数据结构复习-算法设计题1（201211）

2010 1、假设线性表L用带头结点的单链表存储，且至少有两个结点，每个结点数据域为整型值，设计算法判断该链表中每一个结点的值是否等于其后续两个结点的值之和。若满足上述要求，返回1并输出最大值，否则，返回0并输出最小值。

```
int  Linklist_sum_count.(LinkList L) {
    //L是数据域为整数的单链表的头指针
     p=L->next;  //p为链表的工作指针
     while (p->next->next )
       if (p->data== ( p->next->data + p->next->next->data))
           p=p->next;
       else return 0;
    return 1;
} //Linklist_sum_count
```

2009 (2) 设计算法将循环单链表中结点p的直接前驱删除的算法（p为指针，假设p的前驱存在）。

So easy!

```
void Delete_CircleList(LinkList p){
     //p是循环单链表的任一指针，删除p的直接前驱
    q=p->next;
    if(q->next==p){
        p->next=q->next;free q;// 删除p的直接前驱q
    }//if
    else {
        r=q->next;
        while(r->next!=p) {
           q=r;r=r->next;//查找p的前驱
        }
        q->next=p;free r; // 删除p的直接前驱r
} //Delete_CircleList
```

(3) 设L为单链表的头结点地址，其数据结点的数据都是正整数且无相同的，试设计利用直接插入的原则把该链表整理成数据递增的有序单链表的算法。

先断完连，再一个一个插。

```
void LinkListInsertSort(LinkList L){
 //L是带头结点的单链表，其数据域是正整数。
  if(L->next)//链表不为空表。
    { p=L->next->next;//p指向第一结点的后继。
      L->next->next=null;//从第二元素起依次插入。
      while(p)
      { r=p->next;//暂存p的后继。
        q=L->next;
        while(q!=p&&q->data<p->data)
            q=q->next;//查找插入位置。
        p->next=q->next;//将p结点链入链表。
        q->next=p;
        p=r;
}// LinkListInsertSort
```

4、已知单链表 L 是一个递增有序表，设计高效算法，删除表中值大于 min 且小于 max 的结点。

```
void Ddete_Between(Linklist &L,int min,int max){
//删除元素递增排列的链表 L 中值大于 min 且小于 max 的所有元素
  p=L;
  while(p->next->&&p->next->data<=min) p=p->next;
   //p 是最后一个不大于 min 的元素
  if(p->next) {  //如果还有比 min 更大的元素
    q=p->next;
    while(q&&q->data<max) {
        t=q; q=q->next; free t;
    }//while
    p->next=q; //q 是第一个不小于 max 的元素
  }//if
}//Delete_Between
```

5、设有一头指针为 L 的带有表头结点的非循环双向链表，其每个结点中除有 pred（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，在链表被起用前，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的最后，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。

```
DLinkList  locate(DLinkList L, ElemType x)
// L 是带头结点的按访问频度递减的双向链表，本算法先查找数据 x，查找成功时结点的访问频度域增 1，最后将该结点按频度递减插入链表中适当位置。
{ DLinkList p=L->next, q;      //p 为 L 表的工作指针，q 为 p 的前驱，用于查找插入位置。
  while (p && p->data !=x) p=p->next; // 查找值为 x 的结点。
  if (!p) {printf("不存在值为 x 的结点\n"); exit(0);}
  else { p->freq++;                    // 令元素值为 x 的结点的 freq 域加 1 。
         p->next->pred=p->pred;        // 将 p 结点从链表上摘下。
         p->pred->next=p->next;
         q=p->pred;                    // 以下查找 p 结点的插入位置
         while (q !=L && q->freq<p->freq) q=q->pred;
         p->next=q->next; q->next->pred=p; // 将 p 结点插入
         p->pred=q; q->next=p;
       }
  return p;     //返回值为 x 的结点的指针
}  //locate
```

2001 6、设有一个正整数序列组成的单链表（按递增次序有序，且允许有相等的整数存在），试写能实现下列功能的算法：（要求用最少的时间和最少的空间）

（1）确定在序列中比正整数大的数有几个（相同的数只计算一个，如（20，20，17，16，15，15，11，10 ，8，7，7，5，4））中比 10 大的数有 5 个）；

（2）在单链表将比正整数小的数 x 小的数将按递减次序排列；

（3）将正整数 x 大的偶数从单链表删除。

```
void  Exam_L (LinkedList L, int x)
    //L 是递增有序单链表，数据域为正整数。本算法确定比 x 大的数有几个；将比 x 小
    的数按递减排序，并将比 x 大的偶数从链表中删除。)
    {p=L->next; q=p;//p 为工作指针  q 指向最小值元素
     pre=L;            //pre 为 p 的前驱结点指针。
     k=0;              //计数（比 x 大的数）。
     L->next=null; //置空单链表表头结点。
     while (p && p->data<x)        //先解决比 x 小的数按递减次序排列
      {r=p->next;           //暂存后继
       p->next=L->next; //逆置
       L->next=p;
       p=r; //恢复当前指针。退出循环时，r 指向值>=x 的结点。
      }
     q->next=p; pre=q;    //pre 指向结点的前驱结点
        while (p->data==x) {pre=p; p=p->next;} //从小于 x 到大于 x 可能经过等于 x
          while (p)                 //以下结点的数据域的值均大于 x
            {k++; x=p->data;        //下面仍用 x 表示数据域的值，计数
             if (x % 2==0)          //删偶数
                 {while (p->data==x)
                 {u=p; p=p->next; free u; }
                  pre->next=p;   //拉上链
                  }
              else                  //处理奇数
                 while (p->data==x)//相同数只记一次
                 {pre->next=p; pre=p; p=p->next; }
            }//while(p)
          printf ("比值%d 大的数有%d 个\n", x, k);
}//Exam_L
```

(7) 线性表(a1,a2,a3...an)中元素递增有序且按顺序存于计算机内。要求设计一算法完成：

(1) 用最少的时间在表中查找数值为的元素。

(2) 若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递增有序。

```
void  SearchExchangeInsert (ElemType a[]; ElemType x)
    //a 是具有 n 个元素的递增有序线性表，顺序存储。
    { low=0; high=n-1; //low 和 high 指向线性表下界和上界的下标
      while (low<=high)
        { mid= (low+high) /2;              //找中间位置
          if (a[mid]==x)  break;           //找到 x，退出 while 循环。
          else if  (a[mid] <x)  low=mid+1; //到中点 mid 的右半去查。
              else high=mid-1;       //到中点 mid 的左部去查。
          }
         if (a[mid]==x && mid!=n)
            // 若最后一个元素与 x 相等，则不存在与其后继交换的操作。
           {t=a[mid]; a[mid]=a[mid+1]; a[mid+1]=t; }
```

```
                    // 数值 x 与其后继元素位置交换。
                if (low>high)       //查找失败，插入数据元素 x
                  {for (i=n-1; i>high; i--)  a[i+1]=a[i]; //后移元素。
                   a[i+1]=x; //插入 x。
                  } //结束插入
        }//SearchExchangeInsert
```

2009 (8) 设有一个由正整数组成的无序（向后）单链表，编写完成下列功能的算法：

（1）找出最小值结点，且打印该数值；

（2）若该数值是奇数，则将其与直接后继结点的值交换；

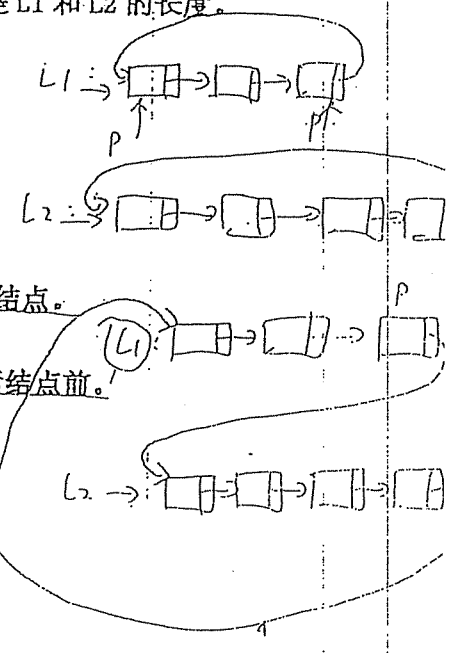
（3）若该数值是偶数，则将其直接后继结点删除。

So easy!

```
void  MiniValue (LinkedList la)
//la 是数据域为正整数且无序的单链表，本算法查找最小值结点且打印。若最小值结点的数值是奇数，则与后继结点值交换；否则，就删除其直接后继结点。
    {p=la->next;      //设 la 是头结点的头指针，p 为工作指针。
     pre=p;           //pre 指向最小值结点，初始假定首元结点值最小。
     while (p->next!=null)  //p->next 是待比较的当前结点。
       {if (p->next->data<pre->data) pre=p->next;
        p=p->next;  //后移指针
       }
     printf ("最小值=%d\n", pre->data);
     if (pre->data%2!=0)     //处理奇数
       if (pre->next!=null) //若该结点没有后继，则不必交换
        {t= pre->data; pre->data=pre->next->data;
           pre->next->data=t; }//交换完毕
     else//处理偶数情况
       if (pre->next!=null) //若最小值结点是最后一个结点，则无后继
          {u=pre->next; pre->next=u->next; free (u); } //释放后继结点空间
```

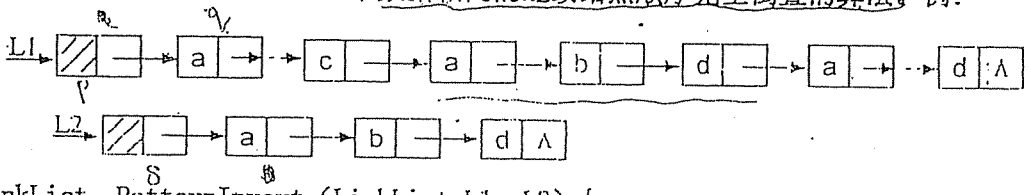
(9) 已知 L1、L2 分别为两循环单链表的头结点指针，m,n 分别为 L1、L2 表中数据结点个数。要求设计一算法，用最快速度将两表合并成一个带头结点的循环单链表。

```
LinkList Union_L(LinkList L1,L2;int m,n){
   //L1 和 L2 分别是两循环单链表的头结点的指针，m 和 n 分别是 L1 和 L2 的长度。
  if(m<0||n<0){
     printf("表长输入错误\n"); exit(0);
  }
  if(m<n){  //若 m<n，则查 L1 循环单链表的最后一个结点。
     if(m==0) return L2;//L1 为空表。
     else{ p=L1;
           while(p->next!=L1) p=p->next;//查最后一个元素结点。
           p->next=L2->next;
           //将 L1 循环单链表的元素结点插入到 L2 的第一元素结点前。
           L2->next=L1->next;
           free L1;//释放无用头结点。
     }// else
  }//if(m<n)处理完 m<n 情况
```



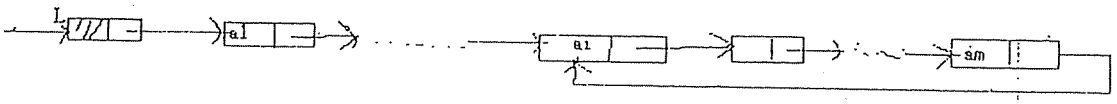
```
    else{// 下面处理 L2 长度小于等于 L1 的情况
        if(n==0)return L1;//L2 为空表。
        else{ p=L2;
              while(p->next!=L2) p=p->next;//查最后元素结点。
              p->next=L1->next;
               //将 L2 的元素结点插入到 L1 循环单链表的第一元素结点前。
              L1->next=L2->next;
              free L2;//释放无用头结点。
        }// else
    }// else 处理完 m>=n 情况
}//Union_L
```

(10) L 1 与 L 2 分别为两单链表头结点，地址指针，且两表中数据结点的数据域均为一个字母。设计把 L 1 中与 L 2 中数据相同的连续结点顺序完全倒置的算法。例：




```
LinkList  PatternInvert (LinkList L1, L2) {
//L1 和 L2 均是带头结点的单链表，数据结点的数据域均为一个字符。
 p=L1;                  //p 是每趟匹配时 L1 中的起始结点前驱的指针。
 q=L1->next;            //q 是 L1 中的工作指针。
 s=L2->next;            //s 是 L2 中的工作指针。
 while (p && s)
    if (q->data==s->data) {q=q->next;s=s->next;} //对应字母相等，指针后移。
    else {p=p->next; q=p->next; s=L2->next; }
          //失配时，L1 起始结点后移，L2 从首结点开始。
 if (s==null) {//匹配成功，这时 p 为 L1 中与 L2 中首字母结点相同数据域结点的前驱，
                //q 为 L1 中与 L2 最后一个结点相同数据域结点的后继。
     r=p->next;             //r 为 L1 的工作指针，初始指向匹配的首字母结点。
     p->next=q;             //将 p 与 q 结点的链接。
     while (r!=q) {         //逐结点倒置。
         s=r->next;         //暂存 r 的后继。
         r->next=p->next;   //将 r 所指结点倒置。
         p->next=r;
         r=s;               //恢复 r 为当前结点。
      }//while
 }//if
 else printf ("L2 并未在 L1 中出现");
} //PatternInvert
```

(11) 已知 L 为链表的头结地址，表中共有 m(m>3)个结点，从表中第 i 个结点(1<i<m)起到第 m 个结点构成一个循环部分链表，设计将这部分循环链表所有结点顺序完全倒置的算法。



```
LinkList PatternInvert1 (LinkList L, int  i, m)
  //L 是有 m 个结点的链表的头结点的指针
    {if (i<1|| i>=m || m<4)
    {printf ("%d,%d 参数错误\n" , i, m); exit (0); }
     p=L->next->next;          //p 是工作指针，初始指向第二结点（已假定 i>1)。
     pre=L->next;              //pre 是前驱结点指针，最终指向第 i-1 个结点。
     j=1;                      //计数器
     while (j<i-1)             //查找第 i 个结点。
       {j++; pre=p; p=p->next; } //查找结束，p 指向第 i 个结点。
     q=p;                      //暂存第 i 个结点的指针。
     p=p->next;                //p 指向第 i+1 个结点，准备逆置。
     j+=2;          //上面 while 循环结束时，j=i-1，现从第 i+1 结点开始逆置。
     while (j<=m) {
         r=p->next;            //暂存 p 的后继结点。
         p->next=pre->next;    //逆置 p 结点。
         pre->next=p;
         p=r;                  //p 恢复为当前待逆置结点。
         j++;                  //计数器增 1。
     } //while
     q->next=pre->next;  //将原第 i 个结点的后继指针指向原第 m 个结点。
} //PatternInvert1
```

2002

12. 已知 L 为没有头结点的的单链表中第一个结点的指针，每个结点数据域存放一个字符，该字符可能是英文字母字符或数字字符或其他字符，编写算法构造三个以带头结点的单循环链表表示的线性表，使每个表中只含同一类字符。（要求用最少的时间和最少的空间）

```
void  OneToThree (LinkList L, la, ld, lo)
  //L 是无头结点的单链表第一个结点的指针，链表中的数据域存放字符。
  {la= (LinkedList) malloc (sizeof (LNode));  //建立三个链表的头结点
   ld= (LinkedList) malloc (sizeof (LNode));
   lo= (LinkedList) malloc (sizeof (LNode));
   la->next=la; ld->next=ld; lo->next=lo; //置三个循环链表为空表
   while (L) {//分解原链表。
     r=L; L=L->next;  //L 指向待处理结点的后继
     if (r->data>= 'a' && r->data<= 'z' || r->data>= 'A' && r->data<= 'Z')
       {r->next=la->next;  la->next=r; }          //处理字母字符。
      else if (r->data>= '0' && r->data<= '9')
          {r->next=ld->next; ld->next=r; }        //处理数字字符
       else {r->next=lo->next; lo->next=r; }      //处理其它符号。
     } //结束 while (L!=null)。
} //OneToThree
```

13. 已知线性表（a1 a2 a3 ...an）按顺序存于内存，每个元素都是整数，试设计用最少时间把所有值为负数的元素移到全部正数元素前边的算法：

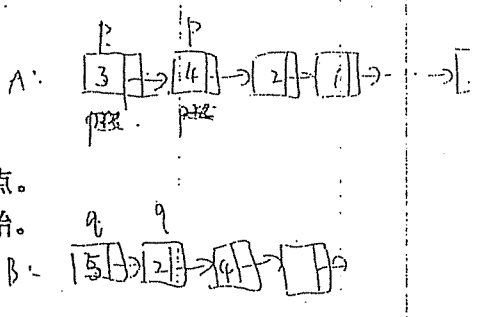
例：（x, -x, -x, x, x, -x ...-x）变为（-x, -x, -x...x x x）

```
int Rearrange (SeqList a; int n)
```

```
//a 是具有 n 个元素的线性表，以顺序存储结构存储，线性表的元素是整数。
{i=0; j=n-1; // i,j 为工作指针（下标），初始指向线性表 a 的第 1 个和第 n 个元素。
 t=a[0];      //暂存枢轴元素。
 while(i<j){
    while(i<j && a[j]>=0) j--;  // 若当前元素为大于等于零，则指针前移。
     if(i<j){a[i]=a[j];i++;}    //  将负数前移。
     while(i<j &&a[i]<0)i++;    // 当前元素为负数时指针后移。
     if(i<j) a[j--]=a[i];       // 正数后移。
 }
 a[i]=t;    //将原第一元素放到最终位置。
}// Rearrange
```

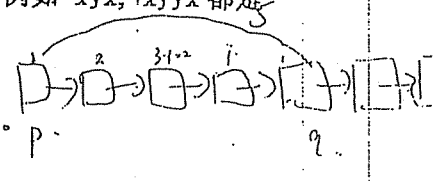
(14) 两个正数序列 A=a1,a2,a3,.....am 和 B=b1,b2,b3,...bn 已经存入两个单链表中，设计一个算法，判别序列 B 是否是序列 A 的子序列。

```
int  Pattern2 (LinkList A, B) {
  //A 和 B 分别是数据域为整数的单链表，本算法判断 B 是否是 A 的子序列。
  //如是，返回 1；否则，返回 0 表示失败。
   p=A;     //p 为 A 链表的工作指针，本题假定 A 和 B 均无头结点。
   pre=p;   //pre 记住每趟比较中 A 链表的开始结点。
   q=B;     //q 是 B 链表的工作指针。
   while (p && q)
     if (p->data==q->data)  {p=p->next; q=q->next; }
     else{pre=pre->next; p=pre;  //A 链表新的开始比较结点。
          q=B; }                 //q 从 B 链表第一结点开始。
    if (q==null) return 1;       //B 是 A 的子序列。
    else   return 0;             //B 不是 A 的子序列。
} //Pattern2
```



2006 (15) 设单链表的表头指针为 h，结点结构由 data 和 next 两个域构成，其中 data 域为字符型。写出算法 dc(h,n)，判断该链表的前 n 个字符是否中心对称。例如 xyx，xyyx 都是中心对称。

```
int  dc_L (LinkList  h, int n)
  // h 是带头结点的 n 个元素单链表，链表中结点的数据域是字符。
  { char s[]; int i=1; //i 记结点个数， s 字符栈
   p=h->next; //p 是链表的工作指针，指向待处理的当前元素。
   for (i=1; i<=n/2; i++) // 链表前一半元素进栈。
    {s[i]=p->data; p=p->next; }
   i--; //恢复最后的 i 值
   if (n%2==1) p=p->next; }  //若 n 是奇数，后移过中心结点。
   while (p && s[i]==p->data)
   {i--; p=p->next; }    //测试是否中心对称。
   if (p==null) return 1; //链表中心对称
   else  return 0;        //链表不中心对称
 }//dc_L
```



① 判断两棵树是否相似

```
int Bitree_Sim(Bitree B1,Bitree B2){ //判断两棵树是否相似的递归算法
    if(!B1&&!B2) return 1;
    else
   if(B1&&B2&&Bitree_Sim(B1->lchild,B2->lchild)&&Bitree_Sim(B1->rchild,B2->rchild))
     return 1;
    else return 0;
}//Bitree_Sim
```

1代表相似. 0代表不相似

2009 ② 设二叉树为二叉链表为存储结构，设计算法将二叉树中各结点的左右孩子位置交换。

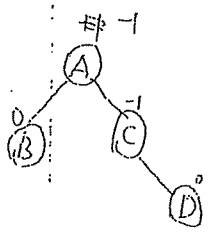
```
void Bitree_Revolute(Bitree T){ //交换所有结点的左右子树
    p=T->lchild; T->rchild= T->lchild; T->rchild=p;   //交换左右子树
    if(T->lchild) Bitree_Revolute(T->lchild);
    if(T->rchild) Bitree_Revolute(T->rchild); //左右子树再分别交换各自的左右子树
}//Bitree_Revolute
```

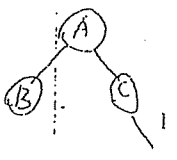
2005 3-1. 求二叉树中以值为 x 的结点为根的子树深度

```
int Get_Sub_Depth(Bitree T,int x){
 //求二叉树中以值为 x 的结点为根的子树深度
 if(T->data==x) {
  printf("%d\n",Get_Depth(T)); //找到了值为 x 的结点,求其深度
  exit 1;
 }
 else {
  if(T->lchild) Get_Sub_Depth(T->lchild,x);
  if(T->rchild) Get_Sub_Depth(T->rchild,x); //在左右子树中继续寻找
 }
}//Get_Sub_Depth
int Get_Depth(Bitree T){//求子树深度的递归算法
 if(!T) return 0;
 else {
  m=Get_Depth(T->lchild);
  n=Get_Depth(T->rchild);
  return (m>n?m:n)+1;
 }
}//Get_Depth
```

2007 3-2. 假设一棵平衡二叉树的每个结点都标明了平衡因子 bf，试设计一个算法，求平衡二叉树的高度。

```
int Height_Bf (BSTree T)
    // 求平衡二叉树 T 的高度
    {level=0; p=T;
     while (p) {
        level++;  // 树的高度增 1
        if (p->bf<0) p=p->rchild; //bf=-1 沿右分枝向下
        else p=p->lchild;         //bf>=0 沿左分枝向下
     }//while
```





```
    return level; //平衡二叉树的高度
  } // Height_Bf
```

2006 (3-3) 设二叉树结点结构为平衡二叉树结点结构，设计一递归算法计算并填写二叉树中每个结点的平衡因子，同时返回二叉树中不平衡结点的个数。

```
int Get_BF(Bitree T,int &count){ //求各结点的平衡因子
 if(T) {
    Get_BF(T->lchild);//遍历左子树
    Get_BF(T->rchild);//遍历右子树
    m=Get_Depth(T->lchild);
    n=Get_Depth(T->rchild);
    T->bf=m-n;
    if (T->bf>1|| T->bf<-1) count++;
 }//if
}//Get_BF
```

2009 4、已知一个二叉树的先序序列和中序序列分别存储与两个一维数组中，设计算法建立二叉树的二叉链表结构。

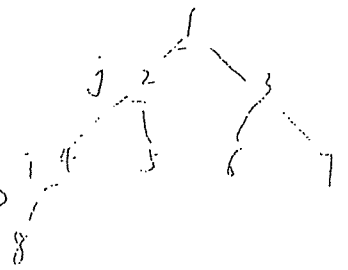
```
BiTree IntoPost((ElemType in[],post[],int l1,h1,l2,h2)
    //l1,h1,l2,h2 分别是两序列第一和最后结点的下标
     T=(BiTree)malloc(sizeof(BiNode));//申请结点
     T->data=post[h2];//后序遍历序列最后一个元素是根结点数据
     for(i=l1;i<=h1;i++)
        if(in[i]==post[h2])break;//在中序序列中查找根结点
     if(i==l1) T->lchild=null;  //处理左子树
        else T->lchild=IntoPost(in,post,l1,i-1,l2,l2+i-l1-1);
     if(i==h1) T->rchild=null;  //处理右子树
        else T->rchild=IntoPost(in,post,i+1,h1,l2+i-l1,h2-1);
     return T;
}//IntoPost
```

(5) 已知二叉树的顺序存储结构建立二叉链表结构

```
bool CreateBitree_SqList(Bitree &T,SqList sa){ //根据顺序存储结构建立二叉链表
   Bitree ptr[sa.last+1]; //该数组储存与 sa 中各结点对应的树指针
   if(!sa.last) {
    T=NULL; //空树
    return true;
   }
   ptr[1]=(BTNode*)malloc(sizeof(BTNode)); } 申请结点+赋值.
   ptr[1]->data=sa.elem[1]; //建立树根
   T=ptr[1]; 与前驱结点.
   for(i=2;i<=sa.last;i++) {
    if(!sa.elem[i]) return false; //顺序错误
    ptr[i]=(BTNode*)malloc(sizeof(BTNode));
    ptr[i]->data=sa.elem[i];
    j=i/2; //找到结点 i 的双亲 j
    if(i-j*2) ptr[j]->rchild=ptr[i]; //i 是 j 的右孩子
```

依次申请结点.
依次赋值.

的在 for 循环里.



```
        else ptr[j]->lchild=ptr[i]; //i 是 j 的左孩子
    }
    return true;
}//CreateBitree_SqList
```

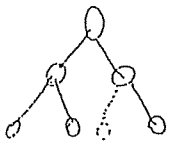
⑥ 二叉树层次遍历

```
void LayerOrder(Bitree T){//层次遍历二叉树
    InitQueue(Q); //建立工作队列
    EnQueue(Q,T);
    while(!QueueEmpty(Q)) {
     DeQueue(Q,p);
     visit(p);
     if(p->lchild) EnQueue(Q,p->lchild);
     if(p->rchild) EnQueue(Q,p->rchild);
    }
}//LayerOrder
```

⑦ 判断二叉树是否完全二叉树

（1）应用队列

```
int IsFull_Bitree(Bitree T){ //判断二叉树是否完全二叉树,是则返回 1,否则返回 0
    InitQueue(Q);
    flag=0;
    EnQueue(Q,T); //建立工作队列
    while(!QueueEmpty(Q)) {
     DeQueue(Q,p);
     if(!p) flag=1;
     else if(flag) return 0;
     else {
      EnQueue(Q,p->lchild);
      EnQueue(Q,p->rchild); //不管孩子是否为空,都入队列
     }
    }//while
    return 1;
}//IsFull_Bitree
```



到此时，if(!p)

flag=1，置为1，让其停止，

返回0，到此以前均为完全二叉

如果不是这种循环停止情况，

那就继续往下走，始返回1代表

完全二叉

（2）应用层次遍历

```
int JudgeComplete(BiTree T) {
    //判断二叉树是否是完全二叉树,如是，返回 1，否则，返回 0
    int tag=0; p=T;
    if(!p) return 1;
      InitQueue(Q); EnQueue(Q, p);  //初始化队列，根结点指针入队
        while (!QueueEmpty(Q)){
          DeQueue(Q, p);  //出队
          if (p->lchild && !tag) EnQueue(Q, p->lchild);  //左子女入队
          else {
            if (p->lchild) return 0; //前边已有结点为空，本结点不空
            else tag=1;              //首次出现结点为空
```

```
            if (p->rchild && !tag) EnQueue(Q, p->rchild);  //右子女入队
            else if (p->rchild) return 0;
                 else tag=1;
          } //while
       return 1;
} //JudgeComplete
```

2011 8、设二叉树以二叉链表为存储结构，设计算法通过一次遍历求二叉树宽度并输出这一层上的所有的叶子结点的算法。所谓宽度是指二叉树的各层上，具有结点数最多的那一层上的结点总数。（二叉树采用二叉链表为存储结构，设计算法求出二叉树中第 i 层和第 i+1 叶子结点个数之和。）

（1）使用数组

```
typedef struct{
        BTNode node;
        int layer;
} BTNRecord; //包含结点所在层次的记录类型
int Width1(Bitree T){ //求一棵二叉树的宽度
    int countd; //count 数组存放每一层的结点数
    InitQueue(Q); //Q 的元素为 BTNRecord 类型
    EnQueue(Q,{T,0});
    while(!QueueEmpty(Q)) {
     DeQueue(Q,r);
     count[r.layer]++;
     if(r.node->lchild) EnQueue(Q,{r.node->lchild,r.layer+1});
     if(r.node->rchild) EnQueue(Q,{r.node->rchild,r.layer+1});
    } //利用层序遍历来统计各层的结点数
    for(maxn=count[0],i=1;count[i];i++)
       if(count[i]>maxn) maxn=count[i]; //求层最大结点数
    return maxn;
}// Width1
```

（2）使用变量

```
int Width2(BiTree T) {//求二叉树 bt 的最大宽度
   if (!T) return 0;  //空二叉树宽度为 0
   else {
     BiTree Q[];//Q 是队列，元素为二叉树结点指针，容量足够大
     front=1;rear=1;last=1;
    //front 队头指针,rear 队尾指针,last 同层最右结点在队列中的位置
     temp=0; maxw=0;     //temp 记局部宽度, maxw 记最大宽度
     Q[rear]=T;          //根结点入队列
     while(front<=last) {
        p=Q[front++]; temp++; //同层元素数加 1
        if (p->lchild)  Q[++rear]=p->lchild;   //左子女入队
          if (p->rchild)  Q[++rear]=p->rchild;   //右子女入队
        if (front>last) {     //一层结束,
            last=rear;
```

```
            if(temp>maxw) maxw=temp;
            //last 指向下层最右元素，更新当前最大宽度
          temp=0;
          }//if
        }//while
        return (maxw);
    }//width2
```

2010 (9) 假设一个仅包含二元运算符的算术表达式以二叉链表形式存放于二叉树 T 中，设计算法后序遍历计算表达式的值。

```
float PostEval(BiTree T) {// 以后序遍历算法求以二叉树表示的算术表达式的值
  float lv,rv;
  if(T){
    lv=PostEval(T->lchild); // 求左子树表示的子表达式的值
    rv=PostEval(T->rchild); // 求右子树表示的子表达式的值
    switch(T->optr){
        case '+' : value=lv+rv; break;
        case '-' : value=lv-rv;break;
        case '*' : value=lv*rv;break;
        case '/' : value=lv/rv;
    }// switch
  }//if
  return(value);
}// PostEval
```

(10) 设计算法利用叶子结点中的空指针域将所有叶子结点链接为一个带有头结点的双链表，算法返回头结点的地址。

```
void CreatLeafList(BiTree T){
  //将 BiTree 树中所有叶子结点链成带头结点的双链表，
  if (T) {
     CreatLeafList (T->lchild) ;  //中序遍历左子树
     if(!T->lchild && !T->rchild)  //叶子结点
        if(!L){//第一个叶子结点
           L=(BiTree)malloc(sizeof(BiNode)); //生成头结点
           L->lchild=null; L->rchild=T;
           //头结点的左链为空,右链指向第一个叶子结点
           T->lchild=L; pre=T;
           //第一个叶子结点左链指向头结点，pre 指向当前叶子结点
        }//if
        else {   //已不是第一个叶子结点
           pre->rchild=T; T->lchild=pre; pre=T;
        } //当前叶子结点链入双链表
        CreatLeafList(T->rchild);  //中序遍历右子树
        pre->rchild=null;     //最后一个叶子结点的右链置空（链表结束标记）
  }//if(T)
}// CreatLeafList
```

2003 (11) 给出中序线索二叉树的结点结构，试编写在不使用栈和递归的情况下先序遍历中序线索二叉树的算法。

```
void Preorder_InThreat(BiThrTree T)
    //按先序 遍历带头结点的中序线索二叉树 T
   {p=T->lchild;//设 p 指向二叉树的根结点
    while(p)
      {while(p->ltag==0) {printf(p->data ); p=p-> lchild ; //遍左
       printf(p->data ); //准备右转
       while(p->rtag==1 && p->rchild!=T) p=p->rchild; //回溯
       if(p->rchild!=T ) p=p->rchild; //遍右
      }// while(p)
   }//Preorder_In
```

(12) 非递归不用栈中序遍历带有双亲指针的三叉链表的二叉树

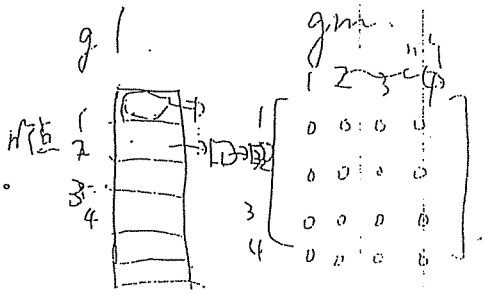
```
typedef struct {
    int data;
    PBTNode *lchild;
    PBTNode *rchild;
    PBTNode *parent;
} PBTNode,PBitree; //有双亲指针域的二叉树结点类型
void Inorder_Nonrecursive(PBitree T){ //不设栈非递归遍历有双亲指针的二叉树
  p=T;
  while(p->lchild) p=p->lchild; //向左走到尽头
  while(p) {
    visit(p);
    if(p->rchild){ //寻找中序后继:当有右子树时
      p=p->rchild;
      while(p->lchild) p=p->lchild; //后继就是在右子树中向左走到尽头
    }
    else if(p->parent->lchild==p) p=p->parent; //当自己是双亲的左孩子时后继就是双亲
    else {
      p=p->parent;
      while(p->parent&&p->parent->rchild==p) p=p->parent;
      p=p->parent;
    } //当自己是双亲的右孩子时后继就是向上返回直到遇到自己是在其左子树中的祖先
  }//while
}//Inorder_Nonrecursive
```

2002 2007 (13) 设有向 G 图有 n 个顶点，e 条边，设计算法根据其邻接表生成其逆邻接表，要求算法复杂性为 O(n+e)。

```
void InvertAdjList(AdjList gin, gout)
    //将有向图的出度邻接表改为按入度建立的逆邻接表
   {for (i=1;i<=n;i++)//设有向图有 n 个顶点，建逆邻接表的顶点向量。
     {gin[i].vertex=gout[i].vertex; gin.firstarc=null; }
    for (i=1;i<=n;i++)    //邻接表转为逆邻接表。
     {p=gout[i].firstarc;//取指向邻接表的指针。
```

```
        while (p!=null)
             { j=p->adjvex;
               s=(ArcNode *)malloc(sizeof(ArcNode));//申请结点空间。
               s->adjvex=i; s->next=gin[j].firstarc; gin[j].firstarc=s;
               p=p->next;//下一个邻接点。
             }//while
      }//for
}// InvertAdjList
```

(14) 写出从图的邻接表表示转换成邻接矩阵表示的算法。



```
void  AdjListToAdjMatrix(AdjList gl, AdjMatrix gm)
      //将图的邻接表表示转换为邻接矩阵表示。
     {for (i=1;i<=n;i++)  //设图有 n 个顶点，邻接矩阵初始化。
        for (j=1;j<=n;j++) gm[i][j]=0;
      for (i=1;i<=n;i++)
       {p=gl[i].firstarc;  //取第一个邻接点。
        while (p!=null) {gm[i][p->adjvex]=1;p=p->next; }//下一个邻接点
        }//for
}// AdjListToAdjMatrix
```

2005 (15) 试写一算法，判断以邻接表方式存储的有向图中是否存在由顶点 V_i 到顶点 V_j 的路径 ($i<>j$)。

设一全程变量 flag。初始化为 0，若有通路，则 flag=1。

```
int visited[]=0;  //全局变量，访问数组初始化
int  dfs_path(AdjList g , vi)
    //以邻接表为存储结构的有向图 g，判断顶点 Vi 到 Vj 是否有通路
   { visited[vi]=1;   //visited 是访问数组，设顶点的信息就是顶点编号。
    p=g[vi].firstarc;  //第一个邻接点。
    while ( p!=null)
       { j=p->adjvex;
        if (vj==j) { flag=1; return (1) ;} //vi 和 vj 有通路。
        if (visited[j]==0) dfs(g, j);
        p=p->next;  }//while
     if (!flag) return(0);
  }// dfs_path
```

2001 (16) 再有向图 g 中，如果 r 到 g 中的每个节点都有路径可达，则称结点 r 为 g 的根结点，编写一个算法完成下列功能：

（1）建立有向图的邻接表存储结构；

（2）判断有向图 g 是否有根，若有，则打印出所有的根结点的值。

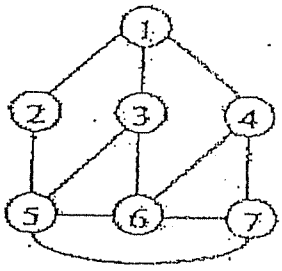
```
int  num=0,  visited[]=0  //num 记访问顶点个数，访问数组 visited 初始化。
const  n=用户定义的顶点数;
AdjList g ;                //用邻接表作存储结构的有向图 g。
void  dfs(v)
 {visited [v]=1;  num++; //访问的顶点数+1
  if (num==n) {printf("%d 是有向图的根。\n",v); num=0;}//if
  p=g[v].firstarc;
```

```
    while (p)
      {if (visied[p->adjvex]==0) dfs (p->adjvex);
       p=p->next;} //while
    visited[v]=0; num--; //恢复顶点 v
   }//dfs
void  JudgeRoot()
    //判断有向图是否有根，有根则输出之。
 {static int i ;
  for (i=1;i<=n;i++ ) //从每个顶点出发，调用 dfs()各一次。
   {num=0; visited[1..n]=0; dfs(i); }
 }// JudgeRoot
```

(17) 图的 D_搜索类似与 BFS,不同之处在于使用栈代替 BFS 中的队列，关出队列的操作改为入出栈的操作，即当一个顶点的所有邻接点被搜索之后,下一个搜索出发点应该是最近入栈(栈顶)的顶点。

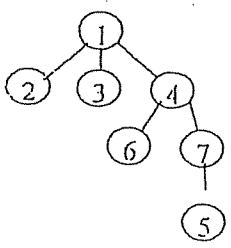
(1). 用邻接表做存储结构,写一个 D_搜索算法;

(2). 用 D_搜索方法的访问次序和相应的生成树，当从某顶点出发搜索它的邻接点，请按邻接点序号递增序搜索，以使答案唯一。




```
void D_BFS(AdjList g ,vertype v0)
    // 从 v0 顶点开始，对以邻接表为存储结构的图 g 进行 D_搜索。
    { int s[], top=0;      //栈，栈中元素为顶点，仍假定顶点用编号表示。
      for (i=1,i<=n;i++) visited[i]=0;  //图有 n 个顶点，visited 数组为全局变量。
      for (i=1,i<=n;i++)  //对 n 个顶点的图 g 进行 D_搜索。
        if (visited[i]==0)
          {s[++top]=i; visited[i]=1; printf( "%3d",i);
           while (top>0)
              {i=s[top--]; //退栈
               p=g[i].firstarc; //取第一个邻接点
               while (p!=null)  //处理顶点的所有邻接点
                 {j=p->adjvex;
                  if (visited[j]==0) //未访问的邻接点访问并入栈。
                    {visited[j]=1; printf( "%3d",i);s[++top]=j;}
                  p=p->next;
                 } //下一个邻接点
               }//while(top>0)
          } //if
    }//D_BFS
```

(2) D 搜索序列：1234675，生成树如图：



2000 18、假设一个有向图 G 已经以十字链表形式存储在内存中，试写一个判断该有向图中是否有环（回路）的算法。

```
int TopsorOrthList  g)
   //判断以十字链表为存储结构的有向图 g 是否存在环路
  {int top=0; //用作栈顶指针
   for (i=1;i<=n;i++)
    //求各顶点的入度。设有向图 g 有 n 个顶点，初始时入度域均为 0
    {p=g[i].firstin;      //设顶点信息就是顶点编号，否则，要进行顶点定位
     while(p)
       {g[i].indegree++; //入度域增 1
        if (p->headvex==i) p=p->headlink; else p=p->taillink;
         //找顶点 i 的邻接点
       }//while(p)  }//for
   for (i=1;i<=n;i++)        //建立入度为 0 的顶点的栈
      if (g[i].indegree==0) {g[i].indegree=top; top=i; }
   m=0;  //m 为计数器，记输出顶点个数
   while (top<>0)
     {i=top; top=g[top].indegree; m++;      //top 指向下一入度为 0 的顶点
      p=g[i].firstout;
      while (p) //处理顶点 i 的各邻接点的入度
       {if (p->tailvex==i) k=p->headvex; else k=p->tailvex;}
           //找顶点 i 的邻接点
        g[k].indegree--; //邻接点入度减 1
        if (g[k].indegree==0) {g[k].indegree=top; top=k; }
          //入度为 0 的顶点再入栈
        if (p->headvex==i) p=p->headlink; else p=p->taillink;
          //找顶点 i 的下一邻接点
       }//while (p)
     }// while (top<>0)
   if (m<n) retun(1);  //有向图存在环路
     else return(0);   //有向图无环路
  }// Topsor
```

2008 19、无向图 G 采邻接矩阵存储，试设计算法删除从顶点 v 到顶点 w 之间的一条边。

```
Status Delete_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上删除边(v,w)
{
 if((i=LocateVex(G,v))<0) return ERROR;
 if((j=LocateVex(G,w))<0) return ERROR;
 if(G.arcs[i][j].adj)
 {
  G.arcs[i][j].adj=0;
  G.arcnum--;
 }
 return OK;
}//Delete_Arc
```

(20) 求邻接表方式存储的有向图 G 的顶点 i 到 j 之间长度为 len 的简单路径条数

```
int GetPathNum_Len(ALGraph G,int i,int j,int len)
//求邻接表方式存储的有向图 G 的顶点 i 到 j 之间长度为 len 的简单路径条数
{
 if(i==j&&len==0) return 1; //找到了一条路径,且长度符合要求
 else if(len>0)
 {
  sum=0; //sum 表示通过本结点的路径数
  visited[i]=1;
  for(p=G.vertices[i].firstarc;p;p=p->nextarc)
  {
   w=p->adjvex;
   if(!visited[w])
    sum+=GetPathNum_Len(G,w,j,len-1)//剩余路径长度减一
  }//for
  visited[i]=0; //本题允许曾经被访问过的结点出现在另一条路径中
 }//else
 return sum;
}//GetPathNum_Len
```

(21) 对于一个使用邻接表存储的有向图 G，可以利用深度优先遍历方法，对该图中结点进行拓扑排序。其基本思想是：在遍历过程中，每访问一个顶点，就将其邻接到的顶点的入度减一，并对其未访问的、入度为 0 的邻接到的顶点进行递归。

```
int visited[]=0; finished[]=0; flag=1;   //flag 测试拓扑排序是否成功
    ArcNode *final=null;     //final 是指向顶点链表的指针，初始化为 0
    void  dfs(AdjList g,vertype v)
      //以顶点 v 开始深度优先遍历有向图 g，顶点信息就是顶点编号.
      {ArcNode *t;  //指向边结点的临时变量
       printf("%d",v); visited[v]=1; p=g[v].firstarc;
       while(p)
         {j=p->adjvex;
          if (visited[j]==1 && finished[j]==0) flag=0 //dfs 结束前出现回边
            else if(visited[j]==0) {dfs(g,j); finished[j]=1;} //if
          p=p->next;
         }//while
       t=(ArcNode *)malloc(sizeof(ArcNode));//申请边结点
       t->adjvex=v; t->next=final; final=t;     //将该顶点插入链表
      } //dfs 结束
    int dfs-Topsort(Adjlist g)
      //对以邻接表为存储结构的有向图进行拓扑排序，拓扑排序成功返回 1，否则返回 0
      {i=1;
       while (flag && i <=n)
         if (visited[i]==0) {dfs(g,i);  finished[i]=1; }//if
       return(flag);
    }// dfs-Topsort
```