第六章 树和二叉树

制作：数据结构在线课程课题组

南京审计大学 信息工程学院

2020.10

二叉树的链式存储表示

1. 二叉链表

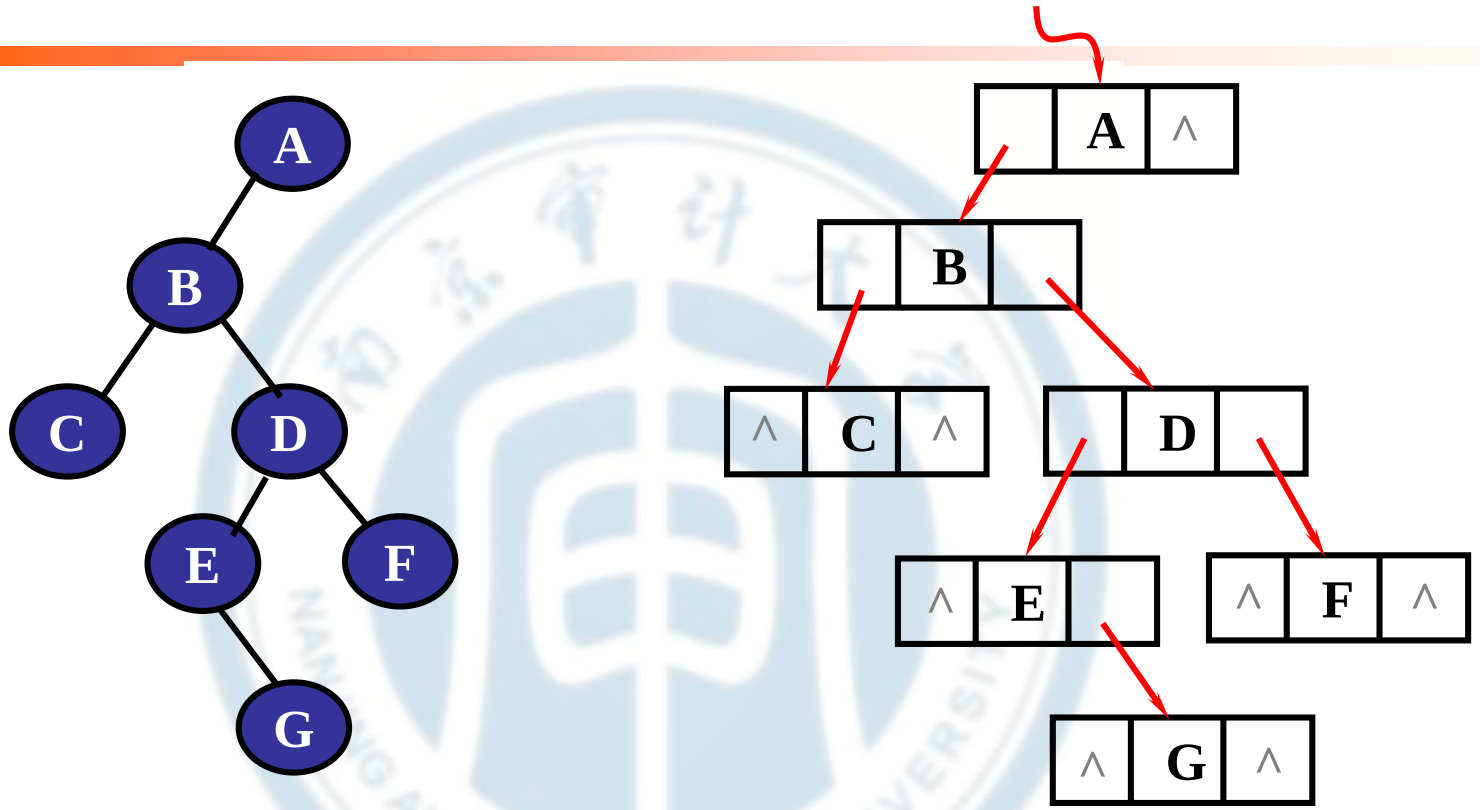
2. 三叉链表

链式存储结构二叉链表

- 结点除包括元素自身的信息外，还包括指向其左、右子树的指针。即结点要包括数据域，左子树指针域和右子树指针域。

❖ 二叉链表的存储表示

```
typedef  struct  BiTNode{
    TElemType  data;
    struct  BiTNode  *lchild, *rchild;
}BiTNode ,*BiTree;
```

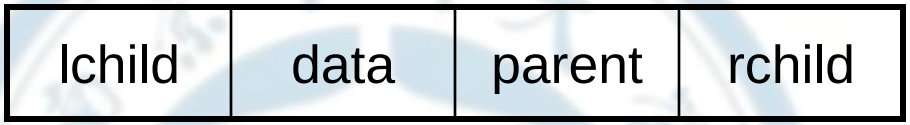


在n个结点的二叉链表中，有n+1个空指针域。

- 如何从一个节点找到其父节点？

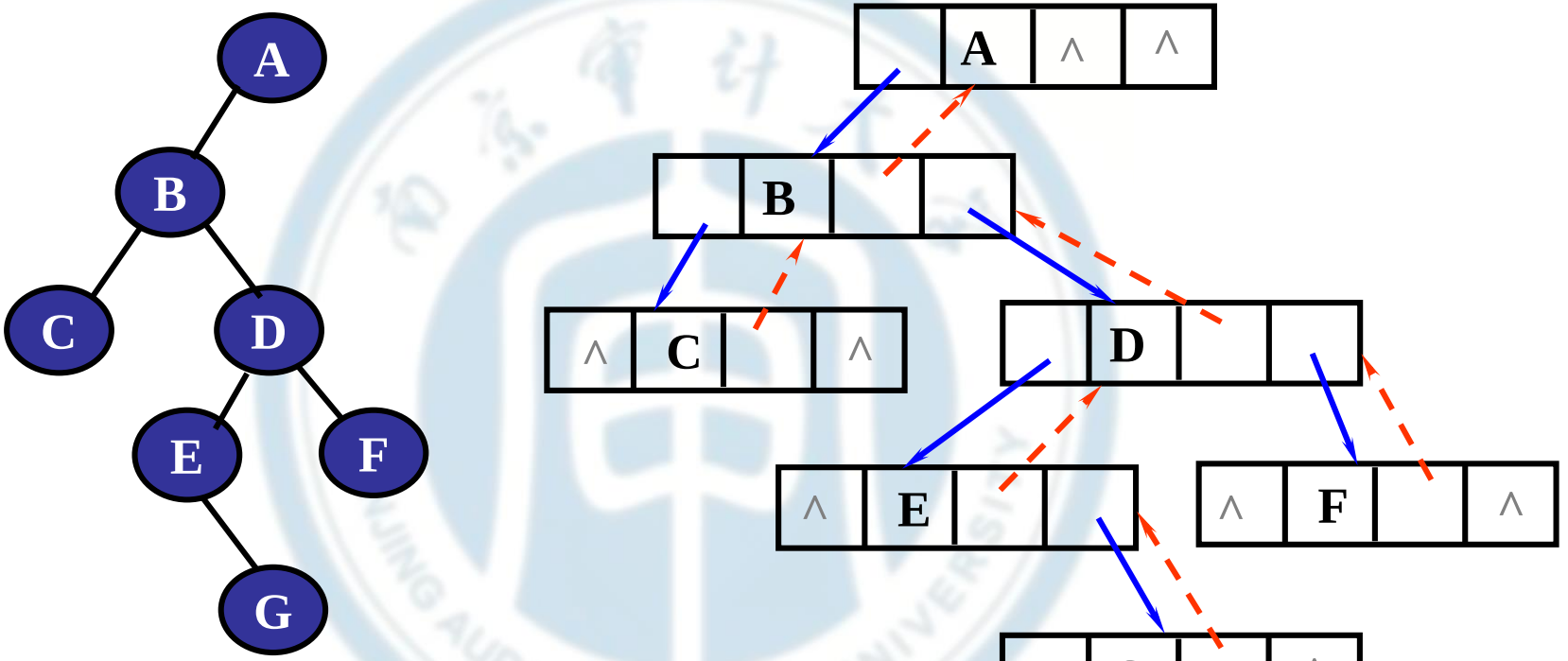
链式存储结构 三叉链表

- 结点包括数据域，左子树指针域、双亲域和右子树指针域。



❖三叉链表的存储表示

```
typedef  struct  TriTNode{
    TElemType  data;
    struct  TriTNode  *lchild, *rchild, *parent;
} TriTNode ,*TriTree;
```


A
B
C
D
E
F
G
A
^
^
B
^
C
^
D
^
E
^
F
^
G
^

6.3 遍历二叉树

- 遍历
  - 按一定的规律，走遍二叉树的每个结点，使每个结点被访问一次，且只被访问一次。
  - 遍历方式
    - 按根、左子树、右子树三个部分进行访问；
    - 按层次访问；

遍历的过程就是把非线性结构的二叉树中的结点排成一个线性序列的过程。

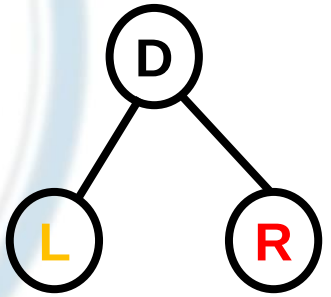
对“二叉树”而言，可以有三条搜索路径：

- 设访问根结点记作 D
- 遍历根的左子树记作 L
- 遍历根的右子树记作 R
- 则可能的遍历次序有

先(根)序　DLR

中(根)序　LDR

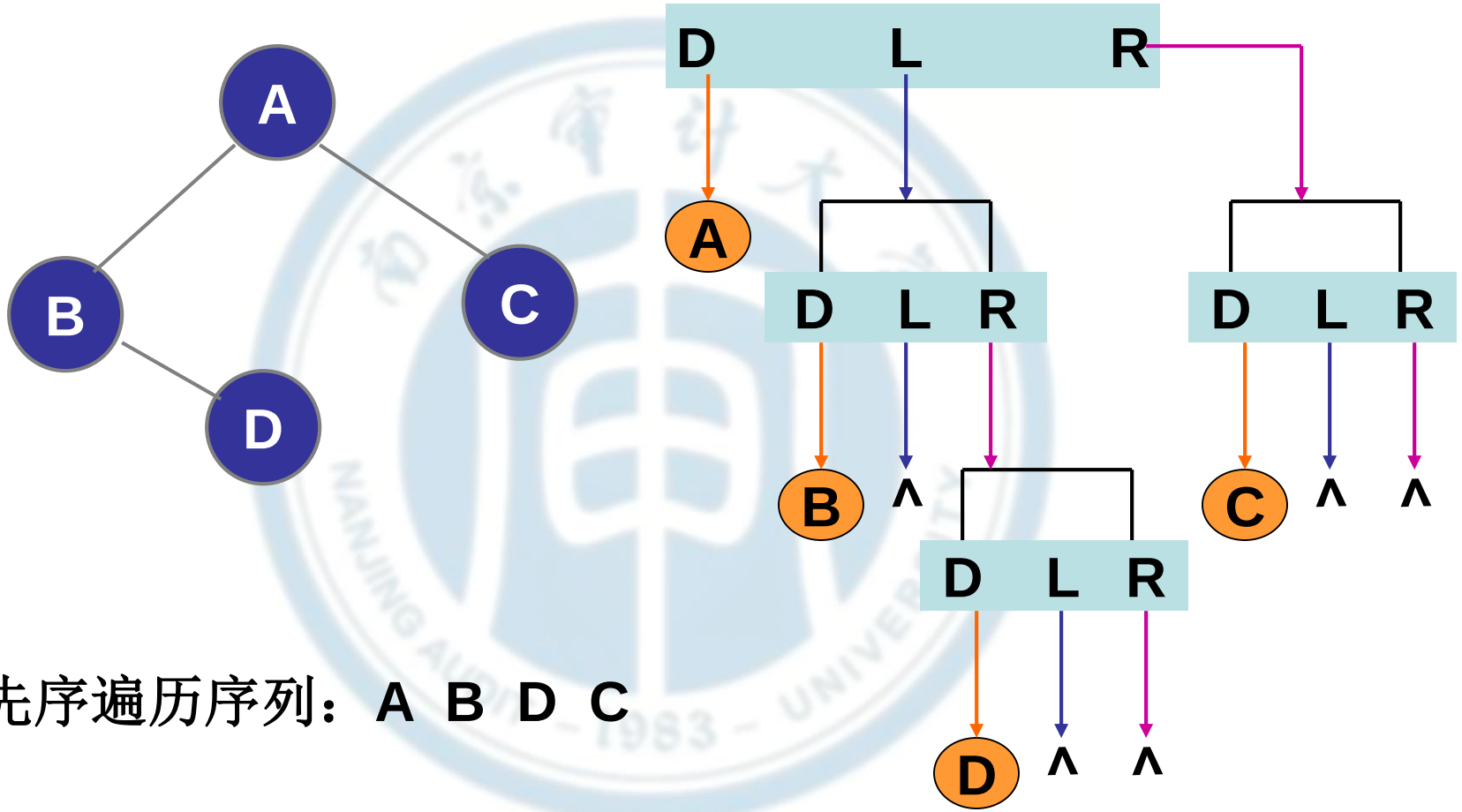
后(根)序　LRD



先序遍历

- 若二叉树为空，则空操作；否则
  - 访问根结点；
  - 先序遍历左子树；
  - 先序遍历右子树。

先序遍历

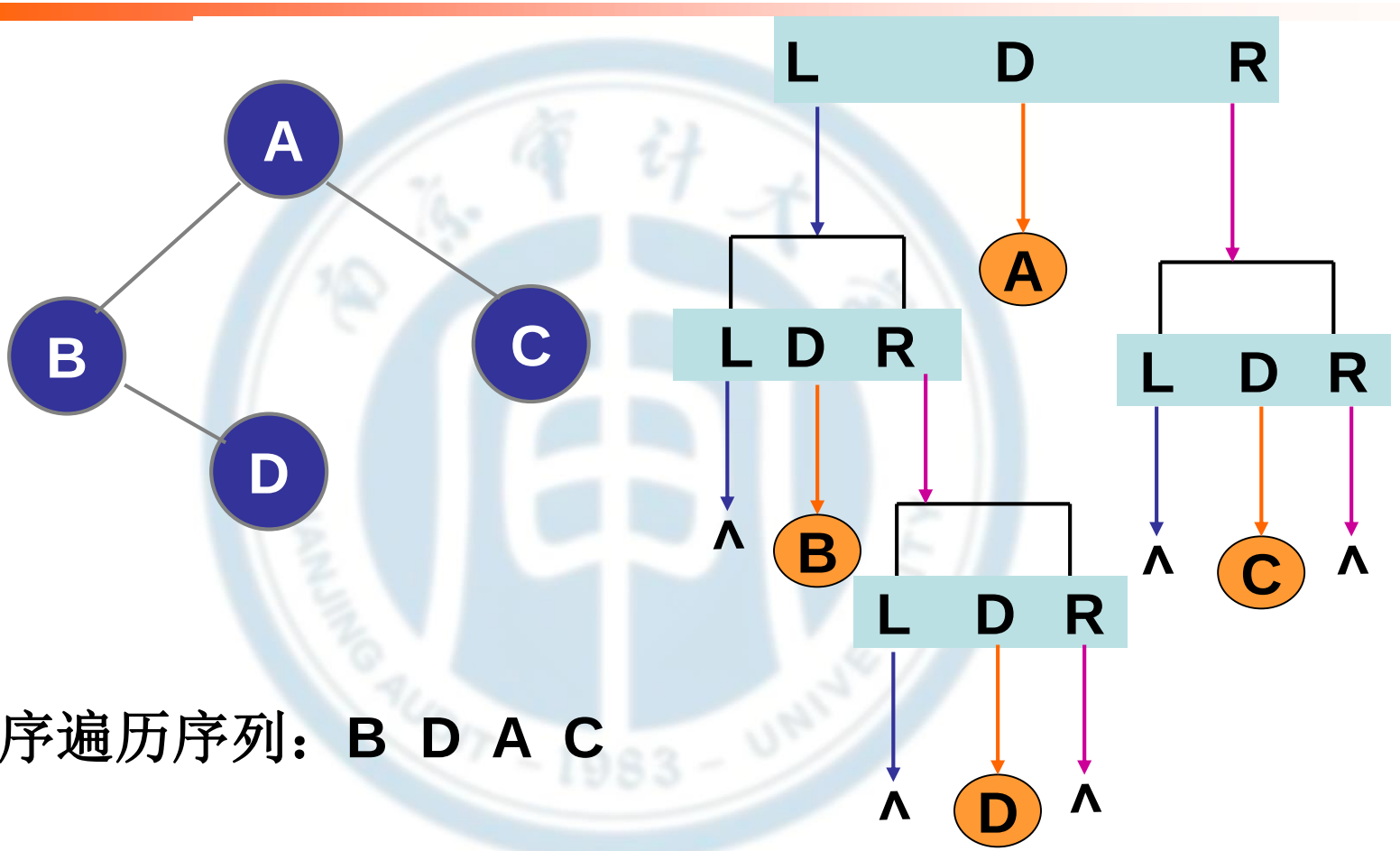


先序遍历序列：A B D C

中序遍历

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

中序遍历

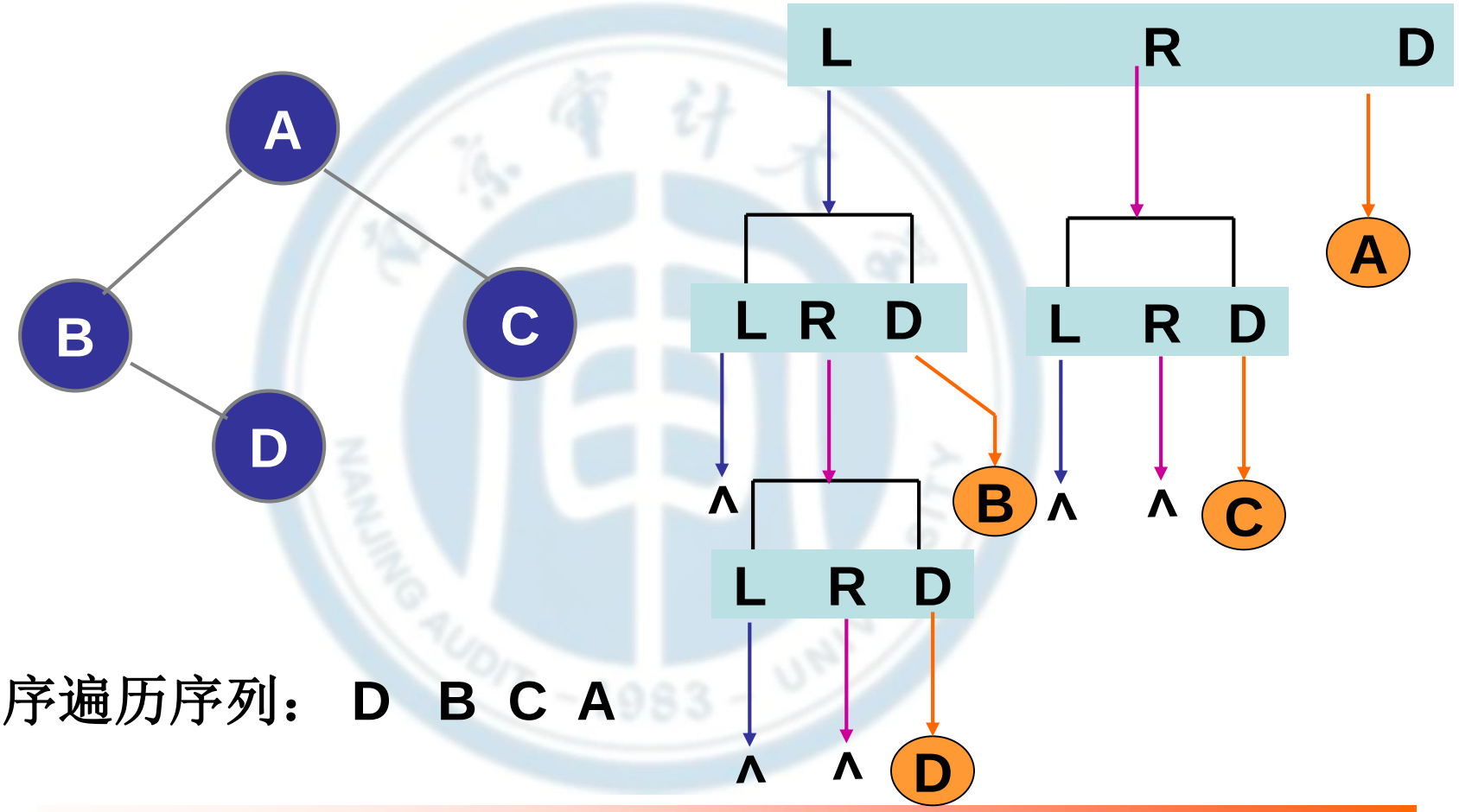


中序遍历序列：B D A C

后序遍历

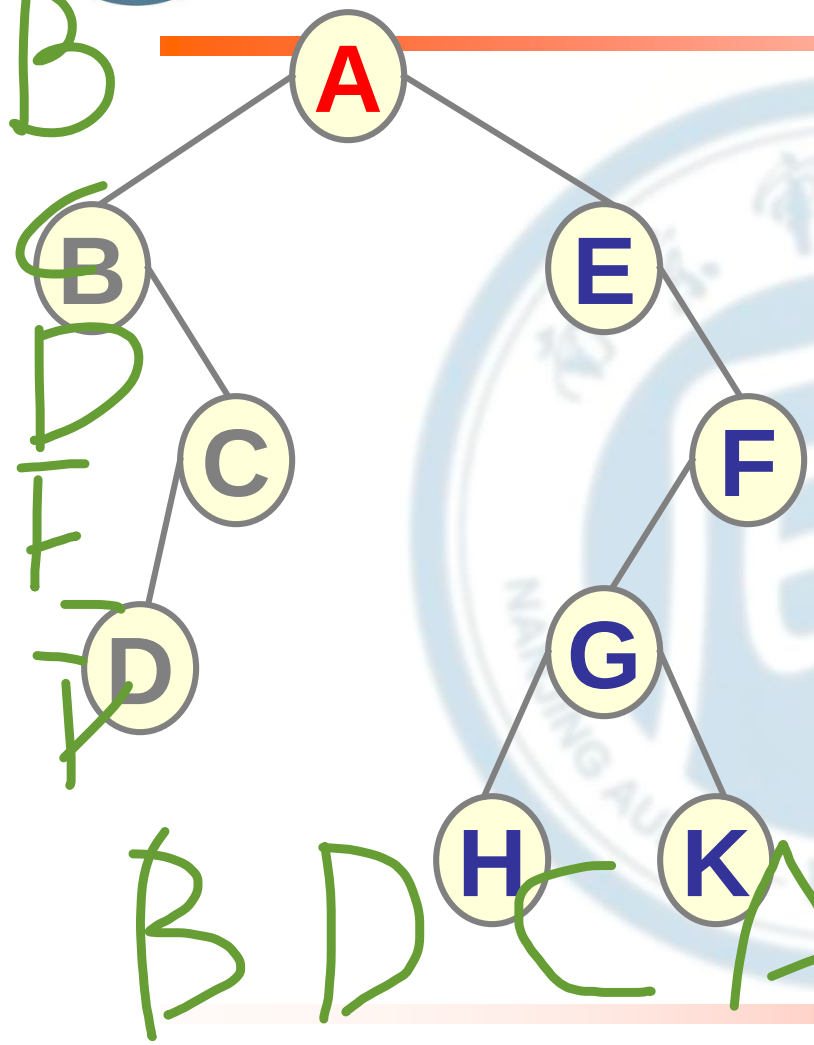
- 若二叉树为空，则空操作；否则
  - 后序遍历左子树；
  - 后序遍历右子树；
  - 访问根结点。

后序遍历



后序遍历序列： D B C A

练习：



先序序列：

A B C D E F G H K

中序序列：

B D C A E H G K F

后序序列：

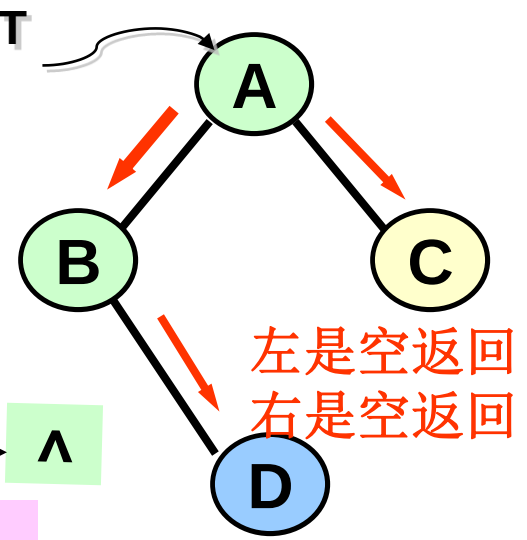
D C B H K G F E A

先序遍历的递归算法

```
Status PreOrderTraverse(BiTree T,Status (*Visit)(TElemType))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数。修改算法6.1
        // 操作结果：先序递归遍历T，对每个结点调用函数Visit一次且仅一次
        if(T) // T不空
        {
                if(Visit(T->data))// 先访问根结点
                        if(PreOrderTraverse(T->lchild,Visit))// 再先序遍历左子树
                                if(PreOrderTraverse(T->rchild,Visit))// 遍历右子树
                                        return OK;
                return ERROR;
        }
        else  return OK;
}
```

递归算法的执行过程

```
void pre(BiTree  T)
{  if (T!=NULL)
   {  printf ("%d\t",T->data);
      pre(T->lchild);
      pre(T->rchild);
   }
}
```



主程序

Pre(T)

T → A
printf(A);
pre(T→ L);
pre(T→ R);

T → B
printf(B);
pre(T→ L);
pre(T→ R);

T → C
printf(C);
pre(T→ L);
pre(T→ R);

T → D
printf(D);
pre(T→L);
pre(T→R);

T → ∧
返回

T → ∧
返回

T → ∧
返回

T → ∧
返回

先序序列：A B D C

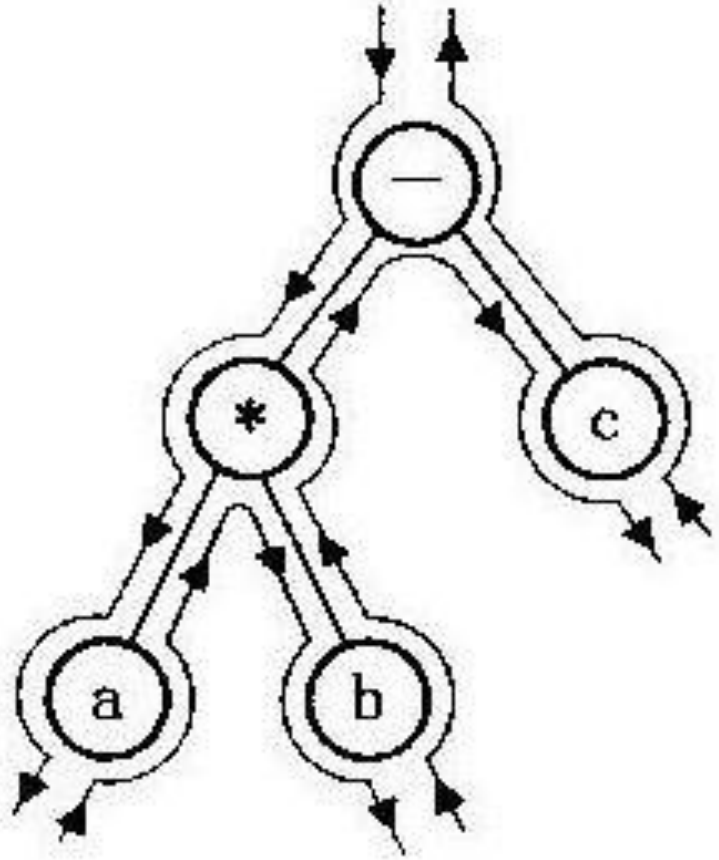
中序遍历的递归算法

- **Status InOrderTraverse(BiTree T，Status(*visit)**
  (TElemType e)) {
  // 采用二叉链表存储结构，visit是对元素操作的应用函数，
  // 中序遍历二叉树T的递归算法，对每个数据元素调用函数visit。
  // 最简单的visit函数是输出元素的值。
- **//留作上机练习**
- **} // InOrderTraverse**

后序遍历的递归算法

- **Status PostOrderTraverse(BiTree T，**
  Status(*visit)(TElemType e)){
  // 采用二叉链表存储结构，visit是对元素操作的应用函数，
  // 先序遍历二叉树T的递归算法，对每个数据元素调用函数visit。
  // 最简单的visit函数是输出元素的值。
- **//留作上机练习**
- **} // InOrderTraverse**

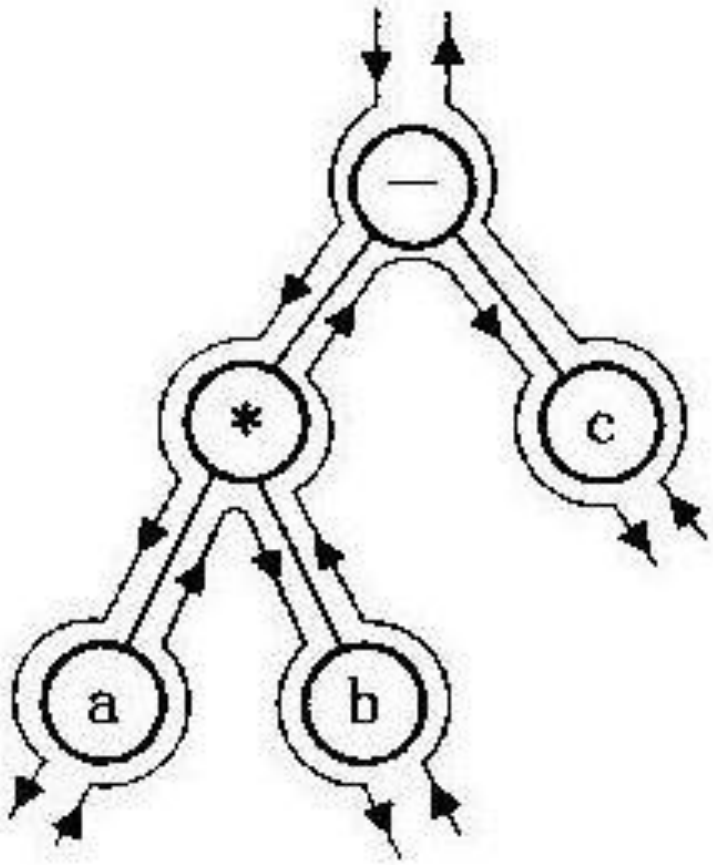
三种遍历算法的比较



相同点：

如果把访问根结点这个不涉及递归的语句抛开，则三个递归算法走过的路线是一样的。

三种遍历算法的比较



不同点：

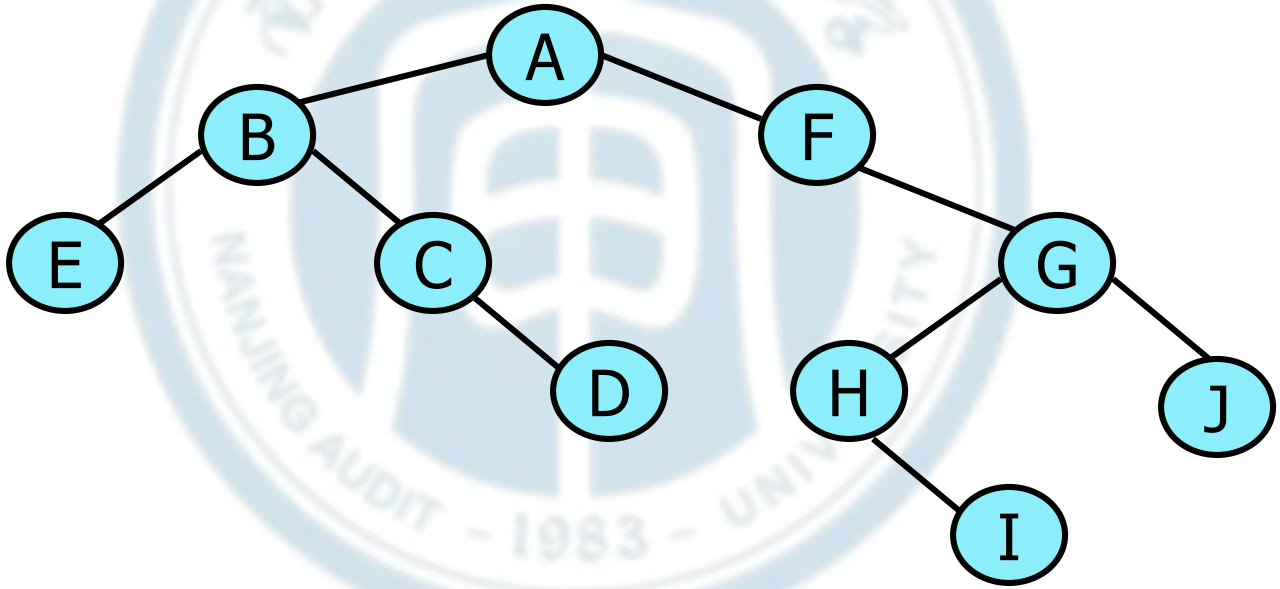
① 前序遍历是每进入一层递归调用时先访问根结点，然后再依次向它的左、右子树执行递归调用；

② 中序遍历是在执行完左子树递归调用后再访问根结点，然后向它的右子树递归调用；

③ 后序遍历则是在从右子树递归调用退出后访问根结点。

练习

- 已知一棵二叉树的先序遍历序列为ABECDFGHI J，中序遍历序列为EBCDAFHIGJ，试画出这棵二叉树



遍历算法的应用举例

1、建立二叉树的存储结构

2、二叉树的输出

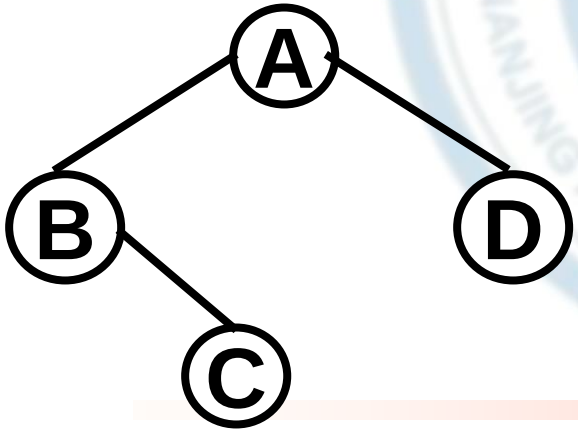
3、统计二叉树中叶子结点的个数

4、求二叉树的深度

1、建立二叉树

空树 以空白字符“■”表示

只含一个根结点的二叉树 Ⓐ 以字符串“A■■”表示



以下列字符串表示

A B ■ C ■ ■ D ■ ■

二叉树的创建

```
Status CreateBiTree(BiTree &T){
// 按先序序列输入二叉树中结点的值（一个字符），空格表示空树，
// 构造二叉链表表示的二叉树T。
    scanf(&ch) ;
    if(ch == ‘ ’) T=NULL;                          // 建空树
    else {
      if(!(T=(BiTNode *)malloc(sizeof(BiTNode)))) exit(OVERFLOW);
      T->data = ch;                                // 生成根结点
      CreateBiTree(T->Lchild);          // 递归建(遍历)左子树
      CreateBiTree(T->Rchild);          // 递归建(遍历)右子树
    } // else
    return OK;
} // CreateBiTree
```

2、二叉树的目录状输出

```
Status PrintTree(BiTree bt,int nLayer)  /* 按竖向树状打印的
    二叉树 */
{
    if(bt != NULL) {
        PrintTree(bt->lchild,nLayer+1);
        for(int i=0;i<nLayer;i++)
            printf(" -");
        printf("%c\n",bt->data);
        PrintTree(bt->rchild,nLayer+1);
    }
    return OK;
}
```

3、统计二叉树中叶结点数

算法基本思想：

先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数。

由此，需在遍历算法中增添一个“计数”的参数，并将算法中“访问结点” 的操作改为:若是叶子，则计数器增1。

```
void CountLeaf (BiTree T,  int &count){
  if ( T ) {
    if ((!T->lchild)&& (!T->rchild))
      count++;    // 对叶子结点计数
    CountLeaf( T->lchild, count);
    CountLeaf( T->rchild, count);
  } // if
} // CountLeaf
```

4、求二叉树的深度

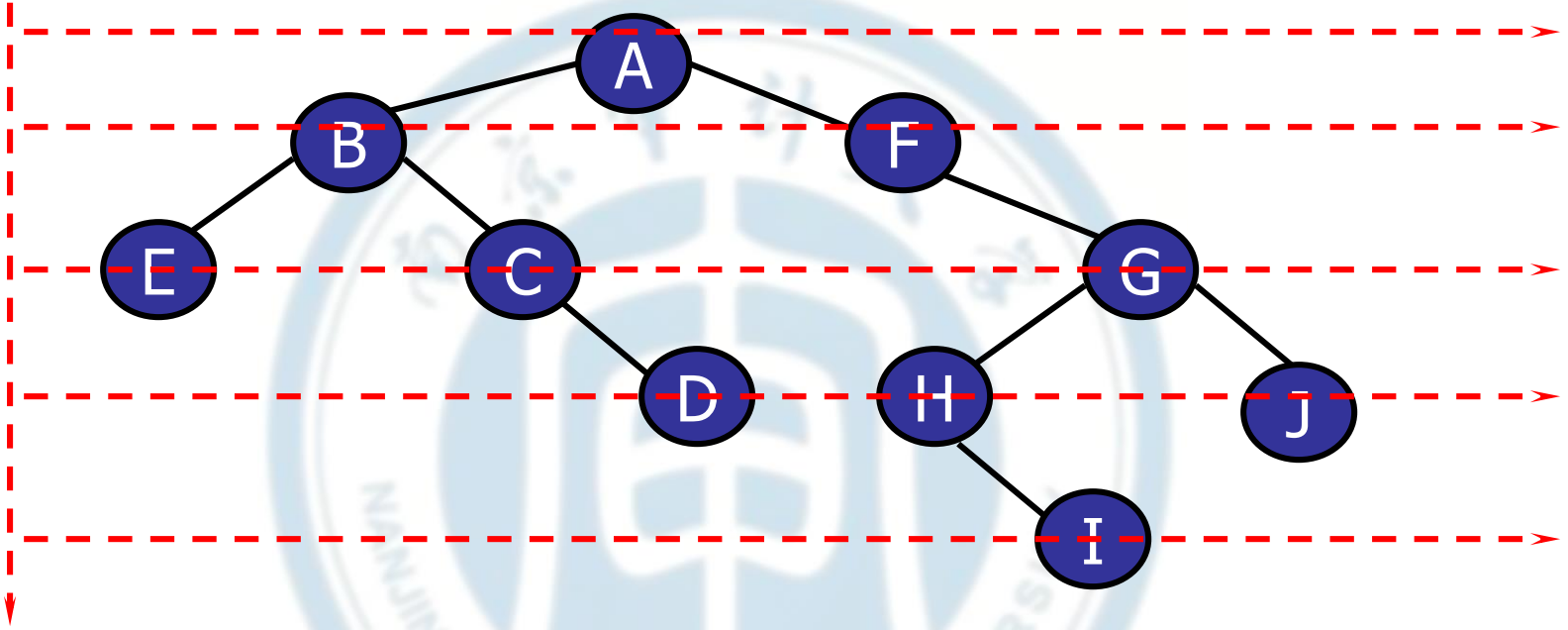
算法基本思想：

首先分析二叉树的深度和它的左、右子树深度之间的关系。

从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。由此，需先分别求得左、右子树的深度，算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1 。

```
int TreeDepth (BiTree T ){ // 返回二叉树的深度
   if ( !T )    depth = 0;
   else   {
     depthLeft = TreeDepth( T->lchild );
     depthRight= TreeDepth( T->rchild );
     depth = 1 + (depthLeft > depthRight  ?
                        depthLeft : depthRight);
   }
   return depth;
}
```

按层次遍历



按层次遍历序列：**ABFECGDHJI**

```
void LevelOrderTraverse(BiTree T,Status (*Visit)(TElemType e))
{ // 初始条件：二叉树T存在，Visit是对结点操作的应用函数
    // 操作结果：层序递归遍历T(利用队列)，对每个结点调用函数Visit一次且仅一次
    LinkQueue q;
    QElemType a;
    if(T){ // T不空
        InitQueue(q); // 初始化队列q
        EnQueue(q,T); // 根指针入队
        while(!QueueEmpty(q)){ // 队列不空
            DeQueue(q,a); // 出队元素(指针)，赋给a
            Visit(a->data); // 访问a所指结点
            if(a->lchild!=NULL) // a有左孩子
                EnQueue(q,a->lchild); // 入队a的左孩子
            if(a->rchild!=NULL) // a有右孩子
                EnQueue(q,a->rchild); // 入队a的右孩子
        }
        printf("\n");
    }
}
```