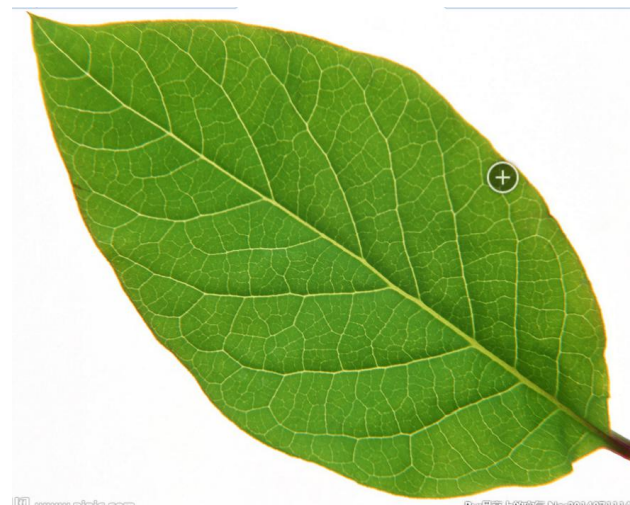
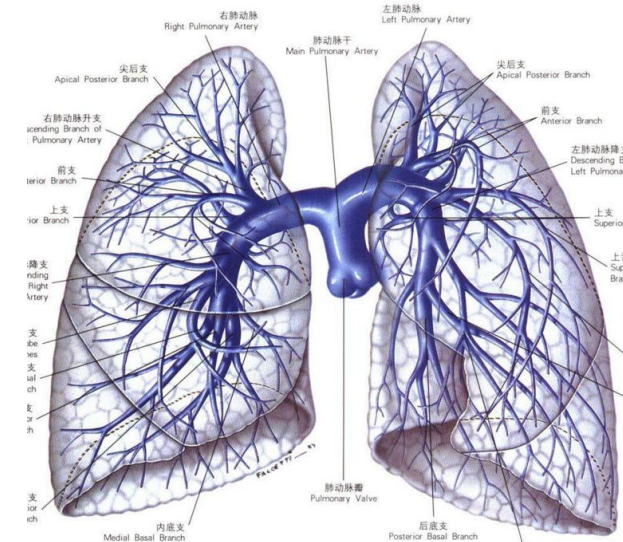
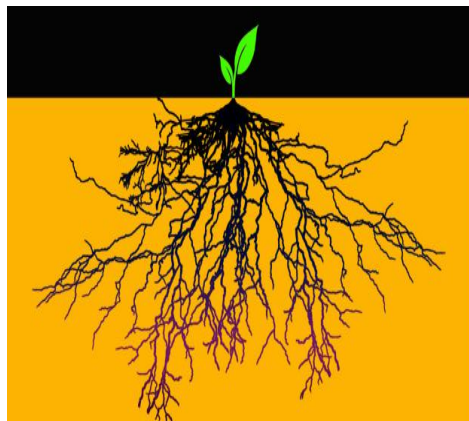
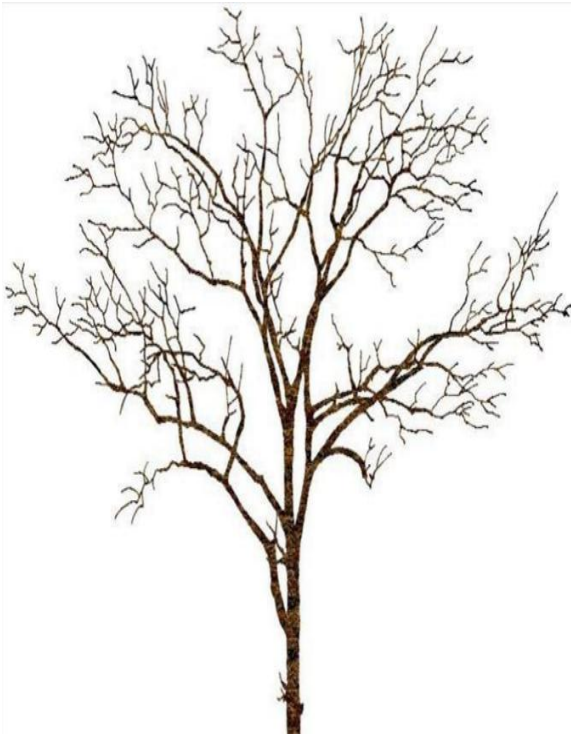


数据结构

Data Structure

2017年秋季学期
刘鹏远

树



- 1、二叉树的层次遍历实现**
- 2、二叉树的递归遍历实现**
- 3、基于递归遍历的应用：
创建、销毁、统计节点数等**

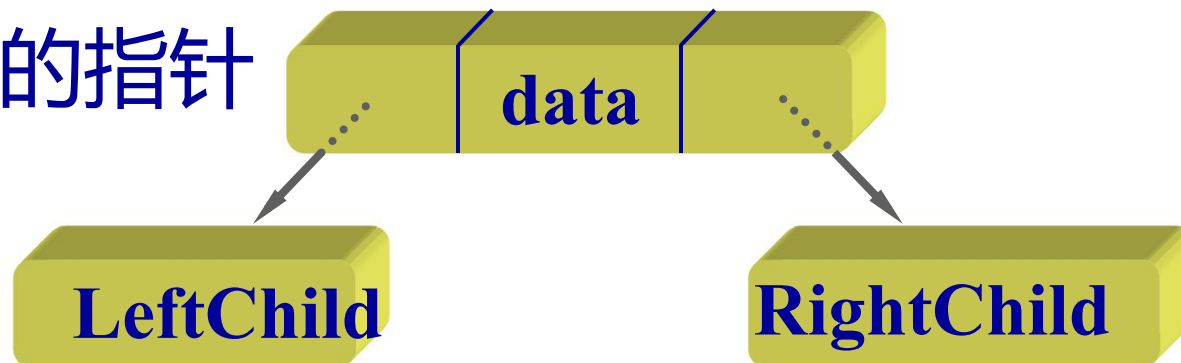
二叉树二叉链表表示与定义

```
typedef struct Node {           //树结点定义
    ElemType data;              //结点数据域
    struct Node *LeftChild, *RightChild; //孩子指针域
} BT_Node, *BinTree;
```



//树定义

//与链表类似，用指向树根的指针



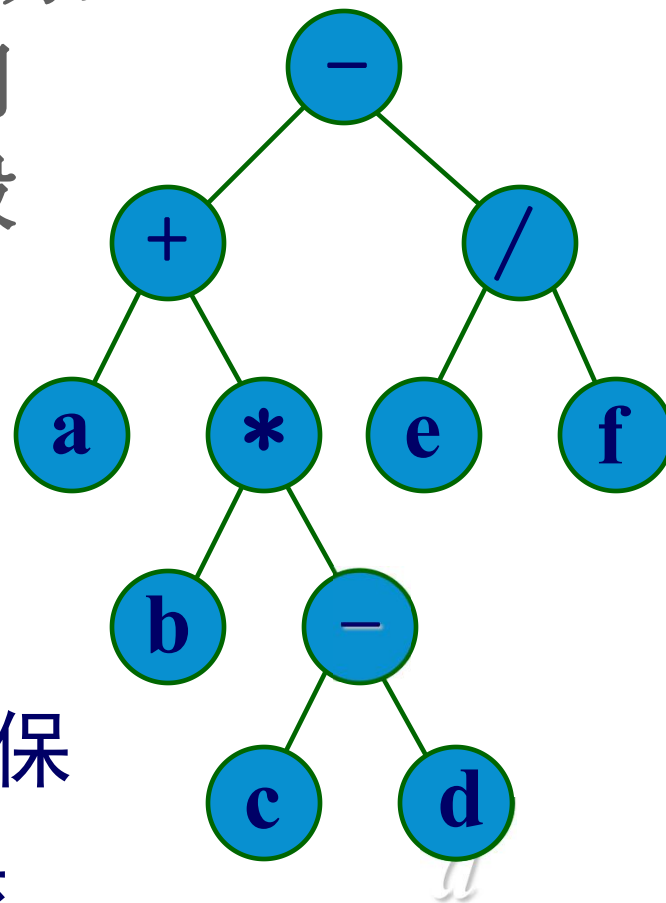
二叉树的层次遍历算法

层次遍历二叉树，是从根结点开始遍历，按层次次序“**自上而下，从左至右**”访问树中的各结点。该遍历非递归定义，一般不采用递归方法。

$- + / a * e f b - c d$

如何实现？

一层层遍历访问，打印/访问当前层节点，保存下一层节点信息。下一层信息由上一层节点提供（像不像杨辉三角？）



二叉树的层次遍历算法

设置一个队列 Q ，初始化时为空。

设 T 是指向根结点的指针变量，层次遍历非递归算法是：

若二叉树为空，则返回；否则，令 $p=T$ ， p 入 Q ；

do (1) 队首元素出队；

(2) 访问该元素所指向的结点；

(3) 将该元素所指向的结点的左、右子结点依次入队。

while(队空)。

(广度优先，后面图的遍历中也会涉及)

```
#define MAX_NODE 50
```

```
void LevelOrder( BinTree T){//伪码
```

```
    Queue Q;InitQueue(&Q) ; BinTree p=T ;
```

```
    if (p){
```

```
        EnQueue(&Q,p); /* 根结点入队 */
```

```
        while (!IsEmpty(Q)){
```

```
            DeQueue(&Q,p); visit( p->data );
```

```
            if (p->LeftChild)
```

```
                EnQueue(&Q,p->LeftChild); /* 左子入队*/
```

```
            if (p->RightChild)
```

```
                EnQueue(&Q,p->RightChild);/*右子入Q*/
```

```
        }}}//visit()可以直接用printf()
```

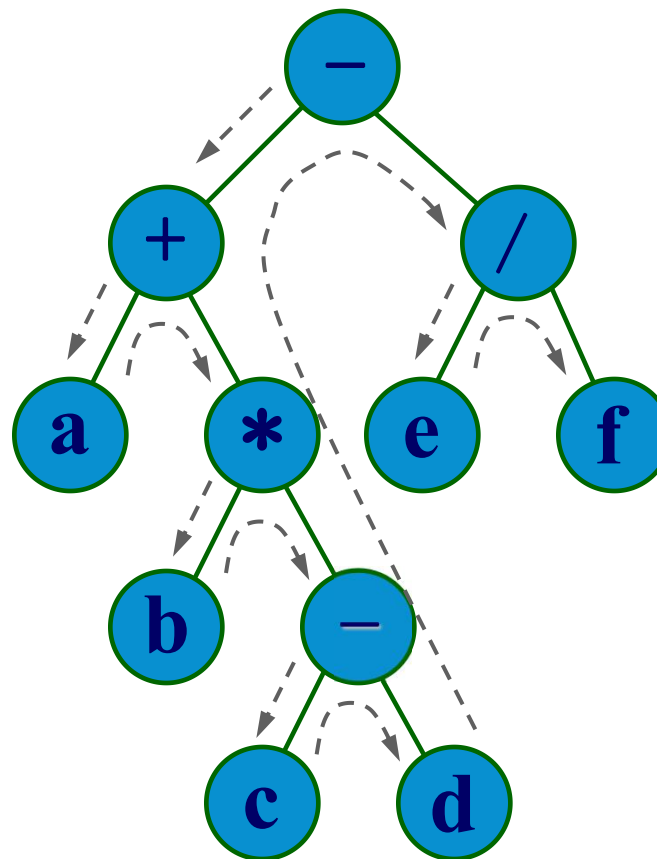
前/先序遍历 (Preorder Traversal)

前序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
 - ◆ 访问根结点 (V)；
 - ◆ 前序遍历左子树 (L)；
 - ◆ 前序遍历右子树 (R)。

遍历结果

$- + a * b - c d / e f$



二叉树递归的前序遍历算法

```
void PreOrder (BinTree T) {  
    if (T) {  
        visit (T->data);  
        PreOrder (T->LeftChild);  
        PreOrder (T->RightChild);  
    }  
}
```

//visit()是输出数据值的操作，实际应用时可用输出信息、修改结点的值及其他计算等各种操作。

对递归的再次思考

对二叉树的递归先序遍历：

写先序遍历算法(假定一定存在且能够被写出)：

PreOrder(T)

{

 如果T为空，则返回（此即递归出口，最终为空树时）

 非空，则访问T

 用这个PreOrder遍历(T->LeftChild)

 用这个PreOrder遍历(T->RightChild)

}

对递归的再次思考

为何可以用这个算法对左/右子树求解？

因为整个求解过程是这样的分解：

访问T <---原子问题

用这个PreOrder遍历(T->LeftChild) <---子问题一

用这个PreOrder遍历(T->RightChild) <---子问题二

原问题分解为规模更小的子问题（二叉树的子树与自身有相似的结构），因此可以用求解原问题的方法求解！

最终均分解为原子问题。

对递归的再次思考

汉诺塔问题：

由Hanoi (n)分解为：

Hanoi (n-1, A, C, B);

move one disk;

Hanoi (n-1, B, A, C);

<---子问题1

原子问题

<---子问题2

最终均分解为原子问题。

对递归的再次思考

迷宫问题问题：

分解为：

从左搜索走

<---子问题1

从下搜索走

<---子问题2

从右搜索走

<---子问题3

从上搜索走

<---子问题4

走一步

<---原子问题

最终均分解为原子问题。停止条件为到达出口。

对递归的再次思考

设计主要考虑如何分解原问题成规模降低的子问题，
或考虑如何由各子问题的求解结果合成原问题。

问题解决的过程就在问题的逐步分解或由子问题逐步
合成原问题的过程中！

问题一旦被正确分解完毕，递归框架即可写毕。

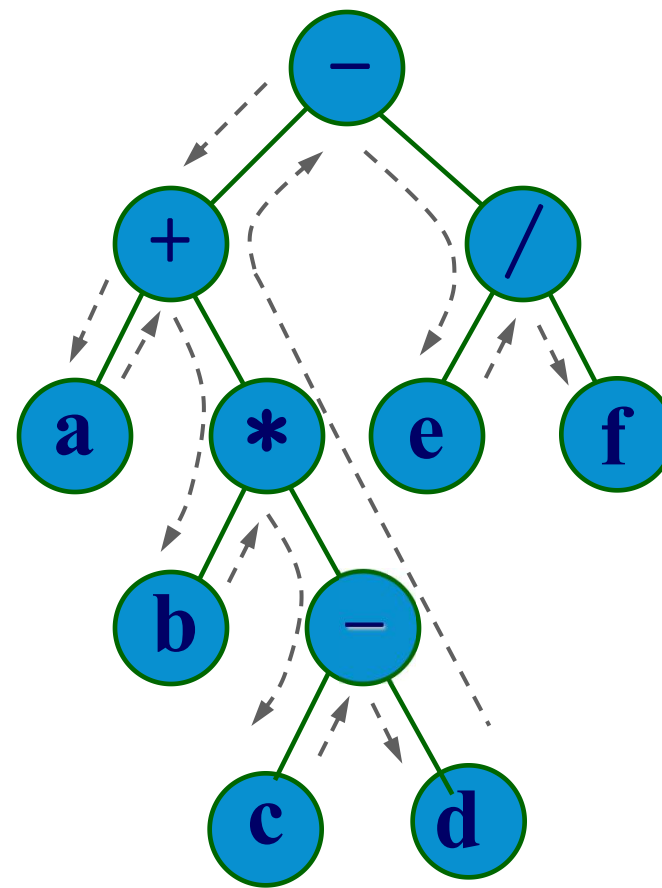
中序遍历 (Inorder Traversal)

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
 - 中序遍历左子树 (L)；
 - 访问根结点 (V)；
 - 中序遍历右子树 (R)。

遍历结果

$a + b * c - d - e / f$



二叉树递归的中序遍历算法

```
void InOrder ( BinTree T ) {  
    if ( T ) {  
        InOrder ( T->LeftChild );  
        visit ( T->data );  
        InOrder ( T->RightChild );  
    }  
}
```

与先序遍历算法相比，**visit()** 操作放在两个子树递归前序遍历的中间。

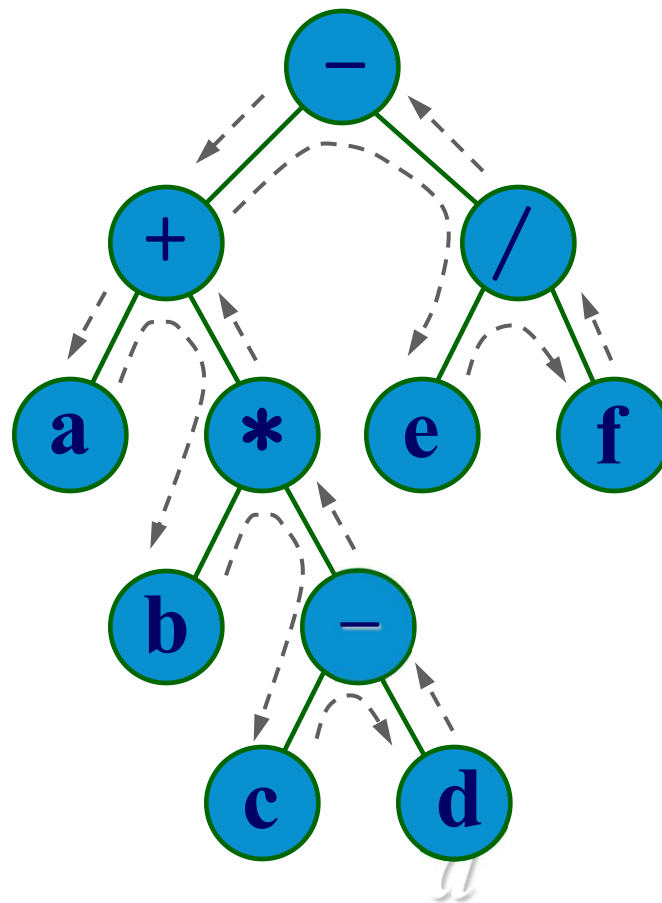
后序遍历 (Postorder Traversal)

后序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
 - ◆ 后序遍历左子树 (L)；
 - ◆ 后序遍历右子树 (R)；
 - ◆ 访问根结点 (V)。

遍历结果

$a b c d - * + e f / -$



二叉树递归的后序遍历算法

```
void PostOrder (BinTree T) {  
    if (T != NULL) {  
        PostOrder (T->leftChild);  
        PostOrder (T->rightChild);  
        visit (T->data);  
    }  
}
```

//与先序遍历算法相比，**visit()** 操作放在两个子树递归前序遍历的最后面。

二叉树递归的后序遍历算法

如果去掉visit，三个递归算法完全相同

从递归执行过程看，三个算法是相同的，区别只在于visit的时机。

二叉树遍历问题目的是遍历，二叉树的结构是递归定义的，分解是一样的：

原子问题（只有一个节点），左子树遍历、右子树遍历。

利用递归遍历的几个应用

1、创建

2、销毁

3、节点数

4、高度

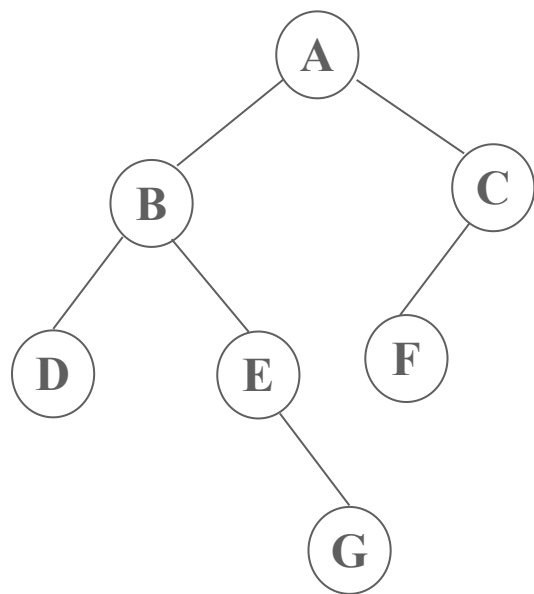
5、删除元素值为x的结点（并销毁其子树）

6、求二叉树先序序列中第k个位置的结点的值

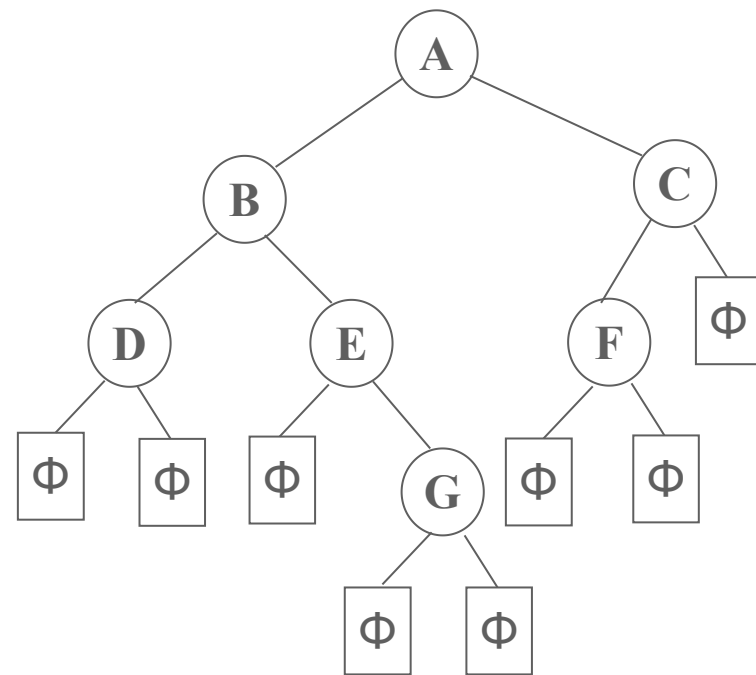
- 分析问题
- 分解问题，选择递归遍历框架（X序）
设置递归出口
- 设计适当VISIT操作
- 合理利用递归调用的返回值

创建CreatTree（递归框架）

扩充的二叉树：对一棵二叉树进行“扩充”（扩充 Φ ），得到由该二叉树所扩充的二叉树。 Φ 即NULL，可利用字符显示标记出来。



(a) 二叉树 T_1



(b) T_1 的扩充二叉树 T_2

对扩充后的二叉树，前序遍历：

ABDΦΦEΦGΦΦCFΦΦΦ

“Φ”代表为NULL，可用任意指定字符，如“#”“?”等。

给定上述遍历序列，计算机依次读入，构造存储二叉树，就实现了二叉树的创建（基于前序遍历）。

如何实现？

算法框架：

```
void PreOrder (BinTree T) {  
    if (T) {  
        visit (T->data);  
        PreOrder (T->LeftChild);  
        PreOrder (T->RightChild);  
    }  
}
```

先序框架

问题分解：

- 1、建立根节点
- 2、建左子树
- 3、建右子树

递归出口 (Φ)

VISIT操作（建立节点）

返回值（非必须）

```
void CreatBinTree(BinTree *T){  
    scanf("%c",&ch);  
    if(ch=='?')    *T=NULL;  
    else{  
        *T=(BTNode*)malloc(sizeof(BTNode));  
        if(*T==NULL) exit(OVERFLOW);  
        (*T)->data = ch;  
        CreatBinTree(&((*T)->left_child));  
        CreatBinTree(&((*T)->right_child));  
    }  
}
```


中序、后序创建的该如何实现？请大家思考，回答！

```
*T=(BTNode*)malloc(sizeof(BTNode));  
if(*T==NULL) exit(OVERFLOW);  
(*T)->data = ch;  
CreatBinTree(&((*T)->left_child));  
CreatBinTree(&((*T)->right_child));
```

结论是不能用中序、后序创建

二叉树的销毁

1、问题分解

销毁根、销毁左子树、销毁右子树{

2、选择递归框架

后序框架最好(why?)

3、设置出口条件

节点==NULL

4、VISIT操作

free, NULL

5、返回值利用

无需

```
void PostOrder (BinTree T)
{
    if (T != NULL) {
        PostOrder (T->leftChild);
        PostOrder (T->rightChild);
        visit (T->data);
    }
}
```

二叉树的销毁

```
void DestroyTree(BinTree *T){  
    if(*T){  
        DestroyTree(&((*T)->LeftChild));  
        DestroyTree(&((*T)->RightChild));  
        free(*T);          *T = NULL;  
    }  
}
```

删除并释放二叉树中以元素值为x的结点作为根的各子树

DelXTree()整体与前面的Destory类似，
可分解为：

1/对当前节点进行比较， Destory或者什么也不做

2/DelXTree(左子树)

3/DelXTree(右子树)

出口：当前节点为NULL

三种递归框架哪种呢？实际上，都可以用。但有一个最优。

DelXTree 后序

```
void DelXTree(BinTree *T, ElemType x)
{ // 基于后序的查找
  if ( *T != NULL ) {
    DeleteXTree((*T)->lchild, x);
    DeleteXTree((*T)->rchild, x);
    if (( *T )->data == x ) DestroyTree(*T);
  }
}
```

DelXTree 中序

```
void DelXTree(BitTree *T, ElemType x)
{ // 基于中序的查找
  if ( *T != NULL ){
    deleteXTree((*T)->lchild, x);
    if ((*T)->data== x ) DestroyTree(*T);
    else deleteXTree((*T)->rchild, x);
  }
}
```

DelXTree 先序

```
void DelXTree(BitTree *T, ElemType x)
{ // 基于先序的查找
  if ( *T != NULL ) {
    if (( *T )->data == x ) DestroyTree(*T);
    else {
      deleteXTree(( *T )->lchild, x);
      deleteXTree(( *T )->rchild, x);
    } } } // 最优, 前两个在 T->data == x 时, 均有多余步骤
```

二叉树的节点个数

1、问题分解

左子树节点个数

右子树节点个数

当前节点个数

2、选择递归框架

三种遍历均可

3、设置出口条件：空树节点为0

4、VISIT操作：计数

5、返回值利用：可返回总结点个数

二叉树的节点个数

```
int Count (BinTree T) {  
    if (!T ) return 0;  
    int left  = Count(T->LeftChild);  
    int right = Count(T->RightChild);  
    int total = left + right + 1;  
    return total;  
    //return 1+ Count (T->LeftChild) + Count (T->RightChild);  
}
```

二叉树中叶子节点个数

整体与统计前类似。

需要判断该节点是否是叶子节点（左右子树空否）

还可以根据节点个数的不同传递方式：

- 1、通过函数返回值
- 2、通过全局变量
- 3、通过函数参数

对同一问题，实现不同的求解函数

二叉树中叶子节点个数 (1)

```
int CountLeaf (BinTree T) { //利用函数返回值, 先序
    if (!T ) return 0;
    int n=0;
    if(T->LeftChild==NULL && T->RightChild==NULL) n=1;
    int left  = CountLeaf (T->LeftChild);
    int right = CountLeaf (T->RightChild);
    return n + left + right;
}
```

二叉树中叶子节点个数 (2)

int n = 0; // 作为叶子节点计数器

void CountLeaf (BinTree T) { // 利用全局变量，中序

if (!T) {

CountLeaf (T->LeftChild);

if (T->LeftChild == NULL && T->RightChild == NULL)

n++;

CountLeaf (T->RightChild);

}}

二叉树中叶子节点个数 (3)

```
void CountLeaf (BinTree T, int *n) { //利用函数参数, 后序
    if (!T) {
        CountLeaf (T->LeftChild, &n);
        CountLeaf (T->RightChild, &n);
        if (T->LeftChild==NULL && T->RightChild==NULL)
            *n++;
    }
}
//第一次调用时, n需要赋值为0
```

求位于二叉树先序序列中第k个位置的结点的值

思路（先序）：

当前节点非空则节点数增加

当前节点位置为k 则记录节点值，返回，否则：

左子树找k位置与值 找到则返回

左子树没有，则查找右子树

- 1、函数需要知道找到或者没有找到 ---可利用返回值T/F
 - 2、函数需要得到当前节点位置 ---可利用全局变量或者参数
- 需用到多个结果的，可同时利用：

全局变量、返回值及(多个)参数进行各类值的传递

求位于二叉树先序序列中第k个位置的结点的值

```
Bool GetPreOrderKNode(BinTree T, int k, ElemType *x, int *n){  
    if (!T) return False;  
    *n++;  
    if (n==k) { *x=T->data; return True;}  
    if (GetPreOrderKNode(T->LeftChild, k, &x, &n))return True;  
    if(GetPreOrderKNode(T->LeftChild, k, &x, &n))return True;  
    return False;  
}
```

二叉树的高度（请大家写出分解，纸上写出代码）

```
int Height (BinTree T) {  
    if (!T) return 0;  
    else {  
        int m = Height (T->LeftChild);  
        int n = Height (T->RightChild);  
        return (m > n) ? m+1 : n+1;  
    }  
}
```


判断两棵二叉树结构是否相似

指的是结构相似，节点内容可以不同（相同就全等了）。

根节点相似

均空 --返回，相同(出口1)

一空一实 --返回，不同(出口2)

均实--可继续：

计算是否左子树结构相似

计算是否右子树结构相似

计算整体是否相似 (请大家在纸上写)

```
int Similar(BinTree T1, BinTree T2)
{
    if(!T1 && !T2) return 1;
    if(T1 xor T2) return 0;
    return similar(T1->LeftChild, T2->LeftChild)
        && similar(T1->RightChild, T2->RightChild);
}
```

应用——给定前序中序遍历序列确定二叉树

对任意特定二叉树，其前、中、后序遍历序列是唯一的，确定的

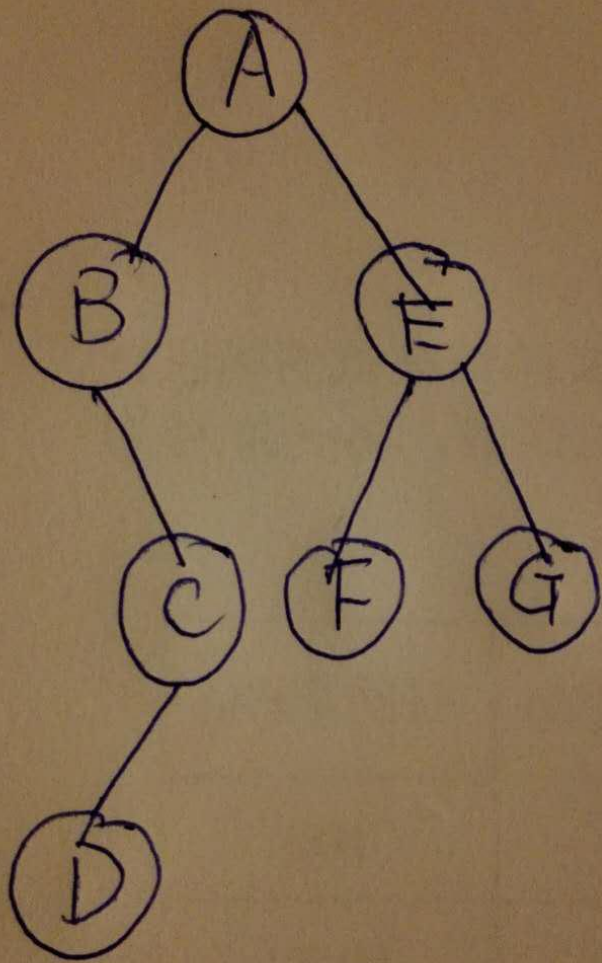
对任意一个前/中/后序遍历序列，是否对应唯一确定的二叉树？

答案是：不能！但若给定三种遍历序列，答案当然是肯定的！

那么给定两个序列呢？

应用——给定前序中序遍历序列确定二叉树

给定



则其遍历序列：

先序： ABCDEFG 中序： BDCAFEG

后序： DCBFGEA

周五进度：

递归框架下的应用（续），二叉树的非递归遍历

下午上机：

1、利用扩充的二叉树进行二叉树创建：

ABDΦΦEΦGΦΦCFΦΦΦ

2、对上述二叉树进行三种递归遍历

3、对该树求高度、节点总数、叶节点个数、先序遍历第4个节点的值，删除节点值为C的子树，销毁该二叉树。

4、建立两个二叉树，比较是否结构相似

5、对上述二叉树进行层次遍历