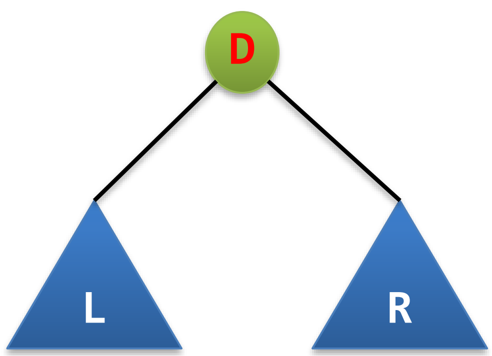
6.3 二叉树先序、中序和后序遍历

6.3.1 二叉树遍历的概念

- 二叉树遍历是指按照一定次序访问二叉树中所有结点，并且每个结点仅被访问一次的过程。
- 设D表示根结点，L、R分别表示左、右子树，则

  有6种遍历方法：DLR，LDR，LRD，DRL，RDL，RLD



若规定先遍历左子树，后遍历右子树：

DLR：先序遍历

LDR：中序遍历

LDN：后序遍历

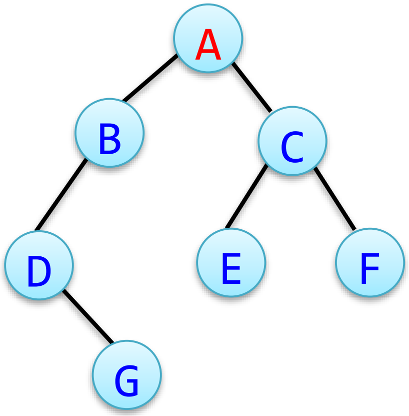
另外，树也可采用层序遍历！

1）先序遍历

①　访问根结点。

②　先序遍历左子树。

③　先序遍历右子树。



先序序列为：ABDGCEF

说明

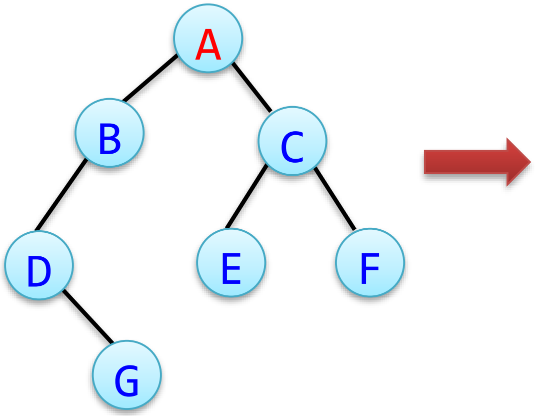
在一棵二叉树的先序序列中，第一个元素即为根结点对应的结点值。

2）中序遍历

① 中序遍历左子树。

② 访问根结点。

③ 中序遍历右子树。



中序序列为：**DGBAECF**。

说明

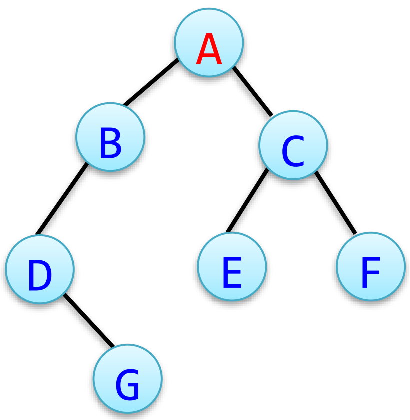
在一棵二叉树的中序序列中，根结点值将其序列分为两部分，前部分为左子树的中序序列，后部分为右子树的中序序列。

3）后序遍历

① 后序遍历左子树。

② 后序遍历右子树。

③ 访问根结点。



后序序列为：**GDBEFCA**。

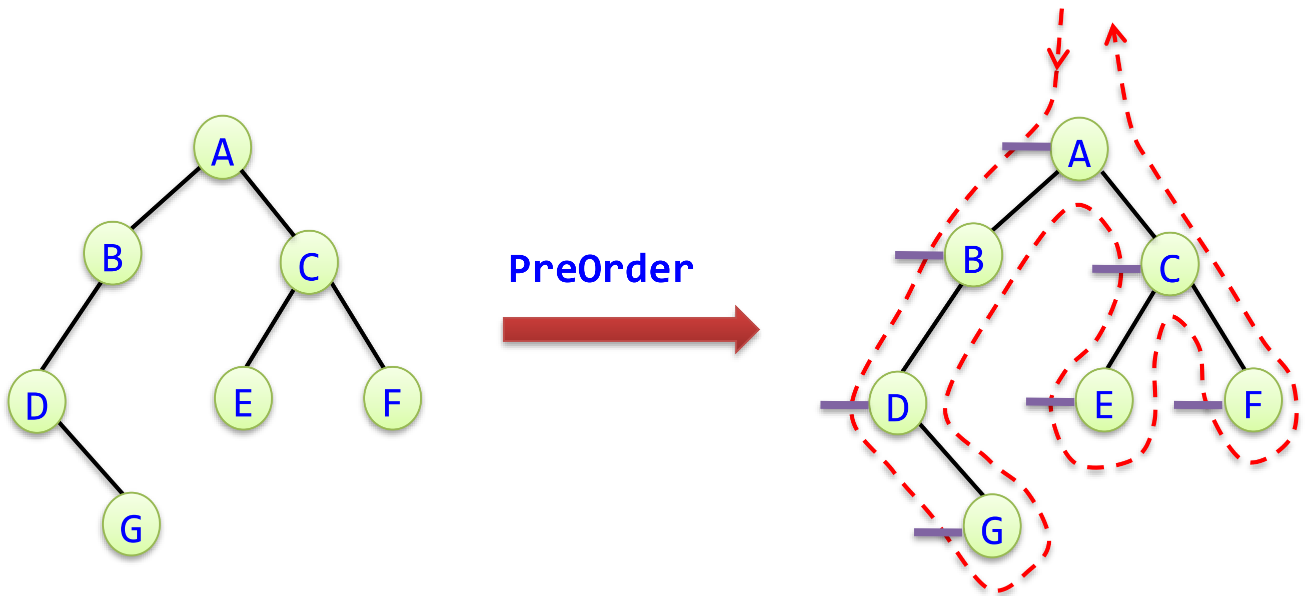
说明

在一棵二叉树的后序序列中，最后一个元素即为根结点对应的结点值。

6.3.2 先序、中序和后序遍历递归算法

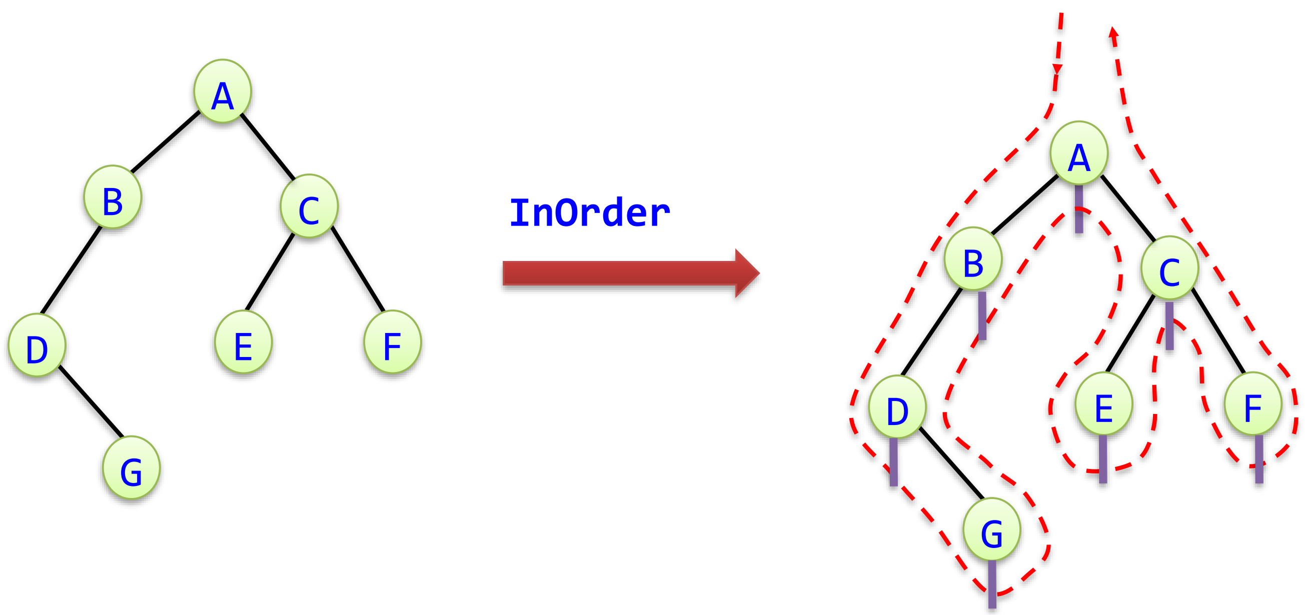
1）先序遍历的递归算法

```
def PreOrder(bt):                       #先序遍历的递归算法
  _PreOrder(bt.b)

def _PreOrder(t):                       #被PreOrder方法调用
  if t!=None:
    print(t.data,end=' ')               #访问根结点
    _PreOrder(t.lchild)                 #先序遍历左子树
    _PreOrder(t.rchild)                 #先序遍历右子树
```


A
B
C
D
E
F
G
PreOrder
A
B
C
D
E
F
G

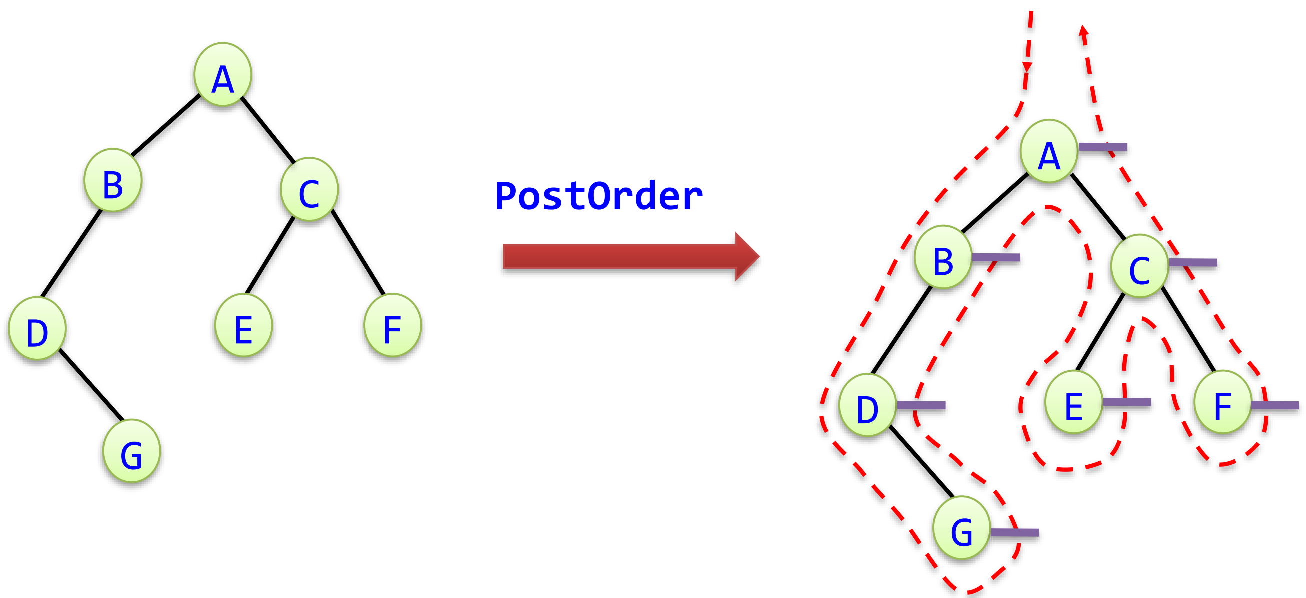
2）中序遍历的递归算法

```
def InOrder(bt):                         #中序遍历的递归算法
  _InOrder(bt.b)

def _InOrder(t):                         #被InOrder方法调用
  if t!=None:
    _InOrder(t.lchild)                   #中序遍历左子树
    print(t.data,end=' ')                #访问根结点
    _InOrder(t.rchild)                   #中序遍历右子树
```


A
B
C
D
E
F
G
InOrder
A
B
C
D
E
F
G

3）后序遍历的递归算法

```
def PostOrder(bt):                        #后序遍历的递归算法
  _PostOrder(bt.b)

def _PostOrder(t):                        #被PostOrder方法调用
  if t!=None:
    _PostOrder(t.lchild)                  #后序遍历左子树
    _PostOrder(t.rchild)                  #后序遍历右子树
    print(t.data,end=' ')                 #访问根结点
```


A
B
C
D
E
F
G
PostOrder
A
B
C
D
E
F
G

6.3.3 递归遍历算法的应用

【例6.9】假设二叉树采用二叉链存储结构存储，设计一个算法求一棵给定二叉树中的结点个数。

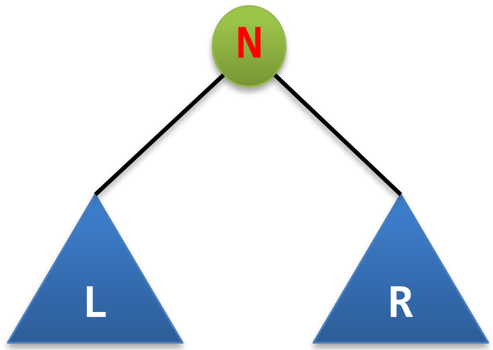
解：求一棵二叉树中的结点个数是以遍历算法为基础的，任何一种遍历算法都可以给出一棵二叉树中的结点个数。

```
def NodeCount1(bt):                  #基于先序遍历求结点个数
  return _NodeCount1(bt.b)

def _NodeCount1(t):
  if t==None:                        #空树结点个数为0
    return 0
  k=1                                #根结点计数1，相当于访问根结点
  m=_NodeCount1(t.lchild)            #遍历求左子树的结点个数
  n=_NodeCount1(t.rchild)            #遍历求右子树的结点个数
  return k+m+n
```

```python
def NodeCount2(bt):                    #基于中序遍历求结点个数
  return _NodeCount2(bt.b)

def _NodeCount2(t):
  if t==None:                          #空树结点个数为0
    return 0
  m=_NodeCount2(t.lchild)              #遍历求左子树的结点个数
  k=1                                  #根结点计数1，相当于访问根结点
  n=_NodeCount2(t.rchild)              #遍历求右子树的结点个数
  return k+m+n
```

```python
def NodeCount3(bt):                     #基于后序遍历求结点个数
  return _NodeCount3(bt.b)

def _NodeCount3(t):
  if t==None:                           #空树结点个数为0
    return 0
  m=_NodeCount3(t.lchild)               #遍历求左子树的结点个数
  n=_NodeCount3(t.rchild)               #遍历求右子树的结点个数
  k=1                                   #根结点计数1，相当于访问根结点
  return k+m+n
```

也可以从递归算法设计的角度来求解。设$f(b)$求二叉树b中所有结点个数，它是“大问题”，f(b.lchild)和f(b.rchild)分别求左、右子树的结点个数。



$f(b)=0$	当b=None
$f(b)=f(b.lchild)+f(b.rchild)+1$	其他情况

$f(b)=0$　　当b=None

$f(b)=f$(b.lchild)+f(b.rchild)+1　　其他情况

```
def NodeCount4(bt):                          #递归求解
  return _NodeCount4(bt.b)

def _NodeCount4(t):
  if t==None:
    return 0                                 #空树结点个数为0
  else:
    return _NodeCount4(t.lchild)+_NodeCount4(t.rchild)+1
```

$f(b)=0$	当b=None
$f(b)=f(\text{b.lchild})+f(\text{b.rchild})+1$	其他情况

其中“+1”相当于访问结点，放在不同位置体现不同的递归遍历思路，NodeCount41()方法是将“+1”放在最后，体现出后序遍历的算法思路。

基于递归遍历思路和直接采用递归算法设计方法完全相同。实际上，当求解问题较复杂时，直接采用递归算法设计方法更加简单方便。

【例6.10】假设二叉树采用二叉链存储结构存储，设计一个算法按从左到右输出一棵二叉树中所有叶子结点值。

解：由于先序、中序和后序递归遍历算法都是按从左到右的顺序访问叶子结点的，所以本题可以基于这三种递归遍历算法求解。

```
def Displeaf1(bt):                    #基于先序遍历输出叶子结点
  _Displeaf1(bt.b)

def _Displeaf1(t):
  if t!=None:
    if t.lchild==None and t.rchild==None:
      print(t.data,end=' ')           #输出叶子结点
    _Displeaf1(t.lchild)              #遍历左子树
    _Displeaf1(t.rchild)              #遍历右子树
```

```python
def Displeaf2(bt):                          #基于中序遍历输出叶子结点
  _Displeaf2(bt.b)

def _Displeaf2(t):
  if t!=None:
    _Displeaf2(t.lchild)                    #遍历左子树
    if t.lchild==None and t.rchild==None:
      print(t.data,end=' ')                 #输出叶子结点
    _Displeaf2(t.rchild)                    #遍历右子树
```

```
def Displeaf3(bt):                      #基于后序遍历输出叶子结点
  _Displeaf3(bt.b)

def _Displeaf3(t):
  if t!=None:
    _Displeaf3(t.lchild)          #遍历左子树
    _Displeaf3(t.rchild)          #遍历右子树
    if t.lchild==None and t.rchild==None:
      print(t.data,end=' ')       #输出叶子结点
```

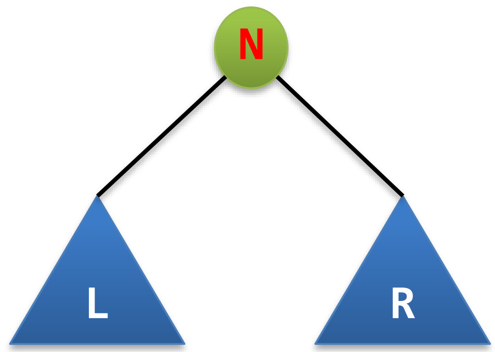
- 也可以直接采用递归算法设计方法求解。
- 设$f(b)$的功能是从左到右输出以b为根结点的二叉树的所有叶子结点值，为"大问题"，显然$f(b.\text{lchild})$和$f(b.\text{rchild})$是两个"小问题"。
- 当b不是叶子结点时，先调用$f(b.\text{lchild})$再调用$f(b.\text{rchild})$。
- 对应的递归模型$f(b)$如下：

$f(b) \equiv$ 不做任何事件	若b=None
$f(b) \equiv$ 输出b结点	若b为叶子结点
$f(b) \equiv f(\text{b.lchild});\ f(\text{b.rchild})$	其他情况

$f(b) \equiv$ 不做任何事件　　若b=None
$f(b) \equiv$ 输出b结点　　若b为叶子结点
$f(b) \equiv f$(b.lchild); f(b.rchild)　　其他情况

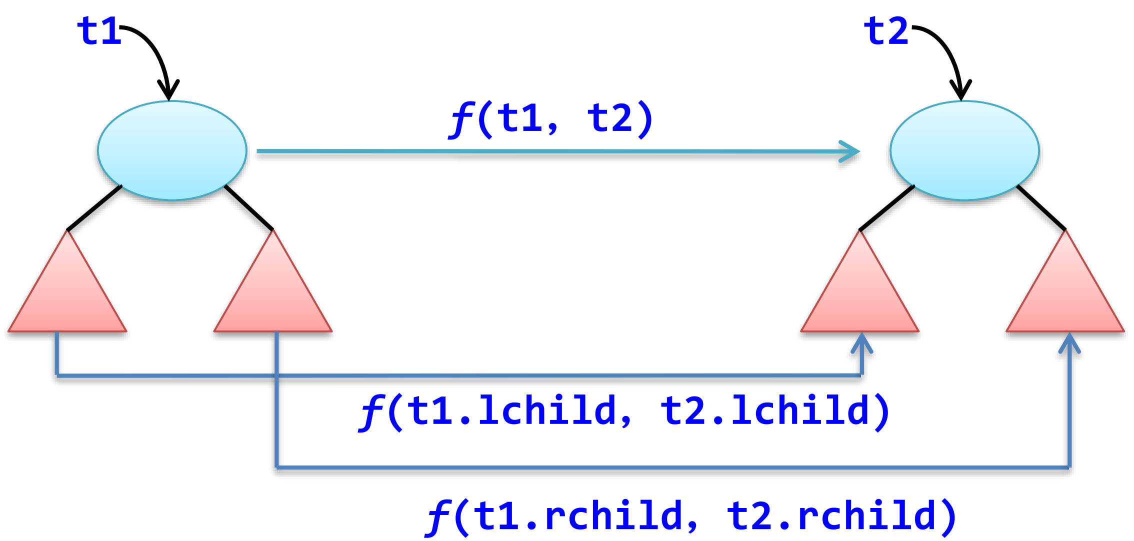
```
def Displeaf4(bt):                               #基于递归算法思路
  _Displeaf4(bt.b)

def _Displeaf4(t):
  if t!=None:
    if t.lchild==None and t.rchild==None:
      print(t.data,end=' ')                      #输出叶子结点
    else:
      _Displeaf4(t.lchild)                       #遍历左子树
      _Displeaf4(t.rchild)                       #遍历右子树
```

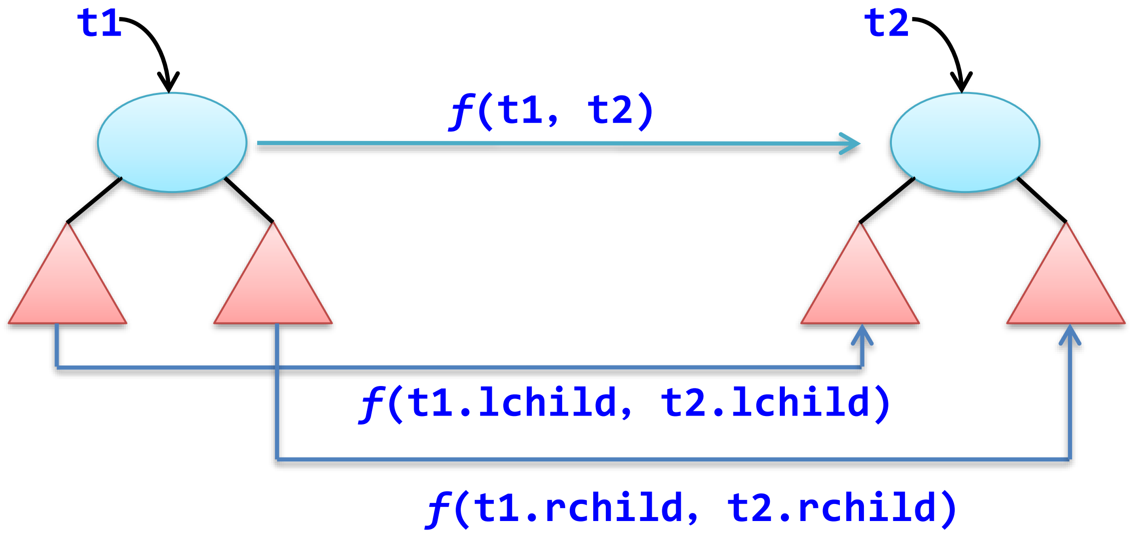
- 从上述两例看出，基于递归遍历思路和直接采用递归算法设计方法完全相同。实际上，当求解问题较复杂时，直接采用递归算法设计方法更加简单方便。
- 仅从递归遍历角度看，上述两例基于3种递归遍历思路中任意一种都是可行的，但有些情况并非如此。
- 一般地，二叉树由根、左右子树3部分构成，但可以看成两类，即根和子树。
- 如果需要先处理根再处理子树，可以采用先序遍历思路。
- 如果需要先处理子树，再处理根，可以采用后序遍历思路。



【例6.11】假设二叉树采用二叉链存储结构存储，设计一个算法将二叉树bt1复制到二叉树bt2。

解：采用直接递归算法设计方法。设f(t1，t2)是由二叉链t1复制产生t2，这是“大问题”。



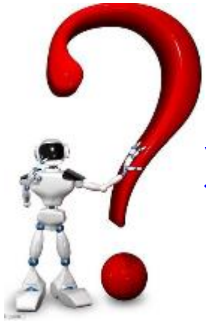


f(t1, t2) ≡ t2=None		当t1=None
f(t1, t2) ≡ 由t1根结点复制产生t2根结点;		当t1≠None
	f(t1.lchild, t2.lchild);	
	f(t1.rchild, t2.rchild);	

```python
def CopyBTree1(bt1):                        #基于先序遍历复制二叉树
  bt2=BTree()
  bt2.SetRoot(_CopyBTree1(bt1.b))
  return bt2

def _CopyBTree1(t1):                        #由t1复制产生t2
  if t1==None:
    return None
  else:
    t2=BTNode(t1.data)                      #复制根结点
    t2.lchild=_CopyBTree1(t1.lchild)        #递归复制左子树
    t2.rchild=_CopyBTree1(t1.rchild)        #递归复制右树
    return t2
```

也可以采用基于后序遍历思路

```
def CopyBTree2(bt1):                #基于后序遍历复制二叉树
  bt2=BTree()
  bt2.SetRoot(_CopyBTree2(bt1.b))
  return bt2

def _CopyBTree2(t1):                #由t1复制产生t2
  if t1==None:
    return None
  else:
    l=_CopyBTree2(t1.lchild)        #递归复制左子树
    r=_CopyBTree2(t1.rchild)        #递归复制左子树
    t2=BTNode(t1.data)              #复制根结点
    t2.lchild=l
    t2.rchild=r
    return t2
```



建议不采用中序遍历思路求解！

【例6.12】假设一棵二叉树采用二叉链存储结构，且所有结点值均不相同，设计一个算法求二叉树中指定结点值的结点所在的层次（根结点的层次计为1）。

解：

- 二叉树中每个结点都有一个相对于根结点的层次，根结点的层次为1，那么如何指定这种情况呢？
- 可以采用递归算法参数赋初值的方法，即设*f*(*b*，*x*，*h*)为“大问题”，增加第3个参数*h*表示第一个参数*b*指向结点的层次，在初始调用时*b*指向根结点，*h*对应的实参为1，从而指定了根结点的层次为1的情况。

$f(b, x, h)=0$	b=None
$f(b, x, h)=h$	当b.data=x
$f(b, x, h)=l$	当$l= f(b$.lchild, x, h+1), 且$l≠0$（在左子树中找到了）
$f(b, x, h)=f$(b.rchild, x, h+1)	其他情况

```python
def Level(bt,x):                                #求解算法
  return _Level(bt.b,x,1)

def _Level(t,x,h):
  if t==None:
     return 0                                   #空树不能找到该结点
  elif t.data==x:
     return h                                   #根结点即为所找,返回其层次
  else:
     l=_Level(t.lchild,x,h+1)                   #在左子树中查找
     if l!=0:
       return l                                 #左子树中找到了,返回其层次l
     else:
       return _Level(t.rchild,x,h+1)            #左子树中未找到,再在右子树中查找
```

递归算法参数赋初值问题

【例6.15】假设二叉树采用二叉链存储结构，且所有结点值均不相同，设计一个算法输出值为x的结点的所有祖先。

解法1

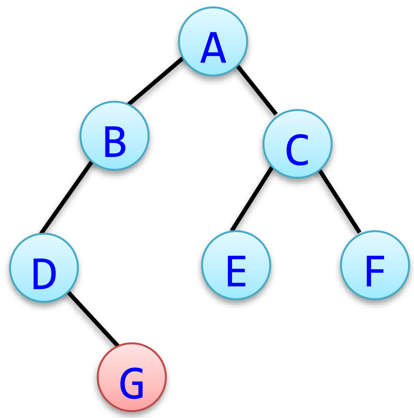
- 根据二叉树中祖先的定义可知，若一个结点的左孩子或右孩子值为*x*时，则该结点是*x*结点的祖先结点；若一个结点的左孩子或右孩子为*x*结点的祖先结点时，则该结点也为*x*结点的祖先结点。
- 设*f*(*t*，*x*)表示*t*结点是否为*x*结点的祖先结点。

```
f(b，x) = false                      若b==None
f(b，x) = true，并输出b结点          若b结点有值为x的左孩子结点
f(b，x) = true，并输出b结点          若b结点有值为x的右孩子结点
f(b，x) = true，并输出b结点          若f(b.lchild，x)为true或
                                          f(b.rchild，x)为true
f(b，x) = false                      其他情况
```

```
def Ancestor1(bt,x):                         #算法1：返回x结点的祖先
   res=[]                                    #存放祖先
   _Ancestor1(bt.b,x,res)
   res.reverse()                             #逆置res
   return res

def _Ancestor1(t,x,res):
   if t==None:                               #空树返回空串
      return False
   if t.lchild!=None and t.lchild.data==x:
      res.append(t.data)                     #t结点是x结点的祖先
      return True
   if t.rchild!=None and t.rchild.data==x:
      res.append(t.data)                     #t结点是x结点的祖先
      return True
   if _Ancestor1(t.lchild,x,res) or _Ancestor1(t.rchild,x,res):
      res.append(t.data)               #t结点的孩子是x的祖先，则t也是x的祖先
      return True
   return False                              #其他情况返回False
```

解法2

- 二叉树中x结点的祖先恰好是根结点到x结点的路径上除了x结点外的所有结点，用全局变量res列表表示。
- 采用先序遍历的思路，采用一个path列表存放路径，当找到x结点时，将path中x结点（最后添加的结点）删除，再将path复制的res中。

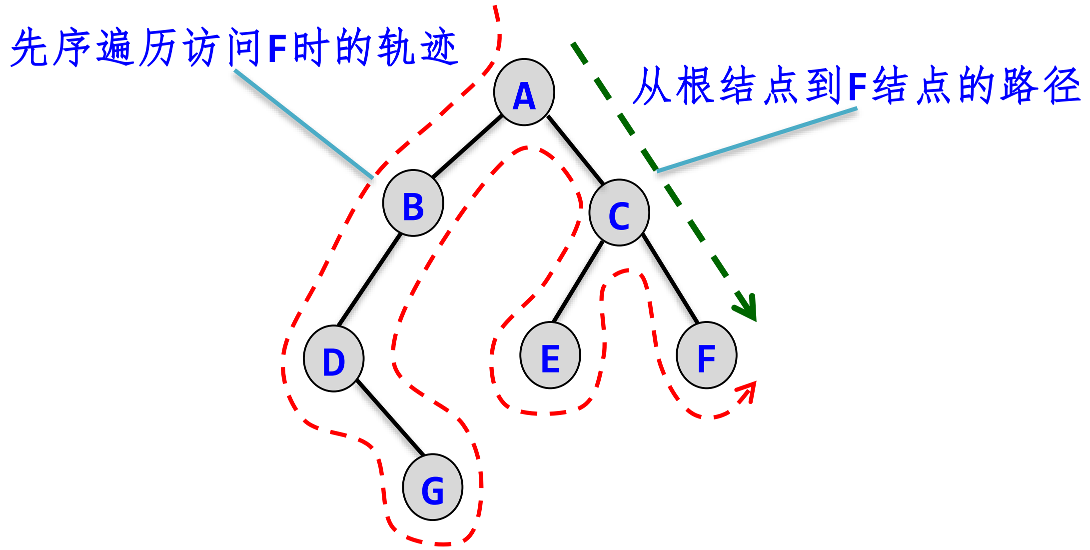


G的祖先：A B D

```python
res=[]                                  #全局变量,存放祖先
def Ancestor2(bt,x):                    #算法2：返回x结点的祖先
   global res
   path=[]
   res=[]
   _Ancestor2(bt.b,x,path)
   return res                           #返回祖先列表res

def _Ancestor2(t,x,path):
   global res
   if t==None: return                   #空树返回
   path.append(t.data)
   if t.data==x:
      path.pop()                        #删除x结点
      res=copy.deepcopy(path)           #深复制，若改为res=path结果是错误的!
      return                            #找到后返回
   _Ancestor2(t.lchild,x,path)          #在左子树中查找
   _Ancestor2(t.rchild,x,path)          #在右子树中查找
   path.pop()    #x结点处理完毕，回退
```

?

在方法_Ancestor2(*t*，*x*，path)中path是可变类型（相当于全局变量）。执行path.append(t.data)语句将当前访问的t结点值添加到path中，如果后面不执行path.pop()回退，则找到*x*结点时path是一个查找轨迹（包含所有遍历中访问的结点）。



改进算法2:

```
def Ancestor3(bt,x):                         #算法3: 返回x结点的祖先
   path=[]
   _Ancestor3(bt.b,x,path)
   return path                               #返回祖先列表res

def _Ancestor3(t,x,path):
   if t==None: return False                  #空树返回
   path.append(t.data)
   if t.data==x:
      path.pop()                             #删除x结点
      return True                            #找到后返回
   if _Ancestor3(t.lchild,x,path) or _Ancestor3(t.rchild,x,path):
      return True                            #在左或者右子树中找到后返回True
   else:                                     #在左或者右子树中都没有找到
      path.pop()                             #x结点处理完毕, 回退
```

改为用path[0..d]存放根到x结点的路径

```
res=[]                                  #全局变量,存放祖先
def Ancestor4(bt,x):                    #算法4：返回x结点的祖先
   global res
   res=[]
   path=[None]*100                      #假设路径长度最大为100
   d=-1
   _Ancestor4(bt.b,x,path,d)
   return res                           #返回祖先列表res

def _Ancestor4(t,x,path,d):
   global res
   if t==None: return False             #空树返回False
   d+=1; path[d]=t.data                 #将t结点值添加到path中
   if t.data==x:
      for i in range(d):                #将path[0..d-1]复制到res中
         res.append(path[i])
      return True                       #找到后返回True
   if _Ancestor4(t.lchild,x,path,d) or _Ancestor4(t.rchild,x,path,d):
      return True                       #在左或者右子树中找到后返回True
```

程序验证

```
#主程序
b=BTNode('A')
p1=BTNode('B');p2=BTNode('C')
p3=BTNode('D');p4=BTNode('E')
p5=BTNode('F');p6=BTNode('G')
b.lchild=p1;b.rchild=p2
p1.lchild=p3;p3.rchild=p6
p2.lchild=p4;p2.rchild=p5
bt=BTree()
bt.SetRoot(b)
print("bt:",end='  ');print(bt.DispBTree())
x='G'
print("解法1 "+x+"的祖先: ",Ancestor1(bt,x))
print("解法2 "+x+"的祖先: ",Ancestor2(bt,x))
print("解法3 "+x+"的祖先: ",Ancestor3(bt,x))
print("解法4 "+x+"的祖先: ",Ancestor4(bt,x))
```