		Description
5	树与二叉树	树的基本概念 二叉树的概念 二叉树的遍历和线索二叉树 树、森林 树与二叉树的应用

5. 已知一棵二叉树按顺序存储结构进行存储，设计一个算法，求编号分别为i和j的两个结点的最近的公共祖先结点的值。

   首先，必须明确二叉树中任意两个结点必然存在最近的公共祖先结点，最坏的情况下是根结点（两个结点分别在根结点的左右分支中），而且从最近的公共祖先结点到根结点的全部祖先结点都是公共的。由二叉树顺序存储的性质可知，任意一个结点i的双亲结点的编号为i/2。求解i和j最近公共祖先结点的算法步骤如下（设从数组下标1开始存储）：

   - 若i>j，则结点i所在层次大于或等于结点j所在层次。结点i的双亲结点为结点i/2，若i/2=j，则结点i/2是原结点i和结点j的最近公共祖先结点，若i/2!=j，则令i=i/2，即以该结点i的双亲结点为起点，采用递归的方法继续查找。
   - 若j>i，则结点j所在层次大于或等于结点i所在层次，结点j的双亲结点为结点j/2，若j/2=i，则结点j/2是原结点i和结点j的最近公共祖先结点，若j/2!=i，则令j=j/2。

   重复上述过程，直到找到它们最近的公共祖先结点为止。

```
ElemType Comm_Ancestor(SqTree T, int i, int j)
{
    //本算法在二叉树中查找结点i和结点j的最近公共祖先结点
    if(T[i]!='#' && T[j]!='#')//结点存在
    {
        while(i!=j)//两个编号不同时循环
        {
            if(i>j)
                i=i/2;//向上找i的祖先
            else
                j=j/2;//向上找j的祖先
        }
        return T[i];
    }
}
```

3. 编写后序遍历二叉树的非递归算法。

   后序非递归遍历二叉树先访问左子树、再访问右子树，最后访问根结点。

   第一步：沿着根的左孩子，依次入栈，直到左孩子为空。此时栈内元素为ABD。

   第二部：读栈顶元素：若其右孩子不空且未被访问过，将右子树转执行第一步；否则栈顶元素出栈并访问。

   栈顶D的右孩子为空，出栈并访问，它是后序序列的第一个结点；栈顶B的右孩子不空且未被访问过，E入栈，栈顶E的左右孩子均为空，出栈并访问；栈顶B的右孩子不空但已被访问，B出栈并访问；栈顶A的右孩子不空且未被访问过，C入栈，栈顶C的左右孩子均为空，出栈并访问；栈顶A的右孩子不空但已被访问，A出栈并访问。由此得到后序序列DEBCA。

   在上述思想的第二步中，必须分清返回时是从左子树返回的还是从右子树返回的，因此设定一个辅助指针r，用于指向最近访问过的结点。也可在结点中增加一个标志域，记录是否已被访问。

```
void PostOrder(BiTree T)
{
    InitStack(S);
    BiTNode *p=T;
    BiTNode *r=NULL;
    while(p||!IsEmpty(S))
    {
        if(p)//走到最左边
        {
            push(S,p);
            p=p->lchild;
        }
        else//向右
        {
            GetTop(S,p);//读栈顶结点（非出栈）
            if(p->rchild&&p->rchild!=r)//若右子树存在，且未被访问过
                p->rchild;//转向右
            else//否则，弹出结点并访问
            {
                pop(S,p);//将结点弹出
                visit(p->data);//访问该结点
                r=p;//记录最近访问过的结点
                p=NULL;//结点访问完后，重置p指针
            }
        }
    }
}
```

4. 试给出二叉树的自下而上、从右到左的层次遍历算法。

一般的二叉树层次遍历是自上而下、从左到右，这里的遍历顺序恰好相反。算法思想：利用原有的层次遍历算法，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即为所求的算法。具体实现如下：

第一步：把根结点入列。

第二步：把一个元素出队列，遍历这个元素。

第三步：依次把这个元素的左孩子、右孩子入队列。

第四步：若队列不同，则跳到第二步，否则结束。

```
void InvertLevel(BiTree bt)
{
    Stack s; Queue Q;
    if(bt!=NULL)
    {
        InitStack(s);//栈初始化，栈中存放二叉树结点的指针
        InitQueue(Q);//队列初始化，队列中存放二叉树结点的指针
        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);//若左子女不空，则入队列
            if(p->rchild)
                EnQueue(Q,p->rchild);//若右子女不空，则入队列
```

```
        }
        while(IsEmpty(s)==false)
        {
            Pop(s,p);
            visit(p->data);
        }//自下而上、从右到左的层次遍历
    }
}
```

5. 假设二叉树采用二叉链表存储结构，设计一个非递归算法求二叉树的高度。

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

```
int Btdepth(BiTree T)
{
    //采用层次遍历的非递归方法求解二叉树的高度
    if(!T)
        return 0;//树空，高度为0
    int front=-1,rear=-1;
    int last=0,level=0;//last指向当前层的最右结点
    BiTree Q[MaxSize];//设置队列Q，元素是二叉树结点指针且容量足够
    Q[++rear]=T;//将根结点入队
    BiTree p;
    while(front<rear)//队不空，则循环
    {
        p=Q[++front];//队列元素出队，即正在访问的结点
        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==last)//处理该层的最右结点
        {
            level++;//层数增1
            last=rear;//last指向下层
        }
    }
    return level;
}
```

   求某层的结点个数、每层的结点个数、树的最大宽度等，都可采用上述思想。也可使用递归算法。

```
int Btdepth2(BiTree T)
{
    if(T==NULL)
        return 0;//空树，高度为0
    ldep=Btdepth2(T->lchild);//左子树高度
    rdep=Btdepth2(T->rchild);//右子树高度
    if(ldep>rdep)
        return ldep+1;//树的高度为子树最大高度加根结点
    else
        return rdep+1;
}
```

6. 二叉树按二叉链表形式存储，试编写一个判别给定二叉树是否是完全二叉树的算法。

   根据完全二叉树的定义，具有n个结点的完全二叉树与满二叉树中编号从1~n的结点一一对应。算法思想：采用层次遍历算法，将所有结点加入队列（包括空结点）。遇到空结点时，查看其后是否有非空结点。若有，则二叉树不是完全二叉树。

```
bool IsComplete(BiTree T)
{
    //本算法判断给定二叉树是否为完全二叉树
    InitQueue(Q);
    if(!T)
        return true;//空树为满二叉树
    EnQueue(Q,T);
    while(!IsEmpty(Q))
    {
        DeQueue(Q,p);
        if(p)//结点非空，将其左、右子树入队列
        {
            EnQueue(Q,p->lchild);
            EnQueue(Q,p->rchild);
        }
        else//结点为空，检查其后是否有非空结点
            while(!IsEmpty(Q))
            {
                DeQueue(Q,p);
                if(p)//结点非空，则二叉树为非完全二叉树
                    return false;
            }
    }
    return true;
}
```

7. 假设二叉树采用二叉链表存储结构存储，试设计一个算法，计算一棵给定二叉树的所有双分支结点个数。

   计算一棵二叉树b中所有双分支结点个数的递归模型f(b)如下：

f(b=0)	若b=NULL
f(b)=f(b->lchild)+f(b->rchild)+1	若*b为双分支结点
f(b)=f(b->lchild)+f(b->rchild)	其他情况（*b为单分支结点或叶结点）

```
int DsonNodes(BiTree b)
{
    if(b=NULL)
        return 0;
    else if(b->lchild!=NULL&&b->rchild!=NULL)//双分支结点
        return DsonNodes(b->lchild)+DsonNodes(b->rchild)+1;
    else
        return DsonNodes(b->lchild)+DsonNodes(b->rchild);
}
```

也可以设置一个全局变量Num，每遍历到一个结点时，判断每个结点是否为分支结点（左、右结点都不为空，注意是双分支），若是则Num++。

8. 设树B是一棵采用链式结构存储的二叉树，编写一个把树B中所有结点的左、右子树进行交换的函数。

   采用递归算法实现交换二叉树的左、右子树，首先交换b结点的左孩子的左、右子树，然后交换b结点的右孩子的左、右子树，最后交换b结点的左、右孩子，当结点为空时递归结束（后序遍历思想）。

```
void swap(BiTree b)
{
    //本算法递归地交换二叉树的左、右子树
    if(b)
    {
        swap(b->lchild);//递归地交换左子树
        swap(b->rchild);//递归地交换右子树
        temp=b->lchild;//交换左、右孩子结点
        b->lchild=b->rchild;
        b->rchild=temp;
    }
}
```

9. 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<=二叉树中结点个数）个结点的值。

   设置一个全局变量i（初值为1）来表示进行先序遍历时，当前访问的是第几个结点。然后可以借用先序遍历的代码模型，先序遍历二叉树。当二叉树b为空时，返回特殊字符'#'；当k==i时，该结点即为要找的结点，返回b->data；当k!=i时，递归地在左子树中查找，若找到则返回该值，否则继续递归地在右子树中查找，并返回其结果。

f(b, k)='#'	当b=NULL时
f(b, k)=b->data	当i=k时
f(b, k)=((ch=f(b->lchild, k))=='#'?f(b->rchild, k) : ch)	其他情况

```
int i=1;//遍历序号的全局变量
ElemType PreNode(BiTree b, int k)
{
    //本算法查找二叉树先序遍历序列中第k个结点的值
    if(b==NULL)//空结点，则返回特殊字符
        return '#';
    if(i==k)//相等，则当前结点即为第k个结点
        return b->data;
    i++;//下一个结点
    ch=PreNode(b->lchild, k);//左子树中递归寻找
    if(ch!='#')//在左子树中，则返回该值
        return ch;
    ch=PreNode(b->rchild,k);//在右子树中递归寻找
    if(ch!='#')//在右子树中，则返回该值
        return ch;
}
```

代码实质上就是一个遍历算法的实现，只不过用一个全局变量来记录访问的序号，求其他遍历序列的第k个结点也采用相似的方法。二叉树的遍历算法可以引申出大量的算法题。

10. 已知二叉树以二叉链表存储，编写算法完成：对于树中每个元素值为x的结点，删除以它为根的子树，并释放相应的空间。

    删除以元素值x为根的子树，只要能删除其左、右子树，就可以释放值为x的根结点，因此宜采用后序遍历。算法思想：删除值为x的结点，意味着应将其父结点的左（右）子女指针置空，用层次遍历易于找到某结点的父结点。要求删除树中每个元素值为x的结点的子树，因此要遍历完整棵二叉树。

```
void DeleteXTree(BiTree &bt)//删除以bt为根的子树
{
    if(bt)
    {
        DeleteXTree(bt->lchild);
        DeleteXTree(bt->rchild);//删除bt的左子树、右子树
        free(bt);//释放被删结点所占的存储空间
    }
}
//在二叉树上查找所有以x为元素值的结点，并删除以其为根的子树
void Search(BiTree bt, ElemType x)
{
    BiTree Q[];//Q是存放二叉树结点指针的队列，容量足够大
    if(bt)
    {
        if(bt->data==x)//若根结点值为x，则删除整棵树
        {
            DeleteXTree(bt);
            exit(0);
        }
        InitQueue(Q);
        EnQueue(Q,bt);
        while(!IsEmpty(Q))
        {
            DeQueue(Q,p);
            if(p->lchild)//若左子女非空
                if(p->lchild->data==x)//左子树符合则删除左子树
                {
                    DeleteXTree(p->lchild);
                    p->lchild=NULL;
                }//父结点的左子女置空
                else
                    EnQueue(Q,p->lchild);//左子树入队列
            if(p->rchild)//若右子女非空
                if(p->rchild->data==x)//右子女符合则删除右子树
                {
                    DeleteXTree(p->rchild);
                    p->rchild=NULL;
                }
                else//父结点的右子女置空
                    EnQueue(Q,p->rchild);//右子女入队列
        }
    }
}
```

11. 在二叉树中查找值为x的结点，试编写算法（用C语言）打印值为x的结点的所有祖先，假设值为x的结点不多于一个。

    采用非递归后序遍历，最后访问根结点，访问到值为x的结点时，栈中所有元素均为该结点的祖先，依次出栈打印即可。

```
typedef struct{
    BiTree t;
    int tag;
}stack;//tag=0表示左子女被访问，tag=1表示右子女被访问
void Search(BiTree bt, ElemType x)
{
    //在二叉树bt中，查找值为x的结点，并打印其所有祖先
    stack s[];//栈容量足够大
    top=0;
    while(bt!=NULL||top>0)
    {
        while(bt!=NULL&&bt->data!=x)//结点入栈
        {
            s[++top].t=bt;
            s[top].tag=0;
            bt=bt->lchild;//沿左右分支向下
        }
        if(bt!=NULL&&bt->data==x)
        {
            printf("所查结点的所有祖先结点的值为:\n");//找到x
            for(int i=1;i<=top;i++)
                printf("%d",s[i].t->data);//输出祖先值后结束
            exit(1);
        }
        while(top!=0&&s[top].tag==1)
            top--;//退栈（空遍历）
        if(top!=0)
        {
            s[top].tag=1;
            bt=s[top].t->rchild;//沿右分支向下遍历
        }
    }
}
```

    因为查找的过程就是后序遍历的过程，所以使用的栈的深度不超过树的深度。

12. 设一棵二叉树的结点为（LLINK, INFO, RLINK），ROOT为指向该二叉树根结点的指针，p和q分别为指向该二叉树中任意两个结点的指针，试编写算法ANCESTOR(ROOT, p, q, r)，找到p和q的最近公共祖先结点r。

    后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。找p和q的最近公共祖先结点r，不失一般性，设p在q的左边。算法思想：采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某节点时，栈中所有元素均为该结点的祖先。后序遍历必然先遍历到结点p，栈中元素均为p的祖先。先将栈复制到另一辅助栈中。继续遍历到结点q时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点p和q的最近公共祖先。

```
typedef struct{
    BiTree t;
    int tag;//tag=0表示左子女已被访问，tag=1表示右子女已被访问
```

```
}stack;
stack s[], s1[];//栈，容量足够大
BiTree ANCESTOR(BiTree ROOT, BiTNode *p, BiTNode *q)
{
    //本算法求二叉树中p和q指向结点的最近公共结点
    top=0;bt=ROOT;
    while(bt!=NULL||top>0)
    {
        while(bt!=NULL)
        {
            s[++top].t=bt;
            s[top].tag=0;
            bt=bt->lchild;
        }//沿左分支向下
        while(top!=0&&s[top].tag==1)
        {
            //假定p在q的左侧，遇到p时，栈中元素均为p的祖先
            if(s[top].t==p)
            {
                for(i=1;i<=top;i++)
                    s1[i]=s[i];
                top1=top;
            }//将栈s的元素转入辅助栈s1保存
            if(s[top].t==q)//找到q结点
                for(i=top;i>0;i--)//将栈中元素的树结点到s1中去匹配
                {
                    for(j=top1;j>0;j--)
                        if(s1[j].t==s[i].t)
                            return s[i].t;//p和q的最近公共祖先已找到
                }
                top--;//退栈
        }
        if(top!=0)
        {
            s[top].tag=1;
            bt=s[top].t->rchild;
        }//沿右分支向下遍历
    }
    return NULL;//p和q无公共祖先
}
```

13. 假设二叉树采用二叉链表存储结构，设计一个算法，求非空二叉树b的宽度（即具有结点数最多的那一层的结点个数）。

    采用层次遍历的方法求出所有结点的层次，并将所有结点和对应的层次放在一个队列中。然后通过扫描队列求出各层的结点总数，最大的层结点总数即为二叉树的宽度。

```
typedef struct{
    BiTree data[MaxSize];//保存队列中的结点指针
    int level[MaxSize];//保存data中相同下标结点的层次
    int front, rear;
}Qu;

int BTWidth(BiTree b)
{
    BiTree p;
```

```
    int k, max, i, n;
    Qu.front=Qu.rear=-1;//队列为空
    Qu.rear++;
    Qu.data[Qu.rear]=b;//根结点指针入队
    Qu.level[Qu.rear]=1;//根结点层次为1
    while(Qu.front<Qu.rear)
    {
        Qu.front++;//出队
        p=Qu.data[Qu.front];//出队结点
        k=Qu.level[Qu.front];//出队结点的层次
        if(p->lchild!=NULL)//左孩子进队列
        {
            Qu.rear++;
            Qu.data[Qu.rear]=p->lchild;
            Qu.level[Qu.rear]=k+1;
        }
    }
    max=0;i=0;//max保存同一层最多的结点个数
    k=1;//k表示从第一层开始查找
    while(i<Qu.rear)//i扫描队中所有元素
    {
        n=0;//n统计第k层的结点个数
        while(i<=Qu.rear&&Qu.level[i]==k)
        {
            n++;
            i++;
        }
        k=Qu.level[i];
        if(n>max)max=n;//保存最大的n
    }
    return max;
}
```

为求二叉树的宽度，队列中的结点出队后仍需保留在队列中，所以设置的队列采用非环形队列，否则在出队后可能被其他结点覆盖。

14. 设有一棵满二叉树（所有结点值均不同），已知其先序序列为pre，设计一个算法求其后序序列post。

对一般二叉树，仅根据先序或后序序列，不能确定另一个遍历序列。但对满二叉树，任意一个结点的左、右子树均含有相等的结点数，同时，先序序列的第一个结点作为后序序列的最后一个结点，由此得到将先序序列pre[l1..h1]转换为后序序列post[l2..h2]的递归模型如下：

f(pre,l1,h1,post,l2,h2)=不做任何事情	h1<l1时
f(pre,l1,h1,post,l2,h2)=post[h2]=pre[l1] 取中间位置half=(h1-l1)/2 将pre[l1+1, l1+half]左子树转换为post[l2, l2+half-1] 即f(pre,l1+1,l1+half,post,l2,l2+half-1) 将pre[l1+half+1,h1]右子树转换为post[l2+half,h2-1] 即f(pre,l1+half+1,h1,post,l2+half,h2-1)	其他情况

其中，post[h2]=pre[l1]表示后序序列的最后一个结点（根结点）等于先序序列的第一个结点（根结点）。

```
void PreToPost(ElemType pre[], int l1, int h1, ElemType post[], int l2, int h2)
{
    int half;
    if(h1>=l1)
    {
        post[h2]=pre[l1];
        half=(h1-l1)/2;
        PreToPost(pre,l1+1,l1+half,post,l2,l2+half-1);//转换左子树
        PreToPost(pre,l1+half+1,h1,post,l2+half,h2-1);//转换右子树
    }
}
```

```
ElemType *pre = "ABCDEFG";
ElemType post[MaxSize];
PreToPost(pre,0,6,post,0,6);
printf("后序序列：");
for(int i=0;i<=6;i++)
    printf("%c",post[i]);
printf("\n");
```

15. 设计一个算法将二叉树的叶结点按从左到右的顺序连成一个单链表，表头指针为head。二叉树按二叉链表方式存储，链接时用叶结点的右指针域来存放单链表指针。

    通常使用的先序、中序和后序遍历对于叶结点的访问顺序都是从左到右，这里选择中序递归遍历。算法思想：设置前驱结点指针pre，初始为空。第一个叶结点由指针head指向，遍历到叶结点时，就将它前驱的rchild指针指向它，最后一个叶结点的rchild为空。

```
LinkedList head, pre=NULL;//全局变量
LinkedList InOrder(BiTree bt)
{
    if(bt)
    {
        InOrder(bt->lchild);//中序遍历左子树
        if(bt->lchild==NULL&&bt->rchild==NULL)//叶结点
            if(pre==NULL)
            {
                head=bt;
                pre=bt;
            }//处理第一个叶结点
            else
            {
                pre->rchild=bt;
                pre=bt;
            }//将叶结点链入链表
        InOrder(bt->rchild);//中序遍历右子树
        pre->rchild=NULL;//设置链表尾
    }
    return head;
}
```

上述算法的时间复杂度为O(n)，辅助变量使用head和pre，栈空间复杂度为O(n)。

16. 试设计判断两棵二叉树是否相似的算法。所谓二叉树T1和T2相似，指的是T1和T2都是空的二叉树或都只有一个根结点；或者T1的左子树和T2的左子树是相似的，且T1的右子树和T2的右子树是相似的。

    采用递归的思想求解，若T1和T2都是空树，则相似；若有一个为空另一个不空，则必然不相似；否则递归地比较它们的左、右子树是否相似。递归函数的定义如下：

f(T1, T2)=1	若T1==T2==NULL
f(T1,T2)=0	若T1和T2之一为NULL，另一个不为NULL
f(T1,T2)=f(T1->lchild,T2->lchild)&&f(T1->rchild,T2->rchild)	若T1和T2均不为NULL

```
int similar(BiTree T1, BiTree T2)
{
    //采用递归的算法判断两棵二叉树是否相似
    int leftS, rightS;
    if(T1==NULL&&T2==NULL)//两棵树皆为空
        return 1;
    else if(T1==NULL||T2==NULL)//只有一棵树为空
        return 0;
    else//递归判断
    {
        leftS=similar(T1->lchild,T2->lchild);
        rights=similar(T1->rchild,T2->rchild);
        return leftS&&rightS;
    }
}
```

17. 二叉树的带权路径长度（WPL）是二叉树中所有叶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：

    - 给出算法的基本设计思想。
    - 使用C或C++语言，给出二叉树结点的数据类型定义。
    - 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。

    二叉树的带权路径长度有两种常见的计算方法：1.根据二叉树的带权路径长度的定义，二叉树的WPL值=树中全部叶结点的带权路径长度之和。2.根据带权二叉树的性质，二叉树的WPL值=树中所有非叶结点的权值之和。

    方法一：可采用递归算法实现，根据定义：

$$
\begin{aligned}
\text{二叉树的}WPL\text{值} &= \text{树中全部叶结点的带权路径长度之和} \\
&= \text{根结点左子树中全部叶结点的带权路径长度之和} + \\
&\quad \text{根结点右子树中全部叶结点的带权路径长度之和}
\end{aligned}
$$

    叶结点的带权路径长度=该结点的weight域的值*该结点的深度

    设根结点的深度为0，若某结点的深度为d时，则其孩子结点的深度为d+1。

在递归遍历二叉树结点的过程中，若遍历到叶结点，则返回该结点的带权路径长度，否则返回其左右子树的带权路径长度之和。

方法二：若借用非叶结点的weight域保存其孩子结点中weight域值的和，则树的WPL等于树中所有非叶结点weight域值之和。

采用后序遍历策略，在遍历二叉树T时递归计算每个非叶结点的weight域的值，则树T的WPL等于根结点左子树的WPL加上右子树的WPL，再加上根结点中weight域的值。在递归遍历二叉树结点的过程中，若遍历到叶结点，则return 0并且退出递归，否则递归计算其左右子树的WPL和自身结点的权值。

二叉树结点的数据类型定义如下

```
typedef struct node{
    int weight;
    struct node *left, *right;
}BTree;
```

方法一

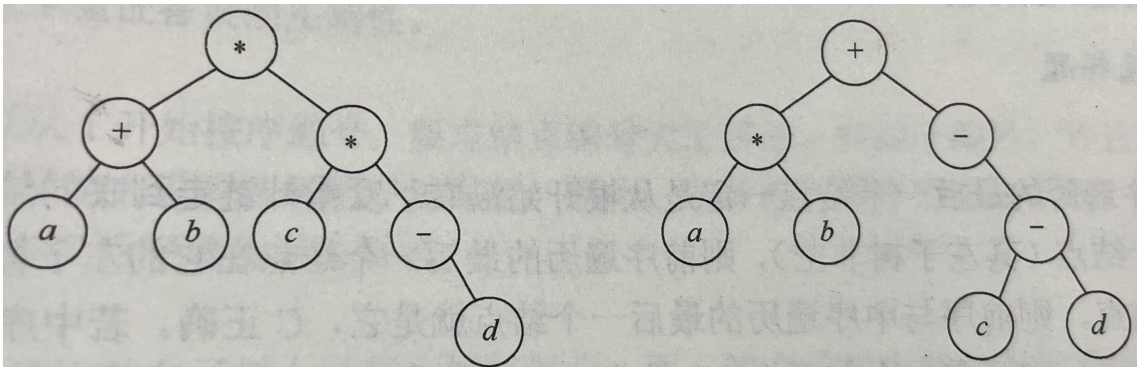
```
int WPL(BTree *root)//根据WPL的定义采用递归算法实现
{
    return WPL1(root, 0);
}
int WPL1(BTree *root, int d)//d为结点深度
{
    if(root->left==NULL&&root->right==NULL)
        return root->weight*d;
    else
        return WPL1(root->left, d+1)+WPL1(root->right,d+1);

}
```

方法二

```
int WPL(BTree *root)//基于递归的后序遍历算法实现
{
    int w_l, w_r;
    if(root->left==NULL&&root->right==NULL)
        return 0;
    else
    {
        w_l=WPL(root->left);//计算左子树的WPL
        w_r=WPL(root->right);//计算右子树的WPL
        root->weight=root->left->weight+root->right->weight;//填写非叶结点的
weight域
        return w_l+w_r+root->weight;//返回WPL值
    }
}
```

18. 请设计一个算法，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，当下列两棵表达式树作为算法的输入时：



输出的等价中缀表达式分别为(a+b)*(c*(-d))和(a*b)+(-(c-d))。

二叉树结点定义如下：

```
typedef struct node{
    char data[10];//存储操作数或操作符
    struct node *left, *right;
}BTree;
```

要求：

- 给出算法的基本设计思想。
- 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。

表达式树的中序序列加上必要的括号即为等价的中缀表达式。可以基于二叉树的中序遍历策略得到所需的表达式。

表达式树中分支结点所对应的子表达式的计算次序，由该分支结点所处的位置决定。为得到正确的中缀表达式，需要在生成遍历序列的同时，在适当位置增加必要的括号。显然，表达式的最外层（对应根结点）和操作数（对应叶结点）不需要添加括号。

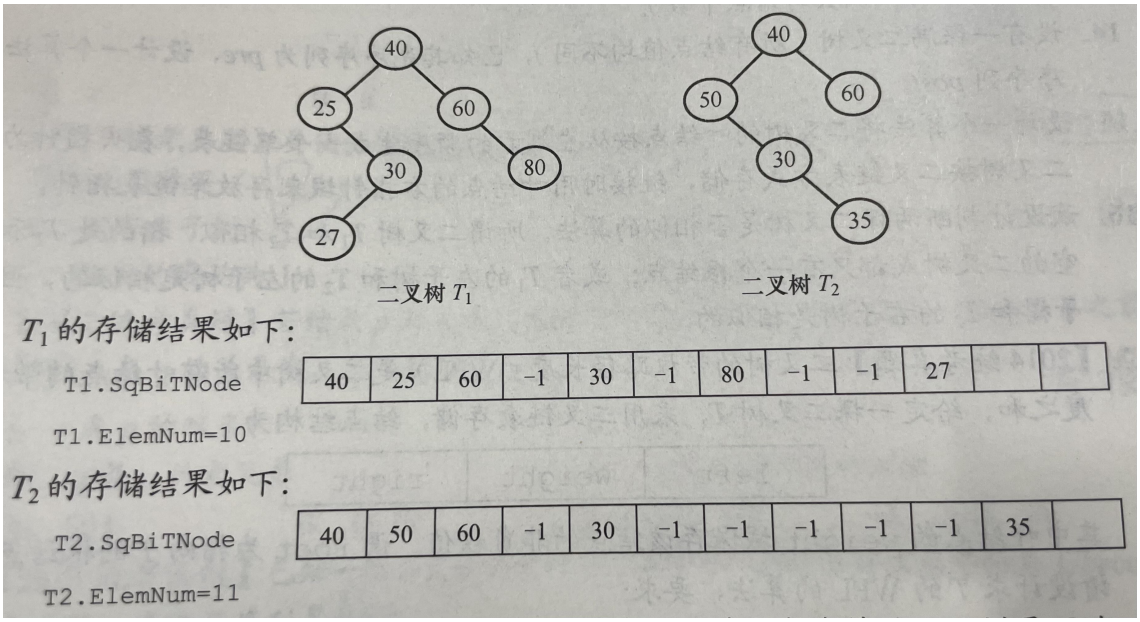
将二叉树的中序遍历递归算法稍加改造。除根结点和叶结点外，遍历到其他结点时在遍历其左子树之前加上左括号，遍历完右子树后加上右括号。

```
void BtreeToE(BTree *root)
{
    BtreeToExp(root,1);//根的高度为1
}
void BtreeToExp(BTree *root, int deep)
{
    if(root==NULL)return;//空结点返回
    else if(root->left==NULL&&root->right==NULL)//若为叶结点
        printf("%s",root->data);//输出操作数，不加括号
    else
    {
        if(deep>1)printf("(");//若有子表达式则加一层括号
        BtreeToExp(root->left,deep+1);
        printf("%s",root->data);//输出操作符
        BtreeToExp(root->right,deep+1);
        if(deep>1)printf(")");//若有子表达式则加一层括号
    }
}
```

19. 已知非空二叉树T的结点值均为正整数，采用顺序存储方式保存，数据结构定义如下：

```
typedef struct{//MAX_SIZE为已定义常量
    int SqBiTNode[MAX_SIZE];//保存二叉树结点值的数组
    int ElemNum;//实际占用的数组元素个数
}SqBiTree;
```

T中不存在的结点在数组SqBiTNode中用-1表示。例如，对于下图所示的两棵非空二叉树T1和T2，



请设计一个尽可能高效的算法，判定一棵采用这种方式存储的二叉树是否为二叉搜索树，若是，则返回true，否则，返回false。要求：

- 给出算法的设计思想。
- 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。

方法一

对于采用顺序存储保存的二叉树，根结点保存在SqBiTNode[0]中；当某结点保存在SqBiTNode[i]中时，若有左孩子，则其值保存在SqBiTNode[2i+1]中；若有右孩子，则其值保存在SqBiTNode[2i+2]中；若有双亲结点，则其值保存在SqBiTNode[(i-1)/2]中。

二叉搜索树需要满足的条件是：任意一个结点值大于其左子树中的全部结点值，小于其右子树中的全部结点值。中序遍历二叉搜索树得到一个升序序列。

使用整型变量val记录中序遍历过程中已遍历结点的最大值，初值为一个负整数。若当前遍历的结点值小于或等于val，则算法返回false，否则，将val的值更新为当前结点的值。

```
#define false 0
#define true 1
typedef int bool;
bool judgeInOrderBST(SqBiTree bt, int k, int *val)//初始调用时k的值是0
{
    if(k<bt.ElemNum&&bt.SqBiTNode[k]!=-1)
    {
        if(!judgeInOrderBST(bt,2*k+1,val))return false;
        if(bt.SqBiTNode[k]<=*val)return false;
        *val=bt.SqBiTNode[k];
        if(!judgeInOrderBST(bt,2*k+2,val))return false;
    }
    return true;
}
```

方法二

设置两个数组pmax和pmin。根据二叉搜索树的定义，SqBiTNode[i]中的值应该大于以SqBiTNode[2i+1]为根的子树中的最大值（保存在pmax[2i+1]中），小于以SqBiTNode[2i+2]为根的子树中的最小值（保存在pmin[2i+1]中）。初始时，用数组SqBiTNode中前ElemNum个元素的值对数组pmax和pmin初始化。

在数组SqBiTNode中从后向前扫描，扫描过程中逐一验证结点与子树之间是否满足上述的大小关系。

```
#define false 0
#define true 1
typedef int bool;
bool judgeBST(SqBiTree bt)
{
    int k,m,*pmin,*pmax;
    pmin=(int *)malloc(sizeof(int)*(bt.ElemNum));
    pmax=(int *)malloc(sizeof(int)*(bt.ElemNum));
    for(k=0;k<bt.ElemNum;k++)//辅助数组初始化
        pmin[k]=pmax[k]=bt.SqBiTNode[k];
    for(k=bt.ElemNum-1;k>0;k--)//从最后一个叶结点向根遍历
    {
        if(bt.SqBiTNode[k]!=1)
        {
            m=(k-1)/2;//双亲
            if(k%2==1&&bt.SqBiTNode[m]>pmax[k])//其为左孩子
                pmin[m]=pmin[k];
            else if(k%2==0&&bt.SqBiTNode[m]<pmin[k])//其为右孩子
                pmax[m]=pmax[k];
            else return false;
        }
    }
    return true;
}
```

4. 编程求以孩子兄弟表示法存储的森林的叶结点数。

当森林（树）以孩子兄弟表示法存储时，若结点没有孩子（fch==NULL），则它必是叶子，总的叶结点个数是孩子子树（fch）上的叶子树和兄弟子树（nsib）上的叶结点个数之和。

```
typedef struct node
{
    ElemType data;//数据域
    struct node *fch, *nsib;//孩子与兄弟域
}*Tree;

int Leaves(Tree t)//计算以孩子兄弟表示法存储的森林的叶子数
{
    if(t==NULL)
        return 0;//树空返回0
    if(t->fch==NULL)//若结点无孩子，则该结点必是叶子
        return 1+Leaves(t->nsib);//返回叶结点和其他兄弟子树中的叶结点树
    else//孩子子树和兄弟子树中叶子树之和
        return Leaves(t->fch)+Leaves(t->nsib);
}
```

5. 以孩子兄弟链表为存储结构，请设计递归算法求树的深度。

   由孩子兄弟链表表示的树，求高度的算法思想是采用递归算法，若树为空，高度为零；否则，高度为第一子女树高度加1的兄弟子树高度的大者。其非递归算法使用队列，逐层遍历树，取得树的高度。

```
int Height(CSTree bt)
{
    //递归求以孩子兄弟链表表示的树的深度
    int hc, hs;
    if(bt==NULL)
        return 0;
    else
    {
        //否则，高度取子女高度+1和兄弟子树高度的大者
        hc=Height(bt->firstchild);//第一子女树高
        hs=Height(bt->nextsibling);//兄弟树高
        if(hc+1>hs)
            return hc+1;
        else
            return hs;
    }
}
```