点空间作为这三个表的节点空间,头节点可另辟空间。

【算法思想】先创建 3 个头节点,用 p 扫描带头节点的单键表 L,将不同类型的数据元素采用头插法插入到相应的循环单链表中。对应的算法如下:

```
void Split(LinkList *L, LinkList * &A,LinkList * &B,LinkList * &C)
{
    LinkList * p= L- > next,* q;
    A= (LinkList *)malloc(sizeof(LinkList) );
    A->next = A;
    B= (LinkList *) malloc(sizeof (LinkList));
    B->next = B;
    C= (LinkList *)malloc(sizeof (LinkList));
    C->next = C;
    while (p!= NULL)
    {
        if (p-> data>='A' && p->data<= 'Z'|| p->data >= 'a' && p->data<='z')
        {
            q = p;p=p->next;
            q->next = A->next;
```

```
                A->next = p;     //采用头插法建表 A
            }
            else if (p->data >='O' && p->data<= '9')
            {
                q=p;p=p->next;
                q->next = A->next;A->next =p;     //采用头插法建表 B
            }
            else
            {
                q = p;p=p->next;
                q->next = C->next;
                C->next = p;         //采用头插法建表 C
            }
        }
}
```

15、已知带头节点的循环单链表 L 中至少有两个节点,每个节点的两个字段为 data 和 next,其中 data 的类型为整型。试设计一个算法判断该链表中每个元素的值是否小于其后续两个节点的值之和。若满足,则返回 true;否则返回 false。

【算法思想】用 p 扫描整个循环单链表 L,一旦找到这样的节点*p,它使得条件 p->data<p->next->data+p->next->next- >data 不成立,则中止循环,返回 0 ；否则继续扫描。当 while 循环正常结束时返回 1。对应的算法如下:

```
bool judge(LinkList *L)
{
    LinkList *p = L->next;
    bool b = true;
    while(p->next->next != L && b)
    {
        if(p->data > p->next->data + p->next->next->data)
            b = false;
        p = p->next;
    }
    return b;
}
```

双指针策略思想（重点）：

- ✔ 一快，一慢
- ✔ 一早，一晚
- ✔ 一左，一右

【2009 年统考】16、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++语言实现），关键之处请给出简要注释。
（1）【算法思想】
定义指针 p 和 q，初始化指针时均指向单链表的首元结点。首先将 p 沿链表移动到第 k 个结点，而 q 指针保持不变，这样当 p 移动到第 k+1 个结点时，p 和 q 所指结点的间隔距离为 k。然后 p 和 q 同时向下移动，当 p 为 NULL 时，q 所指向的结点就是该链表倒数第 k 个结点。
（2）【算法步骤】
①设定计数器 i = 0，用指针 p 和 q 指向首元结点
②从首元结点开始依顺着链表 link 依次向下遍历链表，若 p 为 NULL，则转向步骤⑤。
③若 i 小于 k，则 i+1；否则，q 指向下一个结点。
④p 指向下一个结点，转步骤②。
⑤若 i == k，则查找成功，输出该结点的 data 域的值，返回 1；否则，查找失败，返回 0.

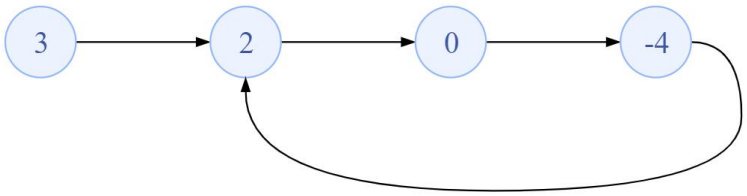
算法如下：

```
typedef struct LNode
{
    int data;
    struct LNode *next;
}LNode, *LinkList;

int Search_k(LinkList list, int k)
{
```

```
    //查找链表上第 k 个位置上的结点
    int i = 0;                    //初始化计数器
    p = q = list->link;          //p，q 均指向首元结点
    while(p != NULL)              //扫描链表，直到 p 为空
    {
        if(i < k)
            i++;                  //计数器+1
        else
            q = q->link;          //q 移到下一个结点
        p = p->link;              //p 移动到下一个结点
    }
    if(i == k)
    {
        cout << q->data;          //查找成功，输出 data 域的值
        return 1;
    }
    else
        return 0;                 //如果链表长度小于 k，查找失败
}
```

17、给定一个链表，判断链表中是否有环。如果有环，返回 1，否则返回 0。



1）给出算法的基本思想
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
下图给出了一个有环的链表。

```
typedef struct ListNode{
int data;
struct ListNode *next;
}
```

【算法思想】
快慢指针遍历链表，快指针步距为 2，慢指针步距为 1，如果链表带环，两指针一定会在环中相遇。

1、判断极端条件，如果链表为空，或者链表只有一个结点，一定不会带环，直接返回 NULL。
2、创建快慢指针，都初始化指向头结点。因为快指针每次都要步进 2 个单位，所以在判断其自身有效性的同时还要判断其 next 指针的有效性，在循环条件中

将两语句逻辑与并列起来。

初始化慢指针 slow = head，每次向后走一步；
初始化快指针 fast =head->next，每次走两步，每走一步判断一次。 如果存在环，fast 和 slow 必定会相遇。

```
int hasCycle(ListNode *head)
{     //判断链表是否有环
      if(head == NULL || head->next == NULL)
            return 0;
      ListNode *fast = head->next;
      ListNode *slow = head;
      while(slow != fast)
      {
            if(fast == NULL || fast->next == NULL)    //链表无环
                  return 0;
            slow = slow->next;
            fast = fast->next->next;
      }
      return 1;
}
```

栈和队列

1. 回文是指正读反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判断给定字符序列是否是回文。（提示：将一般字符入栈。）

【算法思想】

将字符串前一半入栈，然后，栈中元素和字符串后一半比较。即将第一个出栈元素和后一半串中第一个字符比较，若相等，则再出栈一个元素与后一个字符比较，依次类推，直至栈空，在这种情况下可判断该字符序列是回文。如果出栈元素与串中的字符进行比较时出现不等的情况，则可判断该字符序列不是回文。

【算法思想】

```
int isPalindrome(char *t)
{
    InitStack(S);
    len = strlen(t);                    //求字符串长度
    for(int i = 0; i<len/2; i++) //将一半字符入串
       Push(S,t[i]);
    if(len%2 != 0)
       i++;                                  //长度为奇数，跳过中间字段
    while(!EmptyStack(S))             //每弹出一个字符与相应字符比较
    {
        int tmp = Pop(S);
        if(tmp != t[i])
           return 0;                    //不相等返回 0
        else
           i++
    }
    return 1;                                //比较完毕均相等返回 1
}
```

2. 假设以数组 Q[m]存放循环队列的元素，同时设置一个标志域名 tag，以 tag == 0 和 tag == 1 来区别队头指针（front）和队尾指针（rear）相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入（EnQueue）和删除（DeQueue）算法。

【算法思想】

在循环队列中，增设一个 tag 类型的整型变量，进队时置 tag 为 1，出队时置 tag 为 0（因为入队操作可能导致队满，只有出队操作可能会导致队空）。队列 Q 初试时，置 tag == 0，front == rear == 0。这样队列的 4 要素如下：

队空条件：Q.front == Q.rear && Q.tag == 0

队满条件：Q.front == Q.rear && Q.tag == 1

进队操作：Q.data[Q.rear] = x; Q.rear = (Q.rear+1)%MaxSize; Q.tag = 1。

出队操作：x = Q.data[Q.front]；Q.front = (Q.front+1)%MaxSize ; Q.tag = 0;

```
//设 tag 法的循环队列入队算法
int EnQueue(SqQueue &Q, ElemType x)
{
    if(Q.front == Q.rear && Q.tag == 1)   //两个条件都满足时则队满
        return 0;
    Q.data[Q.rear] == x;
    Q.rear = (Q.rear+1)%MaxSize;
    Q.tag = 1;
    return 1;
}

//设 tag 法的循环队列出队算法
int DeQueue(SqQueue &Q, ElemType &x)
{
    if(Q.front == Q.rear && Q.tag == 0)
        return 0;
    x = Q.data[Q.front];
    Q.front = (Q.front+1)%MaxSize;
    Q.tag = 0;
    return 1;
}
```

3. 设计一个算法判断输入的表达式中括号是否配对（假设只含有左、右圆括号）

【算法思想】
该算法在表达式括号匹配时返回真，否则返回假。设置一个链栈 st，扫描表达式 exp，遇到左括号时进栈；遇到右括号时，若栈顶为左括号，则出栈，否则返回假。当表达式扫描完毕且栈为空时返回真；否则返回假。算法如下：

```
bool Match(char exp[], int n)
{
    int i = 0; char e;
    bool match = true;
    LinkStNode *st;
    InitStack(st);                    //初始化栈
    while(i < n && match)
    {
        if(exp[i] == '('          //当前字符为左括号，将其进栈
```

```
            Push(st,exp[i]);
        else if(exp[i] == ')')   //当前字符为右括号
        {
            if(GetTop(st,e) == true)   //成功取栈顶元素 e
            {
                if(e != '(')              //栈顶元素不为 （ 时
                    match = false;        //表示不匹配
                else                      //栈顶元素为 ）时
                    Pop(st,e);            //将栈顶元素出栈
            }
            else match = false;           //无法取栈顶元素时表示不匹配
        }
        i++                                //继续处理其他字符
    }
    if(!StackEmpty(st))                    //栈不空时，表示不匹配
        match = false;
    DestroyStakck(st);
    return match;
}
```

4、【课后变式习题】假设表达式中允许包含 3 种括号：圆括号、方括号和大括号。编写一个算法判断表达式中的括号是否正确匹配。

树和二叉树

1. 二叉树的先序递归遍历与非递归遍历

先序遍历过程：
若二叉树为空，什么都不做，否则
①访问根结点
②先序遍历左子树
③先序遍历右子树

递归算法：

```
void PreOrder(BiTree T)
{
    if(T != NULL)
    {
        visit(T);                    //访问根结点
        PreOrder(T->lchild);     //递归遍历左子树
        PreOrder(T->rchild);     //递归遍历右子树
    }
}
```

非递归算法：

```
void PreOrder(BiTree T)
{
    InitStack(S);
    p = T;
    while(p || !StackEmpty(S))
    {
        if(p)
        {
            printf(p->data);
            push(S,p);
            p = p->lchild;
        }
        else
        {
            pop(S,p);
            p = p->rchild;
        }
    }
}
```

2. 二叉树的中序非递归遍历

中序遍历过程：
若二叉树为空，则什么也不做；否则：
①中序遍历左子树
②访问根结点
③中旬遍历右子树

递归算法：

```
void InOrder(BiTree T)
{
    if(T != NULL)
    {
        InOrder(T->lchild);
        visit(T);
        InOrder(T->rchild);
    }
}
```

非递归算法：

```
void InOrder(BiTree T)
{
    InitStack(S);
    p = T;
    while(p || !StackEmpty(S))
    {
        if(p)
        {
            push(S,p);
            p = p->lchild;
        }
        else
        {
            pop(S,p);
            printf(p->data);
            p = p->rchild;
        }
    }
}
```

3. 二叉树的后序非递归遍历

后序遍历过程：
若二叉树为空，什么也不做，否则，
①后序遍历左子树
②后序遍历右子树
③后序遍历根结点

递归算法：

```
void PostOrder(BiTree T)
{
    if(T != NULL)
    {
        PostOrder(T->lchild);
        PostOrder(T->rchild);
        visit(T);
    }
}
```

非递归算法：

```
void PostOrder(BiTree T)
{
    InitStack(T);
    p = T;
    while(p || !StackEmpty(S))
    {
        if(p)
        {
            push(S,p);
            p = p->lchild;
        }
        else
        {
            pop(S,p);
            if(p->rchild && (p->rchild).tag == 0)
            {
                push(S,p);
                p = p->rchild;
            }
            else
            {
                printf(p->data);
                p.tag = 1;
```

```
                    p = NULL;
                }
            }
        }
}
```

注意：在二叉树的存储结构中每个结点增加一个是否访问的 tag 域，初始值均为 0

4. 二叉树的层序遍历算法

层次遍历的算法过程可以概括为：

- 先将二叉树根结点入队，
- 然后出队，访问该出队结点，
- 若它有左子树，则将左子树根结点入队，
- 若它有右子树，则将右子树根结点入队，
- 然后出队，访问出队结点.....
- 如此反复，直到队列为空。

```
void LevelOrder(BiTree)
{
    InitQueue(Q);                    //初始化辅助队列
    BiTree p;
    EnQueue(Q,T);                    //将根结点入队
    while(!IsEmpty(Q))               //队列循环不为空
    {
        DeQueue(Q,p);                //队头元素出队
        visit(p);                    //访问当前 p 所指向结点
        if(p->lchild != NULL)
            EnQueue(Q,p->lchild); //左子树不为空，左子树入队
        if(p->rchild != NULL)
            EnQueue(Q,p->rchild); //右子树不为空，右子树入队
    }
}
```

注：对于二叉树的层序遍历的算法，大家应该重点背下，当成一个模板，应用到下面各种题目中。

关于二叉树队列的一个问题，EnQueue(p->lchild)或者 Q[++rear] = p->lchild 都是一样的。如果对 EnQueue()使用不熟悉，可以使用 Q[++rear]的方式，看题目的具体使用场景。通过一个层次遍历的算法，给大家总结出课下面的几个例题能够运用到的层序遍历算法的模板。希望大家能够好好的品味！

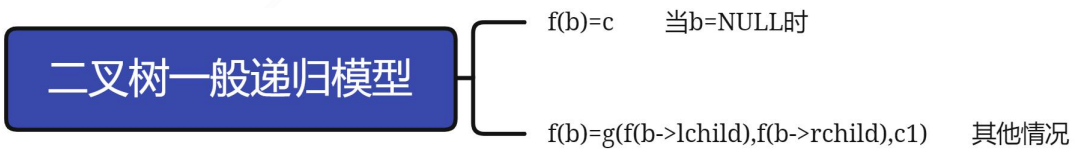
二叉树的递归算法设计策略

递归体的一般格式如下:
f(ds)=g(op(ds),c)

其中，f()为递归函数，ds 为递归数据，op 为基本递归运算符，g 为非递归函数，c 为常量。 对于二叉树，一般递归模型如下:

f(b)=c　　当 b=NULL 时

f(b)=g(f(b->lchild),f(b->rchild),c1)　　其他情况



对应的递归算法如下：

```
DataType fun(BTNode *b)
{
    if(b==NULL)
        return (c);
    else
        return g(fun(b->lchild),fun(b->rchild),c1)
}DataType;
```

5. 试编写二叉树的自下而上、从右到左的层次遍历算法。

【算法思想】
一般的二叉树层次遍历是自上而下，自左到右。在这里刚好相反。我们利用原有的层次遍历算法，各结点出队的同时将各结点指针入栈，在所有结点入栈后再从栈顶开始依次访问。

```
void InvertLevel(BiTree bt)
{
    Stack S; Queue Q;
    if(bt != NULL)
    {    //初始化栈和队列
        InitStack(S);
        InitQueue(Q);
        EnQueue(Q,bt);
        while(IsEmpty(Q) == false)      //从上而下层次遍历
        {
```

```
            DeQueue(Q,p);
            Push(S,p);                          //出队，入栈
            if(p->lchild)
                EnQueue(Q,p->lchild); //左子树不为空，左子树入队
            if(p->rchild)
                EnQueue(Q,p->rchild); //右子树不为空，右子树入队
        } //whiile
        while(IsEmpty(s) == false)
        {
            Pop(S,p);
            visit(p->data);                 //自下而上、从右到左的层次遍历
        }
    }//if
}
```

例题：设二叉树采用二叉链表的存储结构，试着编写非递归算法二叉树的最大高度（二叉树的最大宽度是指二叉树所有层中结点个数的最大值）。

【算法思想】
采用层次遍历的算法，设置变量 level 记录当前结点所在的层次，设置变量 last 指向当前层的最右结点。每层次遍历出队时与 last 指针比较，若两者相等，则层数+1，并且计算下一层的最右结点，直到遍历完成。level 的值即为二叉树的高度

```
int Bidepth(BiTree T)
{
    //求二叉树的最大宽度
    if(T == NULL)
        return 0;
    else
    {
        BiTree Q[MaxSize];              //假设队列的容量足够大
        BiTree p;
        int front = -1, rear = -1;          //初始化队列
        int last = 0, level = 0;            //level 记录当前结点所在的层次，
                                            //last 指向当前层的最右结点
        Q[++rear] = T;                      //将根结点入队
        while(front < rear)
        {
            p = Q[++front];
            if(p->lchild)
                Q[++rear] = p->lchild;    //左孩子入队
            if(p->rchild)
                Q[++rear] = p->rchild;    //右孩子入队
```

```
            if(front == level)
            {
                level++;                    //层数加+1
                last = rear;                //更新 last,指向下一层
            }//if
        }//while
    } //else
}
```

例题：计算二叉树的最大宽度（二叉树的最大宽度是指二叉树所有层中结点个数的最大值）

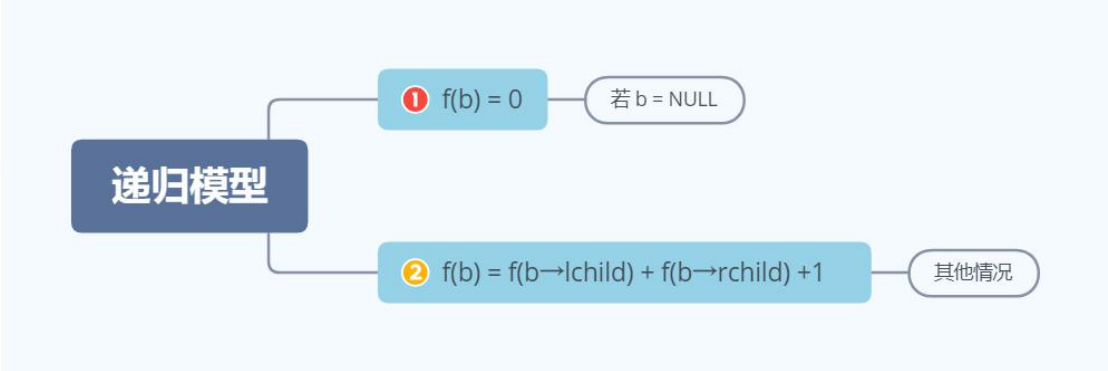
【算法思想】
计算二叉树的最大宽度可采用层序遍历的方法，利用队列来实现。首先判断树是否为空树，如果是空树，则宽度为 0；不为空则分别记录局部的宽度和当前最大宽度，逐层遍历结点，如果结点有孩子结点，则将其孩子结点加入队尾；每层遍历完毕之后，若局部宽度大于当前的最大宽度，则修改最大宽度。

```
int width(BiTree T)
{
    //求二叉树高度
    if(T == NULL)
        return 0;        //空二叉树高度为 0
    else
    {
        BiTree Q[MaxSize];     //Q 是队列，假设容量足够大
        front = rear = 1;      //初始化队列
        int last = 1;          //同层最右结点在队列中的位置
        int tmp = 0;           //局部宽度
        int maxw = 0;          //最大宽度
        Q[rear] = T;           //根结点入队
        BiTree p;
        while(front <= last)
        {
            p = Q[front++];
            tmp++;
            if(p->lchild != NULL)
                Q[++rear] = p->lchild;   //左子树入队
            if(p->rchild != NULL)
                Q[++rear] = p->rchild;   //右子树入队
            if(front > last)
            {
                last = rear;             //last 指向下层最右元素
                if(tmp > maxw)           //更新当前最大宽度
```

```
                    maxw = tmp;
                tmp = 0;
            }//if
        }//while
        return maxw;
    }
}
```

6. **假设二叉树采用二叉链存储结构存储，试设计一个算法，计算一棵给定二叉树的所有节点数。**

解:计算一棵二叉树的所有节点个数的递归模型 f(b)如下。

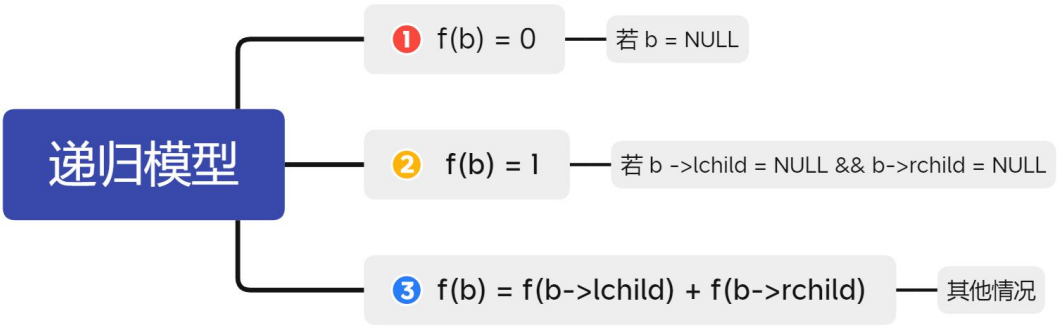



```
int Nodes(BTNode *b)
{ int num1,num2;
    if (b==NULL)
        return 0;
    else
    {   num1=Nodes(b->1child);
        num2=Nodes(b->rchild) ;
        return (numl +num2+1) ;
    }
}
```

7. **假设二叉树采用二叉链存储结构存储，设计一个算法计算一棵给定二叉树的所有叶子节点个数。**

注意：本题是所有叶子结点

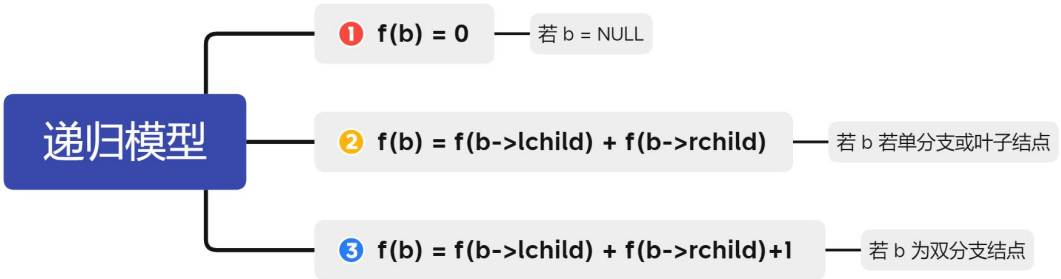
> 解：计算以刻二叉树的所有叶子结点个数的递归模型 f(b)如下：




```
int LeafNodes(BTNode *b)
{     int num1,num2;
      if (b == NULL)
            return 0;
      else if(b->lchild==NULL && b->rchild==NULL)
            return 1;
      else
    {     num1=LeafNodes(b->lchild);
            num2=LeafNodes(b->rchild);
            return(num1+num2);
    }
}
```

8. 假设二叉树采用二叉链存储结构存储，设计一个算法计算一棵给定二叉树的所有双分支节点。

解：计算一棵二叉树的所有双分支结点个数的递归模型 f(b) 如下：

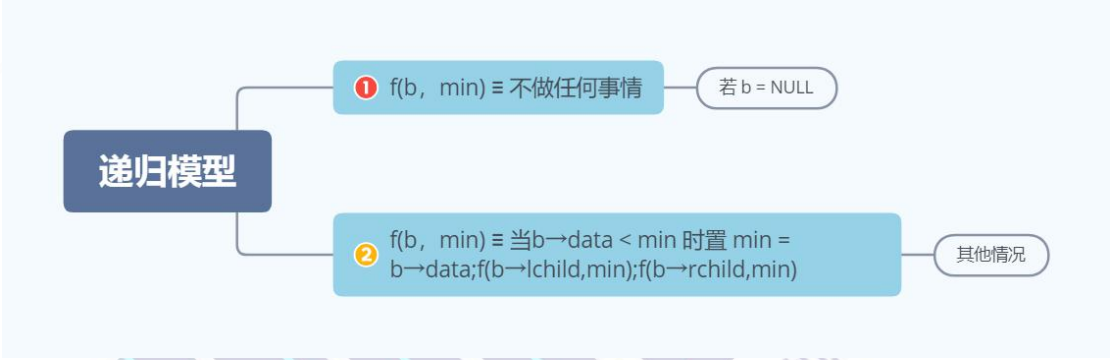



```
int DSonNodes(BTNode *b)
{     int num1,num2,n;
      if (b==NULL)
            return 0;
      else if (b->lchild==NULL || b->rchild==NULL)
            n=0;
```

```
                                        //为单分支或叶子节点时，不计
    else
        n=1;                            //为双分支节点时，计 1
    num1=DSonNodes(b->1child);          //递归求左子树的双分支节点数
    num2=DSonNodes(b->rchild);          //递归求右子树的双分支节点数
    return (num1+num2+n);
}
```

9、假设二叉树采用二叉链存储结构存储，设计一个算法求其中最小值的节点值。

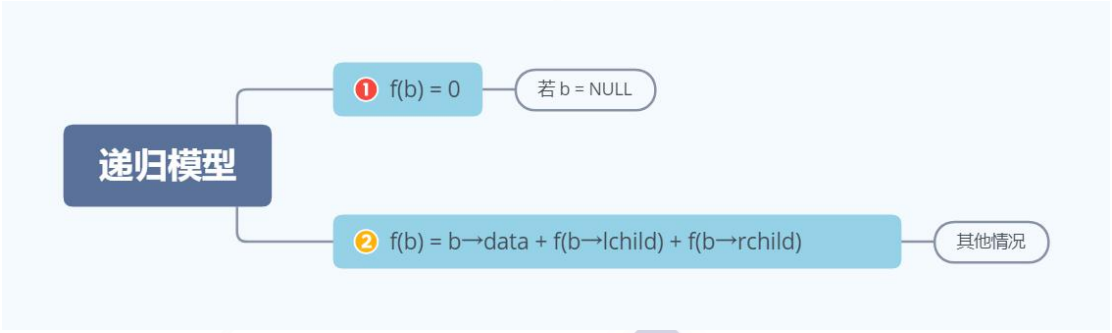
解：设 f(b,min)是在二叉树 b 中寻找最小结点值 min，其递归模型 f(b,min)如下。




```
void FindMinNode(BTNode *b,ElemType &min)
{
    if(b->data < min)
        min = b->data;
    FindMinNode(b->lchild,min); //在左子树中寻找最小节点值
    FindMinNode(b->rchild,min); //在右子树中寻找最小节点值 }
void MinNode(BTNode *b)
{
    if(b!=NULL)
    {
      ElemType min = b->data;
      FindMinNode(b,min);
      printf("Min = %d\n",min);
      }
}
```

10、假设二叉树采用二叉链存储结构存储，所有节点的值为正整数，设计一个算法求所有节点值之和。

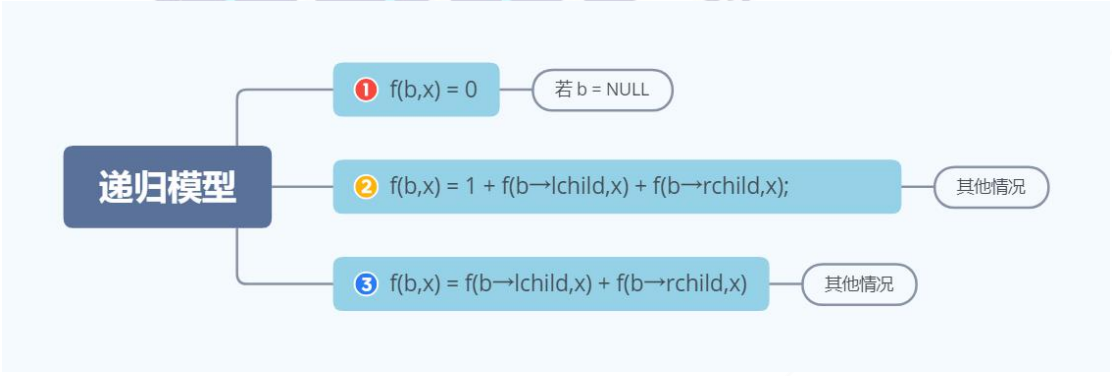
解：设 f(b)返回二叉树 b 中所有结点值之和，其递归模型 f(b)如下：




```
int FindSum(BTNode *b)
{ if(b==NULL)
        return 0;
   else
        return (b->data + FindSum(b->lchild)+FindSum(b->rchild));
}
```

11、假设二叉树采用二叉链存储结构存储，设计一个算法求其中节点值为 x 的节点个数。

解：设 **f(b,x)**返回二叉树 **b** 中所有结点值为 **x** 的结点个数，其递归模型 **f(b,x)** 如下：




```
int FindCount(BTNode *b,ElemType x)
{ if(b==NULL)
      return 0;
   else if(b->data == x)
       return (1+FindCount(b->lchild,x) + FindCount(b->rchild,x));
   else return (FindCount(b->lchild,x) + FindCount(b->rchild,x));
}
```

12、假设二叉树采用二叉链表存储结构，设计一个递归算法求二叉树的高度

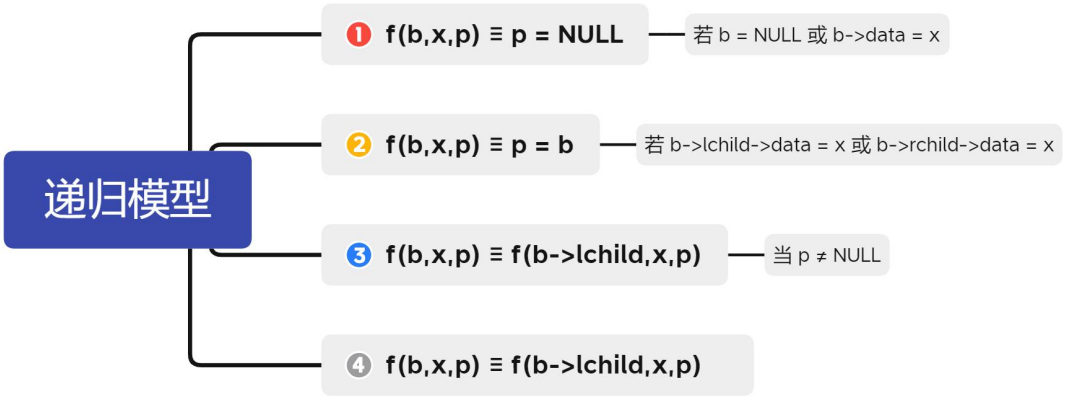
解：递归模型如下：


```
int BTNodeDepth (BTNode *b)
{
    int lchilddep, rchilddep;
    if (b==NULL)
        return(0);                          //空树的高度为 0
    else
    {
        childdep=BTNodeDepth(b->lchild);          //求左子树的高度为 lchilddep
        rchilddep-BTNodeDepth(b->rchild);          //求右子树的高度为 rchi lddep
        return (lchilddep > rchilddep) ?  (lchilddep+1) : (rchilddep+1);
    }
}
```

13、假设二叉树 b 采用二叉链存储结构，设计一个算法 void findparent(BTNode*b,ElemType x,BTNode *&p)求指定值为 x 的节点的双亲节点 p。

提示:根节点的双亲为 NULL，若在 b 中未找到值为 x 的节点,p 亦为 NULL，并假设二叉树中所有节点值是唯一的。

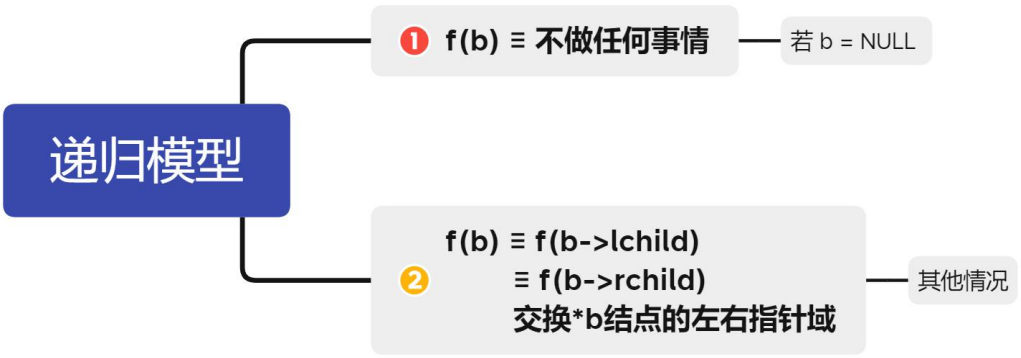
【算法思想】设在二叉树 b 中查找 x 节点的双亲 p 的过程为 f(b,x,p)，找到后 p 指向 x 节点的双亲节点,否则 p=NULL。当 b 为空树或根节点值为 x 时，p=NULL，否则在左子树中查找，若未找到则在右子树中查找。
其递归模型如下:




```
void findparent (BTNode *b,ElemType x, BTNode *&p)
{
    if (b!=NULL)
    {
        if(b->data==X)
            p=NULL;
        else if (b->lchild != NULL && b->lchild->data == x)
            p=b;
        else if (b->rchild != NULL && b->rchild->data == x)
            p=b;
        else
        {
            findparent (b->lchild, x,p);
            if (p==NULL)
            findparent (b->rchild, x,p);
        }
    }
    else p=NULL;
}
```

14、假设二叉树采用二叉链存储结构，设计一个算法把树 b 的左、右子树进行交换。要求算法的空间复杂度为 O(1)。

【算法思想】本题直接交换二叉树 b 的左、右子树，其递归模型如下（基于后序遍历算法)。

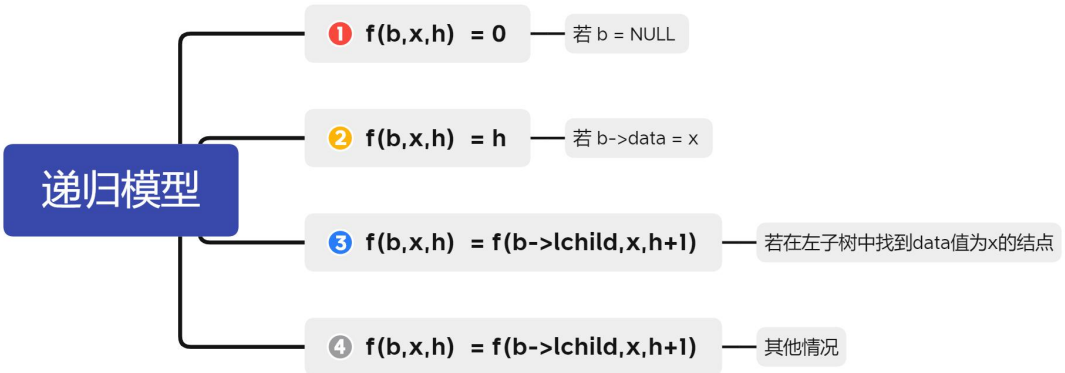



```
void Swap (BTNode * &b)
{
    BTNode *temp;
    if(b!=NULL)
    {
        Swap2 (b->lchild);     //交换左子树
        Swap2 (b->rchild);     //交换右子树
        temp-b->lchild;        //将*b 节点的左、右指针域进行交换
        b->lchild=b->rchild;
        b->rchild=temp;
    }
}
```

本算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

15、假设二叉树采用链式存储结构进行存储，设计一个算法，求二叉树 b 中值为 x 的结点层号

【算法思想】
本题对应的递归模型如下：



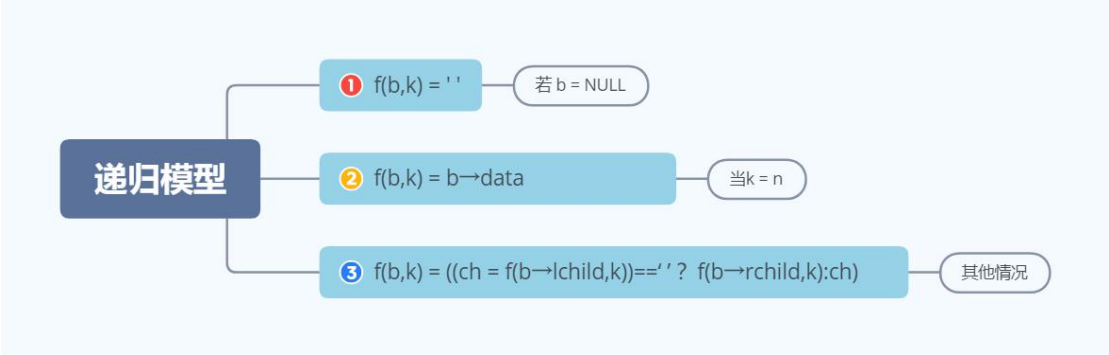
```
int NodeLevel (BTNode *b,ElemType x,int h)//调用本算法时 h 指出根节点的层次即为 1
{
    int hl;
    if(b==NULL)     //空树时返回 0
        return 0;

    else if (b->data==X)    //找到节点 x 时
        return h;
    else
    {
        hl=NodeLevel (b->lchild, x, h+1);    //在左子树中递归查找
        if(hl==0)
            return NodeLevel (b->rchild, x,h+1);//左子树中未找到时在右子树中查找
        else
            return hl;
    }
}
```

16、假设二叉树采用二叉链存储结构存储，设计一个算法，求**先序遍历序列**中第 k(1≤k≤二叉树中节点个数)个节点的值。

➢解:用一个全局变量 n(初值为 1)保存先序遍历时访问节点的序号。当二叉树 b 为空时返回特殊字符‘ ’(‘ ’为空格字符)，当 k==n 时表示找到了满足条件的节点，返回 b->data;当 k≠n 时，在左子树中查找，若找到了返回该值，否则在右子树中查找，并返回其结果。

➢对应的递归模型如下:




```
int n=1;                                    //全局变量
ElemType PreNode(BTNode *b,int k)
{   ElemType ch;
```

```
    if (b==NULL)
        return' ';
    if (n=-k)
        return(b->data);
    n++;
    ch=PreNode(b->lchild,k);                    //遍历左子树
    if(ch!=' ')
        return(ch);                              //在左子树中找到后返回
    ch=PreNode(b->rchild,k);                    //遍历右子树
    return(ch);                                  //返回右子树中的遍历结果
}
```

注：可以尝试用非递归的方法去写，用辅助栈

17、假设二叉树采用二叉链存储结构存储，设计一个算法，求**中序遍历序列**中第 k(1≤k≤二叉树中节点个数)个节点的值。

解:采用类似例上述的递归方法。对应的算法如下。.

```
int n=1;                                        //全局变量
ElemType InNode (BTNode *b, int k)
{ ElemType ch;
    if (b==NULL)
        return' ';
    ch=InNode(b->lchild,k);                     //遍历左子树
    if (ch!=' ')                                //在左子树找到了便返回 ch
        return ch;
    else
    {   if (n==k)
            return b->data;
        n++;
        return InNode(b->rchild,k);     //返回在右子树中查找的结果
    }
}
```

18、假设二叉树采用二叉链存储结构存储，设计一个算法，求**后序遍历序列**中第k(1≤k≤二叉树中节点个数)个节点的值。
解:采用类似上述的递归方法。对应的算法如下。

```
int n=1;                                    //全局变量
ElemType PostNode(BTNode *b,int k)
{ ElemType ch;
    if (b==NULL)
        return ' ';
    ch=PostNode(b->1child,k);               //遍历左子树
    if (ch!=' ')                            //在左子树找到了便返回 ch
        return ch;
    else
    {   ch=PostNode(b->rchild,k);           //遍历右子树
        if (ch!=' ')                        //在右子树找到了便返回 ch
        return ch;
        if (n==k)
            return b->data;
        n++;
    }
}
```

说明：上面给大家总结了 3 道习题，本质上都前、中、后序遍历算法的实现。二叉树遍历的算法可以引申出大量的算法题，因此考生务必熟练掌握二叉树的遍历算法

19、【2014 年统考真题】二叉树的带权路径长度（WPL）是二叉树中所有结点的带权路径长度之和。给定一棵二叉树 T，采用二叉链表存储，结点结构为

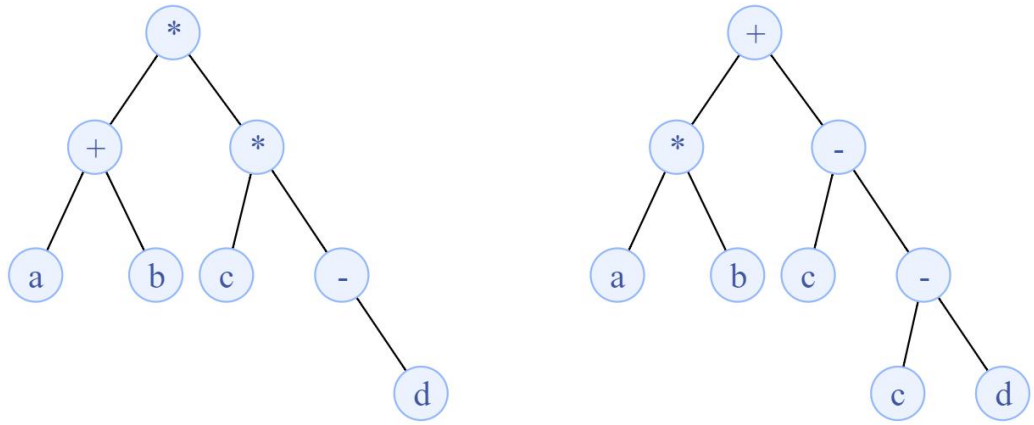
left	Weight	right

其中叶结点的 weight 域保存该结点的非负权值。设 root 为指向 T 的根结点的指针，请设计求 T 的 WPL 的算法，要求“
（1）给出算法的基本设计思想。
（2）适用 C 或 C++语言，给出二叉树结点的数据类型定义
（3）根据设计思想，采用 C 或 C++语言描述算法，关键之初给出注释。

【算法思想】
二叉树的带权路径长度为每个叶结点的深度与权值之积的总和，可以采用先序遍历或层序遍历解决
①基于先序递归遍历的算法思想：采用一个 static 变量记录 wpl,把每个结点的深度作为递归函数的一个参数传递，算法步骤如下：


```
int wpl_PreOrder(BiTree root, int deep)
{
    static int wpl = 0;     //定义一个 static 全局变量存储 wpl
    if(root->lchild == NULL && root->rchild == NULL)
        wpl = wpl + deep*root->weight;     //为叶子结点直接累计 wpl
    if(root->lchild != NULL)
        wpl_PreOrder(root->leight, deep+1);
    if(root->rchild != NULL)
        wpl_PreOrder(root->rchild, deep+1);
    return wpl;
}
```

20、【2017 年统考真题】设计一个算法，将给定的表达式树（二叉树）转换为等价的中缀表达式（通过括号反映操作符的计算次序）并输出。例如，当下列的两棵表达式树作为算法的输入时：



输出等价的中缀表达式分别为（a+b）*（c*（-d））和（a*b）+（-（c-d））。

二叉树的结点定义如下：

```
typedef struct node
{
    char data[10];
    struct node *left, *right;
}BTree;
```

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

解：
（1）基于二叉树的中序遍历方式即可得到该表达式
（2）算法实现：将二叉树的中序遍历递归算法稍加变形即可得到。除根结点而哼叶子结点外，遍历到其他结点是在遍历其左子树之前加上左括号，遍历完右子树后加上右括号：

```
void BtreeToExp(BTree *root, int deep)
{
    if(root == NULL)
        return ;
    else if(root->left == NULL && root->right == NULL) //若为叶子结点
        printf("%s", root->data);
    else
    {
        if(deep > 1)
            printf("(");                          //若有子表达式则加 1 层括号
            BtreeToExp(root->left, deep+1);
            printf("%s", root->data);
            BtreeToExp(root->right, deep+1);
            if(deep > 1)
                printf("）");                     ////若有子表达式则加 1 层括号
    }
}
```

图

引言：在数据结构算法中，图算法设计有一定的难度。要提高这块的设计能力，必须要掌握如下基本结构:

① 图的两种存储结构。图算法都是基于某种存储结构的，如果连存储结构都含糊不清楚，肯定是设计不出来正确的算法，更谈不上设计好的图算法。因此务必去熟悉图的邻接矩阵和邻接表的存储结构

② 图的两种遍历算法。很多图的算法基本都是基于图的遍历算法（正如二叉树的算法是基于二叉树的基本遍历算法一样）。一般涉及不带权图的最短或最长路径时可以考虑采用广度优先遍历算法；而涉及到查找所有简单路径时可考虑采用深度优先遍历算法。

大家可以通过本章节的算法示例总结图算法设计的规律，希望大家可以喜喜品味本章节的算法。

在展示本章节的算法设计题之前，我们先带着大家回顾一下图的两种存储结构的结构体以及图的两种基本遍历的方式。

图的邻接矩阵存储结构定义如下:

```
#define MaxVertexNum 100      //顶点数目的最大值
typedef char VertexType;      //顶点的数据类型
typedef int EdgeType;         //带权图中边上权值的数据类型
typedef struct
{
    VertexType Ver[MaxVertexNum];     //顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum];   //邻接矩阵、边表
    int vexnum, arcnum;
}MGraph;
```

图的邻接表存储结构定义如下：

```
#define MaxVertexNum 100      //顶点数目的最大值
typedef struct ArcNode          //边表结点
{
    int adjvex;      //该弧所指向的顶点的位置
    struct ArcNode *next;   //指向下一条弧的指针
    //InfoType info;     //网的边权值
}ArcNode;
typedef struct VNode             //定点表结点
{
    VertexType data;           //顶点信息
    ArcNode *first;           //指向第一条依附该顶点的弧的指针
}VNode,AdjList[MaxVertexNum];
typedef struct
{
    AdjList vertices;     //邻接表
    int vexnum,arcnum;   //图的顶点数和弧数
}ALGraph;                          //ALGraph  是以邻接表存储的图类型
```

深度优先遍历：

深度优先遍历类似于树的先序遍历，它的基本过程如下：

① 从图中的某个初始顶点 v 出发，首先访问初始顶点 v。

② 选择一个与顶点 v 相邻且没有被访问过的顶点 w 为初始顶点，再从 w 出发进行深度优先遍历，直到图中与当前顶点 v 邻接的所有顶点都被访问过为止。

深度优先遍历递归算法如下：

```
bool visited[MAX_VERTEX_NUM];   //访问标记数组
void DFSTraverse(Graph G)
{
    for(v = 0; v<G.vexnum; v++)
        visited[v] = FALSE;     //初始化已访问标记数组
    for(v = 0; v<G.vexnum; v++)
        if(!visited[v])
            DFS(G,v);
}

void DFS(Graph G, int v)        //从顶点 v 出发，深度优先遍历图 G
{
```

```
    visit(v);
    visited[v] = TRUE;
    for(w = FirstNeighbor(G,v); w>=0; w = NextNeighbor(G,v,w))
        if(!visited[w])         //w 为 u 尚未访问过的邻接顶点
            DFS(G,w);
}
```

广度优先遍历：

深度优先遍历类似于树的层次遍历，它的基本过程如下：

①访问初始点 v，接着访问 v 的所有未被访问过的邻接点 v_1，v_2，…，v_n。

②按照 v_1，v_2，…，v_n 的次序，访问每个顶点的所有未被访问过的邻接点。

③依次类推，直到图中所有和初始点 v 有路径的顶点都被访问过为止。

以邻接表为存储结构，用广度优先遍历图时，需要使用一个辅助队列 Q，以类似于按照层次遍历二叉树一样去遍历图，广度优先遍历算法如下：

```
bool visited[MAX_VERTEX_NUM];       //访问标记数组
void BFSTraverse(Graph G)
{
    for(i = 0; i<G.vexnum; i++)
        visited[i] = FALSE;         //初始化已访问标记数组
    InitQueue(Q);                   //初始化辅助队列 Q
    for(i = 0; i<G.vexnum; i++) //从 0 号顶点开始遍历
        if(!visited[i])             //对每个连通分量调用一次 BFS
            BFS(G,i);               //vi 未被放过过，从 vi 开始 BFS
}

void BFS(Graph G, int v)            //从顶点 v 出发，广度优先遍历图 G
{
    visit(v);                       //访问初始顶点 v
    visited[v] = TRUE;              //对 v 做访问标记
    EnQueue(Q,v);                   //顶点 v 入队
    while(!isEmpty(Q))
    {
        DeQueue(Q,v);               //顶点 v 出队列
        for(w = firstNeighbor(G,v); w>=0; w = NextNeighbor(G,v,w))
                                    //检测 v 的所有邻接点
            if(!visited[w])
```

```
                {    //w 为 v 的尚未访问过的邻接顶点
                     visit(w);          //访问顶点 w
                     visited[w] = TRUE;  //对 w 做已访问标记
                     EnQueue(Q,w);      //顶点 w 入队列
                }
        }
}
```

注：辅助数组 visited[ ]标志顶点是否被访问过，其初始状态为 FALSE。在图的遍历过程中，一旦某个顶点 vi 被访问过，就立刻设 visited[i] = TRUE，防止它被多次访问。

1. 设计一个算法，判断一个无向图 G 是否为一棵数。若是一棵树，则算法返回 true，否则返回 false.

【算法思想】
一个无向图 G 是一棵数的条件，G 必须是无回路的连通图或有 n-1 条边的连通图。这里采用后者作为判断条件。采用深度优先遍历算法在遍历图的过程中统计可能访问到的顶点个数和边的个数，若一次遍历就能访问到 n 个顶点和 n-1 条边，则可断定，此图是一课数，算法如下：

```
bool isTree(Graph &G)
{
    for(i = 0; i<G.vexnum; i++)
        visited[i] = FALSE;              //访问标记数组初始化
    int Vnum = 0, Enum = 0;              //初试化顶点数和边数
    DFS(G,1,Vnum,Enum,visited);
    if(Vnum == G.vexnum && Enum == 2*(G.vexnum-1))
        return true;                     //符合树的条件
    else
        return false;
}
void DFS(Graph &G, int v, int &Vnum, int &Enum, int visited[])
{
    //深度优先遍历
    visited[v] = TRUE; Vnum++;     //做标记访问，顶点计数
    int w = FirstNeighbor(G,v);
    while(w!=-1)
    {
        Enum++;
```

```
        if(!visited[w])
            DFS(G,w,Vnum,Enum,visited);
        w = NextNeighbor(G,v,w);
    }
}
```

2. 写出图的深度优先搜索 DFS 的非递归算法（图采用邻接表存储形式）。

【算法思想】

要实现深度优先遍历的非递归过程，需要借助一个栈来保存访问过的顶点。在深度优先遍历的非递归算法中使用一个栈来记忆下一步将要访问的顶点，同时使用一个访问标记数组 visited[i]来记忆第 i 个顶点是否在栈内。首先将栈初始化为空，然后将起始顶点 v 进栈。判断栈是否为空，若不为空则重复进行下列操作：

①首先，取栈顶元素，如果该顶点未被访问，则访问该顶点，将访问标志改为“已访问”；

②然后，将该顶点的所有未访问过的邻接顶点进栈；直至栈为空，表面图中所有的顶点都被访问过。

写法一：

```
void DFS_NonRC(Graph &G, int v)
{
    //从顶点开始进行深度优先搜索
    int w;
    InitStack(S);

    for(i = 0; i<G.vexnum; i++)
        visited[i] = FALSE;                    //初始化 visited
    Push(S,v)  ;
    visited[v] = TRUE;                         //将v  入栈并置为 true
    while(!IsEmpty(S))
    {
        k = Pop(S);                            //栈中退出一个顶
        visit(k);                              //先访问，再将其子结点入栈
        for(w = FirstNeighbor(G,k) ; w>=0; w = NextNeighbor(G,k,w))
        {     //k  的所有邻接点
            if(!visited[w])                    //未进过栈的顶点进

            {

                Push(S,w);
                visited[w] = true;         //标
            }//if
```

```
            }//for

        } //while
}
```

写法二：

```
void DFS_NonRC(ALGraph G, int v)
{    //从第 v 个顶点出发非递归实现深度优先遍历图 G
      int w;
      InitStack(S);          //初始化栈
      Push(S,v);             //顶点 v 入栈
      while(!StackEmpty(S))
      {
            Pop(S,k);          //栈顶元素 k 出栈
            if(!visited[k])
            {
                  visite(k);
                  visited[k] = true;
                  p = G.vertices[k].firstarc;
                  while(p!=NULL)
                  {
                        w = p->adjvex;          //w 是 k 的邻接点
                        if(!visited[w])        //如果 w 为访问，w 进栈
                              Push(S,w);
                        p = p->nextarc;         //p 指向下一个边结点
                  }//while
            }//if
      }//while
}//DFS_NonRC
```

3. 分别采用基于深度优先遍历和广度优先遍历算法判别以邻接表方式存储的有向图中是否存在有顶点 V_i 到顶点 V_j 的路径（$i \neq j$）。

【算法思想】

两个不同的遍历算法都采用从顶点 V_i 出发，依次遍历图中的每个顶点，直到搜索到顶点 V_j，若能够搜索到 V_j，则说明存在顶点 V_i 到顶点 V_j 的路径。

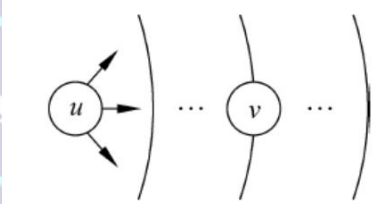
```
bool visited[MaxSize] = false;     //访问标记数组
int Exit_Path_DFS(ALGraph G, int i, int j)
{
    int p;       //顶点序号
    //深度优先遍历判断有向图 G 中顶点 Vi 到 Vj 是否有路径，是则返回 1，否
```

```
则返回 0
    if(i == j)
        return 1;       //i 就是 j 的时候
    else
    {
        visited[i] = 1;     //置访问标记
        for(p = FirstNeighbor(G,i); p>=0; p = NextNeighbor(G,i,p))
        {
            k = p->adjvex;
            if(!visited[p] && Exist_Path_DFS(G,p,j))
                return 1;
        }
    }
    return 0;
}
```

4.假设图 G 采用邻接表存储，设计一个算法，求不带权无向连通图 G 中从顶到 u 到顶点 v 的一条最短路径。

【算法思想】

图 G 是不带权的无向连通图，一条边的长度计为 1,因此求顶点 u 和顶点 v 的最短即求距离顶点 u 到顶点 v 的最少顶点序列。利用广度优先遍历算法，从 u 出发进行广度遍历，类似于从顶点 u 出发一层一层地向外扩展，当第一次找到顶点 v 时队列中便包含了从顶点 u 到顶点 v 地最短路径。



由于要利用队列找出路径，所以要设计成非环形队列，其类型声明如下：

```
typedef struct
{
    int data;          //顶点编号
    int parent;       //前一个顶点的位置
}QUERE;                    //非环形队列类型
```

算法图下：

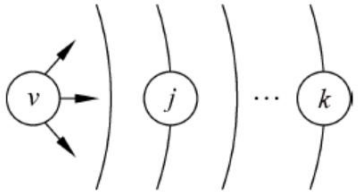
```
void ShortPath(AdjGraph *G, int u, int v)
{   //输出从顶点 u 到顶点 v 的最短逆路径
    ArcNode *p; int w,i;
    QUERE qu[MAXV];
```

```
    int front = -1, rear = -1;        //初始化队列
    int visited[MAXV];                //设置访问标记数组
    for(i = 0; i<G.vexnum; i++)
        visited[i] = 0;               //初始化标记访问数组
    rear++;
    qu[rear].data = u;                //顶点 u 入队
    visited[u] = 1;
    while(front!=rear)
    {
        front++;
        w = qu[front].data;
        if(w == v)
        {
            i = front;
            while(qu[i].parent != -1)
            {
                printf("%2d", qu[i].data);
                i = qu[i].parent;
            }
            printf("%2d\n", qu[i].data);
            return;
        }
        p = G.adjlist[w].firstarc;    //寻找 w 的第一个邻接点
        while(p!=NULL)
        {
            if(visited[p.adjvex] == 0)
            {
                visited[p.adjvex] = 1;
                rear++;               //将 w 的未访问过的邻接点进队
                qu[rear].data = p.adjvex;
                qu[rear].parent = front;
            }
            p = p->nextarc;           //寻找 w 的下一个邻接点
        }
    }
}
```

5、设计一个算法，求不带权无向连通图 G 中距离顶点 v 最远的一个顶点（所谓最远就是到大 v 的路径长度最大）

【算法思想】

图 G 是不带权的无向连通图，一条边长度计为 1，因此求距离顶点 v 的最远的顶点即求距离顶点 v 的边数最多的顶点。基于图的广度优先搜索遍历方式，其体现了图中由某个顶点开始，以近向远扩展的方式遍历图中结点的过程。从顶点 v 出发进行广度优先遍历类似于从顶点 v 出发一层一层地向外扩展，到达顶点 j，……，最后到达的一个顶点 k 即为距离 v 最远的顶点。




```
int maxdist (AGraph *G, int v)
{       ArcNode *p;
        int Qu [MAXV], front=0, rear=0;                  //队列及首、尾指针
        int visited [MAXV],i,j,k;
        for (i=0;i<G->n;i++)                             //初始化访问标志数组
           visited[i]=0;
        rear++;Qu[rear]=v;                               //项点 v 进队
        visited[v]=1;                                    //标记 v 已访问
        while (rear!=front)
        {   front= (front+1) %MAXV;
            k=Qu[front] ;                                //顶点出队
            p=G->adjlist[k] . firstarc;                  //找第 1 个邻接点
            while (p! =NULL)                             //所有未访问过的邻接点进队
            {    j=p->adjvex;
                 if (visited[j]==0)                      //若 j 未访问过
                 {    visited[j]=1;                     //将顶点 j 进队
                      rear= (rear+1) %MAXV;Qu[rear]=j;
                 }
                 p=p->nextarc;                           //找下一个邻接点
            }
        }
        return k;
}
```

6、假设图采用邻接表存储，分别写出基于 DFS 和 BPS 遍历的算法来判别顶点 i 和顶点 j (i≠j）之间是否有路径。

【算法思想】

先置全局数组 visited[]所有元素为 0，然后从顶点 i 开始进行某种遍历，遍历结束之后，若 visited[j]=0，说明顶点 i 与顶点 j 之间没有路径;否则说明它们之间存在路径。

基于 DFS 遍历的算法如下：

```
int DFSTrave(AGraph *G, int i,int j)
{     int k;
      for (k=0;k<G->n;k++)
         visited[k]=0;
      DFS(G,i);                                    //从顶点 i 开始进行深度优先遍历
      if(visited[j]==0)
         return 0;
      else
         return 1;
}
```

基于 BFS 遍历的算法如下：

```
int BFSTrave(AGraph *G,int i, int j)
{      int k;
       for ( k=0;k<G->n;k++)
          visited[k]=0;
       BFS(G,i);                                   //从顶点 i 开始进行广度优先遍历
       if (visited[j]==0)
          return 0;
       else
          return 1;
}
```

7、假设图 G 采用邻接表存储，设计一个算法，判断无向图 G 是否连通。若连通则返回 1;否则返回 0。

解:采用遍历方式判断无向图 G 是否连通。若用深度优先遍历方法，先给 visited[]数组置初值 0，然后从 0 顶点开始遍历该图。在一次遍历之后，若所有顶点 i 的 visited[i]均为 1，则该图是连通的;否则不连通。对应的算法如下。

```
int Connect (AGraph *G)                          //判断无向图 G 的连通性
{     int i,flag=1;
      for (i=0;i<G->n;i++)
         visited[i]=0;
      DFS(G,0);                                  //调用 DFS 算法
      for (i=0;i<G->n;i++)
         if (visited[i]==0)
         {
               flag=0;
```

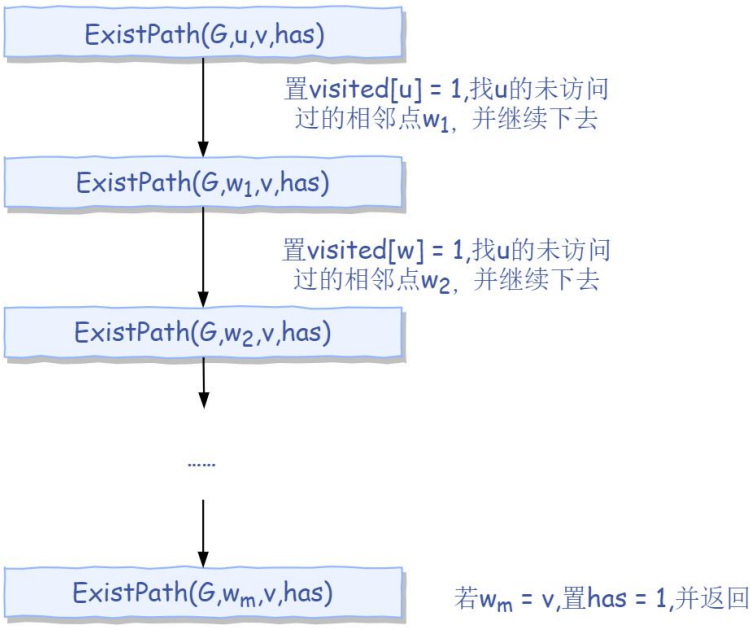
```
            break;
        }
    return flag;
}
```

若用广度优先遍历方法，先给 visited[]数组置初值 0，然后从 0 顶点开始遍历该图。在一次遍历之后，若所有顶点 i 的 visited[i]均为 1，则该图是连通的;否则不连通。对应的算法如下。

```
int Connect1(AGraph *G)                          //判断无向图 G 的连通性
{       int i,flag=1;
        for (i=0;i<G->n;i++)visited[i]=0;BFS(G,0);    //调用 BFS  算法
        for (i=0;i<G->n;i++)
        if (visited[i]==0)
        {
            flag=0;
            break;
        }
    return flag;
}
```

8、假设图 G 采用邻接表存储，设计一个算法，判断图G 中从顶点 u 到v 是否存在简单路径。

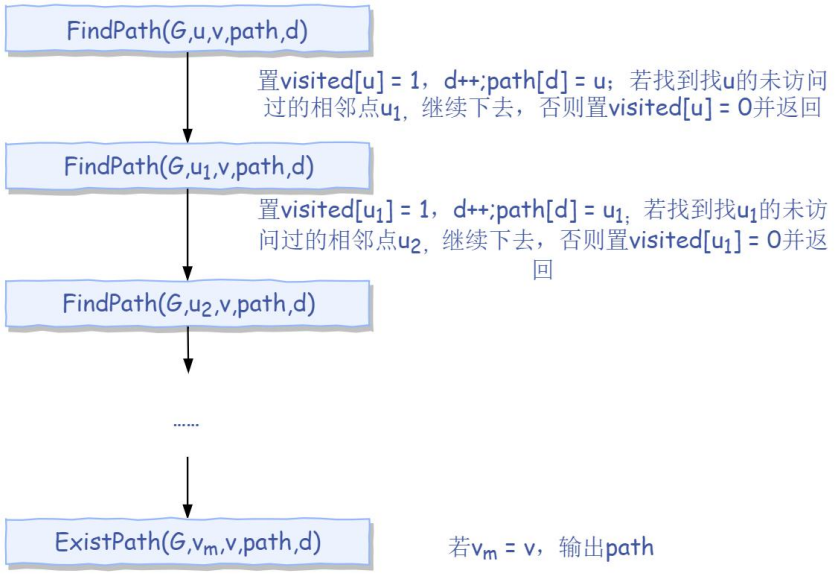
解:采用深度优先遍历的方法，在 DFS 算法基础上改为 G、u、v、has 共4 个形参，其中 has 表示顶点 u 到v 是否有路径，其初值为 0。查找从顶点 u 到v 是否存在简单路径过程如图所示。




```
void ExistPath (AGraph *G,int u,int v,int &has)
{       //has 表示 u 到 v 是否有路径,初值为 0
        int w;
        ArcNode *p;
        visited[u]=1;                           //置已访问标记
        if (u==v)                               //找到了一条路径
        {       has=1;
                return;
        }
      p=G->adjlist[u].firstarc;                 //p 指向顶点 u 的第一个相邻点
      while (p!=NULL)
      {       w=p->adjvex;                      //w 为顶点 u 的相邻顶点
            if(visited[w] ==0)                  //若 w 顶点未访问，递归访问它
               ExistPath(G, w, v, has);
            p=p->nextarc;                       //P 指向顶点 v 的下一个相邻点
      }
}
```

7. 假设图 G 采用邻接表存储，设计一个算法，输出图 G 中从顶点 u 到v 的所有简单路径。

【算法思想】:采用深度优先遍历的方法，在 DFS 算法的基础上改为 G、u、v、path 和d 共5 个形参， 其中 path 数组存放路径（初始为空)，d 表示路径长度（初始为-1)，查找从顶点 u 到v 的所有简单路径过程如图所示。




```
void FindPath(AGraph *G,int u,int v,int path[],int d)
//d 是到当前为止已走过的路径长度,调用时初值为-1
{       int w,i;
        ArcNode *p;
        d++;                                    //路径长度增 1
        path[d] =u;                             //将当前顶点添加到路径中
        visited[u] =1;                          //置已访问标记
        if (u == v)                             //找到一条路径则输出
           printf("%2d" , path[i]);             //输出路径
        p=G->adjlist[u].firstarc;               //p 指向 v 的第一个相邻点
        while (p!=NULL)
        {     w=p->adjvex;                      //w 为顶点 u 的相邻顶点
              if(visited[w]==0)                 //若 w 顶点未访问,递归访问它
                 FindPath(G, w,v , path, d);
              p=p->nextarc;                     //p 指向 v 的下一个相邻点
           }
           visited[u]=0;                        /恢复环境,使该顶点可重新使用
}
```

10、假设图 G 采用邻接表存储，设计一个算法，判断无向图 G 中任意两点之间是否存在一条长度为 k 的简单路径。

【算法思想】

可以利用深度优先搜索的算法递归遍历邻接表来进行判断。设置 visited[]为标记访问数组，已经访问的顶点记为 true。该算法每递归调用一次，长度 k-1，递归结束的条件为首尾顶点相遇且 k==0，此时表面两个顶点之间存在一条长度为 k 的简单路径，返回 1。如果递归结束的条件不成立，则从 vi 的边链表的第一个边界点开始，获取 vi 的邻接点，对 vi 的尚未访问的邻接点递归调用，长度 k-1。如果最后递归结束的条件不成立且无法继续递归下去，并不两点之间不存在长度 k 的简单路径，返回 0。

```
int visited[MaxSize];
int PathLenK(ALGraph G, int i, int k, int k)
{
    if(i == j && k == 0)       //找到一条路径符合要求
       return 1;
    else if(k > 0)
    {
        visited[i] = true;
        for(p = G.vertices[i].firstarc; p; p = p->nextarc)
        {
            //从结点 i 开始遍历，p 为 i 的边链表的第一个边结点
            v = p->adjvex;
            if(!visited[v])                    //v 未被访问
               if(PahtLenK(G,v,j,k-1))
                  return    1;
        }
        visited[i] = false;       //允许曾经被访问过的结点出现在另外一条路径上
    }
    return 0;
 }
```

查找和排序

1. 折半查找非递归算法（二分查找）

```
int Binary_Search(SqList L, ElemType key)
{
    //在有序表 L 中查找关键字为 Key 的元素，若存在则返回其他位置，不存在则返回-1
    int low = 0, high = L.length-1, mid;
    while(low <= high)
    {
        mid = (low + high)/2;       //取中间位置
        if(L.elem[mid] == key)
            return mid;             //查找成功返回所在位置
        else if(L.elem[mid] > key)
            high = mid-1;           //从前半部分继续查找
        else
            low = mid+1;            //从后半部分继续查找
    }
    return -1;
}
```

2. 折半查找递归算法

```
int BinSearch_Cur(SqList L, KeyType key, int low, int high)
{
    if(low > high)
        return 0;       //查找不到时返回 0；
    if(low <= high)
    {
        mid = (low+high)/2;
        if(L.elem[mid] == key)
            return mid;
        else if(key < L.elem[mid])
            return BinSearch_Cur(L, key, low, mid-1);    //对左子表递归查找
        else
            return BinSearch_Cur(L, key, mid+1, high); //对右子表递归查找
    }
}
```

3. 试着写一个算法判断二叉树是否为二叉排序树的算法

【算法思想】
根据二叉排序树的定义，对二叉树进行递归遍历，左子树关键字比根结点关键字小，右子树的关键字比根结点的关键字大，一旦有不满足条件则可判断不是二叉排序树。

通过参数 flag 的值来判断，flag 为 1 表示是二叉排序树，为 0 则表示非二叉排序树，flag 初值为 1。设定全局变量 pre(初始值为 NULL)来指向遍历过程结点的前驱。

```
void JudgeBST(BiTree T, int &flag)
{
    //判断二叉树是否为二叉排序树
    if(T != NULL && flag)
    {
        JudgeBST(T->lchild, flag);        //中序遍历左子树
        if(pre == NULL)                   //中序遍历的第一个结点不必判断
            pre = T;
        else if(pre->data < T->data)
            pre = T;                      //前驱指针指向当前结点
        else flag = 0;                    //不是二叉排序树
        JudgeBST(T->rchild, flag);        //中序遍历右子树
    }
}
```

方法二：对二叉排序树来说，其中序遍历序列为一个递增有序序列，因此，对给定的二叉树进行中序遍历，如果始终能保持前一个值比后一个值小，则说明该二叉树是一棵二叉排序树，算法如下。

```
KeyType predt=-32767; //predt 为全局变量，保存当前节点中序前驱的值，初值为-∞
int judgeBST(BSTNode *bt)
{
    int b1, b2;
    if (bt==NULL)     //空树是一棵二叉排序树
        return 1;
    else
    {
        b1=judgeBST (bt->lchild);   //判断左子树
        if(b1==0ll predt>=bt->key)
            return 0;
        predt=bt->key;       //保存当前节点的关键字
        b2=judgeBST (bt->rchild);     //判断右子树
        return b2;
    }
}
```

5、利用二叉树遍历的思想设计一个判断二叉树是否为平衡二叉树的算法。

【算法分析】设 balance 为平衡二叉树的标记，返回二叉树 bt 是否为平衡二叉树，若为平衡二叉树，balance 为 1;否则为 0。h 为二叉树 bt 的高度。采用递归先序遍历的判断方法。对应的算法如下。

```
void judgeAVL (BSTNode *bt,int &balance,int &h)
{
    int bl, br, hl, hr;
    if(bt==NULL)
    {
        h=0;
        balance=1;
    }
    else if (bt->lchild==NULL && bt->rchild==NULL)
    {
        h=1;
        balance = 1;
    }
    else
    {
        judgeAVL(bt->lchild,h1);
        judgeAVL(bt->rchild,hr);
        h = (h1>h2?h1:hr)+1;
        if(abs(h1-h2)<2)
            balance = b1&br;
        else
            balance = 0;
    }
}
```

6、设计一个算法，计算一棵 AVL 树中所有节点的 bf(平衡因子）值。

解:计算一个节点*b 之 bf 值的递归模型 f( )如下。

```
int Compbf1 (BSTNode * &b)
{
    int max1 , max2;
    if (b-=NULL)
        return 0;
    if(b->lchild==NULL && b->rchild==NULL)
    {
```

```
            b->bf=0;
            printf(" %d: 8d\n", b->key,b->bf);
            return 1;
        }
        else
        {
            max1=Compbf1 (b->lchild);
            max2-Compbf1 (b->rchild);
            b->bf-max1-max2;
            printf(" %d: %d\n",b->key,b->bf);
            return (max1>max2 ?max1+1 :max2+1);
        }
}
```

5、快速排序算法

```
int Partition(SqList &L, int low, int high)
{
    L.r[0] = L.r[low];          //用子表的第一个记录作为枢纽记录
    pivotkey = L.r[low].key;       //枢纽记录关键字
    while(low < high)
    {
        while(low < high && L.r[high].key >= pivotkey)
            --high;
        L.r[low] = L.r[high];     //将比枢纽记录小的移动到低端

        while(low < high && L.r[low].key <= pivotkey)
            ++low;
        L.r[high] = L.r[low];    //将比枢纽记录大的移动到高端
    }
    L.r[low] = L.r[0];
    return low;                              //返回枢纽位置
}

void QSort(SqList &L, int low, int high)
{
    if(low < high)
    {
        pivotloc = Partition(L, low, high);
        QSort(L,low,pivotloc-1);
        QSort(L,pivotloc+1, high);
```

```
    }
}
```

点评：在各种内部排序中，快速排序可谓是相当经典的排序算法，因此，快速排序算法是大家必须中重点掌握了理解的。下面给大家挑选了两道都是采用快速排序算法进行解答的题目

6、【快排的应用】：编写算法，对 n 个关键字取整数值的记录序列进行整理，以使得所有关键字为负值的关键字排列在关键字为非负值的记录之前，要求：
（1）采用顺序存储结构，至多使用一个记录的辅助存储空间
（2）算法的时间复杂度为 O(n)

【算法思想】
此题目借助快速排序中子表划分的算法思想对表中的数据进行划分。附设两个指针 low 和 high，初始时分别指向表的上界和下界。

```
void process(int a[], int n)
{
    int low = 0, high = n-1;
    while(low < high)
    {
        while(low < high && a[low] < 0)
            ++low;
        while(low < high && a[high] > 0)
            --high;
        if(low < high)
        {
          int tmp = a[low];
          a[low] = a[high];
           a[high] = tmp;
          low++;
          high--;
        }
    }
}
```

【变式例题】设有一组初始记录关键字序列（K1，K2，…，Kn），要求设计一个算法能够在 O(n)的时间复杂度内将线性表划分成两部分，其中左半部分的每个关键字均小于 Ki，右半部分的每个关键字均大于等于 Ki。

```
void process(int a[], int n, int w)
{
    int low = 0, high = n-1, x = k[w];
   tmpe = k[i]
```

```
    while(low < high)
    {
        while(low < high && a[low] < x)
            ++low;
        while(low < high && a[high] > x)
            --high;
        if(low < high)
        {
            int tmp = a[low];
            a[low] = a[high];
            a[high] = tmp;
            low++;
            high--;
        }
    }
}
```

7、线性表（$a_1,a_2,a_3,\cdots,a_n$）中元素递增有序且按照顺序存储于计算机内。要求设计一个算法完成：
（1）用最少的时间在表中查找数值为 x 的元素。
（2）若查找到将其与后继元素位置交换。
（3）若找不到将其插入表中并使表中元素仍然递增有序。

【算法思想】
顺序存储的线性表递增有序，可以顺序查找，也可以折半查找。题目要求采用“最少的时间在表中查找数值为 x 的元素”，应使用折半查找，算法如下：

```
void SearchExchageInsert(ElemType a[], ElemType x)
{
    int low = 0, high = n-1;   //low 和 high 指向线性表下界和上界
    while(low <= high)
    {
        mid = (low + high)/2;
        if(a[mid] == x)              //查找到 x，退出循环
            break;
        else if(a[mid] < x)
            low = mid + 1;
        else
            high = mid-1;
    }
    if(a[mid] == x && mid != n)   //若最后一个元素与 x 相等，则不存在
    {    //将其与后继交换
```

```
            int tmp = a[mid];
            a[mid] = a[mid+1];
            a[mid+1] = tmp;
        }
        if(low > high)
        {
            for(i = n-1; i>high; i--)
                a[i+1] = a[i];          //后移元素
            a[i+1] = x;                 //插入 x
        }
}
```

抓机遇，码未来

DRAMA