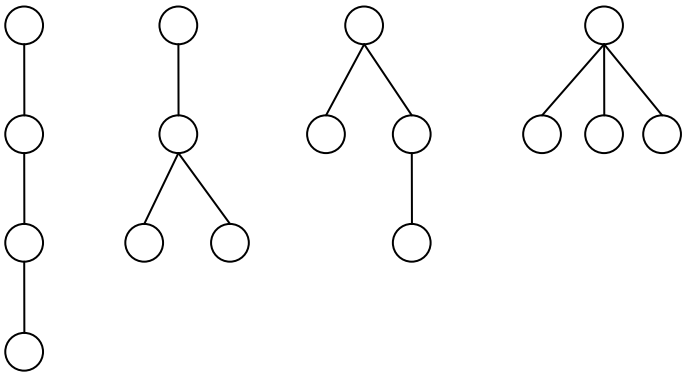
第 5 章习题

5.1 画出由 4 个结点所构成的所有形态的树（假设是无序树）。

【解】可构成 4 棵不同形态的树，如下图。



5.2 已知一棵树的度为 4，其中度为 4 的结点的数目为 3，度为 3 的结点的数目为 4，度为 2 的结点的数目为 5，度为 1 的结点的数目为 2，请求出该树中的叶子结点的数目。

【解】设度为 4、3、2、1、0 的结点数分别为 n_4、n_3、n_2、n_1、n_0，结点总数为 n，由此得方程(1)：

$n= n_4+n_3+n_2+n_1+n_0$ (1)

从根结点往树叶方向看，除了根结点外，每个结点接受 1 个分枝（1 条边），分枝总数为 n-1；

从树叶往树根方向看，4 度结点发出 4 个分枝；3 度 3 个；2 度 2 个；1 度 1 个；叶子结点不发出分枝，即分枝总数为：$4n_4+3n_3+2n_2+n_1$。

树上的分枝树不管从哪个方向看分枝树是相同的，由此得方程(2)：

$n-1=4n_4+3n_3+2n_2+n_1$ (2)

代入已知数，解方程得：$n_0=23$，即叶子结点数为 23 个。

5.3 如果已知一棵二叉树有 20 个叶子结点，有 10 个结点仅有左孩子，15 个结点仅有右孩子，求出该二叉树的结点数目。

【解】二叉树只有度数 2、1、0 三种结点，设度为 2、1、0 的结点数分别为 n_2、n_1、n_0，结点总数为 n，由此得方程(1)：

$n= n_2+n_1+n_0$ (1)

从根结点往树叶方向看，除了根结点外，每个结点接受 1 个分枝（1 条边），分枝总数为 n-1；

从树叶往树根方向看，2 度结点发出 2 个分枝；1 度 1 个；叶子结点不发出分枝，即分

枝总数为：$2n_2+n_1$。

树上的分枝树不管从哪个方向看分枝树是相同的，由此得方程(2)：

$n-1=2n_2+n_1$ (2)

代入已知数，解方程得：n=64，即二叉树结点数为 64 个。

5.4 已知某完全二叉树有 100 个结点，试用三种不同的方法求出该二叉树的叶子结点数。

【解】

方法一：

根据完全二叉树的性质，可以计算出第 100 个结点的父结点编号为 100/2=50，所以从 51 号到 100 号结点是叶子结点，共 50 个。

方法二：

根据完全二叉树性质知此树高度为 7，且前 6 层构成高度 6 的满二叉树，共有 63 个结点，其中第 6 层有 32 个结点。最后一层剩下 37 个结点，全部为叶子结点，且他们要用掉第 6 层 19 个结点作为父结点，这样第 6 层剩下 32-19=13 个叶子结点，加起来叶子结点数为 37+13=50 个。

方法三：

由方法二可知最后一层有 37 个结点，按完全二叉树要求，将有 1 个结点的度为 1，其它都是度为 2 和叶子结点，可以得方程：

$100=n_2+n_1+n_0$ (1)

$100-1=2n_2+n_1$ (2)

解得 $n_0=50$，即叶子结点 50 个。

5.5 如果已知完全二叉树的第 6 层有 5 个叶子，试画出所有满足这一条件的完全二叉树，并指出结点数目最多的那棵完全二叉树的叶子结点数目。

【解】根据完全二叉树性质，第 6 层共有 32 个结点，除了 5 个叶子结点，剩下 27 个结点皆有孩子结点，其中此 27 个结点中最后一个可以只有左孩子结点，也可以有左右孩子结点，所以存在 2 中可能形态的完全二叉树。

结点数最多时，应该是第 7 层放 54 个结点，这样叶子总数为 59 个。

5.6 在编号的完全二叉树中，判断编号为 i 和 j 的两个结点在同一层的条件是什么？

【解】

解法一：

对于标号 i 的结点，我们只看前 i 个结点，也是一棵完全二叉树，高度为$\lfloor \log_2^i \rfloor+1$。

编号 j 一样，以其为最后结点的完全二叉树的高度为为$\lfloor \log_2^j \rfloor+1$。

如果 i、j 处于同一层，则上述 2 棵二叉树高等相等，即：$\lfloor \log_2^j \rfloor+1=\lfloor \log_2^i \rfloor+1$。

所以 i、j 处于同一层的条件为：$\lfloor \log_2^i \rfloor=\lfloor \log_2^i \rfloor$。

解法二：

对 i、j 同步循环整除 2，如果两者在大于 0 时，出现 i==j，则 i、j 在同一层。

【算法描述】

```
bool sameLayer(seqList T,int i,int j)
{
    while(i>0 && j>0)
    {
        if(i==j)
            return true;
        i=i/2;
        j=j/2;
    }
    return false;
}
```

5.7 设计算法以求解编号为 i 和 j 的两个结点的最近的公共祖先结点的编号。

【解】

解法一：

算法思想：结点 i 的祖先依次是：i/2，$i/2^2$，…，1，j 祖先结点依次是：j/2，$j/2^2$，…，1。他们祖先结点的交叉点即为最近的公共祖先结点。对于 i 的每个祖先，从 j/2 开始，循环比较，每次循环 j=j/2，如果出现两者的某个祖先结点编号相同，即为最近公共祖先结点。

【算法描述】

```
int commonAncestor(seqList T,int i,int j)
{
    int k;
    while(i>0)          //对于 i 的每个祖先，检查是否在 j 的祖先路径上，
    {                    //当两个祖先结点编号相同，即为最近公共祖先
        k=j;
        while(k>0)
        {
            if(i==k)
                return i;
            k=k/2;
        }
        i=i/2;
    }
    return 0;
}
```

解法二：

解上述算法的双重循环合并为一个循环，当 i、j 都大于 0 时循环处理，如果 i==j，

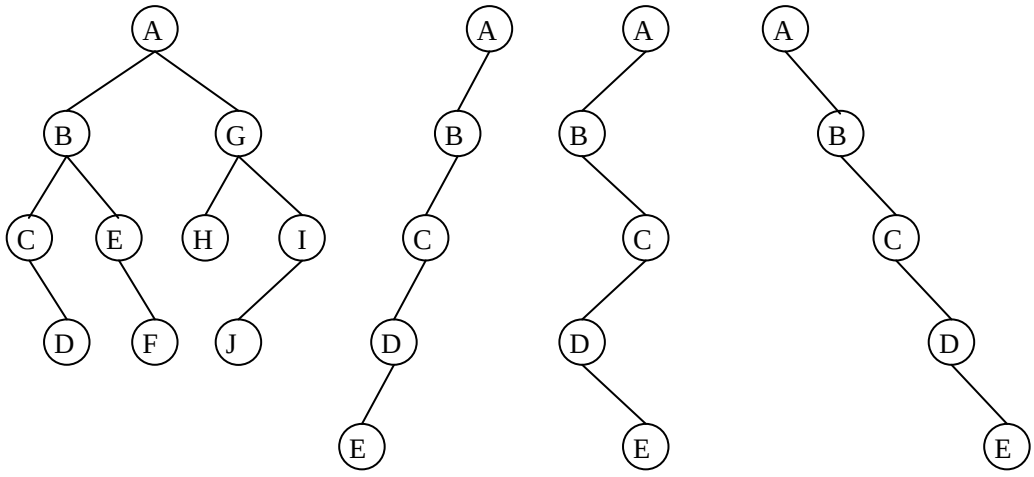
则编号 i 的结点即公共祖先结点，返回公共祖先结点编号；如果 i>j，则让 i=i/2；如果 j>i，则让 j=j/2。循环结束，说明无公共祖先，返回 0。

【算法描述】

```
int commonAncestor1(seqList T,int i,int j)
{
    int k;

    while(i>0 && j>0)
    {
        if(i==j)
            return i;   //找到公共祖先，返回祖先编号
        else if(i<j)
            j=j/2;       //j 指向其父结点
        else              //i 指向父结点
            i=i/2;
    }
    return 0;             //循环退出，没有共同祖先
}
```

5.8 分别求出下图中二叉树的三种遍历序列。



a　　b　　c　　d

图 5-1　题 5.8 图

【解】

第一棵树先序：ABCDEFGHIJ，中序：CDBEFAHGJI，后序：DCFEBHJIGA。

第二棵树先序：ABCDE，中序：EDCBA，后序：EDCBA。

第三棵树先序：ABCDE，中序：DECBA，后序：EDCBA。

第四棵树先序：ABCDE，中序：ABCDE，后序：EDCBA。

5.9 分别描述满足下面条件的二叉树特征：

（1）先序序列和中序序列相同。

（2）先序序列和后序序列相反。

【解】

（1）每个结点只有右分支。

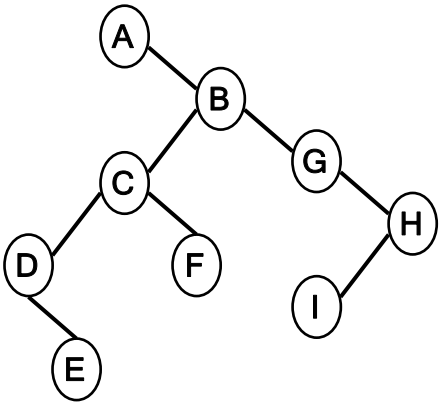
（2）只有左子树，或只有右子树的二叉树。

5.10 证明：由二叉树的先序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：

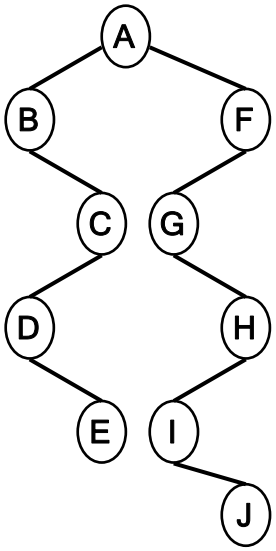
①先序：ABCDEFGHI　②先序：ABCDEFGHIJ

中序：ADECFBGIH　中序：BDECAGIJHF

【解】



① 对应二叉树



② 对应二叉树

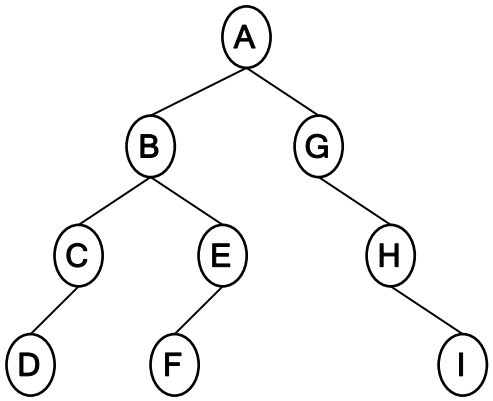
5.11 证明：由二叉树的后序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：

①后序：DCFEBIHGA　②后序：DECBGIHFA

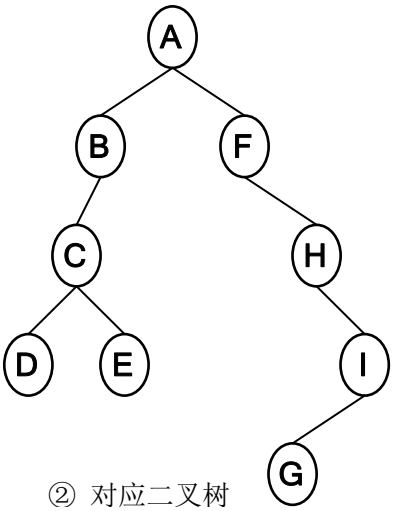
中序：DCBFEAGHI　中序：DCEBAFHGI

【解】

后序序列中最后结点即为根结点。由根结点，在中序序列中就可以区分出左子树和右子树结点。对左右子树做相同的处理，最终即可重建二叉树。



① 对应二叉树



② 对应二叉树

5.12 已知一棵二叉树的先序、中序和后序序列如下，其中各有一部分未给出其值，请构造出该二叉树。

先序：A_CDEF_H_J

中序：C_EDA_GFI_

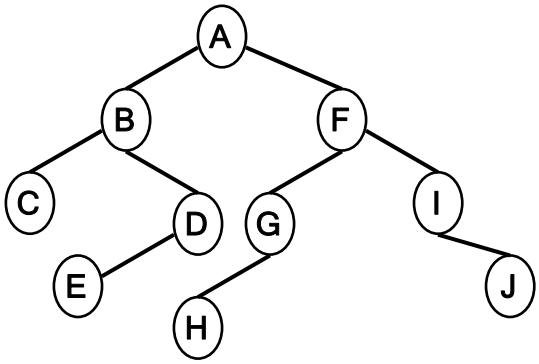
后序：C_ _BHGJI_ _

【解】

先序：A B C D E F G H I J

中序：C B E D A H G F I J

后序：C E D B H G J I F A



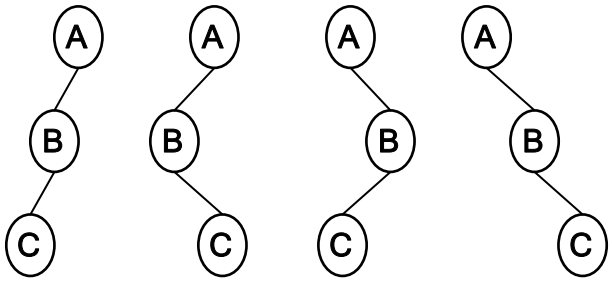
5.12 证明：任意一棵非空的二叉树的先序序列的最后一个结点一定是叶子结点。

【证明】反证法：假定线序序列最后一个结点不是叶子结点，则他必定存在左子树或右子树。按照先序遍历方法，访问子树根结点后，还要先序遍历其左子树、右子树，那么这个结点就不是最后一个结点。所以，线序序列最后一个结点一定是叶子结点。

5.13 用反例证明：由二叉树的先序序列和后序序列不能唯一确定一棵二叉树。

【解】

假定有先序序列 ABC，后序序列 CBA，可能重构出下列不同形态的二叉树，所以由先序序列和后序序列不能唯一确定一棵二叉树。



图

5.14 设计算法以输出二叉树中先序序列的前 k（k>0）个结点的值。

【解】算法思想：本题有不同解法。

解法一：执行先序遍历，将遍历结果保存到一个队列或数组中，遍历结束后再输出前 k 个结点。这种方法要遍历整棵二叉树，效率不高。

解法二：改造先序遍历算法，设置两个整数控制先序遍历，k 为控制遍历的结点数，i 为遍历计数器，计数已经访问的结点数。k、i 可以用全局遍历，也可用函数参数实现，用全局变量实现起来更简单一点。下面为解法二的算法描述。初始调用为 preOutK(T, 0, k)。

【算法描述】

```
void preOutK( btNode *T, int &i, int &k )
{
  if( T && i<k )
  {
            cout<<T->data<<" "<<i<<",    ";     //访问结点
            i++;                                  //访问结点数加 1

            preOutK(T->lChild,i,k);          //先序遍历左子树
            preOutK(T->rChild,i,k);          //先序遍历右子树
  }
}
```

5.15 设计算法按中序次序依次输出各结点的值及其对应的序号。例如，下图中的二叉树的输出结果是(C,1) (B,2) (E,3) (D,4) (F,5) (A,6) (H,7) (J,8) (I,9) (G,10)。

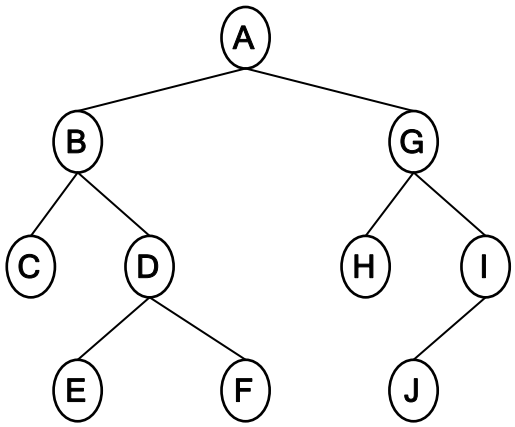


图 5-2　题 5.15 图

【解】算法思想：三种遍历算法改造后均可实现输出结点值同时输出层次数，最简单方法是增加一个传值的函数参数用来记录当前结点的层数，进入左右子树时，层数加 1。

也可以通过全局变量或传址的函数参数来记录层数，在进入子树时层数加 1，遍历结束时，层数需要减 1，且在不同情况下打印层次时，也需要做减 1 处理。

下面改造中序遍历算法，增加一个整数计数变量 i，从 1 开始，进入子树时 i 加 1（增加一层），这种方式每个结点的层次数单独保存在一个变量中。

初始调用时皆为 i=1。

【传值计数中序算法输出层次数描述】

```
void inOrderPrint( btNode *T, int i )
{
    if( T )
    {
        inOrderPrint(T->lChild,i+1);   //进入左子树，层次加 1
        cout<<"("<<T->data<<","<<i<<")   ";
        inOrderPrint(T->rChild,i+1);   //进入右子树，层次加 1
    }
}
```

【引用变量计数中序算法输出层次数描述】

```
void inOrderPrint( btNode *T, int &i )
{
    if( T )
    {
        i++;       //层次数加 1，以便递归访问左右子树
        inOrderPrint(T->lChild,i);
        cout<<"("<<T->data<<","<<i-1<<")   ";   //T 的层次应该减 1，因前面 i 加了 1
        inOrderPrint(T->rChild,i);
        i--;       //左右子树遍历结束，回到根结点，层数减 1
    }
```

```
}
```

全局变量记录层次情况类似。

也可改造非递归遍历算法输出层次，但需要借助一个数组来保存每个结点的层次数。比如当前结点的层次数保存在变量 i 中；保存层次数的数组为 C[ ]。入栈时，结点的层次数保存到数组下标为栈顶指针的单元，即 C[S.top]=i。进入左子树，或进入右子树时，结点的层次数都要加 1。弹栈前，需要把栈顶结点的层次数从数组中取回到变量 i，即 i=C[S.top]，然后再出栈。

【非递归中序遍历输出层次号】

```
void inTraverseNR(btNode* T)
{
    seqStack S;
    btNode *p;
    initialStack(S);
    p=T;
    int i=1;          //保存结点层次号，从 1 开始
    int c[200];       //保存栈中结点对应的层次号

    while( p || !stackEmpty(S))
    {
        if(p)
        {
            push(S,p);      //子树根结点入栈
            p=p->lChild;    //中序遍历左子树

            c[S.top]=i;     //保存当前结点层次号到数组
            i++;            //进入左子树，层次加 1
        }
        else
        {
            i=c[S.top];     //取出栈顶结点的层次号到 i
            pop(S,p);       //取栈顶、出栈
            cout<<"("<<p->data<<",  "<<i<<")";      //输出结点和层次
            p=p->rChild;    //中序遍历右子树
            i++;        //进入右子树，层次加 1
        }
    }
}
```

5.16 设计算法以输出二叉树每个结点到根结点之间的路径上的所有结点的值。

【解】路径问题是二叉树求解的一类问题，变形的问题比如：输出所有叶子结点到根结点的路径，结点值等于 x 的结点到根结点的路径；结点值大于（小于）x 的结点到根结点的

路径等。

求解这类问题都可通过改造二叉树的遍历算法完成，递归遍历和非递归遍历算法都可以改造来输出路径。

本题是一个基本问题，其它路径求解问题要加判定条件，在满足指定条件时输出路径即可。

下面讨论改造递归遍历算法输出路径：

利用先序、中序、后序遍历算法都可以，需要借助栈、队列、数组来保存一个结点到根结点路径上途经的结点。以栈为例，可以把栈作为函数参数通过传值方式传递给递归函数。每遇到一个结点就入栈，则栈中保存的就是当前结点到根结点路径途径的所有结点。改造visit(T)函数，此处输出（打印）栈中所有元素值，就是当前结点到根结点的路径。

【先序遍历输出路径算法描述】

```
void prePath( btNode *T, seqStack S )
{
    if(T)
    {
        push(S,T);          //遇到结点就入栈，栈中保存的即当前结点到根结点的路径
                            //输出结点路径
        for( int i=S.top; i>=0; i-- )
            cout<<S.data[i]->data<<"   ";          //输出栈中结点值，即路径
        cout<<endl;

        prePath( T->lChild, S );      //先序遍历左子树
        prePath( T->rChild, S );      //先序遍历右子树
    }
}
```

【中序遍历输出路径算法描述】

```
void inPath( btNode *T, seqStack S )
{
    if(T)
    {
        push(S,T);          //遇到结点入栈，栈中元素即当前结点到根结点的路径

        inPath( T->lChild, S );       //中序遍历左子树
                //输出路径
        for( int i=S.top; i>=0; i-- )
            cout<<S.data[i]->data<<"   ";          //输出栈中结点值，即路径
        cout<<endl;

        inPath( T->rChild, S );       //中序遍历右子树
    }
```

```
}

【后序遍历输出路径算法描述】
void postPath( btNode *T, seqStack S )
{
    if(T)
    {
        push(S,T);          //遇到结点入栈，栈中即当前结点到根结点的路径

        postPath( T->lChild, S );      //后序遍历左子树
        postPath( T->rChild, S );      //后序遍历右子树
                //输出路径
        for( int i=S.top; i>=0; i-- )
            cout<<S.data[i]->data<<"   ";          //输出栈中结点值，即路径
        cout<<endl;
    }
}
```

其它路径问题只要在输出路径时增加相应条件即可，比如输出所有叶子结点的路径，输出部分代码如下：

```
if( S.data[S.top]->lChild==NULL && S.data[S.top]->rChild==NULL )  //栈顶结点为叶子
    for(int i=S.top;i>=0;i--)
        cout<<S.data[i]->data<<"   ";          //输出路径
    cout<<endl;
```

这种传值方式递归算法输出路径空间效率较低，采用函数传值参数方式，每个结点都要用一个栈来保存路径。

也可以用全局栈，或栈传址方式共用一个栈来保存路径，但每个结点在遍历完左右子树，且已经输出路径后，需要把当前栈顶元素出栈，三种遍历都可以用来输出路径。下面是改造递归后序遍历算法以引用传址方式共用栈保存路径的算法描述：

```
void postPath(btNode *T,seqStack &S)
{
    if(T)
    {
        push(S,T);          //当前结点入栈，栈中即为当前结点到根结点的路径

        postPath (T->lChild, S);      //后序遍历左子树
        postPath (T->rChild, S);      //后序遍历右子树

        for(int i=S.top;i>=0;i--)
```

```
            cout<<S.data[i]->data<<"    ";              //输出路径
        cout<<endl;

        btNode *u;
        pop(S,u);           //当前结点出栈
    }
}
```

求解路径算法也可通过改造非递归遍历算法完成，且非递归算法本身使用栈，不需要另外的缓存机制来保存路径，空间效率显然比递归算法好。

改造非递归算法输出路径时，使用非递归后序遍历算法最为方便，把上述输出路径代码放在遍历左子树、遍历右子树、访问根结点 3 个位置都可以输出路径。但非递归先序遍历和非递归中序遍历算法在遍历右子树时，根结点就出栈了，所以栈中并没有保存全部路径上结点，要想改造来输出路径，要像后序遍历类似改造，遍历右子树时，根结点不出栈，右子树遍历完成再出栈。下面给出非递归后序遍历输出路径，路径输出代码出入到结点入栈位置，插入另外 2 个位置也可以。

```
//非递归后序遍历输出路径
void postPathNR(btNode *T)
{
    seqStack S;
    btNode *p;
    int tag[MAXLEN];
    initialStack(S);
    p=T;
    while( p || !stackEmpty(S))
    {
        if(p)
        {
            push(S,p);

            for(int i=S.top;i>=0;i--)    //输出路径代码
                        cout<<S.data[i]->data<<"    ";              //输出路径
            cout<<endl;

            tag[S.top]=0;
            p=p->lChild;          //后序遍历左子树
        }
        else
        {
            getTop(S,p);          //取栈顶到 p
            if(tag[S.top]==0)
```

```
            {
                        //----输出路径代码也可以插入这里
                tag[S.top]=1;
                p=p->rChild;    //后序遍历右子树
            }
            else                //tag[S.top]==1，左右子树遍历结束，访问子树根结点
            {
                        //----输出路径代码还可以插入这里
                //cout<<p->data<<"   ";
                pop(S,p);         //p 为根子树后序遍历结束，出栈
                p=NULL;           //p 置为空
            }
        }
    }
}
```

5.17 设计算法将一棵以二叉链表形式存储的二叉树转换为顺序存储形式存储到数组A[n]中，并将其中没有存放结点值的数组元素设置为 NULL。

【解】

算法思想：改造二叉树的一种遍历算法实现，这里改造先序遍历算法完成。在访问结点时，将结点值存入数组 A[]。算法设置一个表示结点编号的参数 i，初始为 1。i 作为结点的数组下标。递归处理左子树树时结点编号为 2*i，递归处理右子树时结点编号为 2*i+1。

此外，算法设置一个整型参数 num，用以记录树上最后一个有效结点的编号。

【算法描述】

```
void Bi2SeqTree(btNode *T, int i,elementType A[], int &num)
{
        //i--结点按完全二叉树的编号，从 1 开始
        //num--返回二叉树最后结点的编号
        //A[]--二叉树的顺序存储，按完全二叉树存储
    if(T)
    {
        A[i]=T->data;
        if(i>num)
            num=i;
        Bi2SeqTree(T->lChild,2*i,A,num);
        Bi2SeqTree(T->rChild,2*i+1,A,num);
    }
}
```

算法初始调用：Bi2SeqTree(T, 1, A, num)。

5.18 设计算法将一棵以顺序存储方式存储在数组 A 中的二叉树转换为二叉链表形式。

【解】

算法思想：改造二叉树的一种遍历算法完成，这里改造先序遍历实现。算法包括两个整型参数 i 和 num。i 为结点在完全二叉树上的编号，从 1 开始；num 为二叉树上最后有效结点编号。

【算法描述】

```
void seq2BiTree(btNode *&T,elementType A[],int i,int num)
{
        //i 为当前结点编号，从 1 开始
        //num 为最后有效结点编号
    if(i<=num && A[i])
    {
        T=new btNode;
        T->data=A[i];
        T->lChild=NULL;
        T->rChild=NULL;

        seq2BiTree(T->lChild,A,2*i,num);
        seq2BiTree(T->rChild,A,2*i+1,num);
    }
}
```

5.19 分别设计出先序、中序和后序遍历二叉树的非递归算法。

【解】

```
//非递归：先序遍历
void PreTraverseNR(BiNode* pBT)
{
    BiNode* p;
    seqStack S;

    initStack(S);   //初始化栈

    p=pBT;
    while(p || !stackEmpty(S))
    {
        if(p)
        {
            cout<<p->data<<", ";   //访问根节点
            pushStack(S, p);   //p 指针入栈
            p=p->lChild;   //遍历左子树
        }
```

```
            else
            {
                popStack(S, p);   //p 为空时，将上一层的根节点指针弹出
                p=p->rChild;     //遍历右子树
            }
        }
    }

    //非递归：中序遍历
    void InTraverseNR(BiNode* pBT)
    {
        BiNode* p;
        seqStack S;

        initStack(S);   //初始化栈

        p=pBT;
        while(p || !stackEmpty(S))
        {
            if(p)
            {
                pushStack(S,p);   //根节点先行入栈，以便左子树遍历结束，返回访问根节
点
                p=p->lChild;        //遍历左子树
            }
            else   //p 为空--访问根节点、遍历右子树
            {
                popStack(S, p);   //某子树根节点出栈
                cout<<p->data<<", ";   //访问某子树根节点
                p=p->rChild;   //遍历 p 的右子树
            }
        }
    }

    //非递归：后序遍历
    void PostTraverseNR(BiNode* pBT)
    {
        BiNode* p;
        seqStack S;
        int tag[MaxLen];   //标记左子树、右子树
        int n;
```

```
    initStack(S);   //初始化栈
    p=pBT;

    while(p || !stackEmpty(S))
    {
        if(p)
        {
            pushStack(S,p);
            tag[S.top]=0;   //标记遍历左子树
            p=p->lChild;    //循环遍历左子树
        }
        else   //p==NULL 但是栈不空
        {
            stackTop(S,p);   //取栈顶，但不退栈，以便遍历 p 的右子树
            if(tag[S.top]==0)   //说明 p 的左子树已经遍历，右子树尚未遍历
            {
                tag[S.top]=1;   //设置当前结点遍历右子树标记
                p=p->rChild;    //遍历右子树
            }
            else   //tag[S.top]==1，说明 p 的左右子树皆已经遍历，
            {
                popStack(S,p);   //退栈
                cout<<p->data<<", ";   //访问某子树根节点
                p=NULL;   //上面出栈的 p 已经没用，回去循环取栈顶的下一个元素
            }
        }
    }
}
```

5.20 设计算法将值为 x 的结点作为右子树的（后序序列的）第一个结点的左孩子插入到后序线索二叉树中。

5.21 分别设计出先序、中序和后序线索化算法。

【解】

```
//先序线索化
void preThreading(btNode *&T, btNode *&pre)
{
        //pre 为前一次访问的结点指针（前驱结点指针）；T 为当前正在访问的结点指针
    btNode *lp,*rp;    //T 的左右孩子结点的临时指针
```

```
        if(T)
        {
            lp=T->lChild;   //保留 T 的左右孩子指针，以便后序递归线索化，因为此前左右孩子可能已经成为线索
            rp=T->rChild;
            if(pre && lp==NULL)   //T 的左孩子指针变为前驱线索
            {
                T->lChild=pre;
                T->lTag=1;
            }
            if(pre && pre->rChild==NULL)   //pre 的右孩子指针变为后继线索
            {
                pre->rChild=T;
                pre->rTag=1;
            }
            pre=T;     //更新 pre 为 T
            preThreading(lp,pre);       //递归线索化 T 的左右子树
            preThreading(rp,pre);
        }
    }
    //先序线索化控制函数
    int preThread(btNode *T,btNode *&Tbt)
    {
        btNode *pre=NULL;
        preThreading(T,pre);
        Tbt=T;
        return 1;     //标记先序线索化完成过
    }

    //中序线索化
    void inThreading(btNode *T,btNode *&pre)
    {
        btNode *lp,*rp;
        if(T)
        {
            lp=T->lChild;
            rp=T->rChild;
            inThreading(lp,pre);
            if(pre && T->lChild==NULL)     //T 的左孩子指针变为指向前驱 pre 的线索
            {
                T->lChild=pre;
```

```
            T->lTag=1;
        }
        if(pre && pre->rChild==NULL)     //pre 的右孩子指针变为指向后继 T 的线索
        {
            pre->rChild=T;
            pre->rTag=1;
        }
        pre=T;     //更新 pre 为 T
        inThreading(rp,pre);
    }
}
```

5.22 分别画出下图所示的森林的双亲表示形式、孩子链表表示形式和二叉链表表示形式。

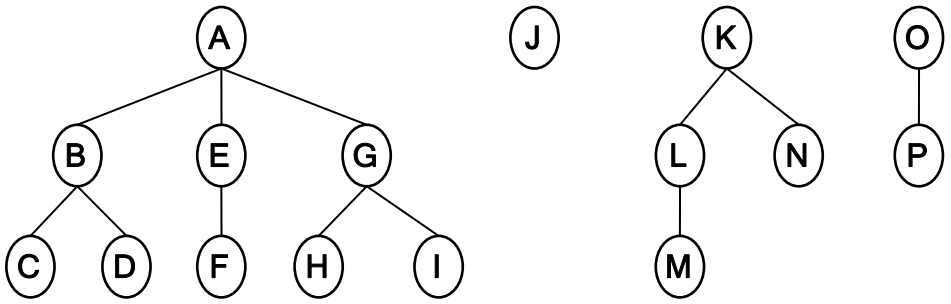


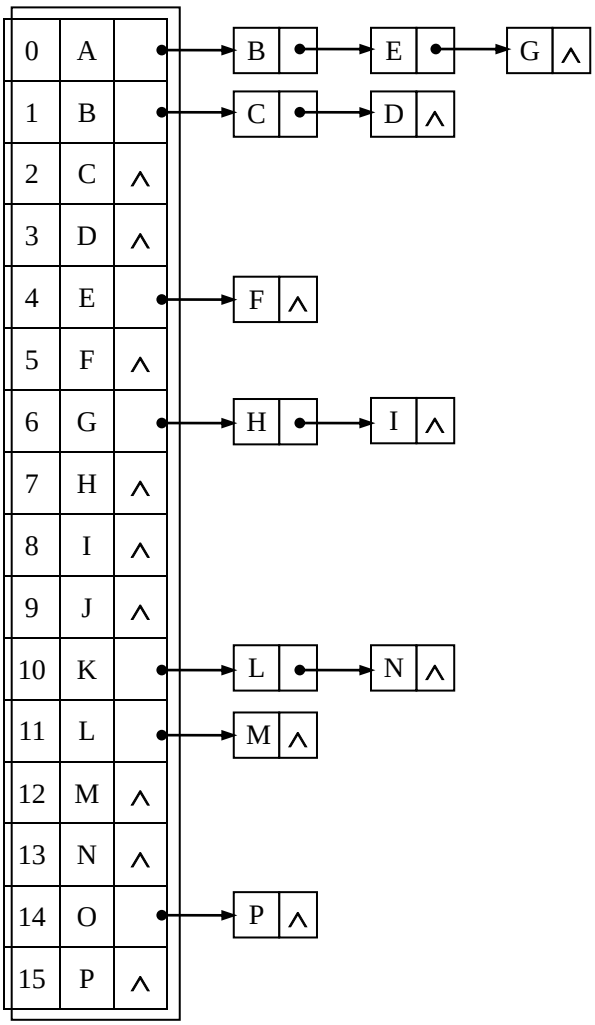
图 5-3　题 5.22 图

【解】

双亲表示：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	0	0	0	1	1	2	3	3	-1	-1	10	10	11	-1	14
A	B	E	G	C	D	F	H	I	J	K	L	N	M	O	P

孩子链表表示：

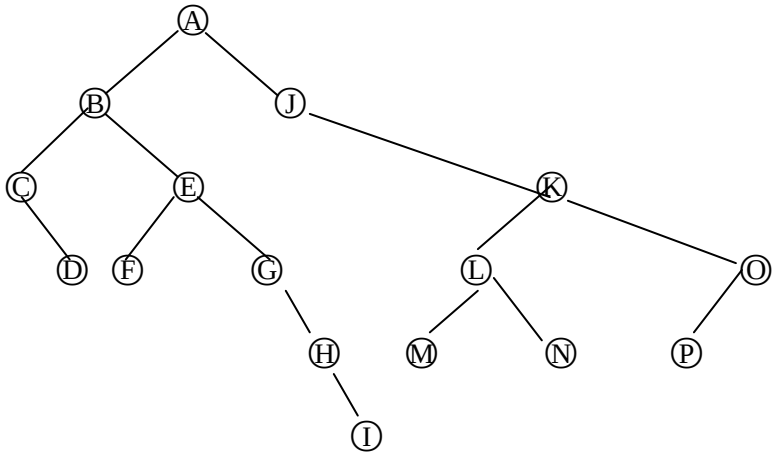


二叉链表表示：

二叉链表表示如下题（题 5.23）解图。

5.23 将图 5-3 中的森林转换为对应的二叉树。

【解】转换后的二叉树如下图



5.24 设计算法以实现将以二叉链表形式存储的树（森林）转换为对应的双亲表示形式。

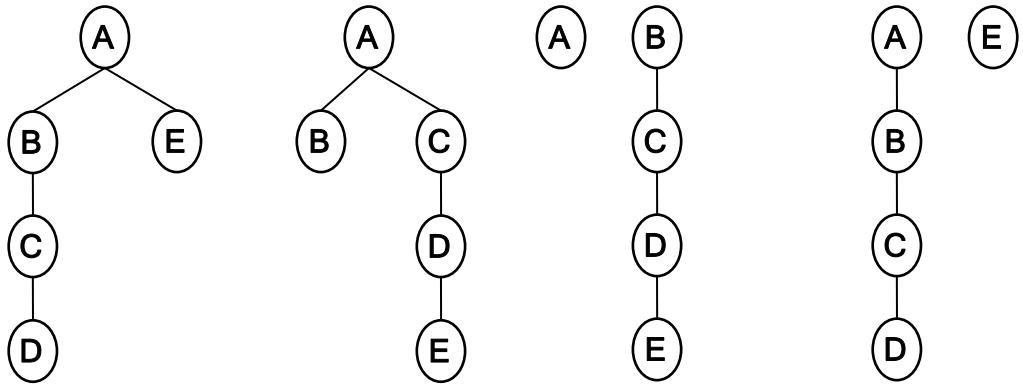
【解】算法思想：改造一种遍历算法，下面算法通过改造先序遍历完成，把双亲结点的下标作为函数参数，初始调用时为-1。

【算法描述】

```
void linkedToParent(csNode *&T,pTree &T1,int &f)
{
                    //f 为双亲结点的下标
    int f1;
    if(T)
    {
        T1.node[T1.n].data=T->data;
        T1.node[T1.n].parent=f;
        f1=T1.n;            //更新第一个孩子结点的双亲
        T1.n++;             //结点数加 1
        linkedToParent(T->firstChild,T1,f1);     //双亲变为 f1
        linkedToParent(T->nextSibling,T1,f);    //双亲仍为 f
    }
}
```

5.25 已知树（森林）的高度为 4，所对应的二叉树的先序序列为 ABCDE，请构造出所有满足这一条件的树或森林。

【解】符合条件的树和森林如下图：



a　　b　　c　　d

图　题 5.25 解图

5.26*设计算法将一个以孩子链表形式表示的森林转换为二叉链表形式。

5.27 将下图中的二叉树转换为对应的森林。

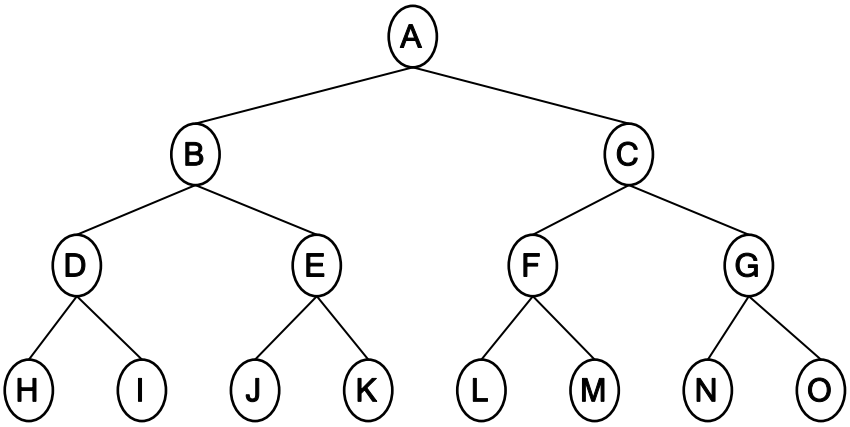


图 5-4　题 5.27 图

【解】对应的森林由四棵树组成，如下图所示。

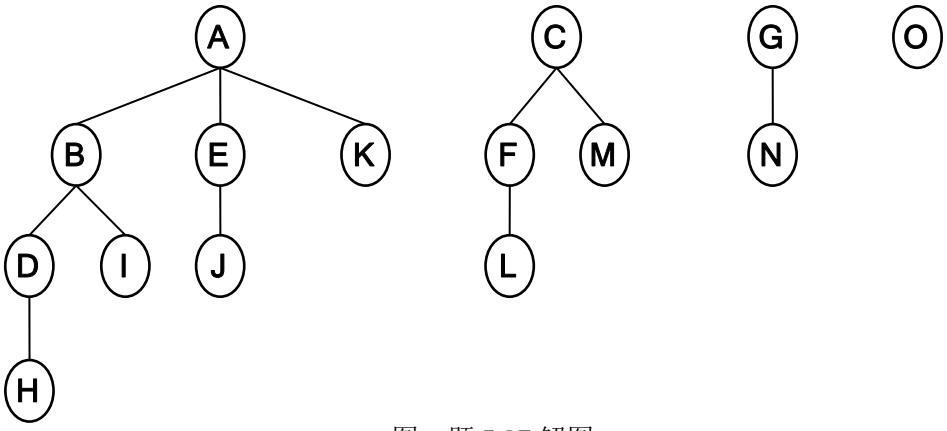


图　题 5.27 解图

5.28 设计算法按先序次序输出森林中每个结点的值及其对应的层次数。

【解】算法思想：改造先序遍历算法，增加一个结点层次参数，当递归遍历 T->firstChild 子树时，结点层次为 T 的层次加 1；递归遍历 T->nextSibling 子树时，结点层次与 T 相同。输出形式：(结点值，层次)。初始调用 outPreOrder(T,1)。假定采用二叉链表结构。

【算法描述】

```
//先序输出结点及层次号
```

```
    void outPreOrder(csNode *T, int level)
    {
        if(T)
        {
            cout<<"("<<T->data<<","<<level<<") ";     //访问根结点
            outPreOrder(T->firstChild,level+1);        //T 的孩子层次为 T 的层次加 1
            outPreOrder(T->nextSibling,level);         //T 的兄弟层次与 T 相同
        }
}
```

5.29 设计算法以求解森林的高度。

【解】算法思想：假定树（森林）以孩子兄弟链表表示。如果森林不空，求出每棵树高度，取最大值返回即森林高度。

【算法描述】

```
int height(csNode *T)
{
    int h,h1;
    if(T==NULL)
        return 0;
    else
    {
        h=1+height(T->firstChild);   //第一棵子树高度
        h1=height(T->nextSibling);   //下一棵子树高度
        if(h>h1)
            return h;
        else
            return h1;
    }
}
```

5.30 设计算法以输出森林中的所有叶子结点的值。

【解】算法思想：二叉链表结构，改造一种遍历算法即可。在二叉链表中叶子结点的条件是：T->firstChild==NULL。

【算法描述】

```
void outLeaf(csNode *T)
{
    if(T)
    {
        if(T->firstChild==NULL)       //叶子结点，输出
            cout<<T->data<<"\t";
        outLeaf(T->firstChild);       //递归搜索第一个孩子子树的叶子
```

```
        outLeaf(T->nextSibling);          //递归搜索下一个兄弟树的叶子
    }
}
```

5.31 设计算法逐层输出森林中的所有结点的值。

【解】算法思想：最外层循环处理森林中每一棵树，在二叉链表结构中，森林中的树通过根结点 T 的 nextSibling 指针找到。利用队列，下面算法定义队列元素包含 2 个分量，一个是结点指针，另一个是结点层次。对每棵树，先将根结点入队。队列不空循环处理，队头出队，找到队头的所有孩子结点入队，方法是通过 firstChild 找到第一个孩子，再通过此孩子结点的 nextSibling 找到所有兄弟。孩子结点的层次在父结点（出队结点）层次上加 1。

【算法描述】

```
void hieOrder(csNode *T)
{
    seqQueue Q;
    eleType s,x;     //队列元素
    csNode *u,*n;
    initialQueue(&Q);
    if(T==NULL)
        return;
    n=T;
    while(n)                    //循环处理森林中每一棵树
    {
        s.p=n;
        s.level=1;
        enQueue(&Q,s);     //根结点入队
        while(!queueEmpty(Q))
        {
            getFront(Q,x);
            cout<<"("<<x.p->data<<","<<x.level<<")    ";       //打印结点及层次
            u=x.p->firstChild;
            while(u)        //队头结点的所有孩子结点 u 入队，层次加 1
            {
                s.p=u;
                s.level=x.level+1;       //双亲结点为 x
                enQueue(&Q,s);
                u=u->nextSibling;       //所有右分支处于同一层
            }
            outQueue(&Q,x);              //队头出队
        }
        n=n->nextSibling;                   //指向森林中下一棵树
    }
```

}

5.32*设计算法将森林中的结点以广义表的形式输出。例如，下图中的森林的输出结果为：

(A,((B,((E,(K)),F,G)),(C,(H,I)),(D,(J)))) , (L,(M,N)), (O,(P))

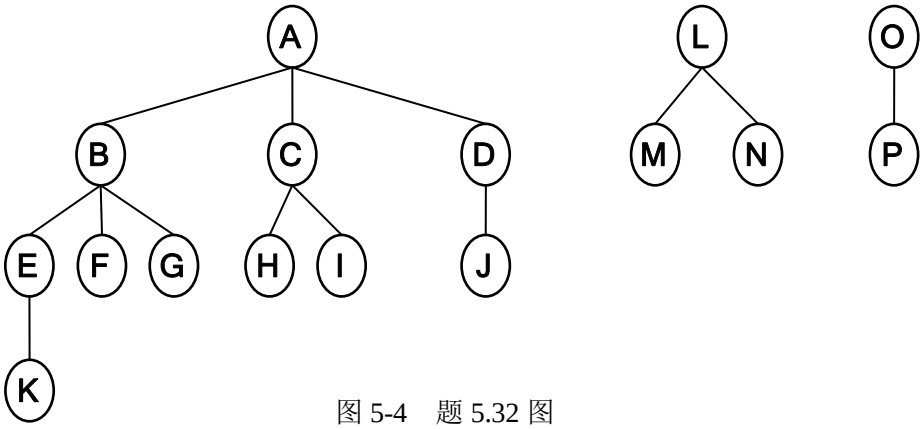


图 5-4　题 5.32 图

【解】算法思想：改造先序遍历算法完成，访问结点为打印结点值，判断如果 firstChild 不为空，说明结点有子树递归遍历，但在遍历之前先要打印“(”；判断 nextSibling 是否为空，不空时，要打印“,”，为空时打印“)”表示子树遍历结束。这个算法有个缺点：根结点层次最后会多出一个“)”，需要手工在调用函数前打印一个“(”，表示整个树（森林）。

【算法描述】

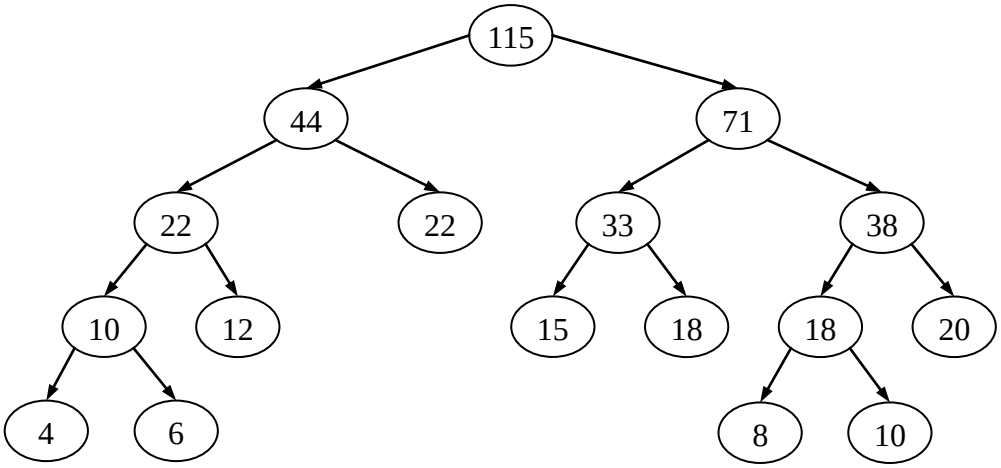
```
void outGList(csNode *T)
{
    if(T)
    {
        cout<<T->data;          //访问根结点
        if(T->firstChild)
        {
            cout<<"(";          //T 有孩子结点，打印子表开始标记
            outGList(T->firstChild);
        }
        if(T->nextSibling)
        {
            cout<<",";          //T 有兄弟结点，打印“,”分割
            outGList(T->nextSibling);
        }
        else                    //T 的兄弟结点遍历结束，打印子表结束标记“)”
            cout<<")";
    }
}
```

5.33 以数据集合{4,6,8,10,12,15,18,20,22}中的元素为叶子结点的权值构造一棵哈夫曼树，并计算其带权路径长度。

【解】

WPL=115+44+71+22+33+38+10+18=351

=4*(4+6+8+10)+3*(12+15+18+20)+2*22=351



5.34 已知一个文件中仅有 10 个不同的字符，各字符出现的个数分别为100,150,180,200,260,300,350,390,400,500。试对这些符号重新编码，以压缩文件的规模，并求出其压缩后的规模以及压缩比（压缩前后的规模比）。

【解】

解：采用哈夫曼编码，根据题意得到如下哈夫曼树和哈夫曼编码。

等长编码：

若采用等长编码，10 个不同字符需要 4 位编码，

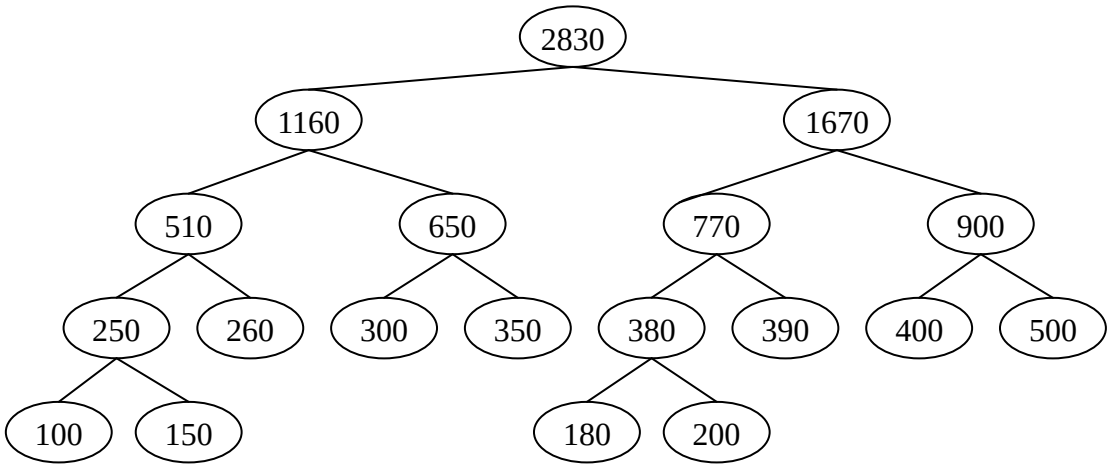
则总码长度=4*(100+150+180+200+260+300+350+390+400+500)=11320

Haffman 编码，码长=WPL

WPL=2830+1160+1670+510+650+770+900+250+380=9120

=4*(100+150+180+200)+3*(260+300+350+390+400+500)=9120

压缩比=9120/11320=80.6%



5.35 设计一个程序以对文件进行压缩，计算其压缩比例，然后对所压缩的文件进行还原。

【解】

```
//文本文件哈夫曼编码
void fileHfmCoding()
{
    ifstream fileIn;
    ofstream fileOut;
    string fileName;
    string str,rstr;

    int n=0,k,i;                //n 为叶子结点个数
    int mid;                    //二分查找时，中间元素的下标
    int min1,min2;              //生成哈夫曼树时选取的两个权值最小的根结点下标
    hfmNode T[MAXLEN];          //哈夫曼树存放数组
    char **hfmCodeList;         //哈夫曼编码表
    weightType x;
    char ch;

    cout<<"请输入要进行哈夫曼编码的文件名（.txt）：";
    cin>>fileName;

    fileIn.open(fileName.c_str());
    if(!fileIn)
    {
        cout<<"文件打开错误。"<<endl;
        return;
```

```
}
str="";        //保存文本源文
while(!fileIn.eof())
{
    fileIn.get(ch);
    mid=BiSearch(T,ch,n);      //T 中二分查找当前元素 ch
    if(mid!=-1)
        T[mid].weight++;        //元素 ch 在哈夫曼树中，权值加 1
    else
    {                                   //元素不在 T[]中，按元素编码递增插入
        k=n-1;
        while(k>=0 && T[k].data>ch)   //比 ch 大的元素后移
        {
            T[k+1]=T[k];
            k--;
        }
                //找到插入位置 k+1，插入元素 ch
        T[k+1].data=ch;
        T[k+1].weight=1;
        T[k+1].parent=-1;
        T[k+1].lChild=-1;
        T[k+1].rChild=-1;
        n++;      //叶子结点数加 1
    }
    str=str+ch;
    //cout<<ch;
}
    //生成哈夫曼树
k=n;
while(k<2*n-1)              //合并 2 棵子树为一棵子树
{
    min1=-1;
    min2=-1;
    x=INF;                    //初始化权值为无穷大
    for(i=0;i<k;i++)     //搜索未合并子树第一个最小权值根结点
    {
        if(T[i].weight<x && T[i].parent==-1)
        {
            min1=i;
            x=T[i].weight;
        }
```

```
        }
        x=INF;
        for(i=0;i<k;i++)     //搜索未合并子树第二个最小权值根结点
        {
            if(T[i].weight<x && T[i].parent==-1 && min1!=i)
            {
                min2=i;
                x=T[i].weight;
            }
        }
        if(min1==-1 || min2==-1)
            break;
             //合并 2 棵根结点权值最小子树
        T[k].weight=T[min1].weight+T[min2].weight;
        T[k].lChild=min1;
        T[k].rChild=min2;
        T[k].parent=-1;
        T[min1].parent=k;
        T[min2].parent=k;
        k++;
    }
            //生成哈夫曼编码表
    hfmCoding(T,hfmCodeList,n);
            //对读入的文本进行编码
    hfmEncode(T,hfmCodeList,str,rstr,n);
    cout<<"哈夫曼编码文本："<<endl;
    cout<<rstr<<endl<<endl;
            //保存用于解码的哈夫曼树，文件扩展名".hfm"
    fileName=fileName.substr(0,fileName.rfind('.'));     //反向找文件扩展名的分界符'.'
    fileName=fileName+".hfm";     //生成哈夫曼树文件名
    //cout<<fileName<<endl;

    fileOut.open(fileName.c_str());
    if(!fileOut)
    {
        cout<<"保存哈夫曼树出错。"<<endl;
        fileIn.close();
        fileOut.close();
        return;
    }
            //哈夫曼树存入文件
```

```
        fileOut<<"[Huffman Tree]\n";
        for(i=0;i<2*n-1;i++)
        {
            fileOut<<(int)T[i].data   <<" "    //转为 ASCII 码存储，考虑到一些不可打印字
符和中文等
                    <<T[i].weight<<" "
                    <<T[i].parent<<" "
                    <<T[i].lChild<<" "
                    <<T[i].rChild<<"\n";
        }
        fileOut.close();
        cout<<"保存哈夫曼树完成。"<<endl;
        cout<<fileName<<endl<<endl;
                //保存哈夫曼编码文件，文件扩展名".cod"
        fileName=fileName.substr(0,fileName.rfind('.'));    //反向找文件扩展名的分界符'.'
        fileName=fileName+".cod";

        fileOut.open(fileName.c_str());
        if(!fileOut)
        {
            cout<<"保存哈夫曼编码文件出错。"<<endl;
            fileIn.close();
            fileOut.close();
            return;
        }

        fileOut<<"[Huffman Coding Text]\n";
                //哈夫曼编码文本存入文件
        fileOut<<rstr;
                //哈夫曼译码
        //str="";
        //hfmDecode(T,rstr,str,n);
        //cout<<"哈夫曼译码文本："<<endl;
        //cout<<str<<endl<<endl;

        cout<<"文本文件哈夫曼编码完成。"<<endl;
        cout<<fileName<<endl<<endl;

        fileIn.clear();
        fileOut.clear();
        fileIn.close();
```

```
    fileOut.close();
}

//文件哈夫曼解码
void fileHfmDecoding()
{
    string fileName,fileName1;
    ifstream fileIn,f1;      //编码文件
    ofstream fileOut;        //解码文件
    string str,rstr;
    hfmNode T[MAXLEN];       //保存哈夫曼树
    int i,n,p;
    int d;

    cout<<"请输入哈夫曼编码文件名（*.cod）: ";
    cin>>fileName;
            //生成对应哈夫曼树存放文件名
    fileName1=fileName.substr(0,fileName.rfind('.'));    //反向找文件扩展名的分界符'.'
    fileName1=fileName1+".hfm";
            //读入哈夫曼树数据
    fileIn.open(fileName1.c_str());
    if(!fileIn)
    {
        cout<<"载入哈夫曼树出错。"<<endl;
        fileIn.close();
        fileOut.close();
        return;
    }
            //判定是否哈夫曼树
    getline(fileIn,str);
    if(str.find("[Huffman Tree]")==-1)
    {
        cout<<"哈夫曼树文件类型错误。"<<endl;
        fileIn.close();
        fileOut.close();
        return;
    }

    n=0;
    while(!fileIn.eof())
    {
```

```
        fileIn>>d;
        T[n].data=(char)d;          //从 ASCII 码转换为字符
        fileIn>>T[n].weight;
        fileIn>>T[n].parent;
        fileIn>>T[n].lChild;
        fileIn>>T[n].rChild;
        n++;
}

cout<<"打印哈夫曼树："<<endl;
cout<<"数据\t"<<"权值\t"<<"父结点\t"<<"左孩子\t"<<"右孩子\t"<<endl;
                //打印哈夫曼树
for(i=0;i<n-1;i++)
{
        cout<<T[i].data<<"\t"
            <<T[i].weight<<"\t"
            <<T[i].parent<<"\t"
            <<T[i].lChild<<"\t"
            <<T[i].rChild
            <<endl;
}
cout<<endl;

fileIn.clear();     //清楚标志位
fileIn.close();     //关闭文件流
                //读入哈夫曼编码数据
fileIn.open(fileName.c_str());
if(!fileIn)
{
        cout<<"载入哈夫曼编码数据出错。"<<endl;
        fileIn.clear();
        fileIn.close();
        fileOut.close();
        return;
}
                        //判定是否哈夫曼编码
getline(fileIn,str);
if(str.find("[Huffman Coding Text]")==-1)
{
        cout<<"哈夫曼编码文件类型错误。"<<endl;
        fileIn.close();
```

```
            fileOut.close();
            return;
        }
        fileIn>>str;       //哈夫曼编码文本读入到字符串 str。
                           //打印哈夫曼编码文本
        cout<<"哈夫曼编码文本："<<endl;
        cout<<str<<endl<<endl;
                           //以下开始哈夫曼解码
        i=0;
        rstr="";           //存放解码后的文本
        p=n-2;             //p 指向哈夫曼树根结点
        while(i<str.length())    //依次取出连续的哈夫曼码，从根结点开始，0 走左分支；1
走右分支，直到叶子结点，即为解码字符
        {
            if(str[i]=='0')
                p=T[p].lChild;
            else
                p=T[p].rChild;
            if(T[p].lChild==-1 && T[p].rChild==-1)
            {
                rstr=rstr+T[p].data;
                p=n-2;     //p 指向哈夫曼树根结点
            }
            i++;
        }
        cout<<"哈夫曼解码文本："<<endl;
        cout<<rstr<<endl<<endl;        //打印解码文件

        cout<<"哈夫曼文本解码完成。"<<endl<<endl;

        fileIn.clear();
        fileOut.clear();
        fileIn.close();
        fileOut.close();
    }
```

5.36 设计算法以产生哈夫曼树中各叶子结点的哈夫曼编码。

【解】

哈夫曼树存放在一维数组中，元素的结构定义如下；

```
typedef struct hfmNode
{
```

```
    elementType data;                //元素值
    weightType weight;               //结点权值
    int parent,lChild,rChild;        //父结点、左右孩子结点指针
}hfmNode;


//哈夫曼编码表
void hfmCoding(hfmNode T[],char **&hfmCodeList,int n)
{
        //输出 n 个叶子结点的哈夫曼编码表 ， n 为叶子结点数
        // T[]中存放顺序存储的哈夫曼树
        // hfmCodeList[][]输出所有叶子结点的哈夫曼编码
    hfmCodeList=new char*[n];       //n 个哈夫曼编码字符串
    char *cd=new char[n];           //临时存放编码的数组

    int i,j,p,pf;
    for(i=0;i<n;i++)          //每个叶子结点往根结点方向搜索确定哈夫曼编码
    {
        pf=T[i].parent;     //当前结点 p 的父结点
        p=i;                    //p 指当前结点，从当前处理的叶子开始
        j=0;
        while(pf!=-1)
        {
            if(T[pf].lChild==p)
                cd[j]='0';
            else
                cd[j]='1';

            j++;
            p=pf;
            pf=T[pf].parent;
        }
             //哈夫曼编码写入编码表
        hfmCodeList[i]=new char[j];
        for(p=0;p<=j-1;p++)
            hfmCodeList[i][p]=cd[j-1-p];;
        hfmCodeList[i][j]='\0';        //标记字符串结束
    }

    //打印哈夫曼编码表
    cout<<"哈夫曼编码表："<<endl;
```

```
    for(i=0;i<n;i++)
    {
        cout<<T[i].data<<"\t"<<T[i].weight<<":\t";
        for(j=0;j<strlen(hfmCodeList[i]) && hfmCodeList[i][j!='\0'];j++)
            cout<<hfmCodeList[i][j];
        cout<<endl;
    }

    delete cd;
}
```