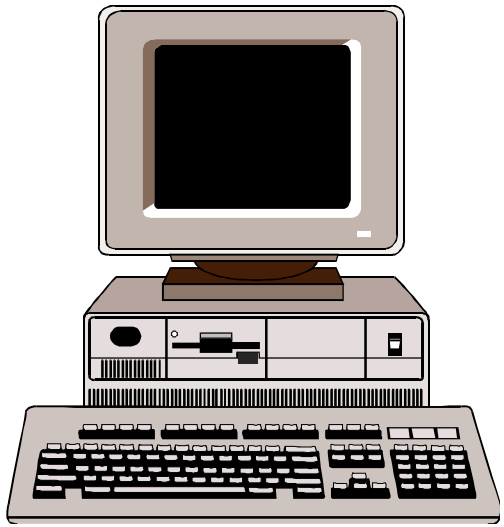
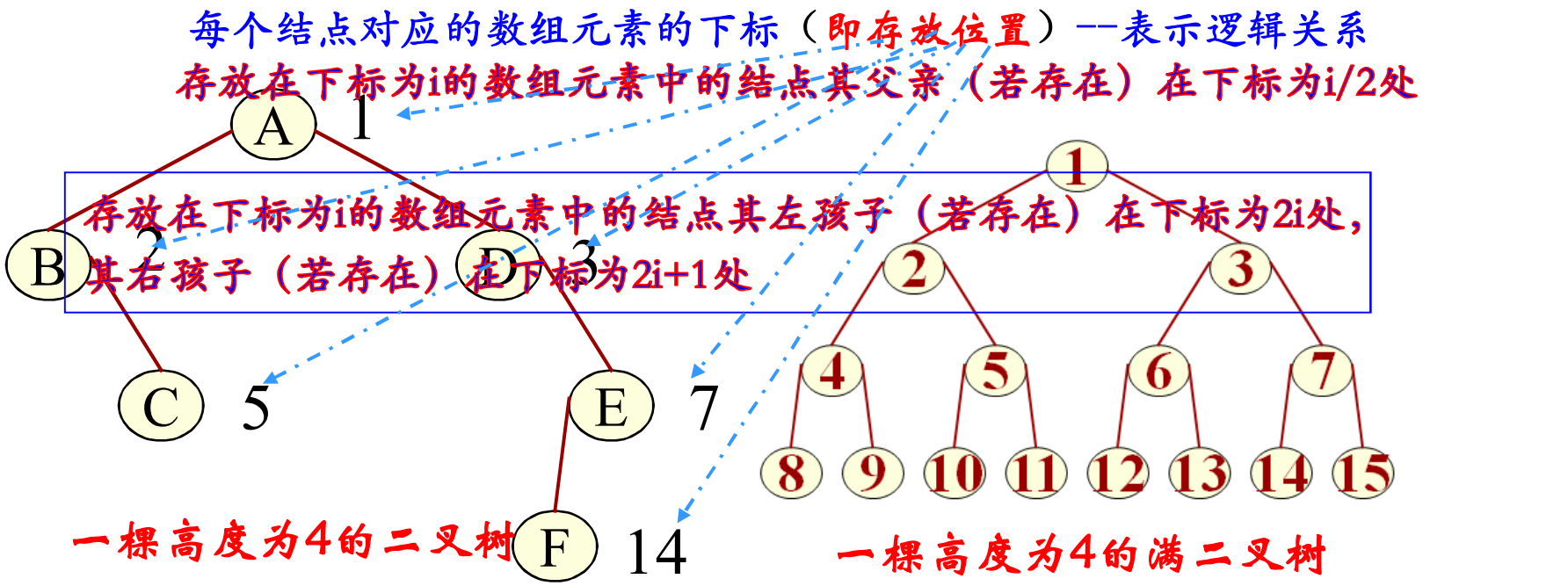
6.2.3 二叉树的存储结构

- 二叉树可以采用：
  - 顺序存储表示
  - 链式存储表示



二叉树的顺序存储表示

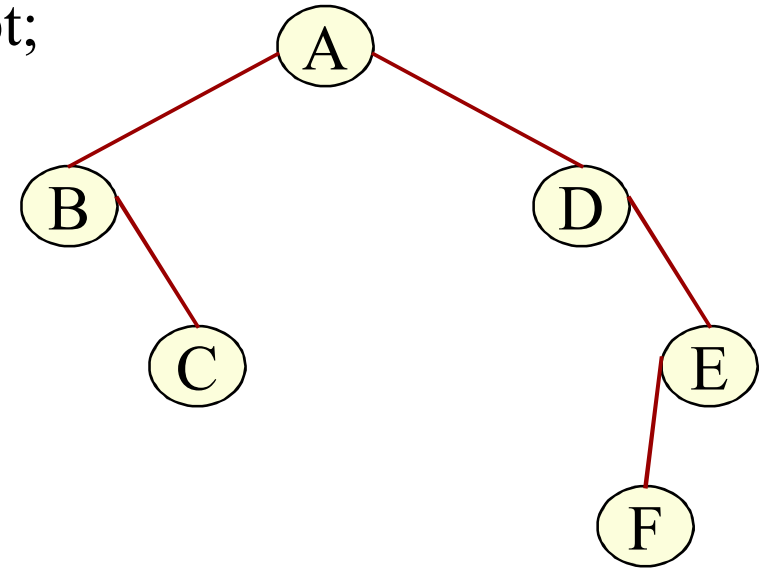
一维数组存放二叉树的每个结点；每个结点存放的数组元素的下标为高度相同的满二叉树中对应结点的层次编号



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	B	D		C		E							F

二叉树的顺序存储表示

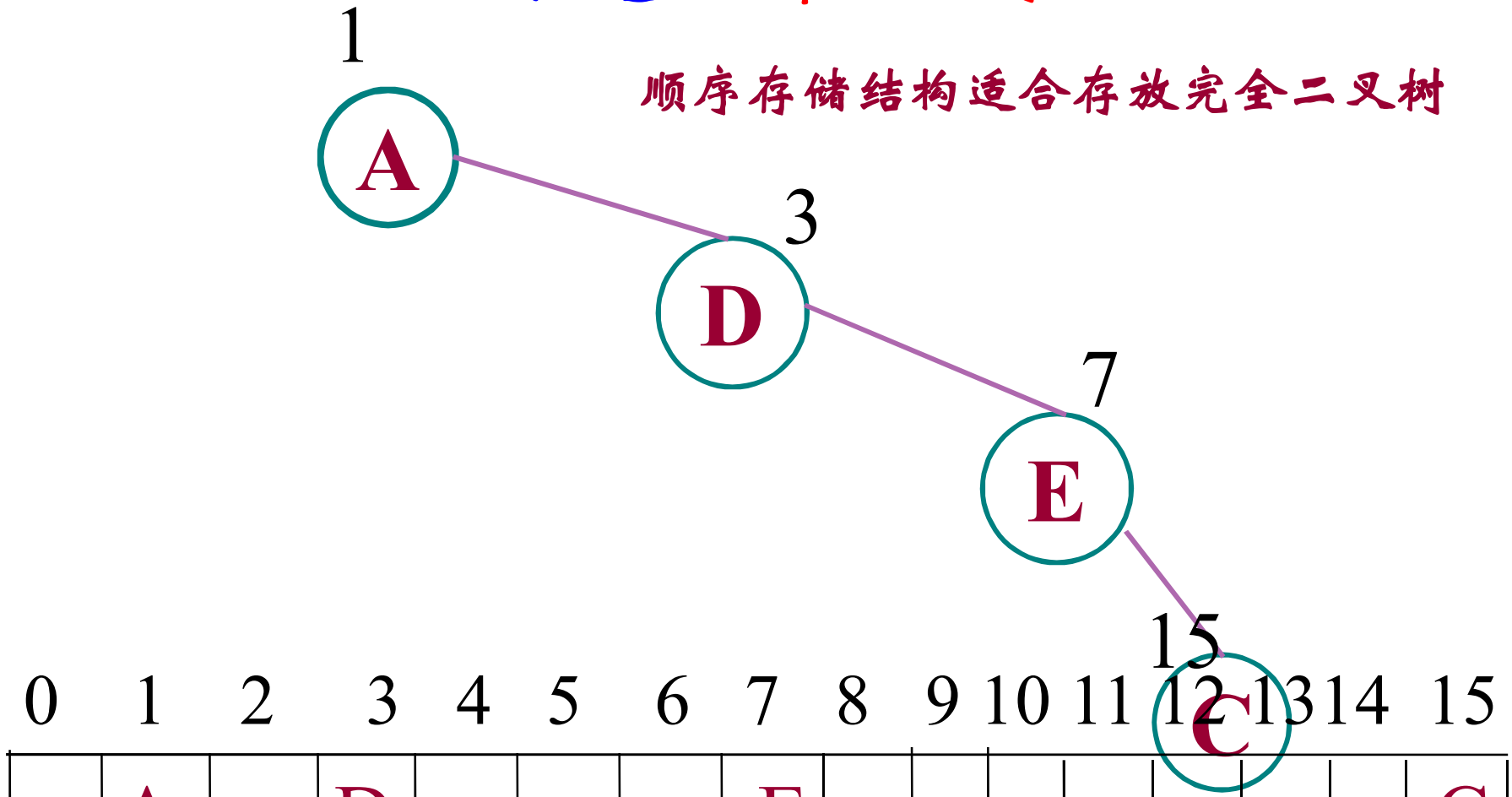
```
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree bt;
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	B	D		C		E							F

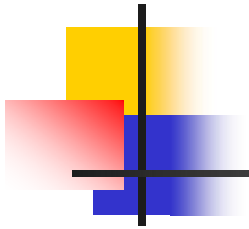
bt[0] bt[1]

问题---单枝树？



顺序存储结构适合存放完全二叉树

顺序存储结构存放单支树浪费存储



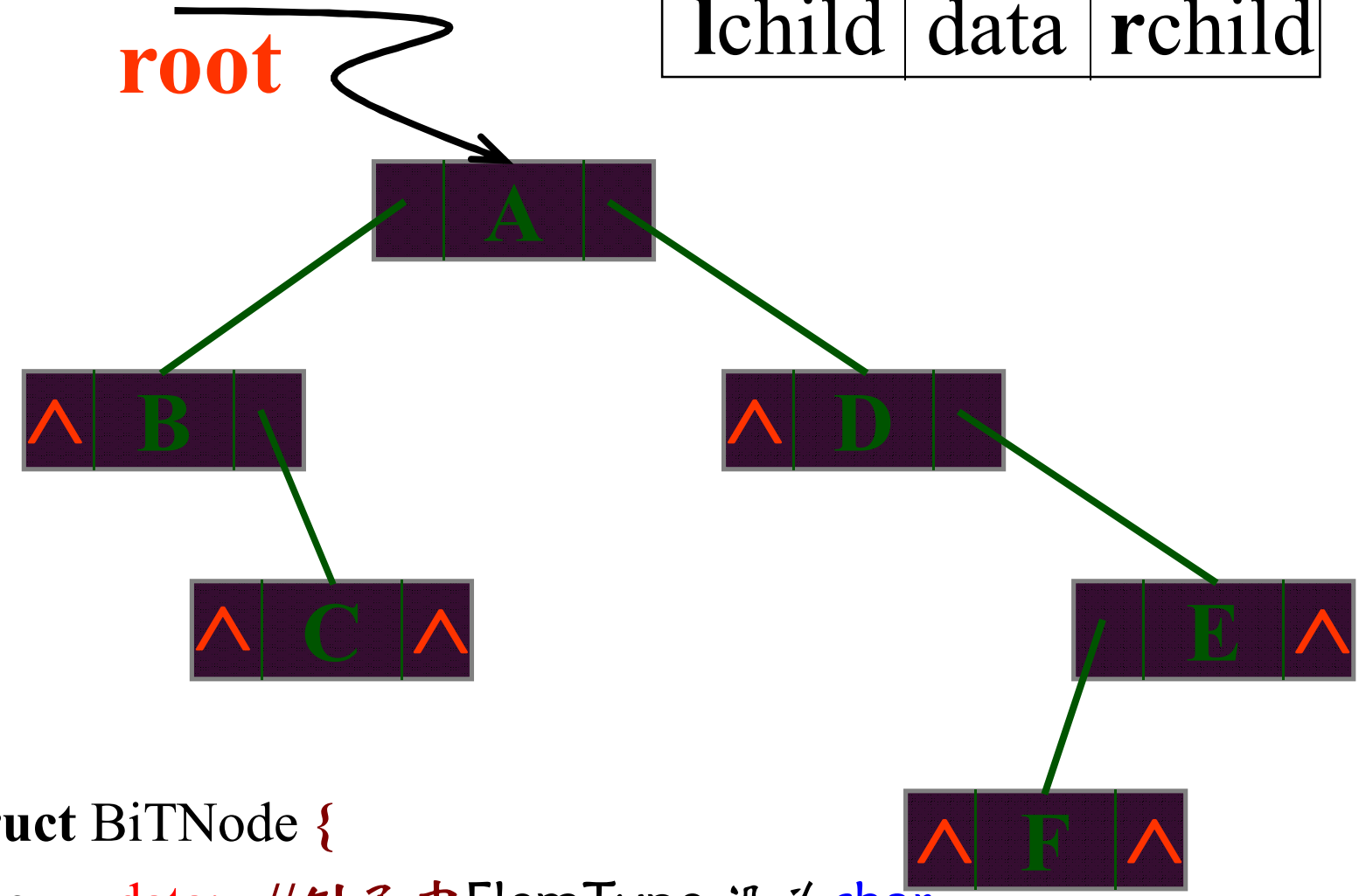
二叉树的链式存储表示

- 二叉链表
- 三叉链表
- 双亲数组
- 线索链表

二叉链表

结点结构：

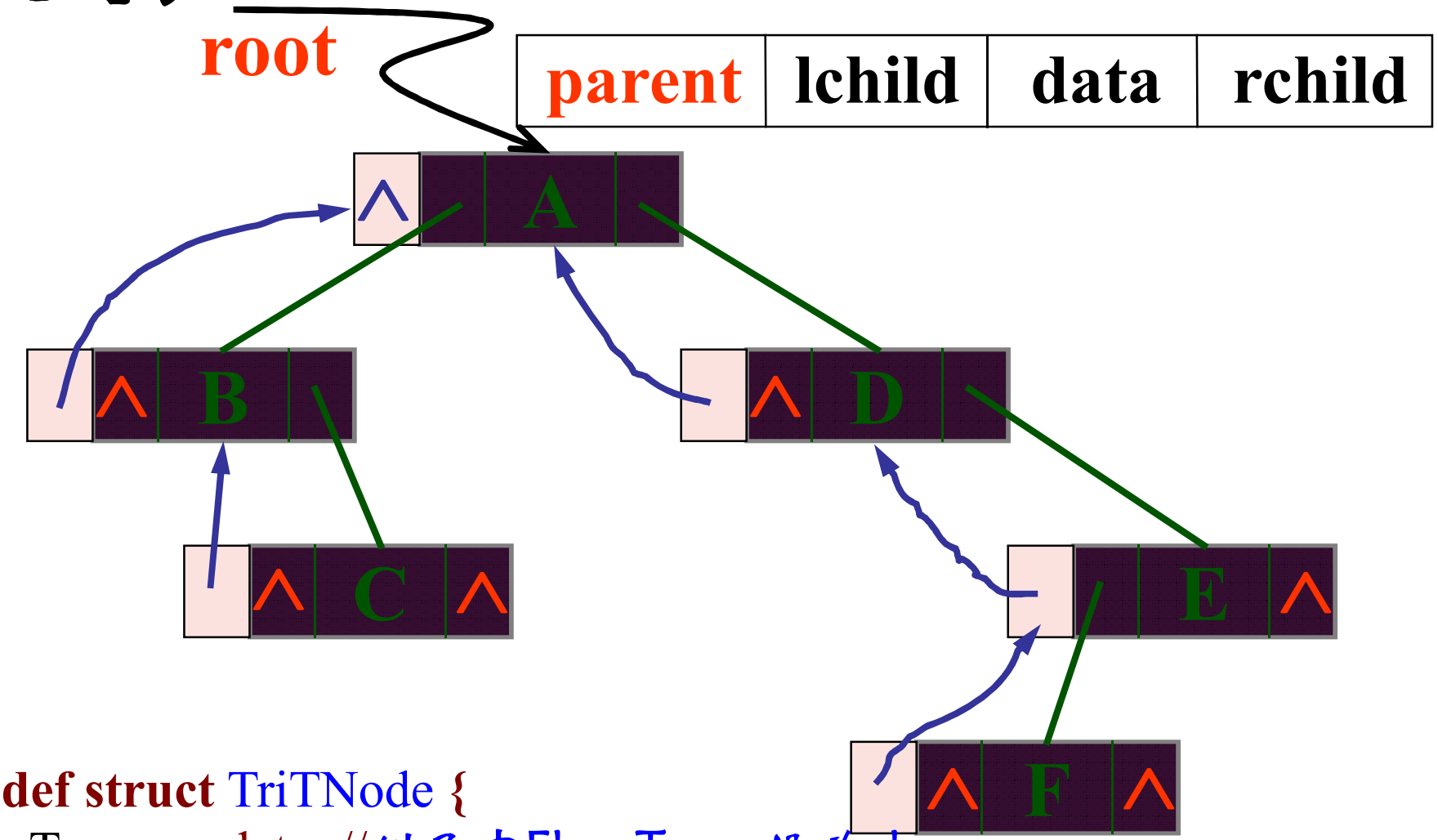
lchild	data	rchild




```
typedef struct BiTNode {
    ElemType      data;   //例子中ElemType设为char
    struct BiTNode  *lchild, *rchild; //左右子树根结点地址
} BiTNode, *BiTree;
```

三叉链表

结点结构：




```
typedef struct TriTNode {
 ElemType       data; //例子中ElemType设为char
 struct TriTNode  *lchild, *rchild; //左右子树根结点地址
 struct TriTNode  *parent; //双亲结点地址
 } TriTNode, *TriTree;
```

双亲数组存放二叉树，一个数组元素存放二叉树中一个结点：该结点的值，其双亲的存放位置（在数组中的下标），以及他是其双亲的左还是右孩子

双亲数组

0	B	2	L
1	C	0	R
2	A	-1	
3	D	2	R
4	E	3	R
5	F	4	L
6			

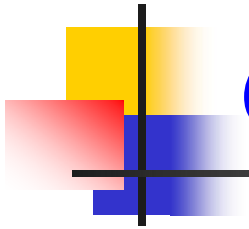
结点结构：

data	parent	LRTag

```
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
typedef struct BPTNode {
    ElemType data;//结点的值
    int parent; //双亲结点的位置
    char LRTag; //左孩子、右孩子的标记
} BPTNode
typedef struct BPTree{
    BPTNode nodes[MAX_TREE_SIZE];
    int n, r; //n为树中结点数，r为根结点位置
} BPTree
```

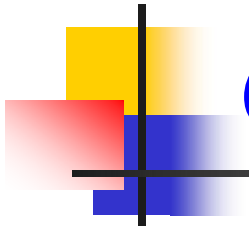
！！！：双亲数组不是二叉树的顺序存储结构，而是链式存储结构！它不是通过存放位置表示逻辑关系，而是需要通过parent这个“指针”明确父子关系

说明：双亲数组表示法不能保证二叉树的根结点一定存放于下标为“0”的第一个数组元素。在一些应用中，初始时无法确定根结点，随着逻辑关系的分析，一步步最后确定根结点，这样无法在建立双亲数组时确保根结点一定会放在下标为“0”处



6.3 遍历二叉树和线索二叉树

- 本节主要内容:
  - 二叉树遍历的递归算法
  - 二叉树遍历的非递归算法
  - 线索二叉树

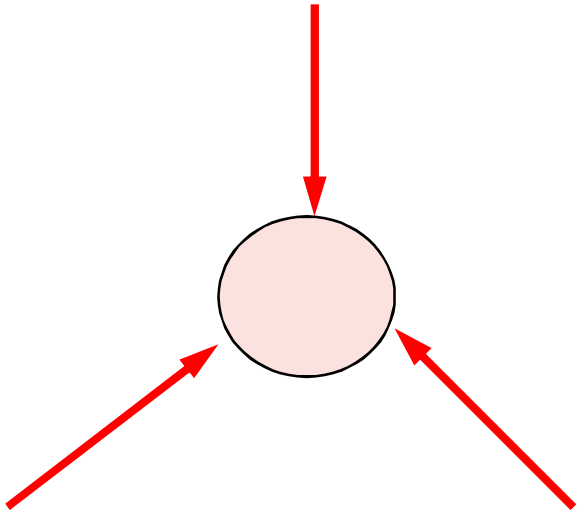


6.3.1遍历二叉树

- 遍历：顺着某一条搜索路径巡访二叉树中的结点，使得每个结点均被访问一次，而且仅被访问一次。
- “访问”的含义可以很广，如：对结点进行处理、输出结点的信息等。

6.3.1遍历二叉树----策略

约定：左子树先于右子树访问

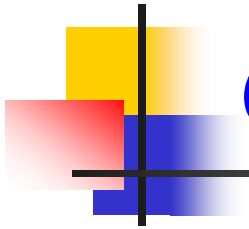


先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法

层次遍历算法



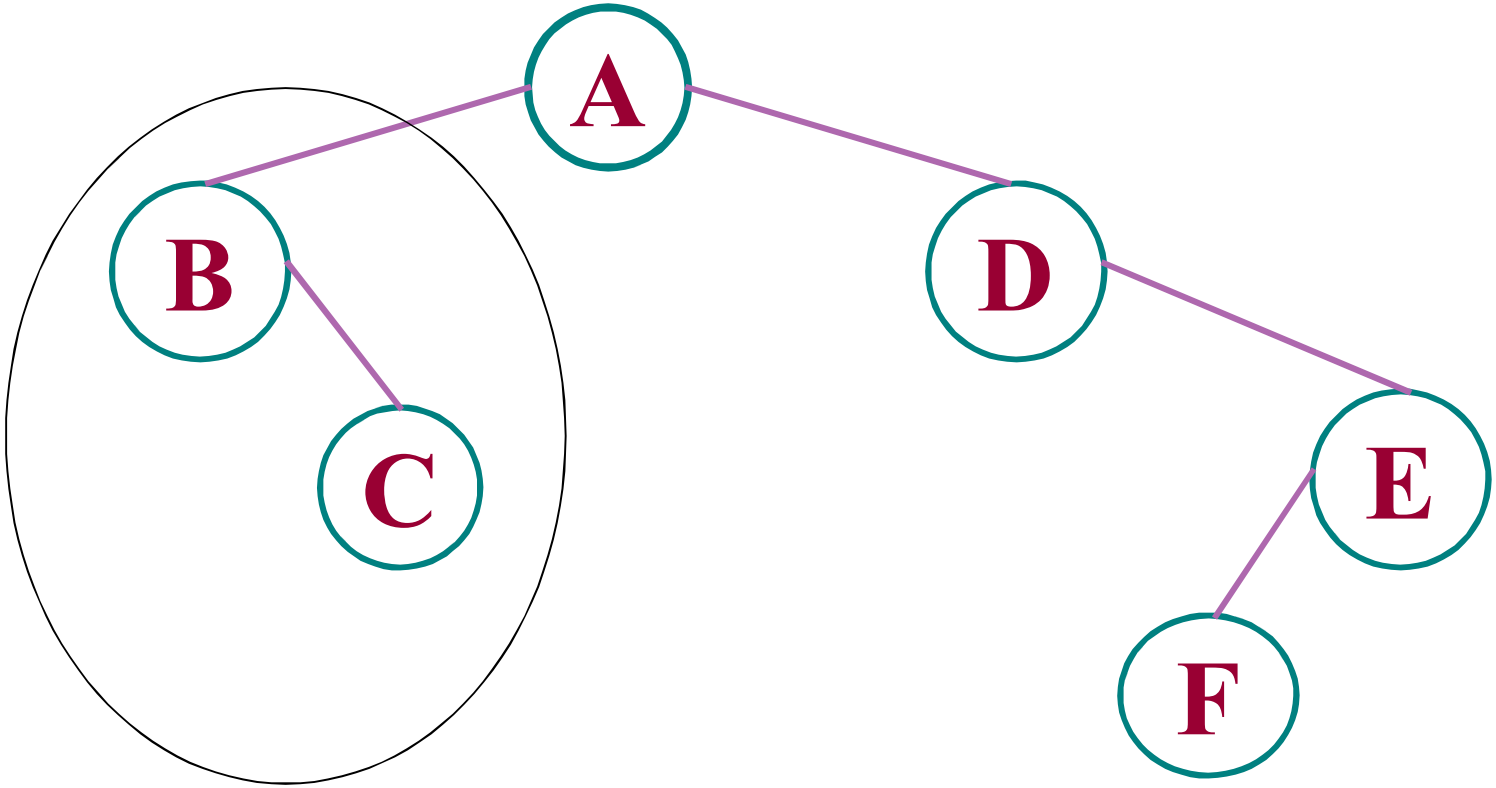
6.3.1遍历二叉树--先序（根）遍历

若二叉树为空树，则空操作；否则，

（1）访问**根**结点；

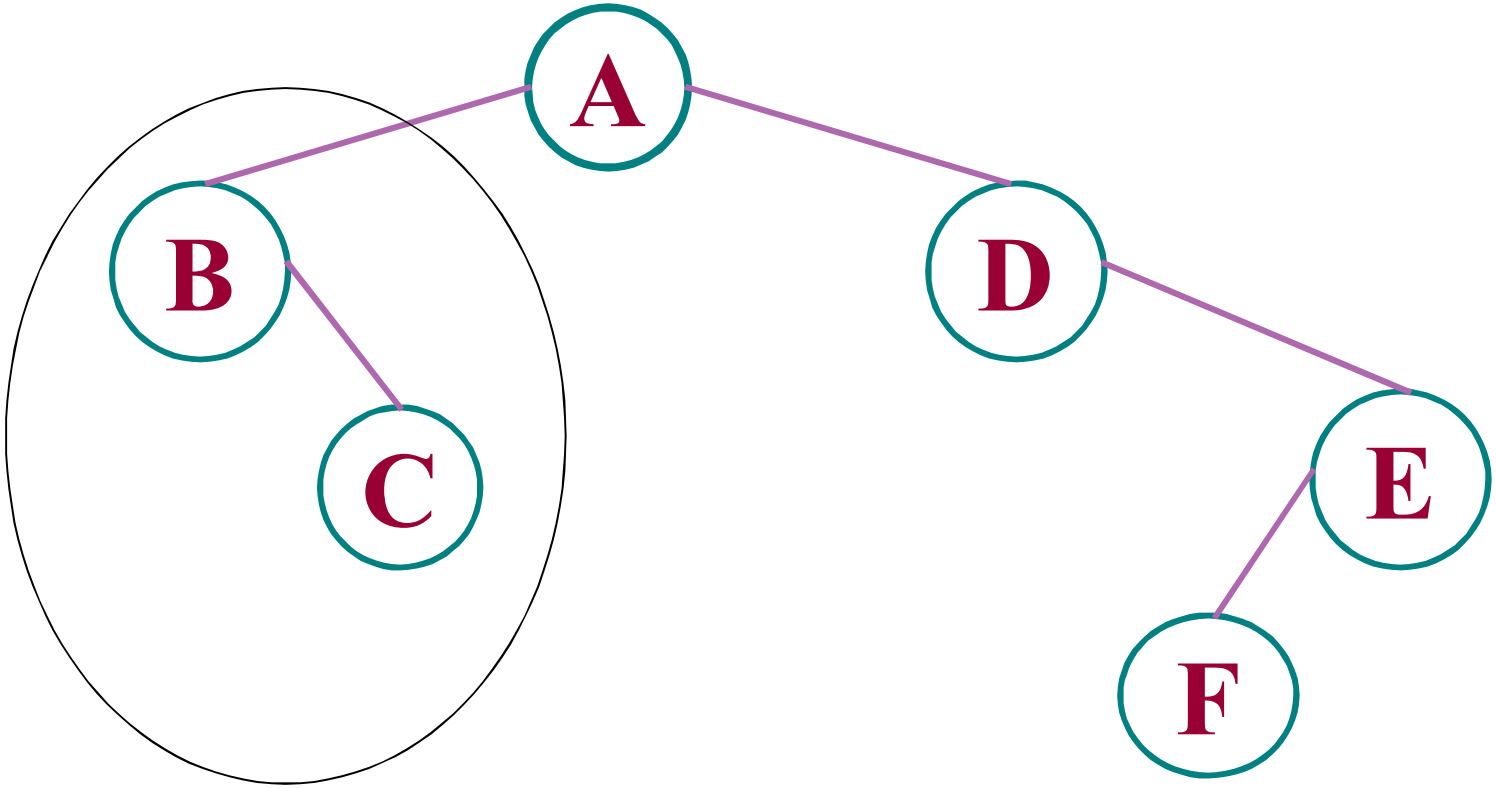
（2）**先序**遍历**左**子树；

（3）**先序**遍历**右**子树。



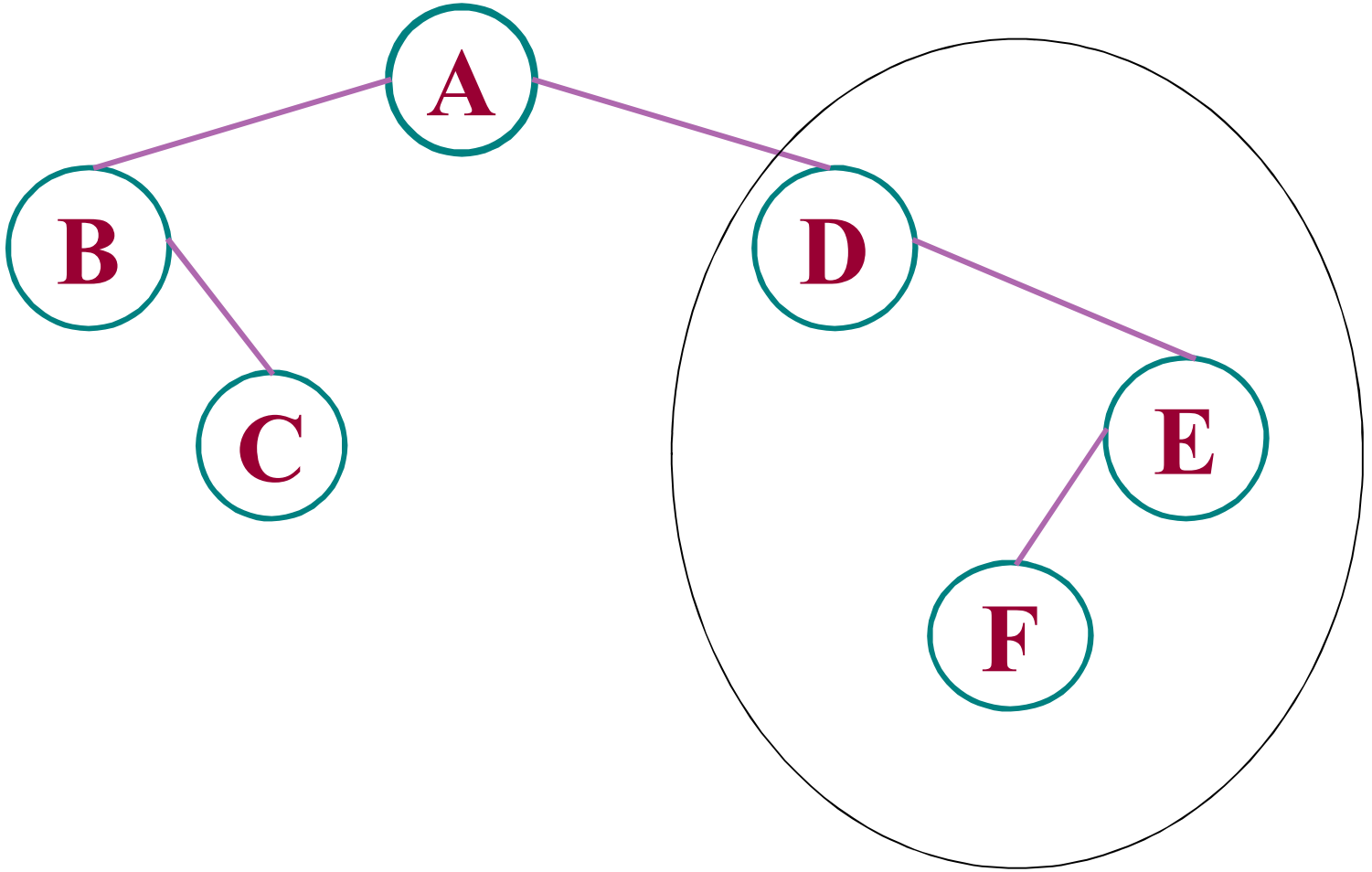
A 左子树 右子树

先序遍历



A　　BC　　右子树

先序遍历

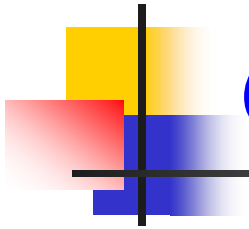


A BC D EF

先序遍历

先序遍历的递归算法

```
void xxbl (BiTree T )
{
  if (T) {
    printf("%c",T->data);   // 访问结点
    xxbl(T->lchild); // 遍历左子树
    xxbl(T->rchild);// 遍历右子树
  }
}
```



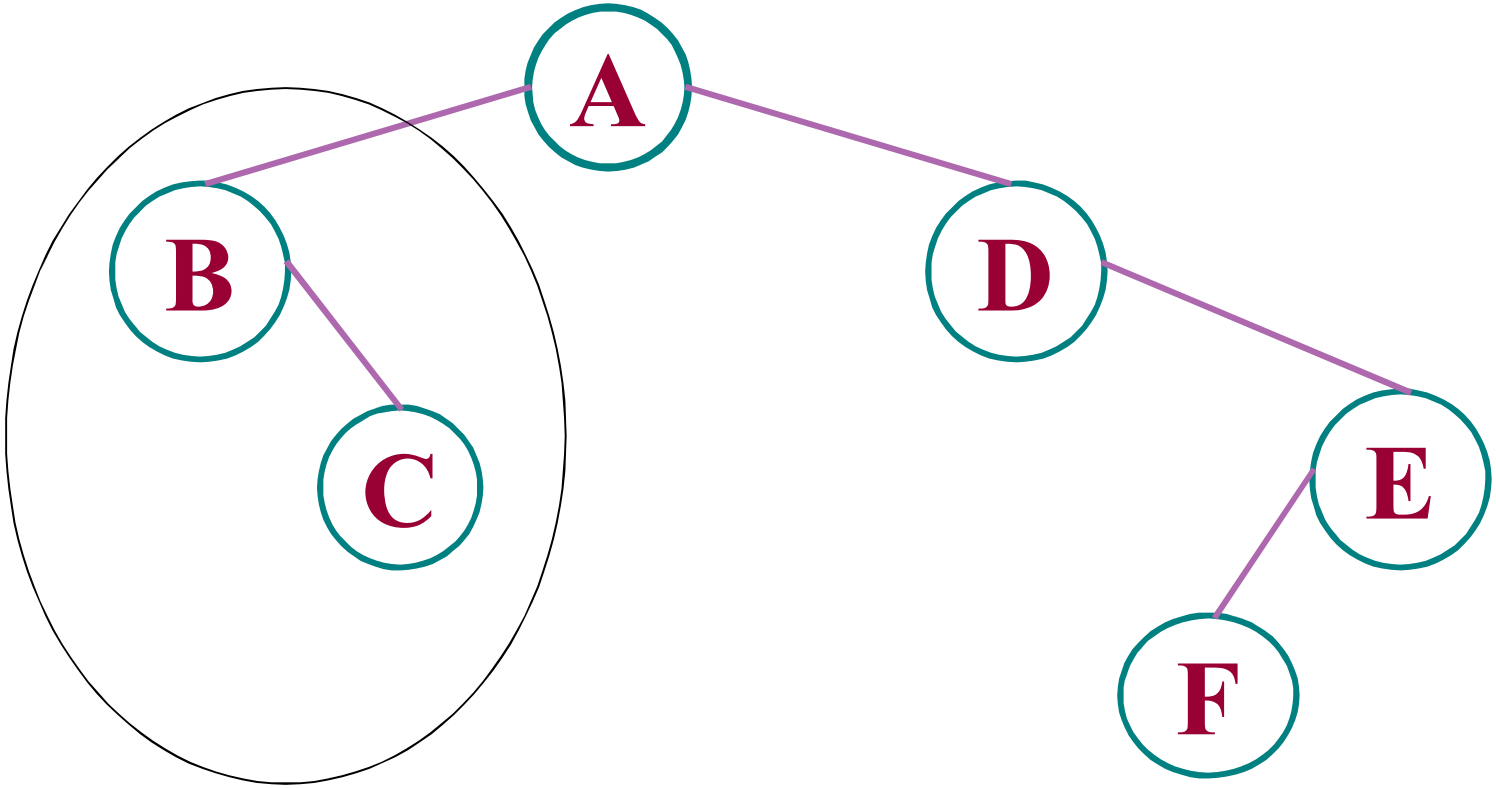
6.3.1遍历二叉树——中序（根）遍历

若二叉树为空树，则空操作；否则，

（1）**中序**遍历**左**子树；

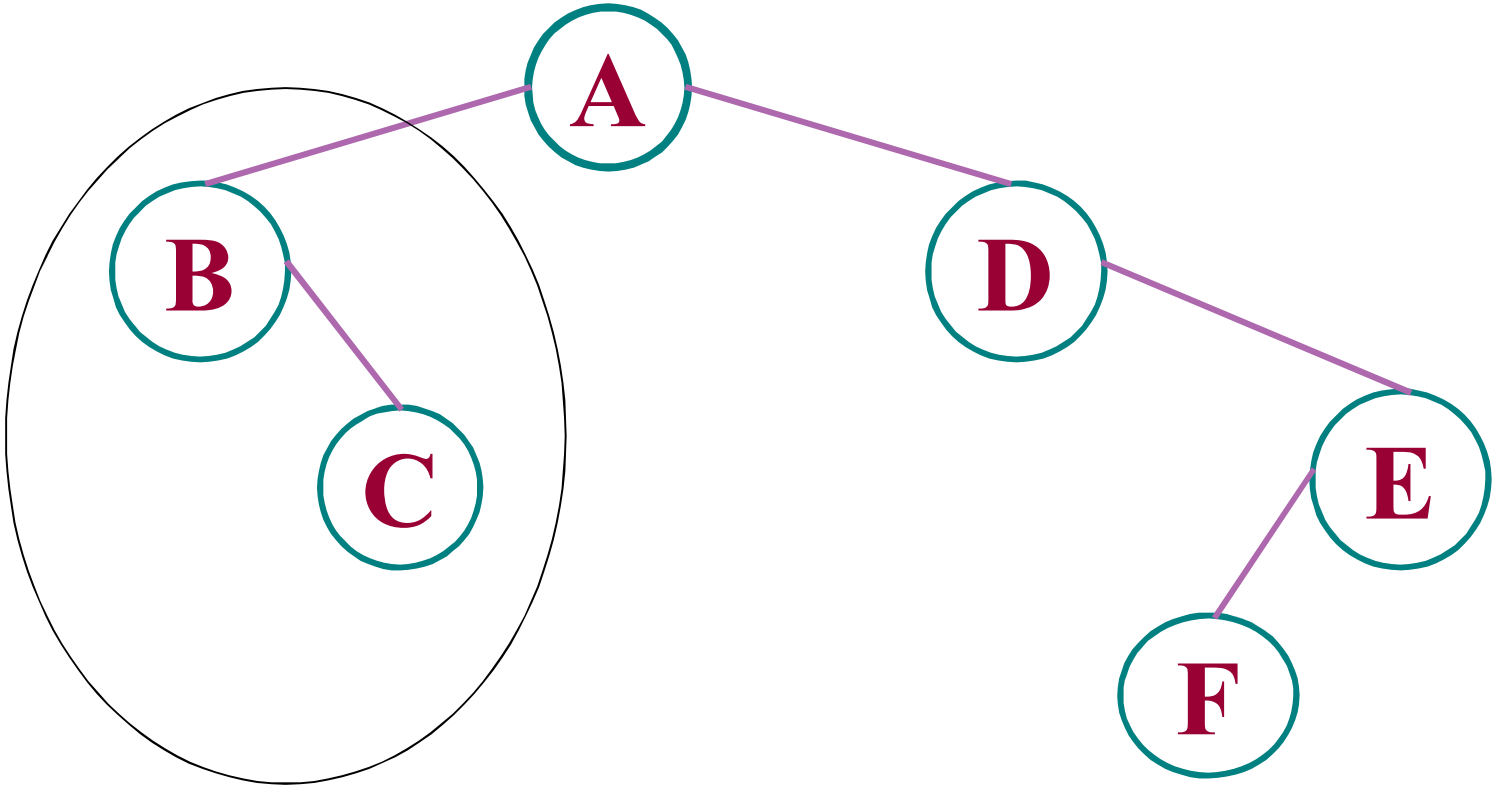
（2）访问**根**结点；

（3）**中序**遍历**右**子树。



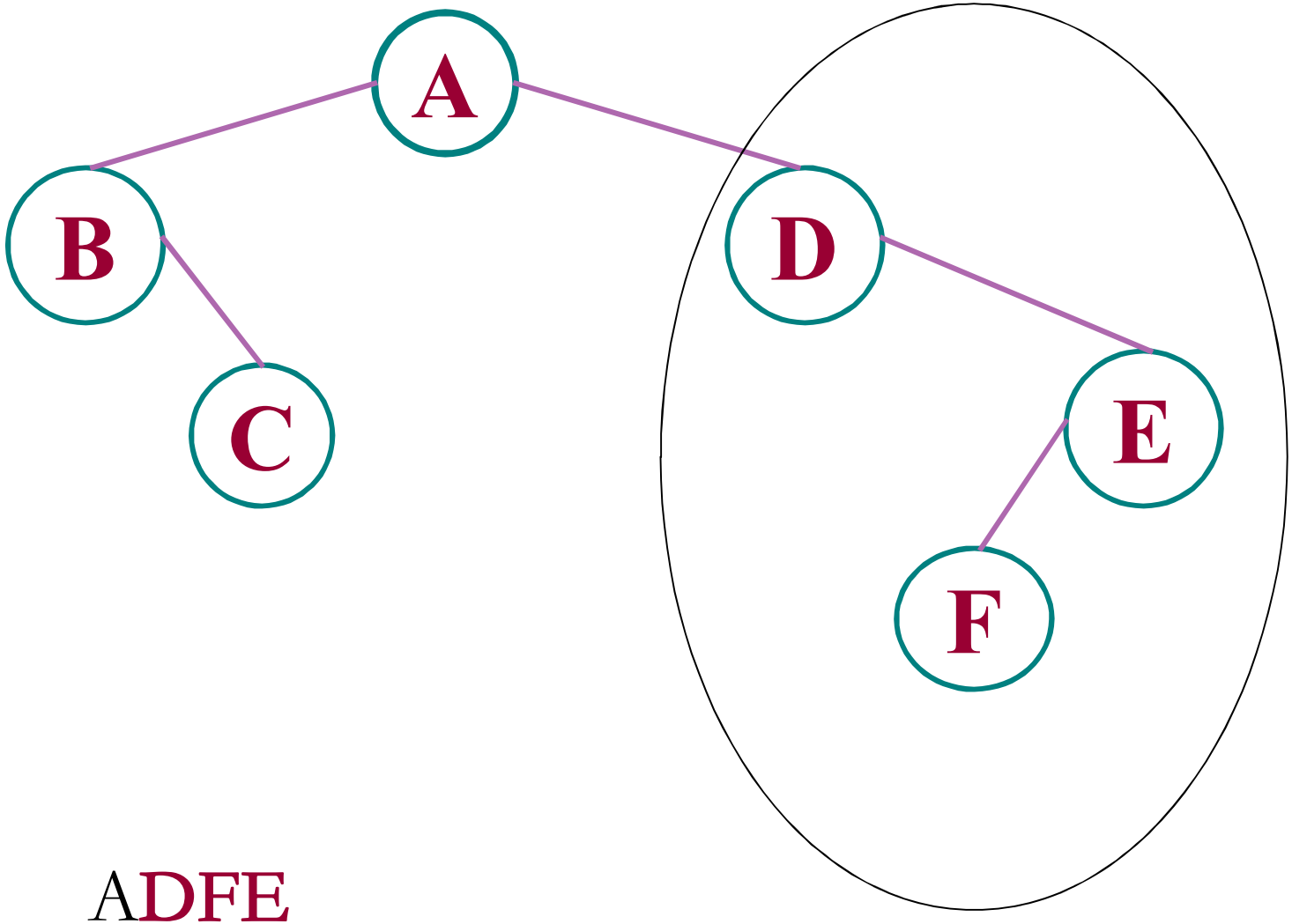
- 左子树 A 右子树

中序遍历



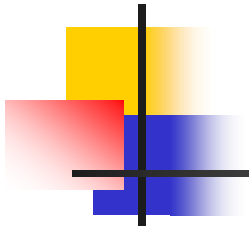
- BC　A右子树

中序遍历



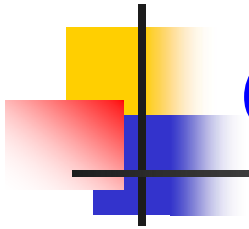
- BC　　ADFE

中序遍历



中序遍历的递归算法

```
void zxbl (BiTree T )
{
  if (T) {
     zxbl(T->lchild);
     printf("%c",T->data);
     zxbl(T->rchild);
  }
}
```



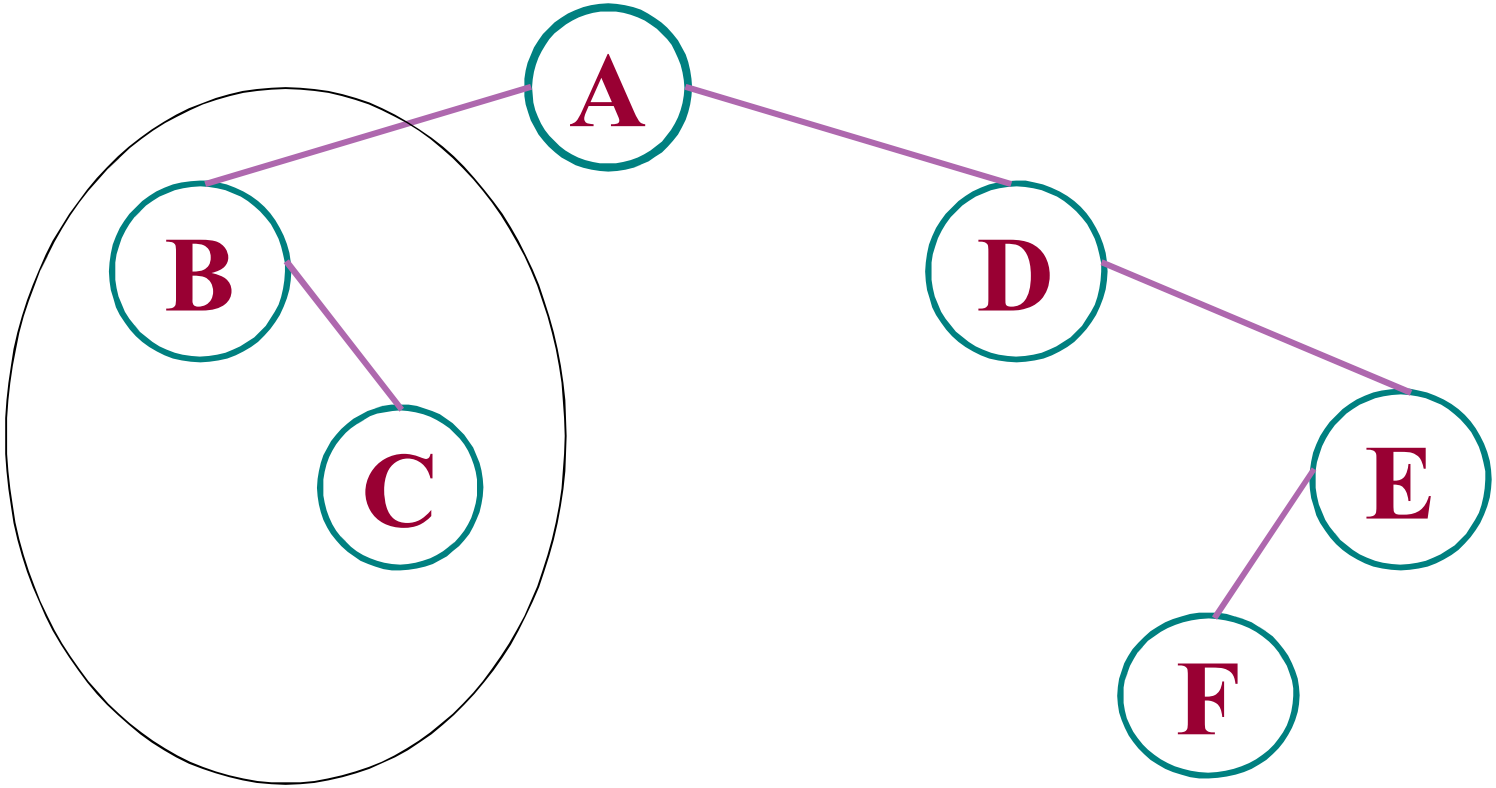
6.3.1遍历二叉树——后序（根）遍历

若二叉树为空树，则空操作；否则，

（1）**后序**遍历**左**子树；

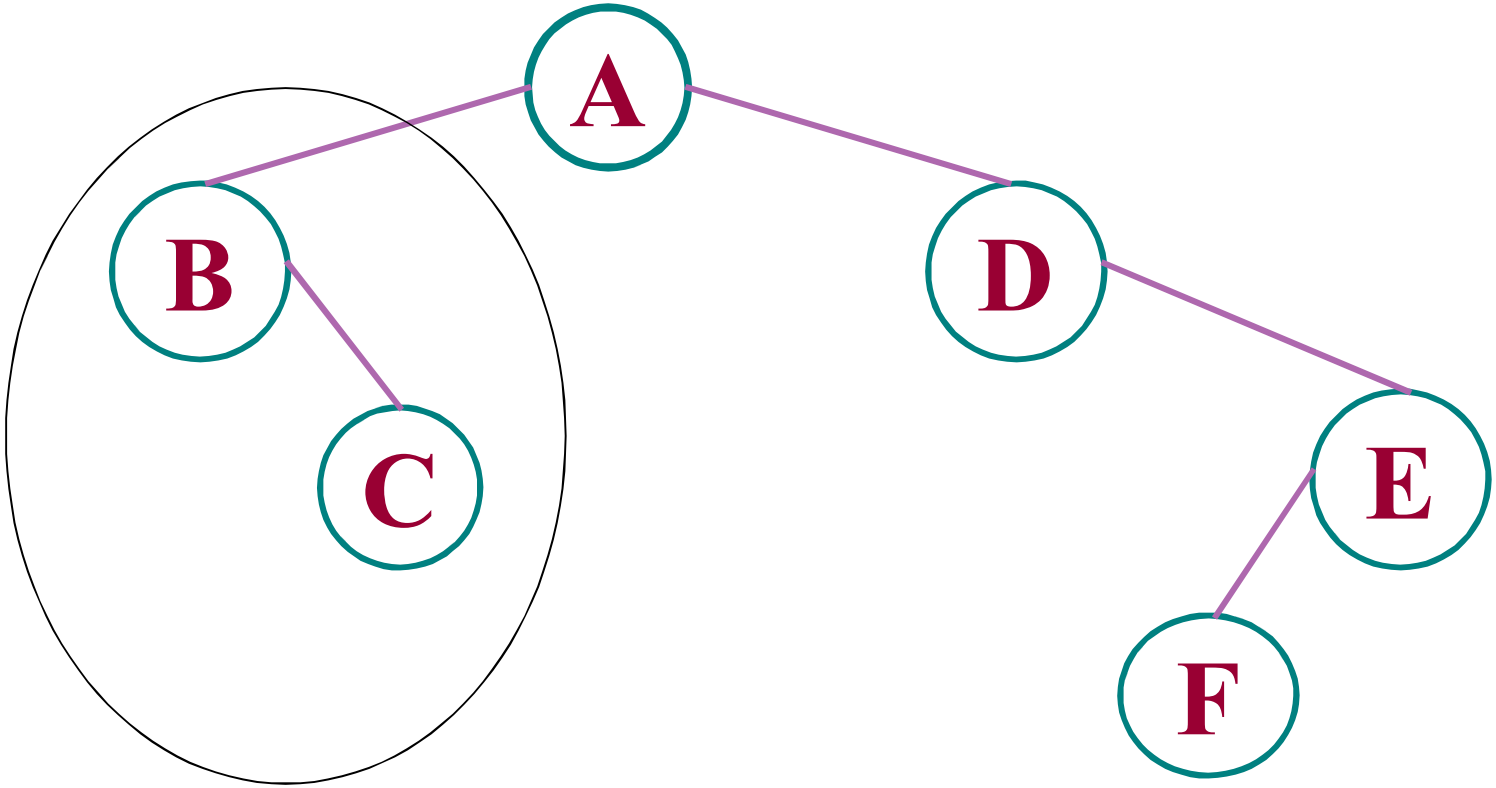
（2）**后序**遍历**右**子树；

（3）访问**根**结点。



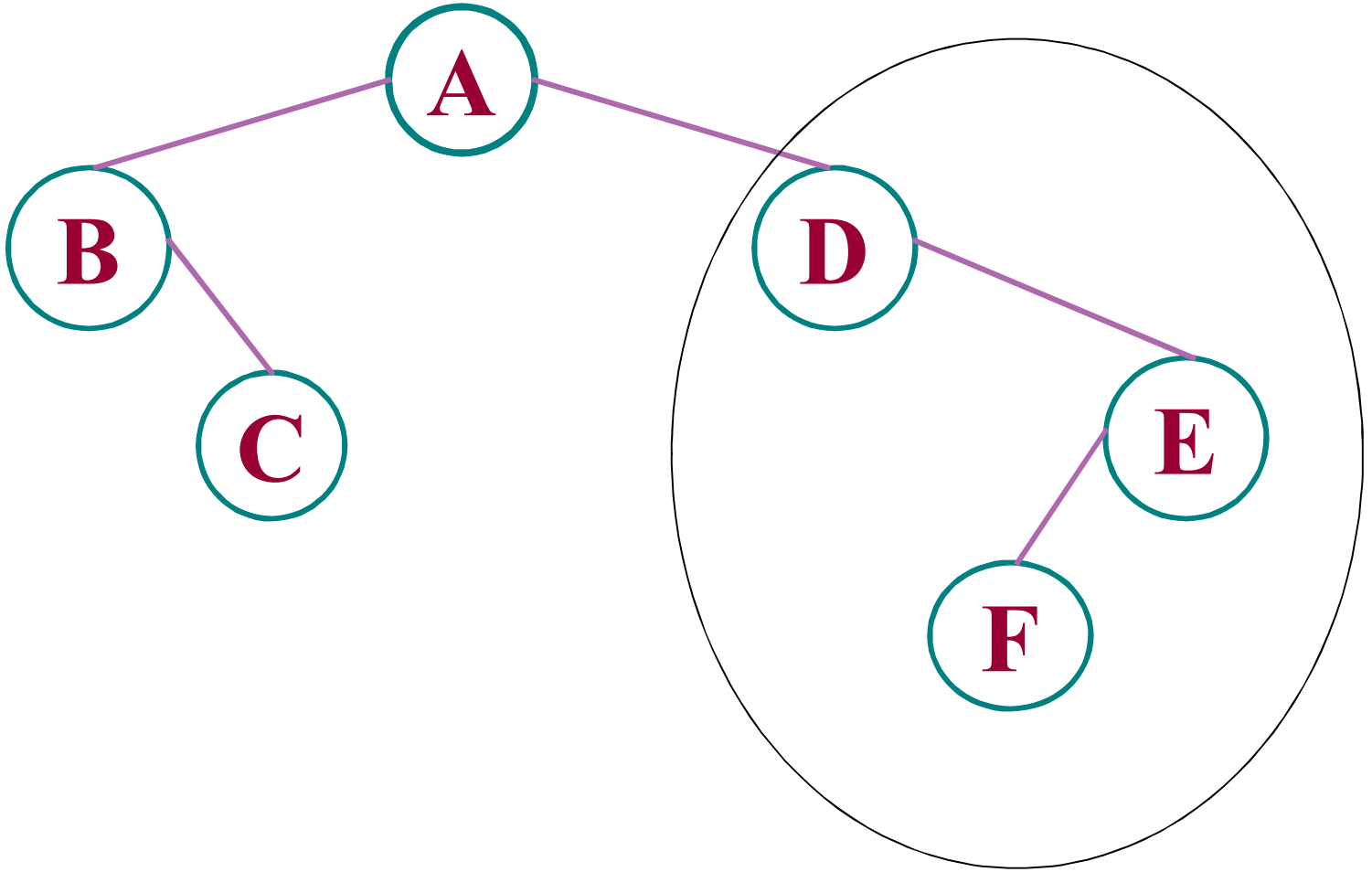
- 左子树 右子树 A

后序遍历



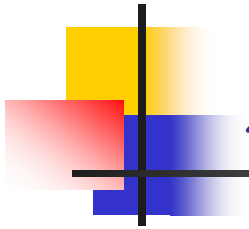
- CB　　右子树A

后序遍历



- CB　　FED A

后序遍历



后序遍历的递归算法

```
void hxbl (BiTree T )
{
  if (T) {
    hxbl(T->lchild);
    hxbl(T->rchild);
    printf("%c",T->data);
  }
}
```

6.3 遍历二叉树----应用

- 计算二叉树中叶子结点的个数
- 求二叉树的高
- ……

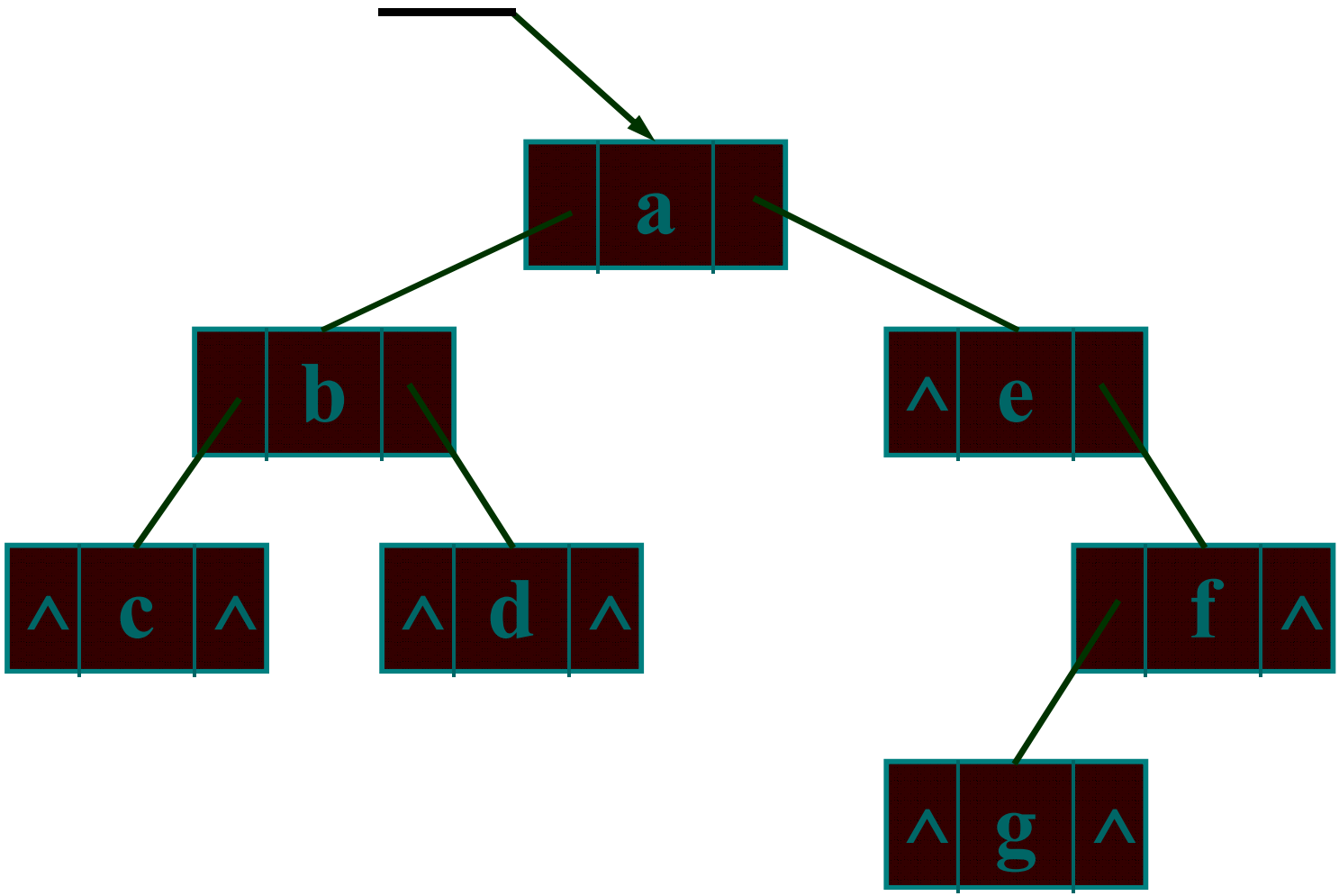
6.3 遍历二叉树--相关结论

- 树中结点均不相同
- 先序+中序 ⟶ 唯一确定一棵二叉树
  - 已知二叉树的先序序列“*abcdefg*”，能否唯一确定一棵二叉树?
  - 如果同时已知二叉树的中序序列“*cbdaegf*”，能否唯一确定一棵二叉树?
  - 二叉树的先序序列
  - 二叉树的中序序列

例如：

a b c d e f g　先序序列

c b d a e g f　中序序列

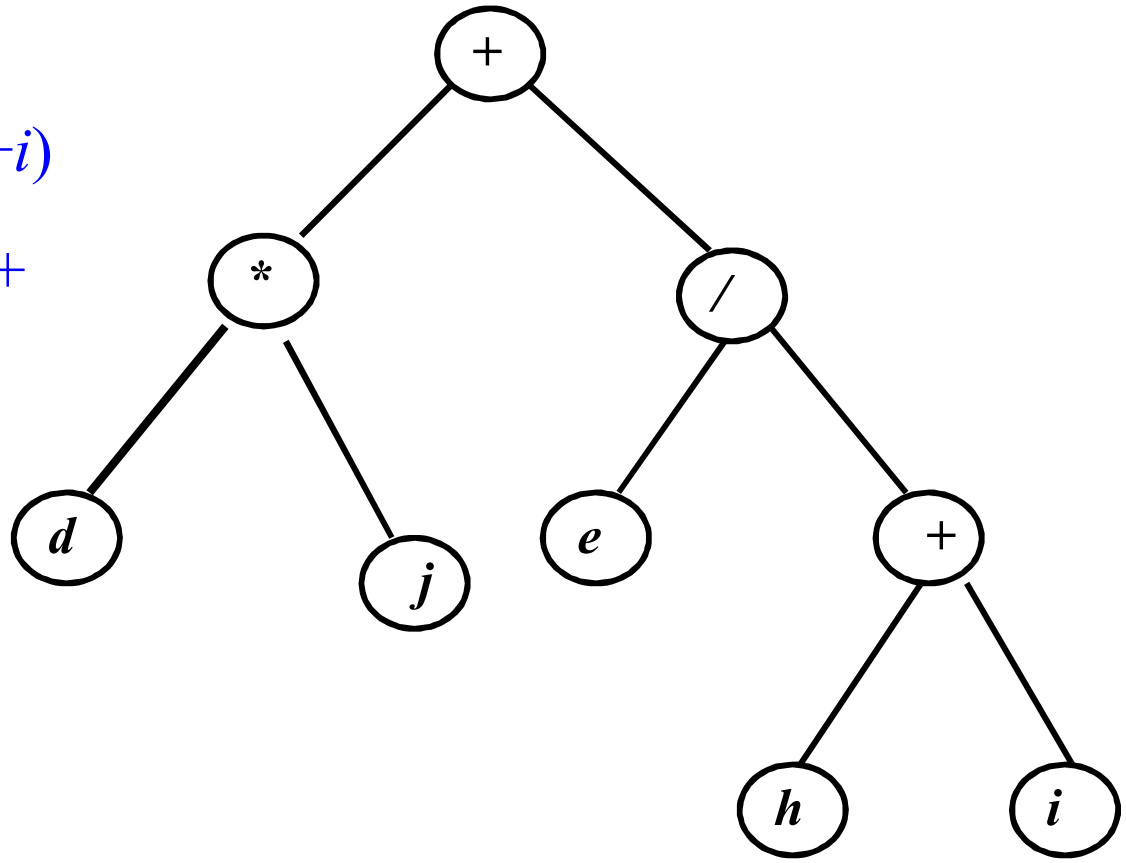


6.3 遍历二叉树--相关结论

- 树中结点均不相同
- 先序+中序 ⟶ 唯一确定一棵二叉树
- 后序+中序 ⟶ 唯一确定一棵二叉树
- 先序+后序 ⟶ 不能唯一确定一棵二叉树

6.3 二叉树—应用之一

- 二叉树存放表达式
- 波兰式： +*dj/e+hi
- 中缀表示 d*j+e/(h+i)
- 逆波兰式 dj*ehi+/+

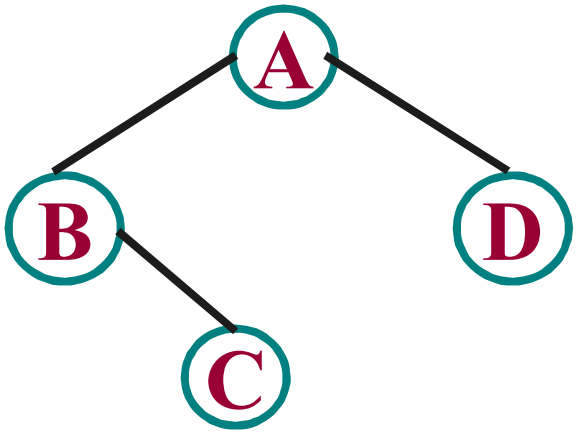


6.3 二叉树的建立

- **以先序序列定义一棵二叉树**：输入所要建立的二叉树的**先序序列**，建立二叉链表
- 在输入二叉树的**先序序列**中，加入**空树**的明确表示
- 例如：空树的**先序序列**以“#”表示
- 只含一个根结点A的二叉树的**先序序列**以字符串“A##”表示

6.3 二叉树的建立

- **以先序序列定义一棵二叉树**：输入所要建立的二叉树的**先序序列**，建立二叉链表
- 在输入二叉树的**先序序列**中，加入**空树**的明确表示



AB # C##D##

6.3 二叉树的建立

```
void CreateBiTree(BiTree &T) {
    scanf("%c",&ch);
    if (ch=='# ') T = NULL;
    else {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        T->data = ch;              // 生成根结点
      CreateBiTree(T->lchild);   // 构造左子树
      CreateBiTree(T->rchild);   // 构造右子树
        }
    }
```

上页算法执行过程举例如下：

A B # C# #D# #