遍历的算法实现 - 先序遍历

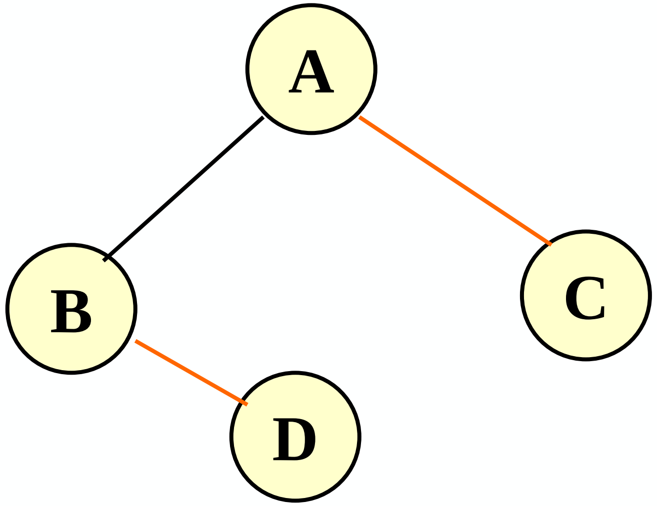
若二叉树为空，则空操作

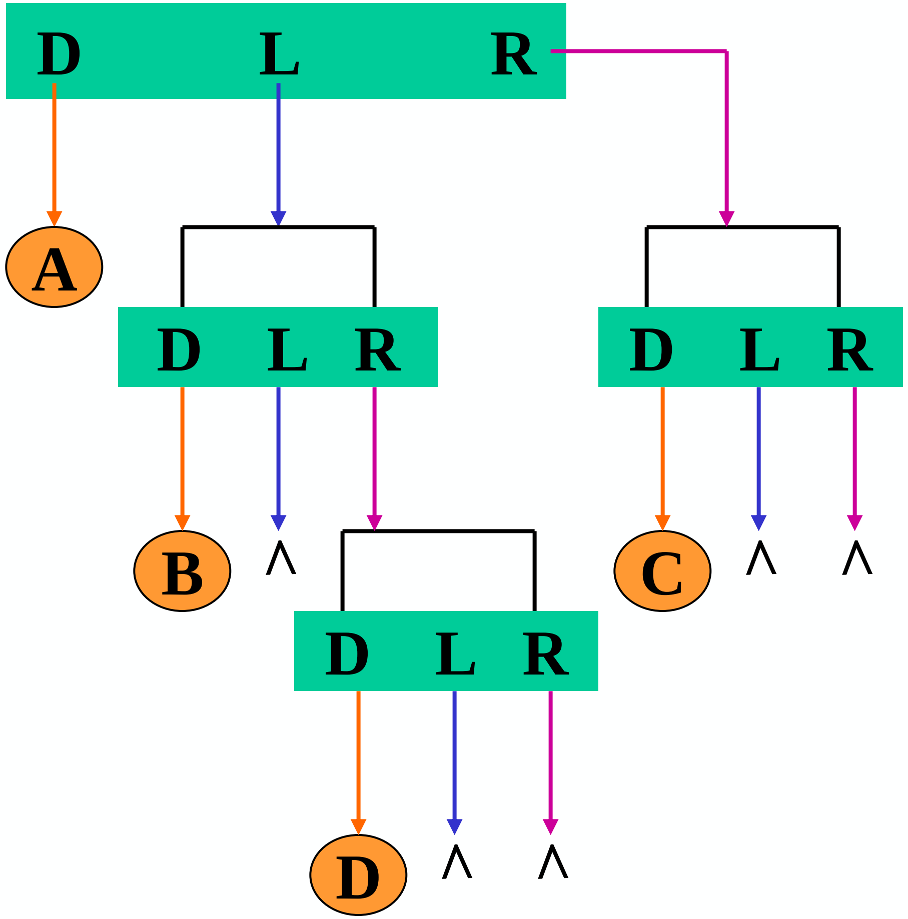
否则

访问根结点 (D)

前序遍历左子树 (L)

前序遍历右子树 (R)





先序遍历序列：A B D C

遍历的算法实现 － －用递归形式格外简单！

回忆:

```
long Factorial ( long n ) {
    if ( n == 0 ) return 1;//基本项
    else return n * Factorial (n-1); //归纳项}
```

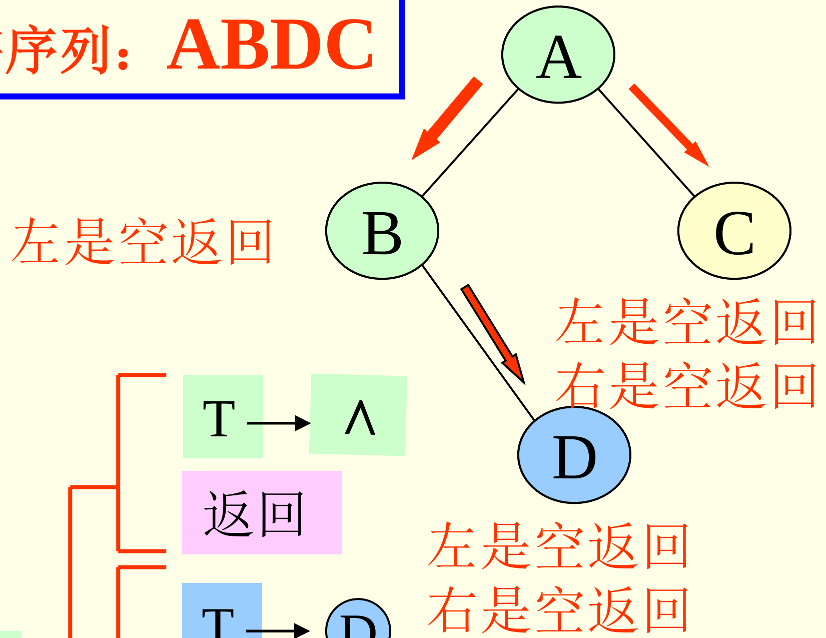
则三种遍历算法可写出:

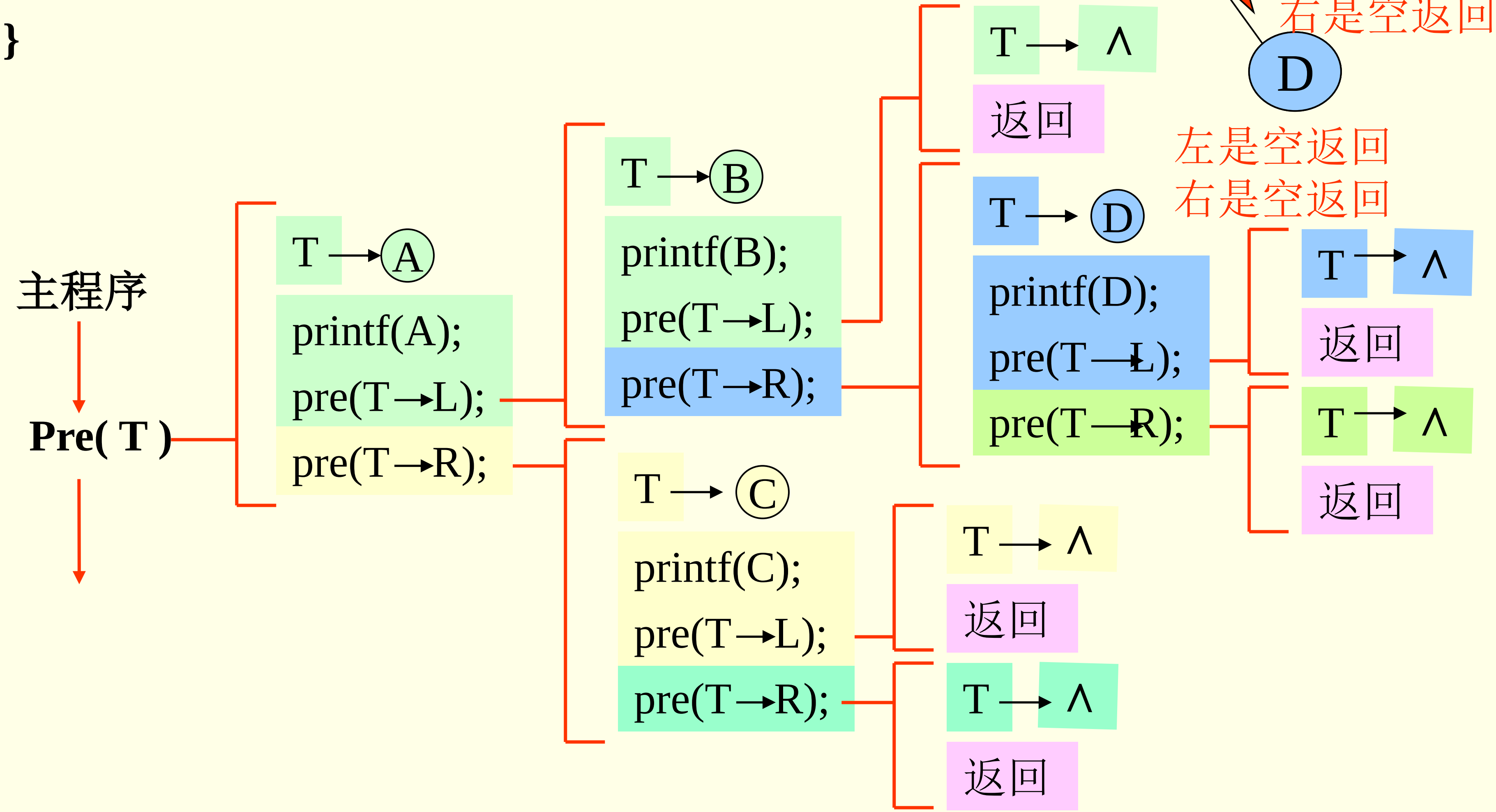
先序遍历算法

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     cout<<T->data; //访问根结点
     PreOrderTraverse(T->lchild); //递归遍历左子树
     PreOrderTraverse(T->rchild); //递归遍历右子树
    }
}
```

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK; else{
    cout<<T->data;
    PreOrderTraverse(T->lchild);
    PreOrderTraverse(T->rchild); }
}
```

先序序列：ABDC





遍历的算法实现 - 中序遍历

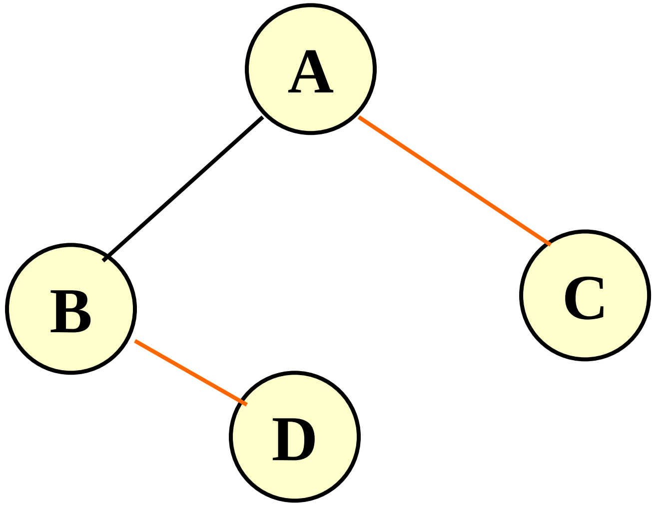
若二叉树为空，则空操作

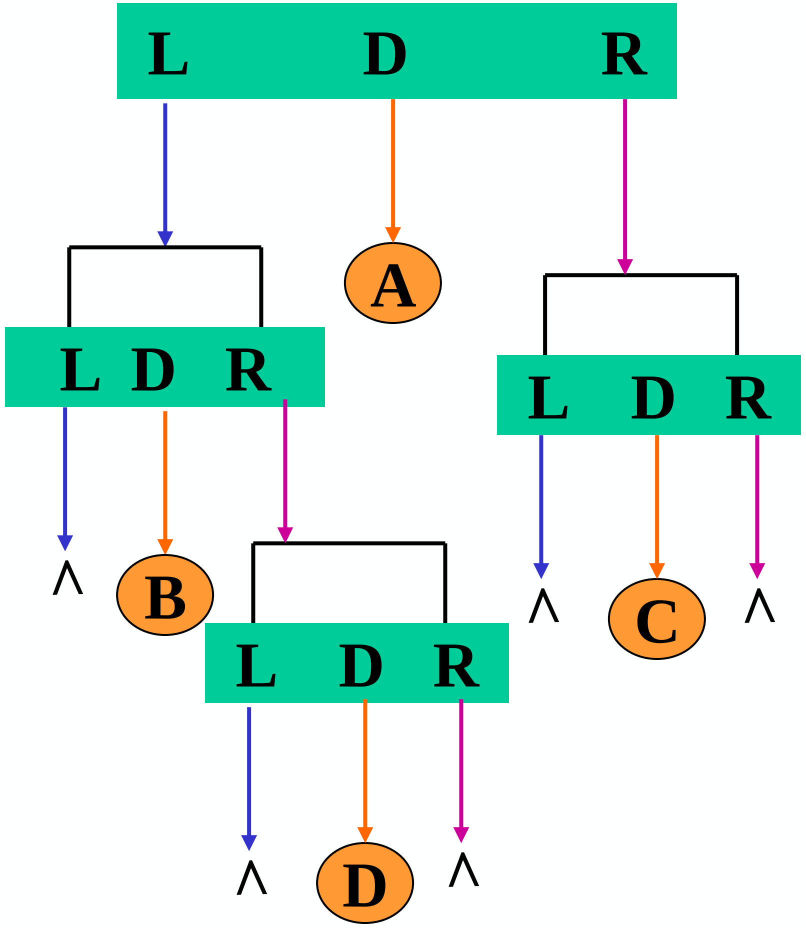
否则:

中序遍历左子树 (L)

访问根结点 (D)

中序遍历右子树 (R)





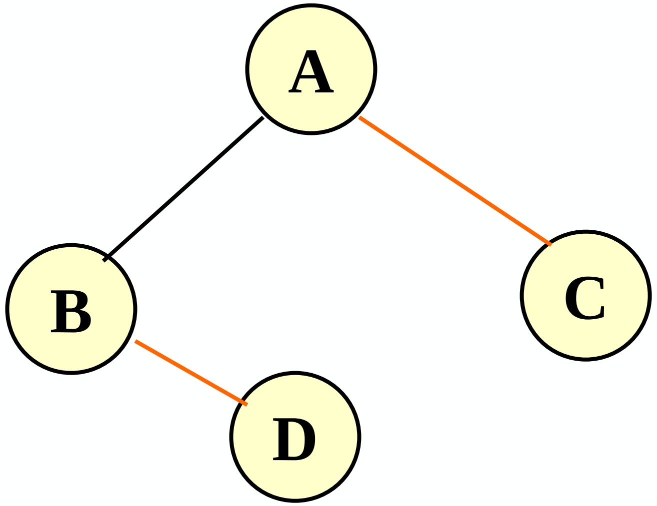
中序遍历序列：B D A C

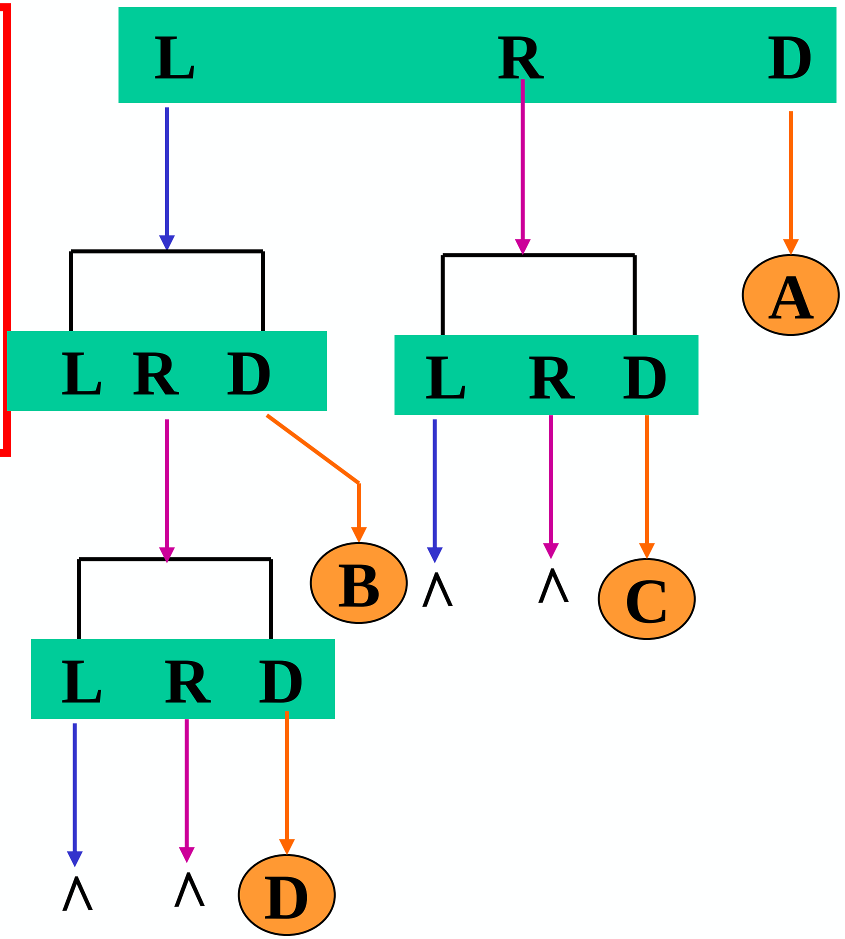
中序遍历算法

```
Status InOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     InOrderTraverse(T->lchild); //递归遍历左子树
   cout<<T->data; //访问根结点
     InOrderTraverse(T->rchild); //递归遍历右子树
    }
}
```

遍历的算法实现 - 后序遍历

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 (D)





后序遍历序列： D　B　C　A

后序遍历算法

```
Status PostOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     PostOrderTraverse(T->lchild); //递归遍历左子树
     PostOrderTraverse(T->rchild); //递归遍历右子树
     cout<<T->data; //访问根结点
    }
}
```

遍历算法的分析

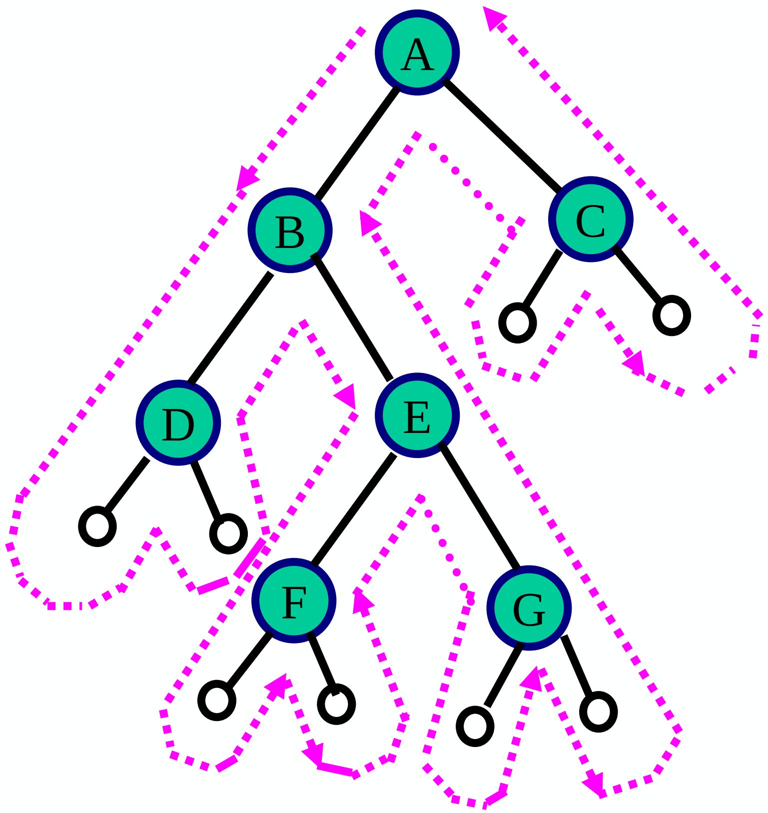
```
Status InOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     InOrderTraverse(T->lchild);
     cout<<T->data;
     InOrderTraverse(T->rchild);
    }
}
```

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     cout<<T->data;
     PreOrderTraverse(T->lchild);
     PreOrderTraverse(T->rchild);
    }
}
```

```
Status PostOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     PostOrderTraverse(T->lchild);
     PostOrderTraverse(T->rchild);
     cout<<T->data;
    }
}
```

遍历算法的分析

如果去掉输出语句，从递归的角度看，三种算法是完全相同的，或说这三种算法的**访问路径是相同的，只是访问结点的时机不同**。

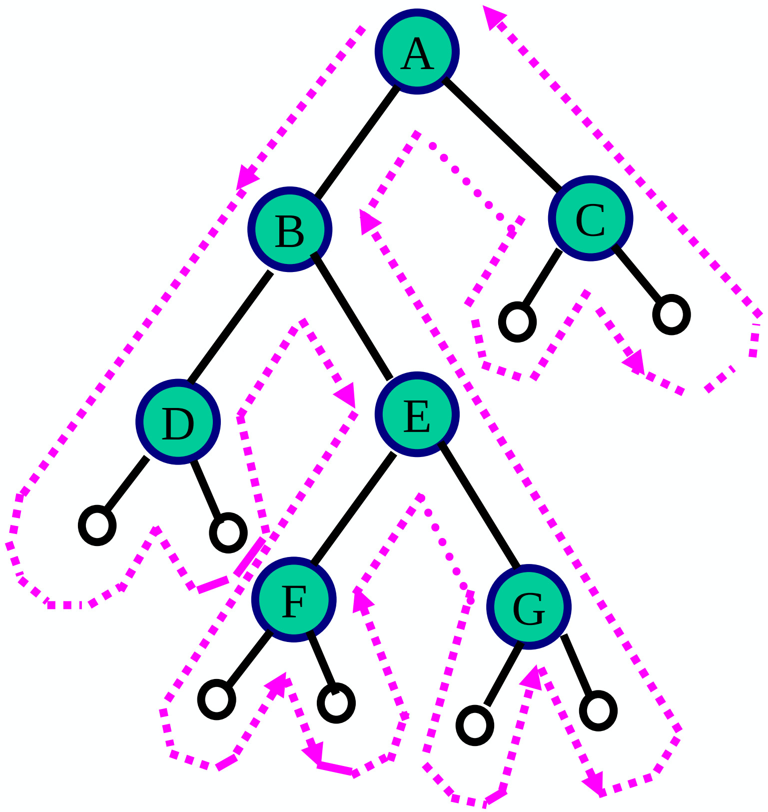


从虚线的出发点到终点的路径上，每个结点经过**3次**。

第1次经过时访问＝**先序**遍历
第2次经过时访问＝**中序**遍历
第3次经过时访问＝**后序**遍历

遍历算法的分析

时间效率:O(n) //每个结点只访问一次
空间效率:O(n) //栈占用的最大辅助空间

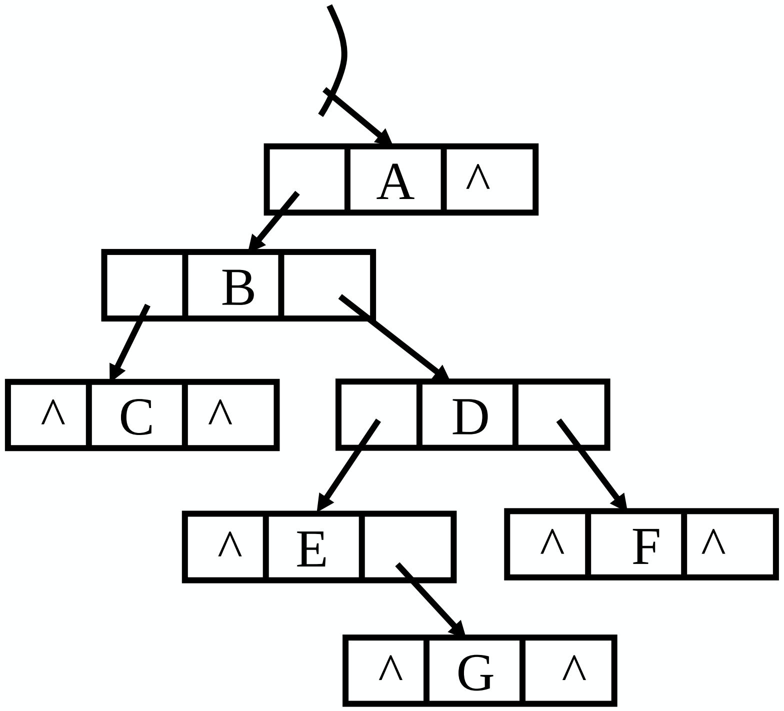


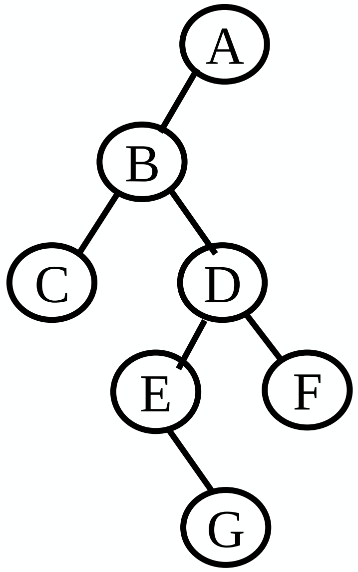
二叉树的建立（算法5.3）

按先序遍历序列建立二叉树的二叉链表

例：已知先序序列为：

A B C Φ Φ D E Φ G Φ Φ F Φ Φ Φ **（动态演示）**





二叉树的建立（算法5.3）

```
void CreateBiTree(BiTree &T) {
cin>>ch;
if (ch==’#’)   T=NULL;      //递归结束，建空树
else{
    T=new BiTNode;    T->data=ch;
        //生成根结点
    CreateBiTree(T->lchild);  //递归创建左子树
    CreateBiTree(T->rchild); //递归创建右子树
  }

}
```

二叉树遍历算法的应用

✓计算二叉树结点总数

- 如果是空树，则结点个数为0；
- 否则，结点个数为左子树的结点个数+右子树的结点个数再+1。

算法5.6

```
int NodeCount(BiTree T){
  if(T == NULL ) return 0;
   else return NodeCount(T->lchild)+NodeCount(T->rchild)+1;
}
```

二叉树遍历算法的应用

✓计算二叉树叶子结点总数

- 如果是空树，则叶子结点个数为0；
- 否则，为左子树的叶子结点个数+右子树的叶子结点个数。
- ？？？

```
int LeadCount(BiTree T){
    if(T==NULL)         //如果是空树返回0
        return 0;
    if (T->lchild == NULL && T->rchild == NULL)
        return 1; //如果是叶子结点返回1
    else return LeafCount(T->lchild) + LeafCount(T->rchild);
}
```

二叉树遍历算法的应用

✓计算二叉树深度

- 如果是空树，则深度为0；
- 否则，递归计算左子树的深度记为m，递归计算右子树的深度记为n，二叉树的深度则为m与n的较大者加1。

重要结论

若二叉树中各结点的值均不相同，则：
由二叉树的前序序列和中序序列，或由其后序序列和中序序列均能唯一地确定一棵二叉树，
但由前序序列和后序序列却不一定能唯一地确定一棵二叉树。

练习

已知一棵二叉树的中序序列和后序序列分别是 BDCEAFHG 和 DECBHGFA，请画出这棵二叉树。

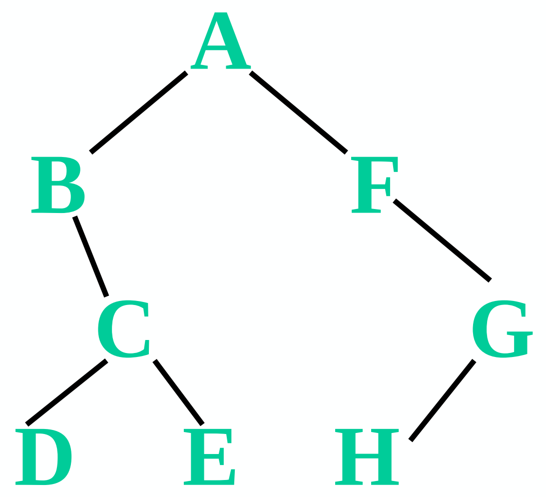
①由后序遍历特征，根结点必在后序序列尾部（A）；

②由中序遍历特征，根结点必在其中间，而且其左部必全部是左子树子孙（BDCE），其右部必全部是右子树子孙（FHG）；

③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。

中序遍历：B D C E A F H G

后序遍历：D E C B H G F A



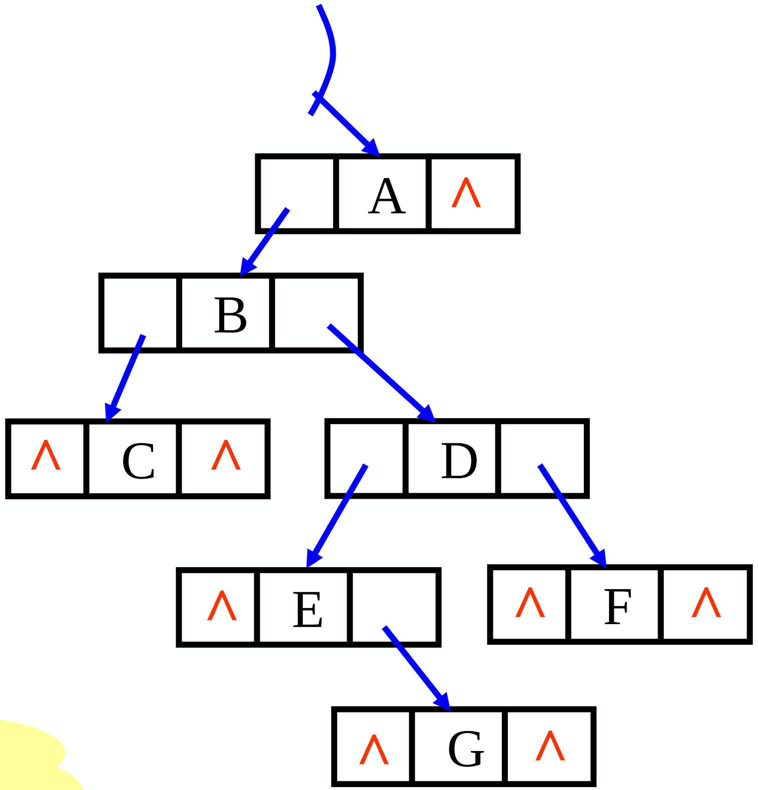
（D C E）　　（H G）

思考

在n个结点的二叉链表中，有n+1个空指针域

二叉链表空间效率这么低，能否利用这些空闲区存放有用的信息或线索？

——可以用它来存放当前结点的**直接前驱和后继**等线索，以加快查找速度。



线索化二叉树

线索化二叉树

普通二叉树只能找到结点的左右孩子信息，而该结点的直接前驱和直接后继只能在遍历过程中获得

若将遍历后对应的有关前驱和后继预存起来，则从**第一个结点**开始就能很快“顺藤摸瓜”而遍历整个树

可能是根、或最左（右）叶子

例如中序遍历结果：B D C E A F H G，实际上已将二叉树转为线性排列，显然具有唯一前驱和唯一后继！

线索化二叉树

如何保存这类信息?

两种解决方法

- 增加两个域：fwd和bwd；
- 利用空链域（n+1个空链域）

线索化二叉树

1）若结点有左子树，则lchild指向其左孩子；
否则， lchild指向其直接前驱(即线索)；

2）若结点有右子树，则rchild指向其右孩子；
否则， rchild指向其直接后继(即线索) 。

为了避免混淆，增加两个标志域

lchild	LTag	data	RTag	rchild

线索化二叉树

lchild	LTag	data	RTag	rchild

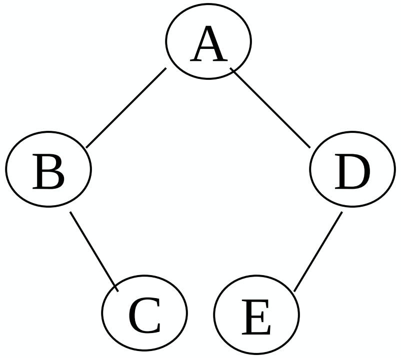
LTag :若 LTag=0, lchild域指向左孩子;
若 LTag=1, lchild域指向其前驱。

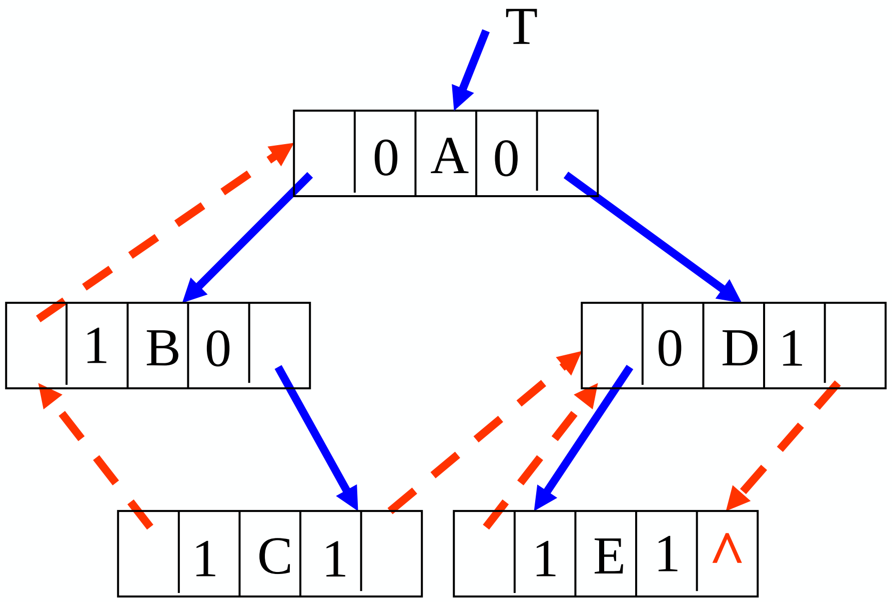
RTag :若 RTag=0, rchild域指向右孩子;
若 RTag=1, rchild域指向其后继。

先序线索二叉树

LTag=0, lchild域指向左孩子
LTag=1, lchild域指向其前驱
RTag=0, rchild域指向右孩子
RTag=1, rchild域指向其后继

lchild	LTag	data	RTag	rchild





先序序列：ABCDE