数据结构与算法

主讲教师： 刘峤

第6章 树与二叉树

第6章 内容提要

6.1 树的定义与基本操作

6.2 二叉树

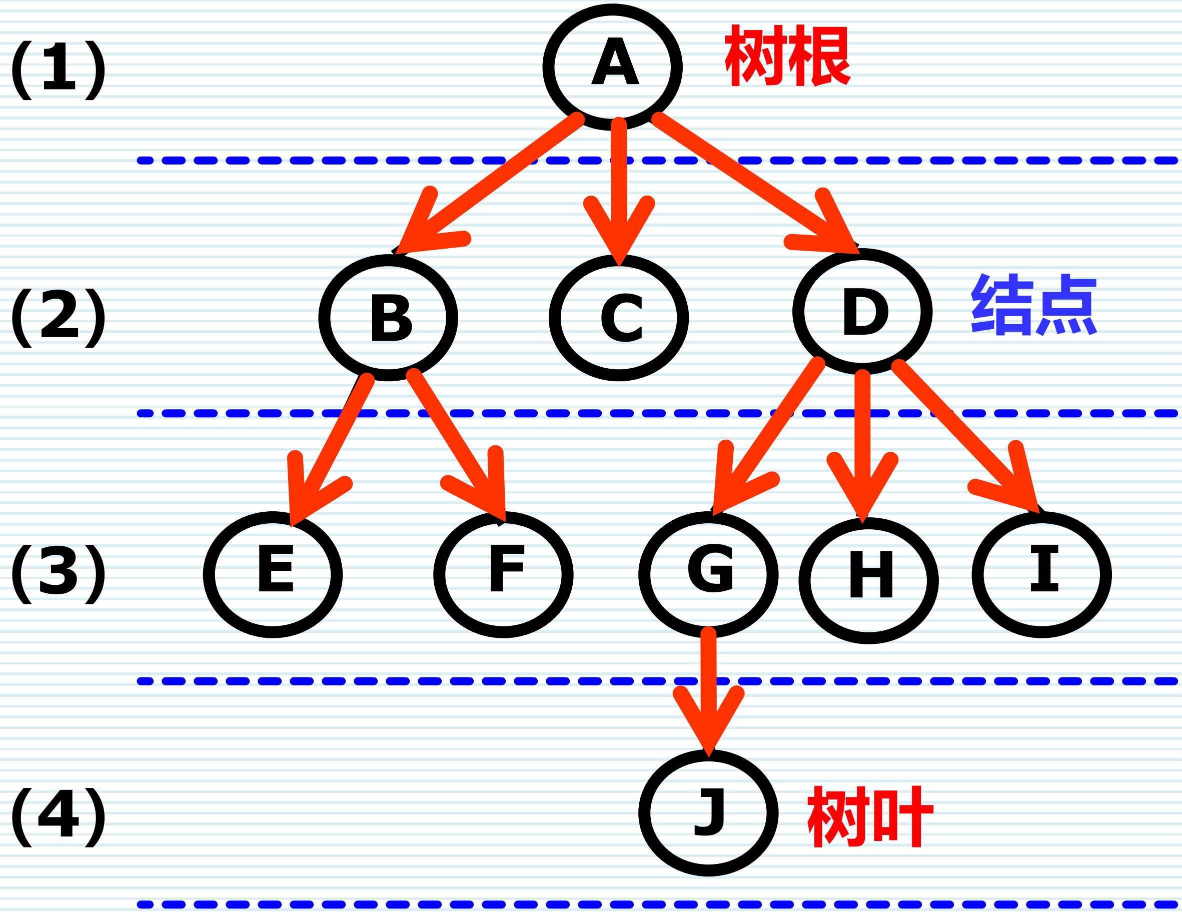
6.3 树和森林

6.4 哈夫曼树与哈夫曼编码

6.5 堆排序算法

1. 树的定义与基本操作

树的概念



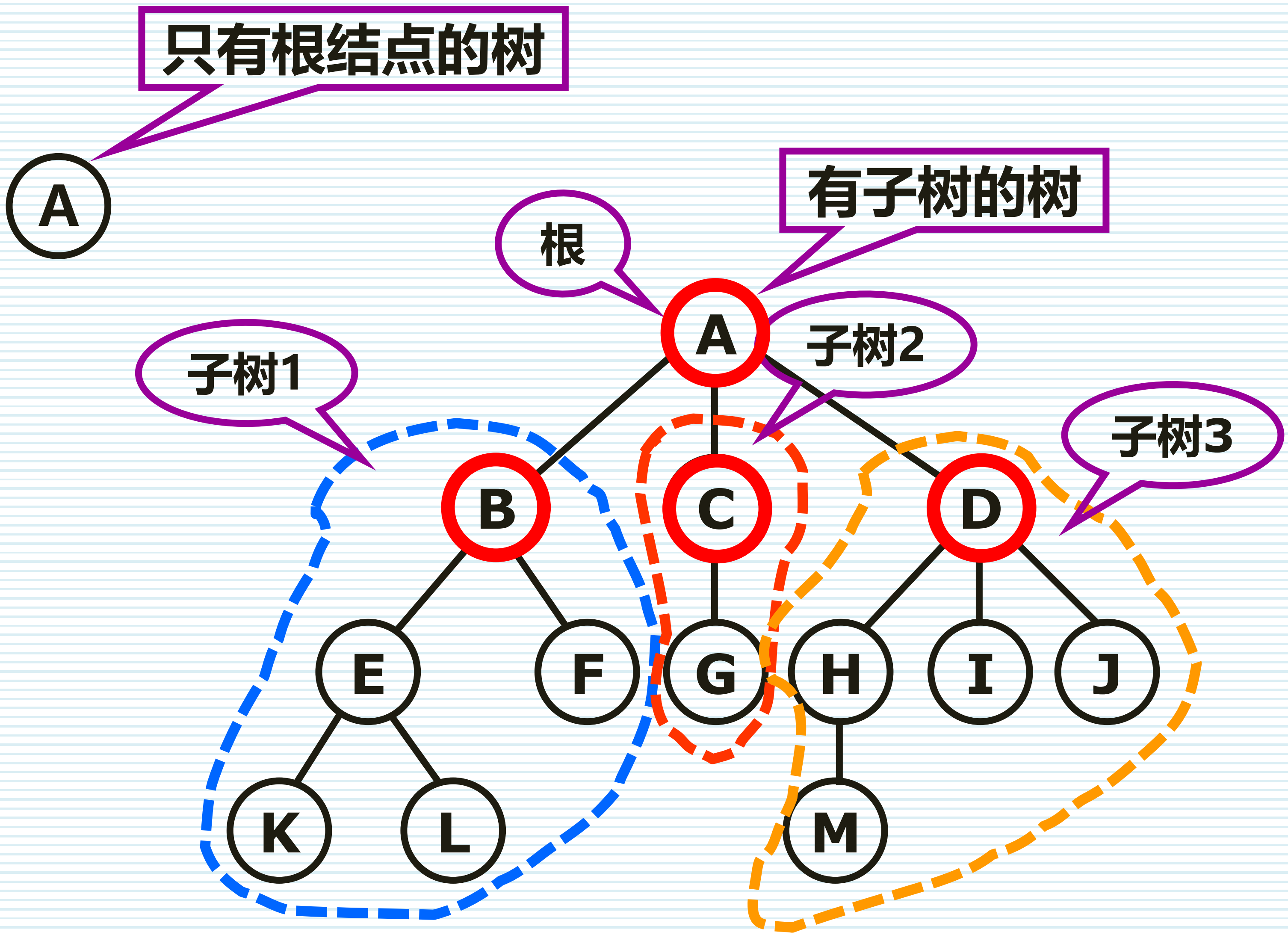
树描述的是一种层次结构

元素间是一对多的逻辑关系

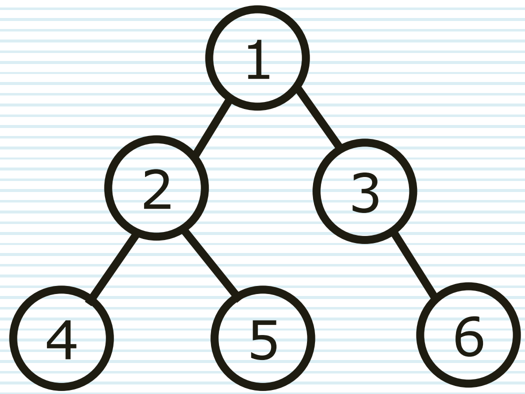
树的概念

- 树（Tree）：是定义在一对多关系上的层次化数据结构
- 树的定义：树是n（n≥0）个结点的有限集T，其中：
  - 有且仅有一个特定的结点，称为树的根（root）
  - 当n>1时，其余结点可分为m个互不相交的有限集
    - $T_1, T_2, \ldots\ldots T_m$
  - 其中每一个集合本身又是一棵树
    - 这些子集构成的树称为根的子树（subtree）
- 树的特点：
  - 树中至少有一个结点：根
  - 树中各子树是互不相交的集合

树的概念

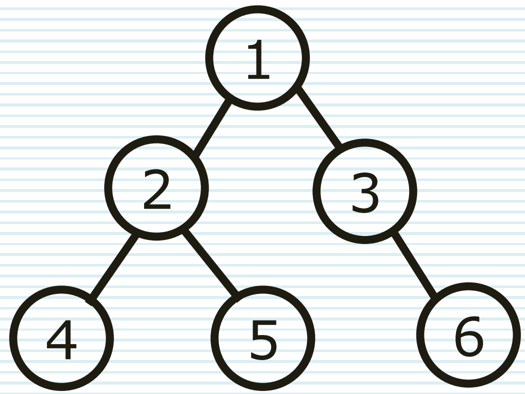


树的基本术语



- 结点（**node**）：表示树中的元素（包括数据项及分支项）
- 结点的度（**degree**）：结点拥有的子树个数
  - 树的度：一棵树中最大的结点度数
- 叶结点（**leaf**）：度为0的结点
- 子结点（**child**）：结点子树的根称为该结点的子结点
- 父节点（**parents**）：子结点的上层结点叫该结点的父节点

树的基本术语



- 兄弟结点（**sibling**）：同一双亲的孩子
- 堂兄弟（**cousin**）：双亲在同一层的结点互为堂兄弟
- 结点的层次（**level**）：从根结点算起
  - 根为第一层，它的孩子为第二层……依此类推
- 树的深度（**depth**）：树中结点的最大层次数
- 森林（**forest**）：m（m≥0）棵互不相交的树的集合

结点A的度: **3**

结点B的度: **2**

结点M的度: **0**

树的度: **3**

树的深度: **4**

结点I的父结点: **D**

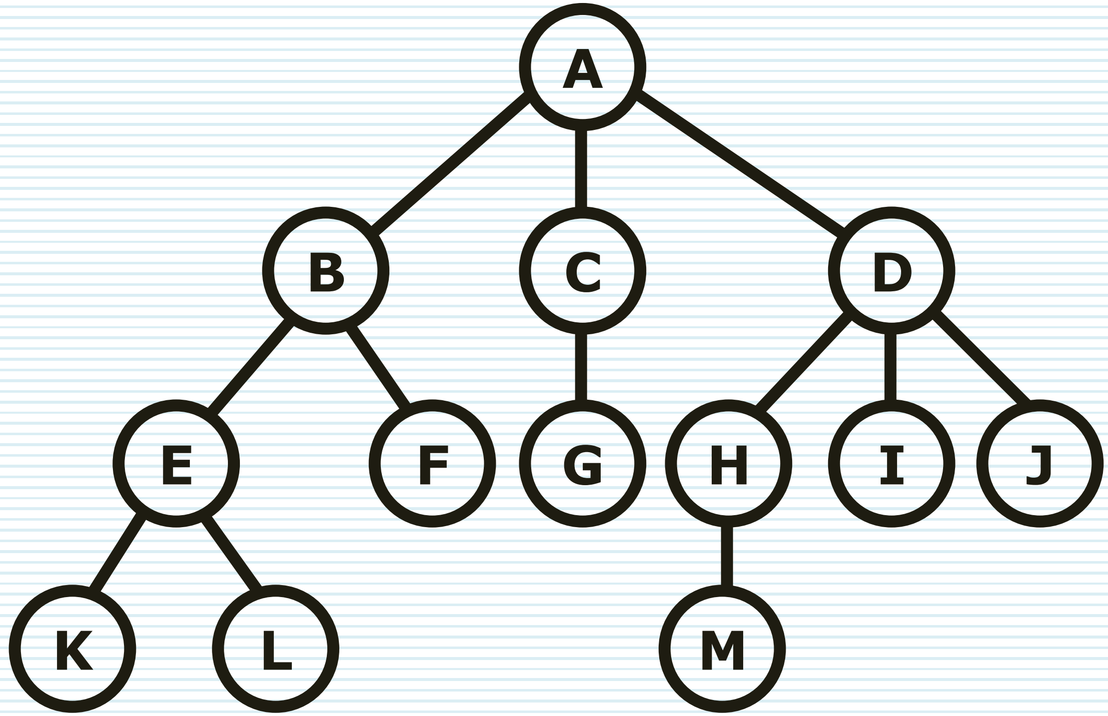
结点L的父结点: **E**

结点A的层次: **1**

结点M的层次: **4**

结点A的孩子: **B, C, D**

结点B的孩子: **E, F**



结点B，C，D的关系: 兄弟

结点K，L的关系: 兄弟

结点F，G的关系: 堂兄弟

叶结点: **K, L, F, G, M, I, J**

树结构和线性结构的比较

线性结构	树结构
第一个数据元素 **无前驱**	根结点 **无双亲**
最后一个数据元素 **无后继**	叶结点 **无孩子**
其它数据元素 **一个前驱；一个后继**	其它结点 **一个双亲；多个孩子**
元素之间的逻辑关系 **一对一**	结点之间的逻辑关系 **一对多**

树的基本操作

- InitTree(T) // 树初始化操作
- CreateTree(&T,definition) // 构造树T
- DestroyTree(&T) // 删除树T
- ClearTree(&T) // 清空树T
- TreeEmpty(&T) // 判树空
- TreeDepth(T) // 求树的深度
- Root(T) // 求树根
- Parent(T,x) // 求双亲结点
- LeftChild(T,x) // 求左孩子
- RightSibling(T,x) // 求右兄弟
- InsertChild(&T,&p,i,x) // 插入孩子
- DeleteChild(&T,&p,i) // 删除孩子
- TraverseTree(T) // 树的遍历

2. 二叉树

二叉树的主要内容概览

1. 二叉树的定义与基本操作

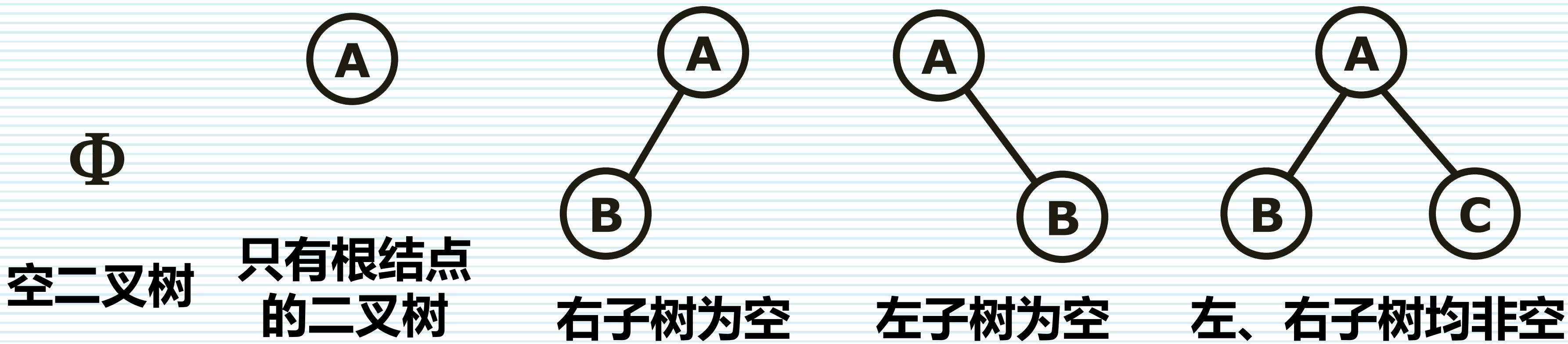
2. 二叉树的性质

3. 二叉树的存储结构

4. 二叉树的遍历

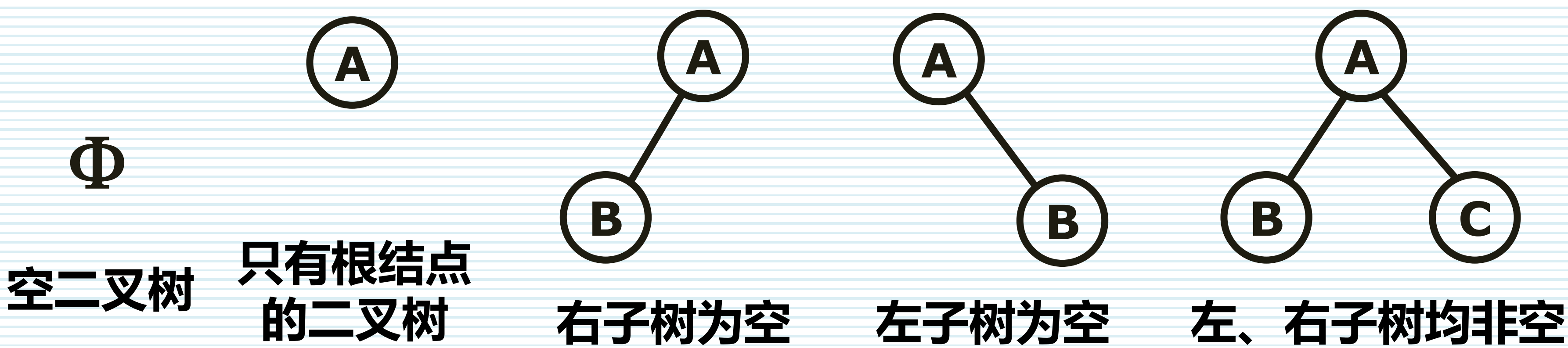
5. 线索化二叉树（自学）

二叉树的定义



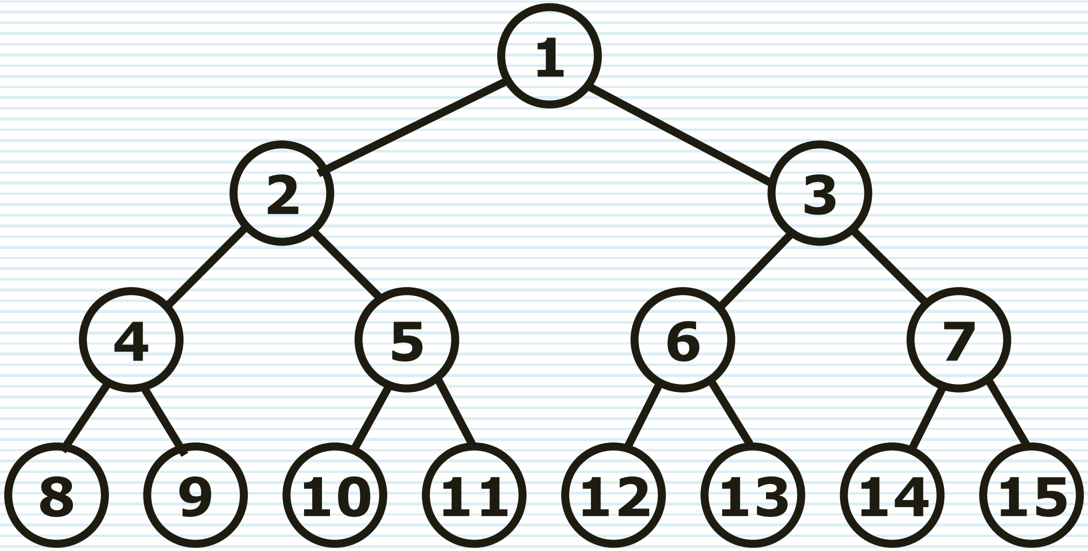
- 定义：二叉树是n（n≥0）个结点的有限集
  - 它或者为空树（n=0）
  - 或者由一个根结点和两棵互不相交的二叉树构成
    - 分别称为左子树和右子树
- 二叉树的基本形态如图

二叉树的定义



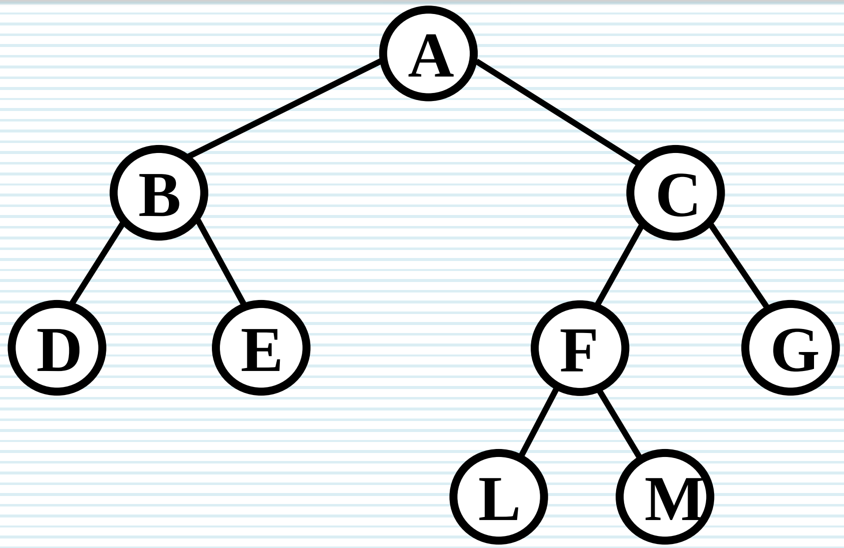
- 二叉树的特点
  - 每个结点至多有二棵子树（即不存在度大于2的结点）
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒
- 研究二叉树的意义?
  - 将树转换为二叉树，从而利用二叉树解决树的有关问题。

满二叉树



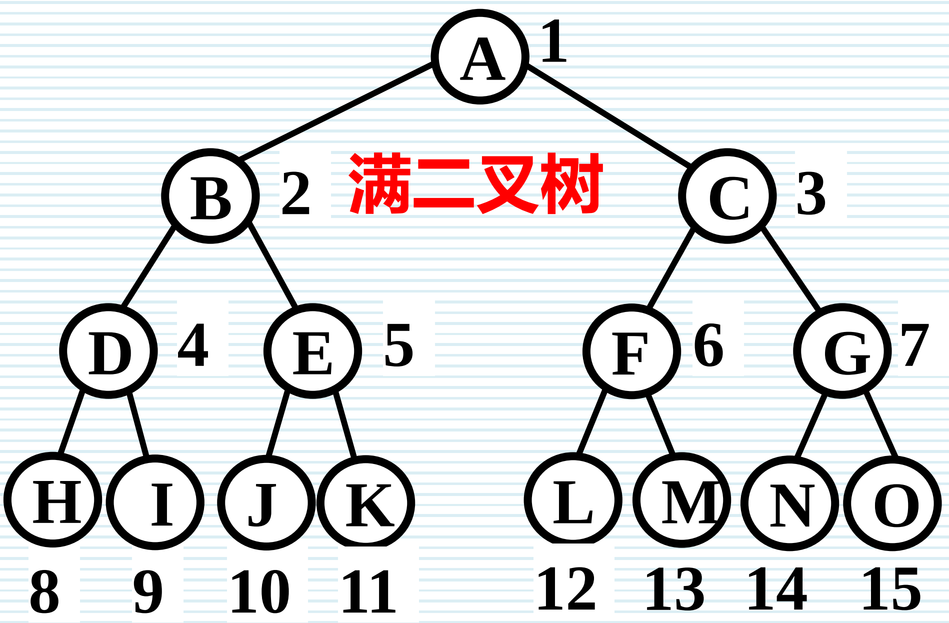
- 满二叉树
  - 深度为k且有2^k-1个结点的二叉树称为满二叉树
- 满二叉树的特点
  - 叶结点只能出现在整棵树的最底层（第k层）
  - 只有度为0和度为2的结点
  - 二叉树中每一层的结点数都达到最大

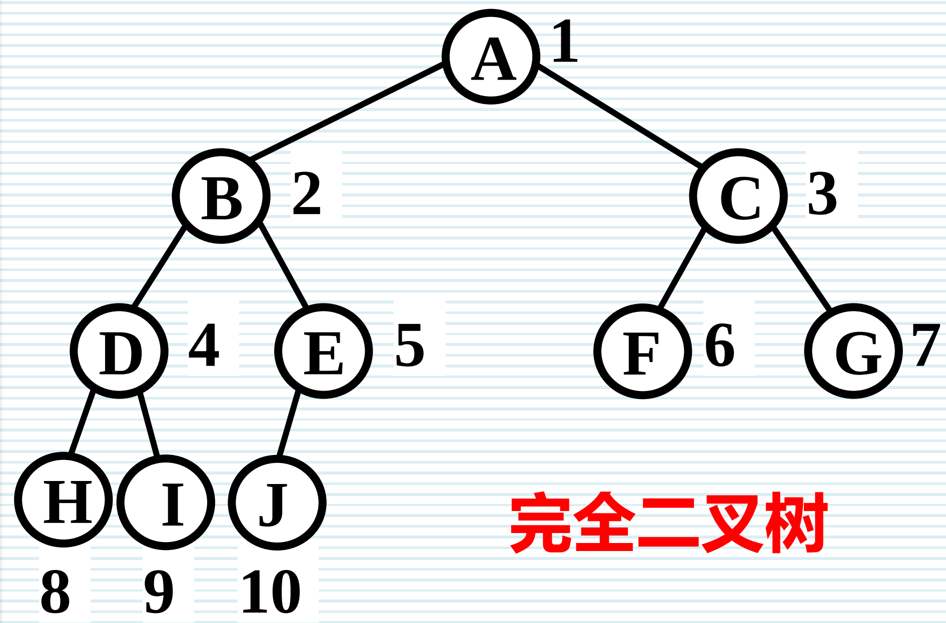
满二叉树



- 思考：如图所示的二叉树是否是满二叉树?
  - 不是：虽然所有内结点都有左右子树，但叶结点不在最底层
- 满二叉树的特点
  - 满二叉树在同样深度的二叉树中**结点**个数最多
  - 满二叉树在同样深度的二叉树中**叶结点**个数最多

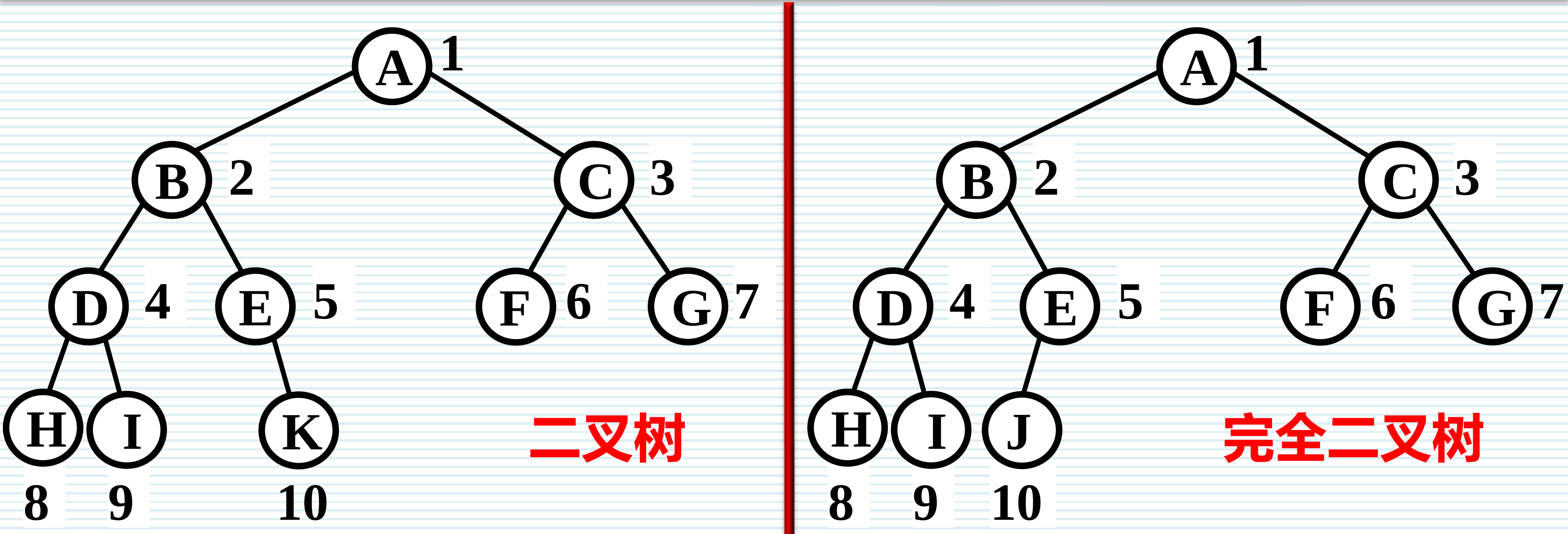
完全二叉树





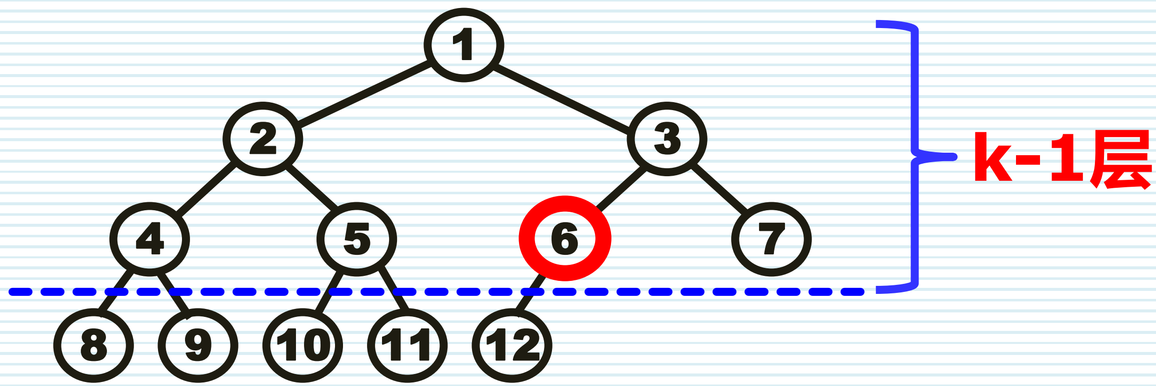
- 完全二叉树
  - 对一棵具有n个结点的二叉树T按层序编号
  - 如果编号为i（1≤i≤n）的结点与同样深度的满二叉树中编号为i的结点在二叉树中的位置完全相同
  - 则称T为完全二叉树

完全二叉树



- 完全二叉树的特点
  - 在满二叉树中，从最后一个结点开始连续去掉任意个结点
  - 即得到一棵完全二叉树
    - 除最后一层外，其余各层都是满的
    - 最后一层或者是满的，或者是右边缺少连续的若干结点

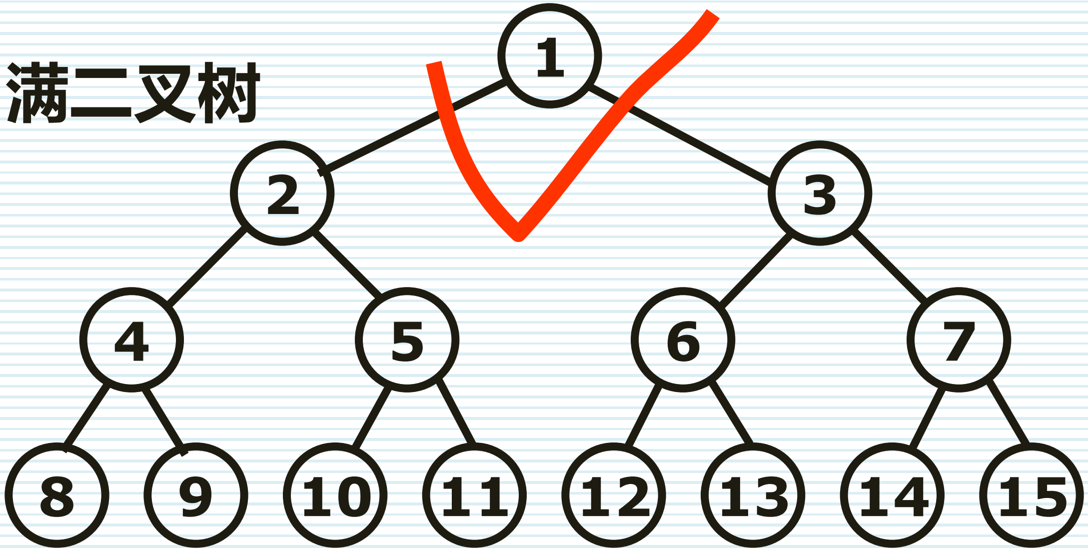
完全二叉树

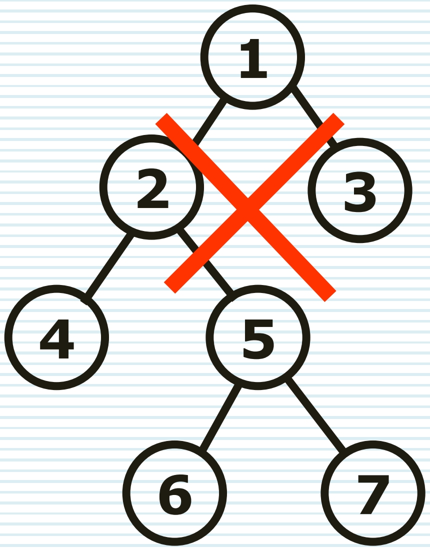


- 完全二叉树的特点
  - 叶结点只能出现在最下两层
    - 且最底层的叶结点都集中在二叉树的左侧
  - 完全二叉树中：度为1的结点只可能有一个
    - 且该结点只有左孩子
  - 深度为k 的完全二叉树在k-1层之上一定是满二叉树
  - 满二叉树是完全二叉树，完全二叉树不一定是满二叉树

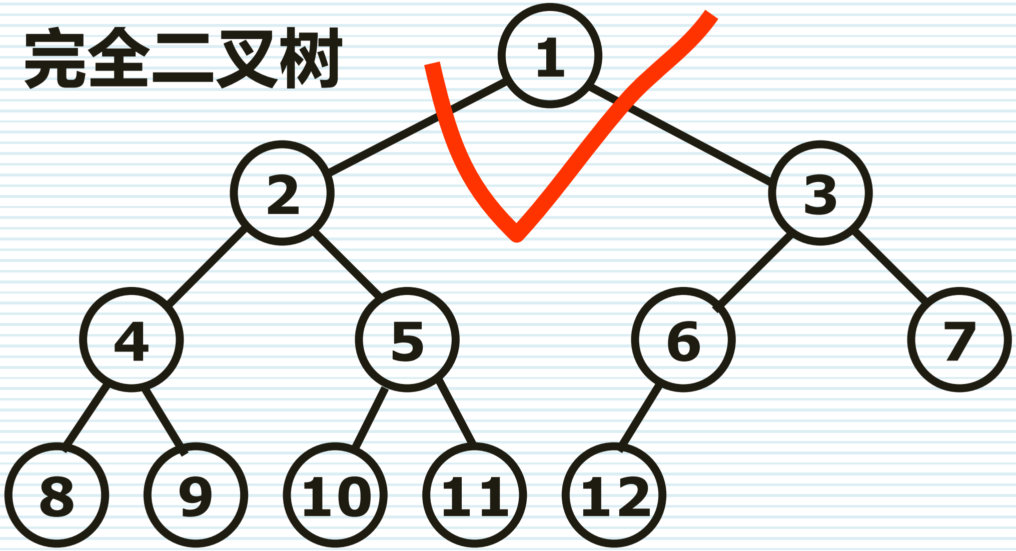
请问：以下哪些树是完全二叉树?

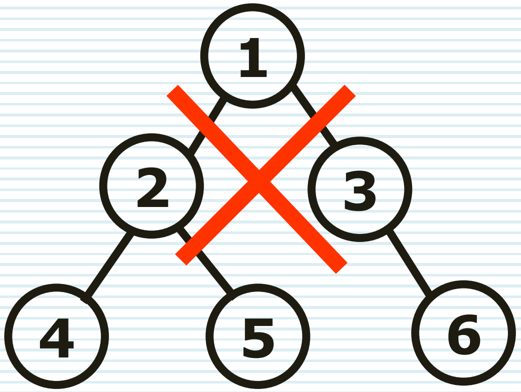
满二叉树





完全二叉树





二叉树的性质

二叉树的性质

性质1：在二叉树的第 i 层上至多有2^{i-1}个结点（i≥1）

- 证明：采用数学归纳法
  - 当i=1时，只有一个根结点，$2^{i-1} = 2^{0} = 1$，命题成立
  - 假设对所有k（1≤k<i）命题成立（第k层至多有2^{k-1}个结点）
  - 则：目标转化为：证明 k = i 时命题成立
    - 由归纳假设：第 i-1 层至多有 2^{i-2} 个结点
    - 因为：二叉树每个结点的度至多为2
    - 所以：第 i 层上最大结点数是第i-1层的2倍
  - 即： $2 \times 2^{i-2} = 2^{i-1}$ 故命题得证 □

二叉树的性质

性质2：深度为k的二叉树至多有2^k-1个结点（k≥1）

- 证明：由性质1可知， 1至 *k* 层各层最多的结点数分别为
  - $2^0, 2^1, 2^2, 2^3, \ldots, 2^{k-1}$ 这是以2为比值的等比数列
  - 前 *k* 项之和即为深度为 k 的二叉树最大结点数

$$\sum_{i=1}^{k}(\text{第}i\text{层的最大结点数}) = \sum_{i=1}^{k} 2^{i-1}$$

$$= 1 + 2 + 2^2 + \ldots + 2^{k-1}$$

$$= \frac{2^k - 1}{2 - 1} = 2^k - 1$$

二叉树的性质

- 性质3：对任何一棵二叉树T有如下关系：$\mathbf{n_0 = n_2 + 1}$
  - 其中：n_0为T中的叶结点数， n_2为T中度为2的结点数
- 证明：设n_1为二叉树T中度为1的结点数

$\because$ 二叉树中的结点总数：$n = n_0 + n_1 + n_2$

$\because$ 二叉树中除根结点外，其余结点的入度均为1

$\therefore$ 设B为二叉树中的分支总数，则：$n = B + 1$

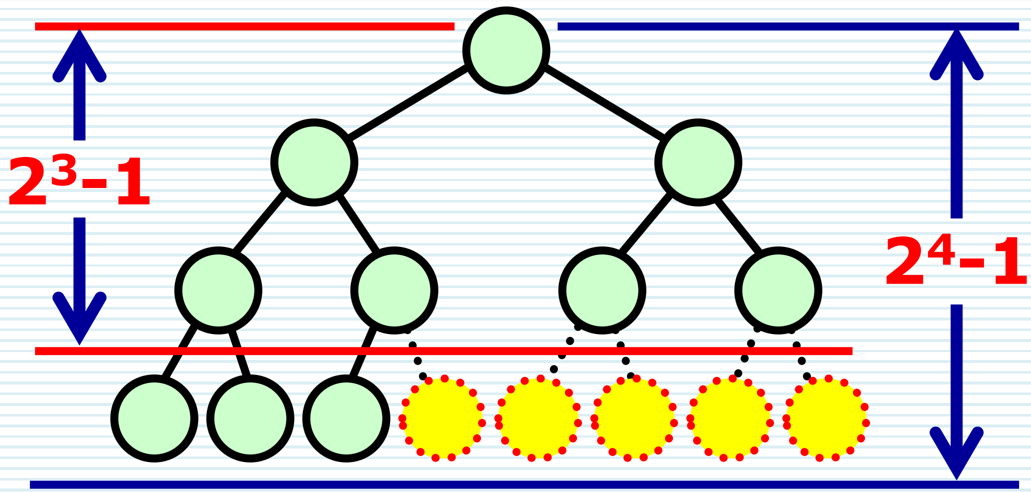
$\because$ 二叉树中的分支由度为1和度为2的结点产生

$\therefore$ $B = n_1 + 2\times n_2$

$\therefore$ $n = B+1 = \cancel{n_1} + \cancel{2}\times n_2+1 = n_0 + \cancel{n_1} + \cancel{n_2}$

$\therefore$ $n_0 = n_2 + 1$ 故命题得证 □

二叉树的性质



性质4：具有n个结点的完全二叉树的深度 $k=\lfloor \log_2 n \rfloor+1$

- 证明：已知：深度为k的二叉树至多有2^k-1个结点

  且已知：深度为k的完全二叉树T中至少包含2^{k-1}个结点

  ∴ 结点总数n满足：$2^{k-1} \le n < 2^k$

  ∴ 取对数：$k-1 \le \log_2 n < k$ 即：$k \le \log_2 n+1 < k+1$

  由于：树的深度k只能取整数值，故命题得证 □

二叉树的性质

性质5：如果对一棵有n个结点的**完全二叉树**的结点按层序编号

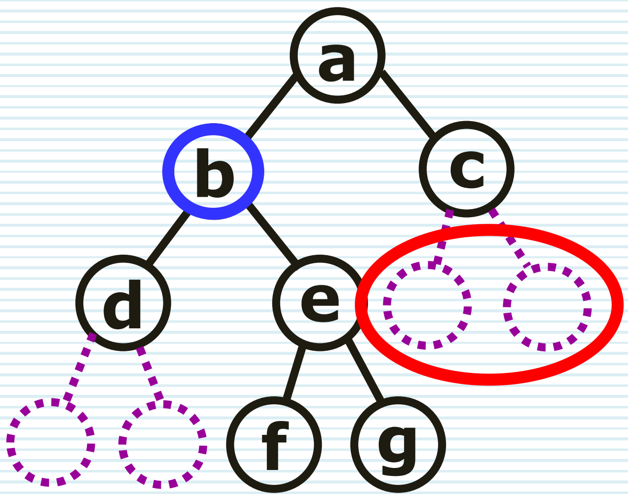
- 则对任一结点 i（$1 \le i \le n$）有：
  - 如果 **i=1**：则结点 **i** 是二叉树的根（无双亲）
  - 如果 **i>1**：则结点i的双亲是 $\lfloor i/2 \rfloor$
  - 如果 **2i>n**：则结点 **i** 无左孩子
    - 如果 **2i≤n**：则结点 i 的左孩子是**2i**
  - 如果 **2i+1>n**：则结点 **i** 无右孩子
    - 如果 **2i+1≤n**：则其右孩子是**2i+1**

二叉树的性质

- 证明:
  - 设结点 i 在第 k 层，其双亲 j 在第 k-1层第 q 个结点
  - 则：$j = 2^{k-2}-1+q$
  - 若：结点 i 是结点 j 的左孩子，则
    - $i = 2^{k-1}-1+2(q-1)+1 = 2^{k-1}-2+2q$，可得 $i = 2j$
  - 若结点 i 是结点 j 的右孩子，则
    - $i = 2^{k-1}-1+2(q-1)+2 = 2^{k-1}-1+2q$，可得 $i = 2j+1$
  - 故：$j = \lfloor i/2 \rfloor$
    - 若结点 j 有左孩子：则 $2j \le n$，其左孩子为 $2j$
    - 若结点 j 有右孩子：则 $2j+1 \le n$，其右孩子为 $2j+1$

二叉树的存储结构

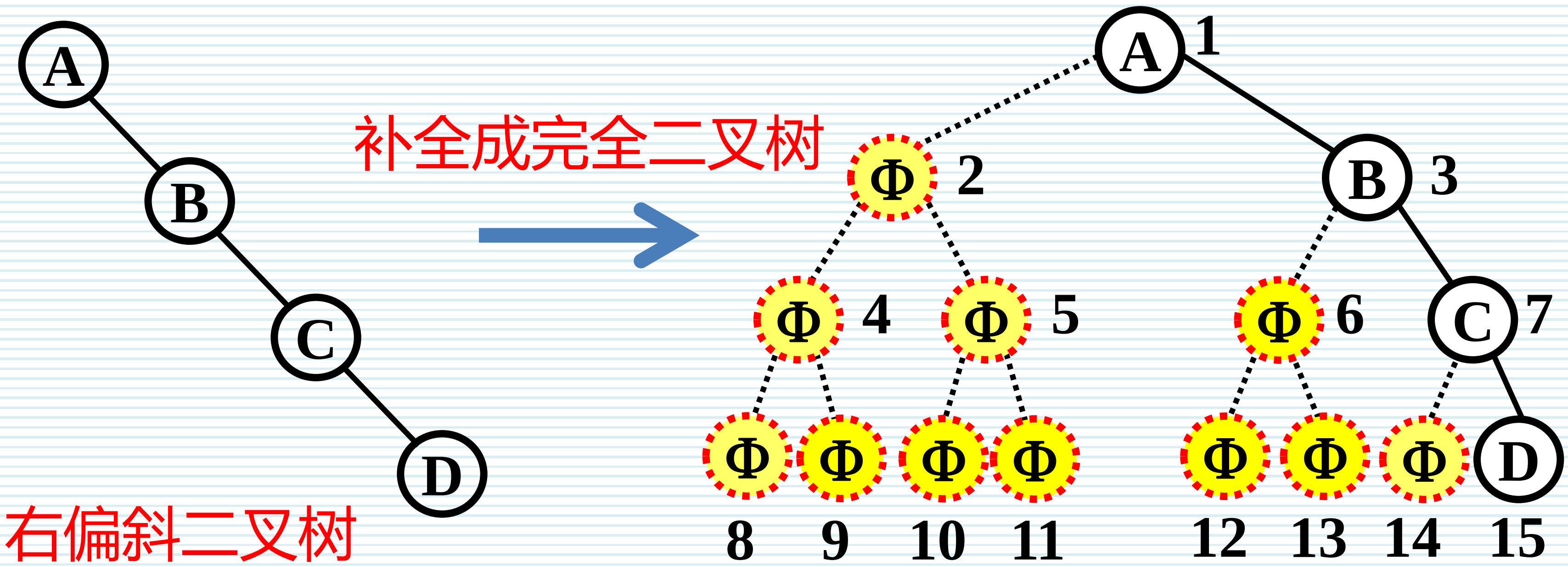
二叉树的顺序存储结构



0	1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	0	0	0	0	f	g

- 存储方式
  - 用一组地址连续的存储单元存放二叉树T上的结点元素
    - 按完全二叉树的结构自上而下、自左至右对结点编号
    - 存储时将T中结点存储在一维数组的对应位置
  - 特点：结点间关系蕴含在其存储位置中
    - 浪费空间（适于存满二叉树和完全二叉树）

二叉树的顺序存储结构



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0	B	0	0	0	C	0	0	0	0	0	0	0	D

- 深度为100的右偏斜二叉树
  - 需要2^{100} - 101个额外空间!
  - 顺序存储结构适用于存储满二叉树和完全二叉树

二叉树的链式存储结构

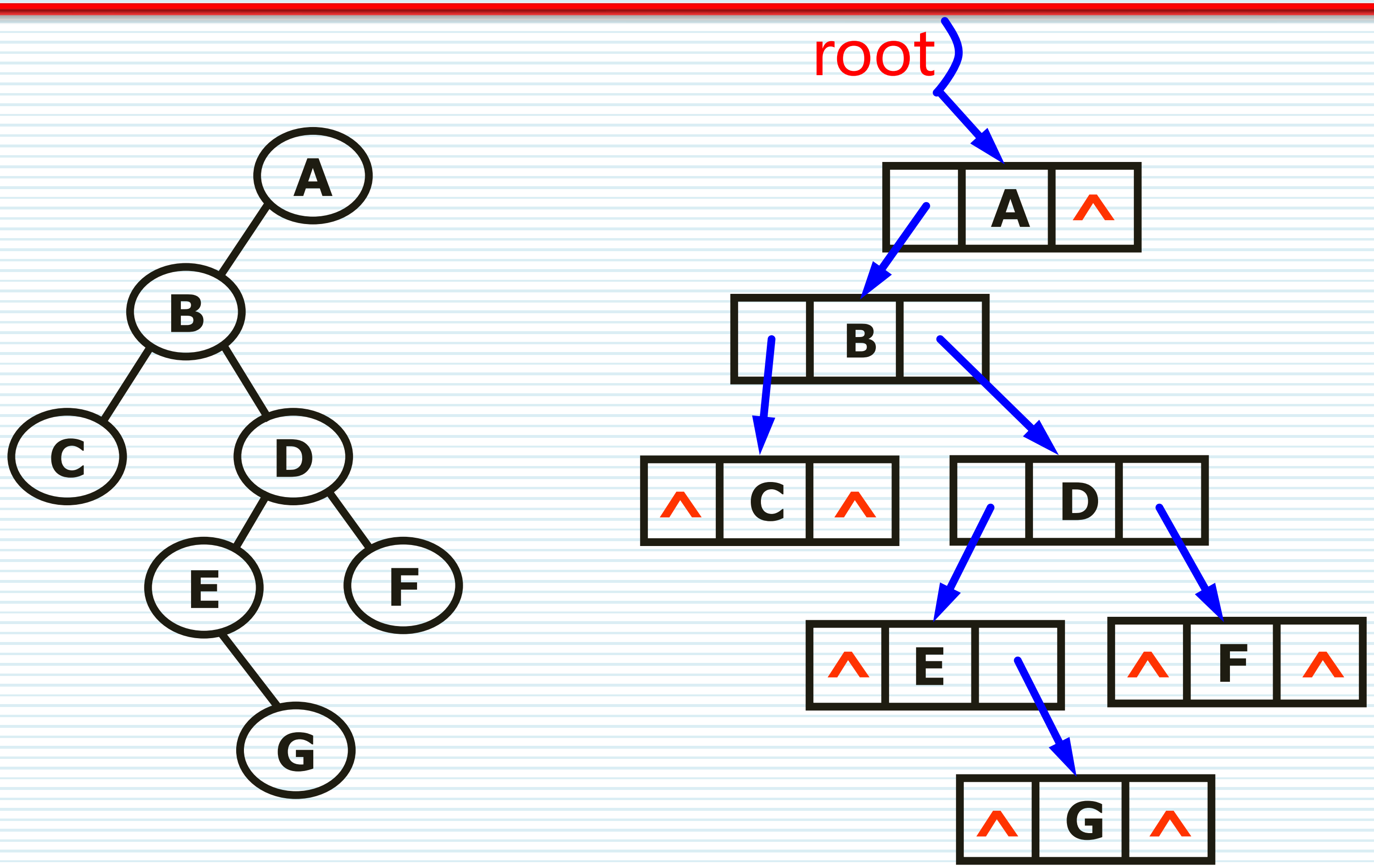
结点结构：

lchild	data	rchild

```
typedef struct node {
    ElemType    data;                    // 数据域
    struct  node  *lchild,  *rchild;  //指针域
} TNode, *PBT;
```

- 二叉树的链式存储结构：二叉链表
  - 基本思想：令二叉树的每个结点对应一个链表结点
    - 除存放二叉树结点的数据信息外
    - 还包含指向左右孩子结点的指针

二叉链表



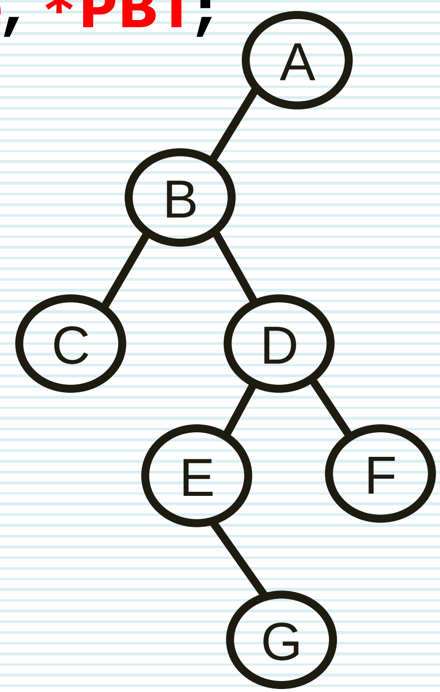
- 二叉链表示例 **空指针域数目** = $2n_0+n_1 = n_0+n_1+n_2+1 = n+1$
  - 问题：如图所示的二叉链表中有多少指针为空?
  - 在n个结点的二叉链表中，有n+1个空指针域

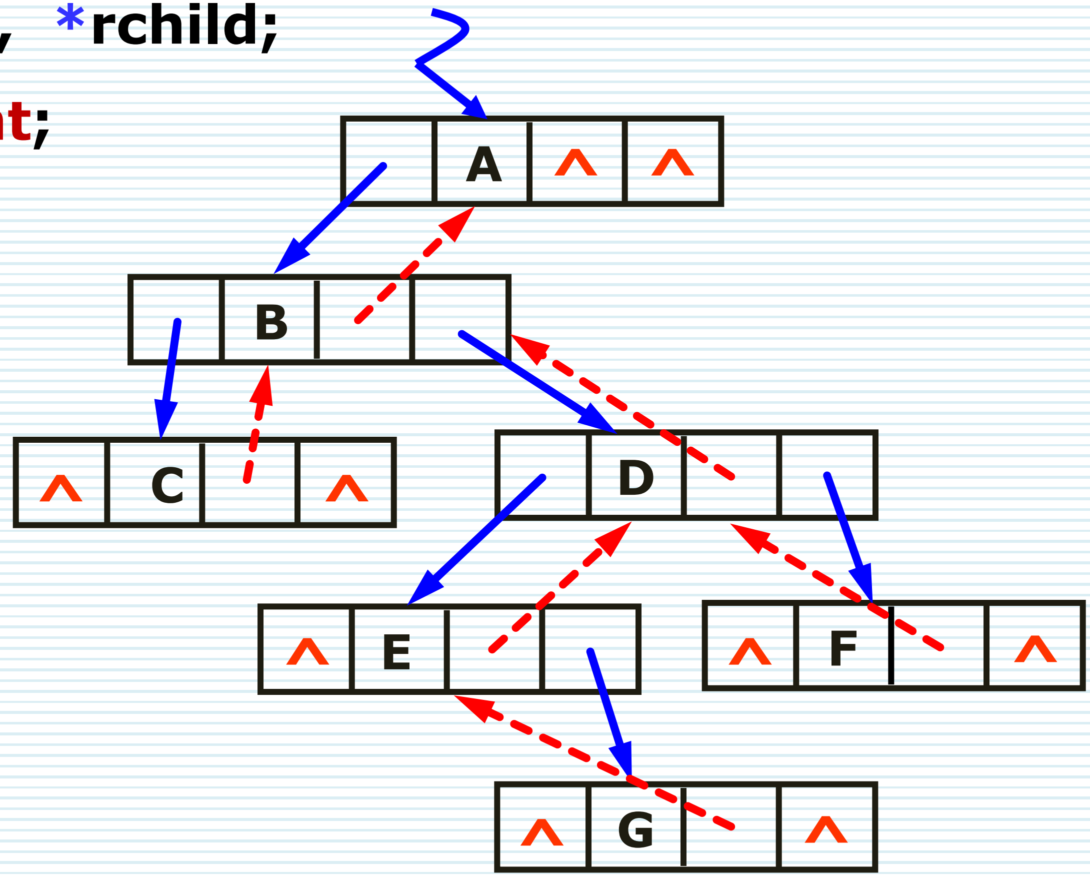
三叉链表

- 三叉链表：在二叉链表的基础上增加一个指向双亲的指针域

```
typedef  struct  node {
     Elemtype   data;
     struct  node  *lchild,  *rchild;
     struct  node  *parent;
} TNode, *PBT;
```

lchild	data	parent	rchild



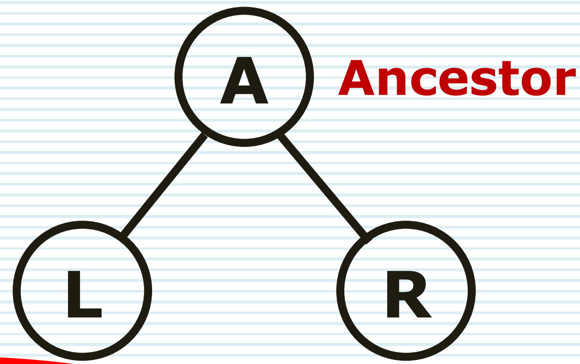


二叉树的遍历

树的遍历

- 按一定规律走遍树的各顶点，且使每一顶点仅被访问一次
- 即：采用一定的方法得到树中所有结点的一个线性排列

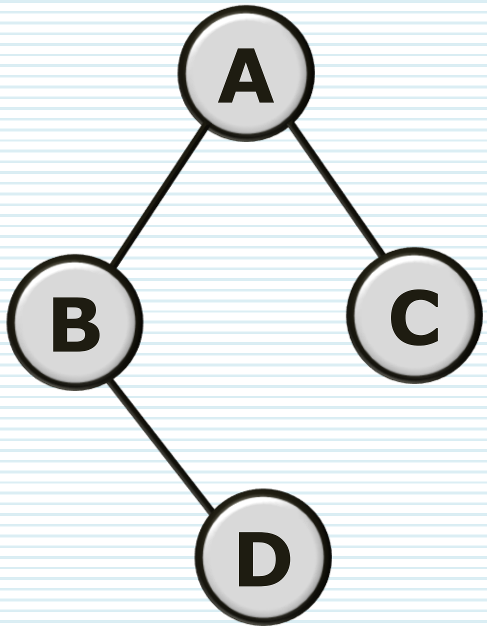
二叉树的四种遍历算法

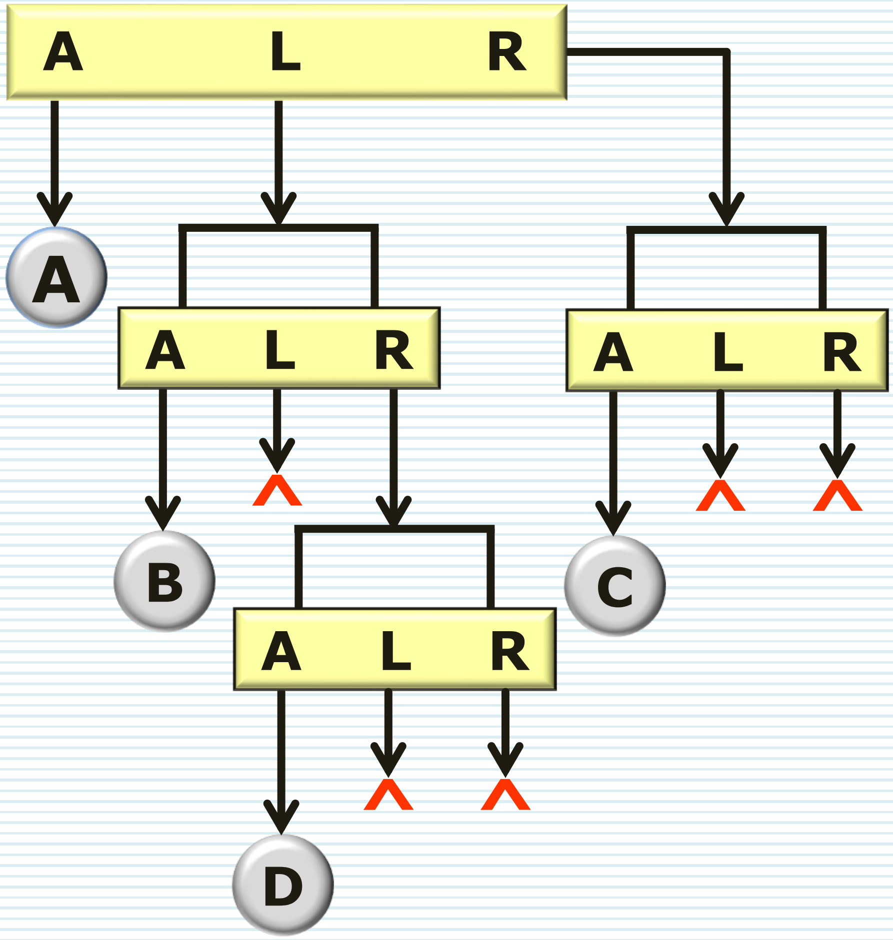


LAR、LRA、ALR、~~RAL、RLA、ARL~~

- 前序遍历：先访问根结点，然后分别前序遍历左子树、右子树
- 中序遍历：中序遍历左子树，访问根结点，中序遍历右子树
- 后序遍历：先后序遍历左、右子树，然后访问根结点
- 按层次遍历：从上到下、从左到右访问各结点

二叉树的前序遍历（ALR）



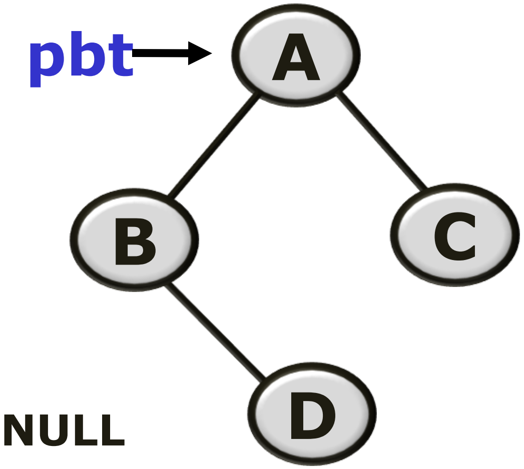


前序遍历结果序列：A B D C

前序遍历的递归算法

```
void preorder(PBT pbt){
    if(pbt != NULL){        // 如果pbt为空指针则结束递归
        printf("%c\t", pbt->data);    // 访问根结点
        preorder(pbt->lchild);    // 前序遍历左子树
        preorder(pbt->rchild);    // 前序遍历右子树
    }
    return;
}
```

前序遍历递归算法详解




```
void preorder(PBT pbt){
    if(pbt != NULL){
        printf("%c\t", pbt->data);
        preorder(pbt->lchild);
        preorder(pbt->rchild);
    }
    return;
}
```

主程序

Pre(bt)

pbt→Ⓐ
print "A"
Pre(bt -> L)
Pre(bt -> R)
return

pbt→Ⓑ
print "B"
Pre(pbt->L)
Pre(pbt->R)
return

pbt=NULL
return

pbt→Ⓓ
print "D"
Pre(pbt-> L)
Pre(pbt-> R)
return

pbt=NULL
return

pbt=NULL
return

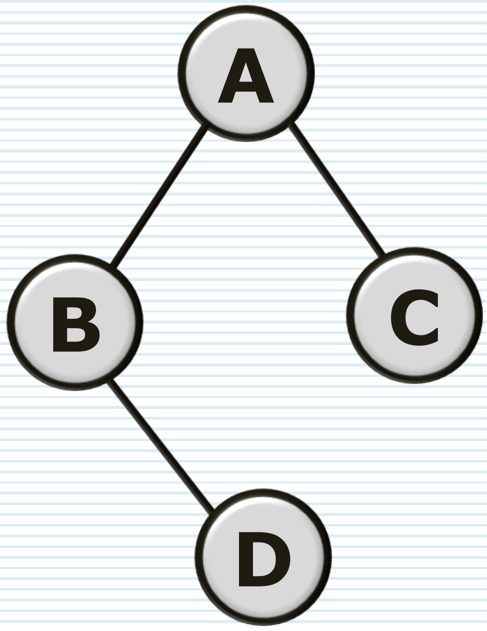
bt →Ⓒ
print "C"
Pre(pbt->L)
Pre(pbt->R)
return

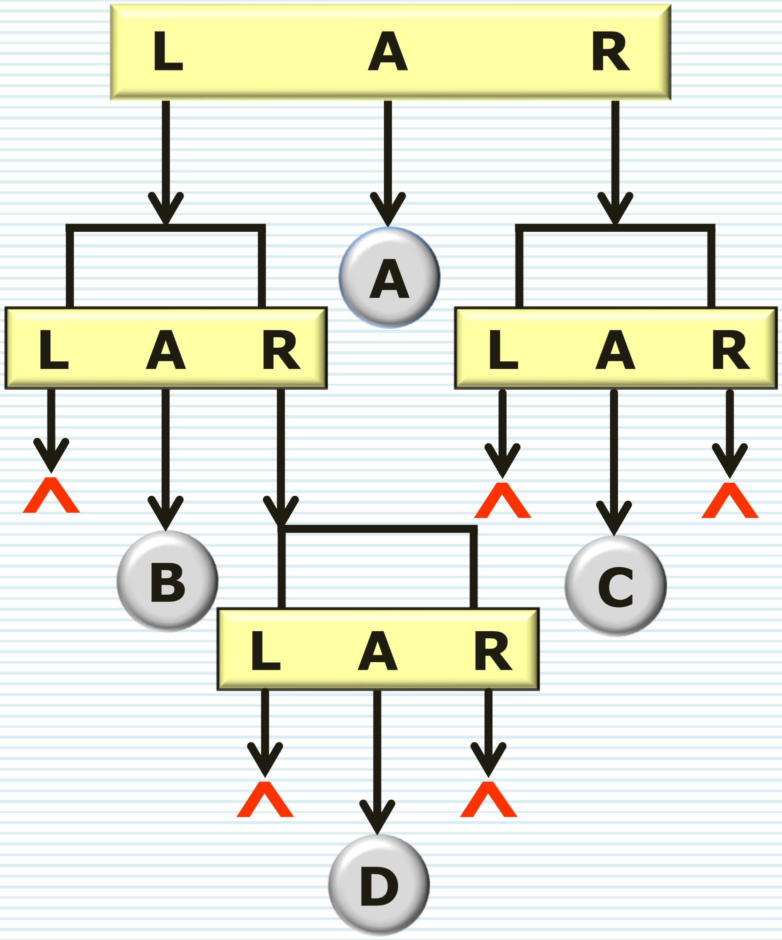
pbt=NULL
return

pbt=NULL
return

前序序列： A B D C

二叉树的中序遍历（LAR）



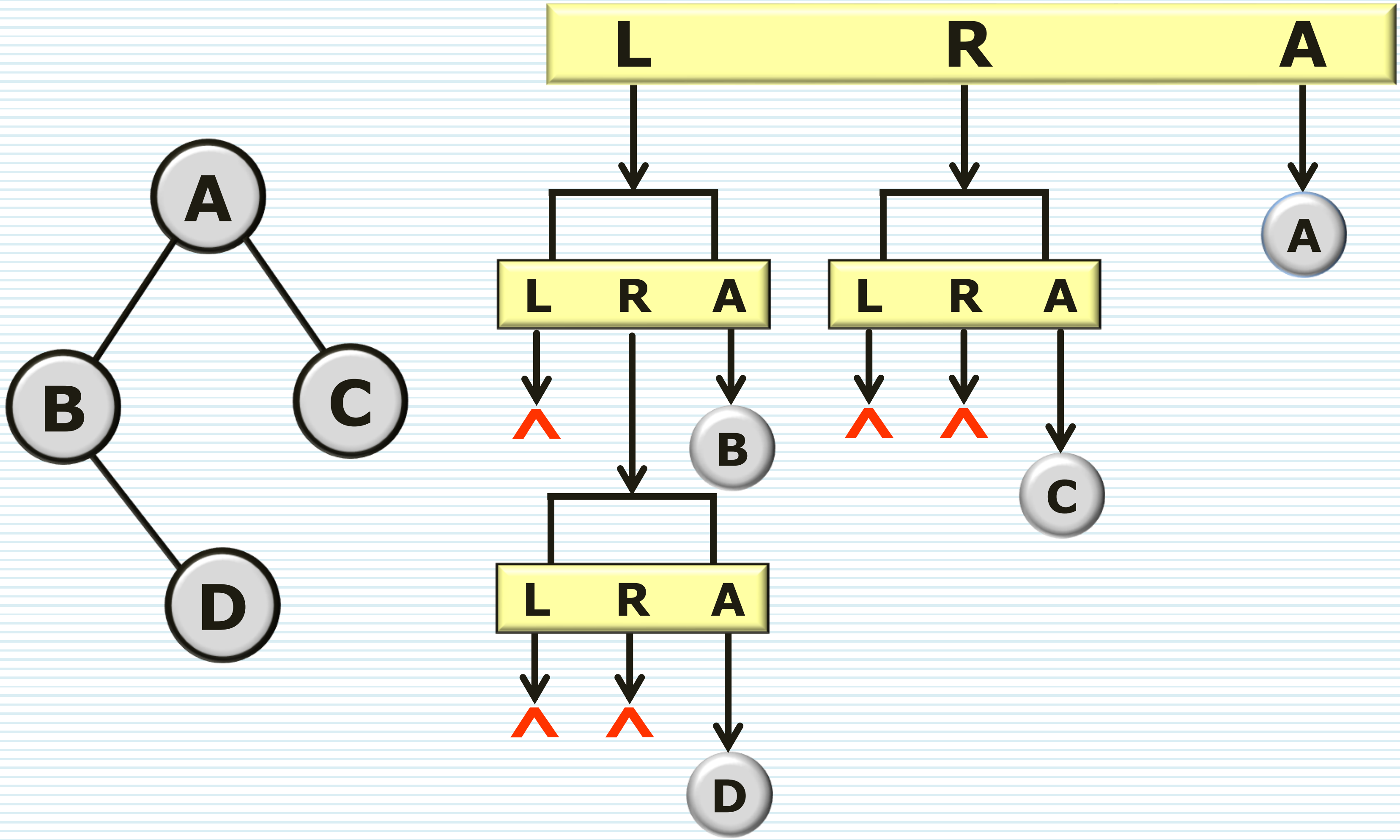


中序遍历结果序列：B D A C

中序遍历的递归算法

```
void inorder(PBT pbt){
    if(pbt != NULL){        // 如果pbt为空指针则结束递归
        inorder(pbt->lchild);    // 中序遍历左子树
        printf("%c\t", pbt->data);   // 访问根结点
        inorder(pbt->rchild);   // 中序遍历右子树
    }
    return;
}
```

二叉树的后序遍历（LRA）



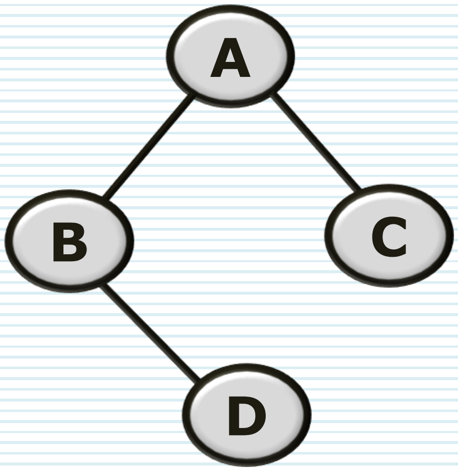
中序遍历结果序列：D B C A

后序遍历的递归算法

```
void postorder(PBT pbt){
    if(pbt != NULL){        // 如果pbt为空指针则结束递归
        postorder(pbt->lchild);    // 后序遍历左子树
        postorder(pbt->rchild);   // 后序遍历右子树
        printf("%c\t", pbt->data);   // 访问根结点
    }
    return;
}
```

二叉树的层次遍历算法

- 层次遍历算法
  - 从根节点开始**从上到下**逐层遍历
  - 同一层中**从左到右**依次访问二叉树结点
  - 思考：采用哪种抽象数据结构？　**队列！**



- 算法描述：

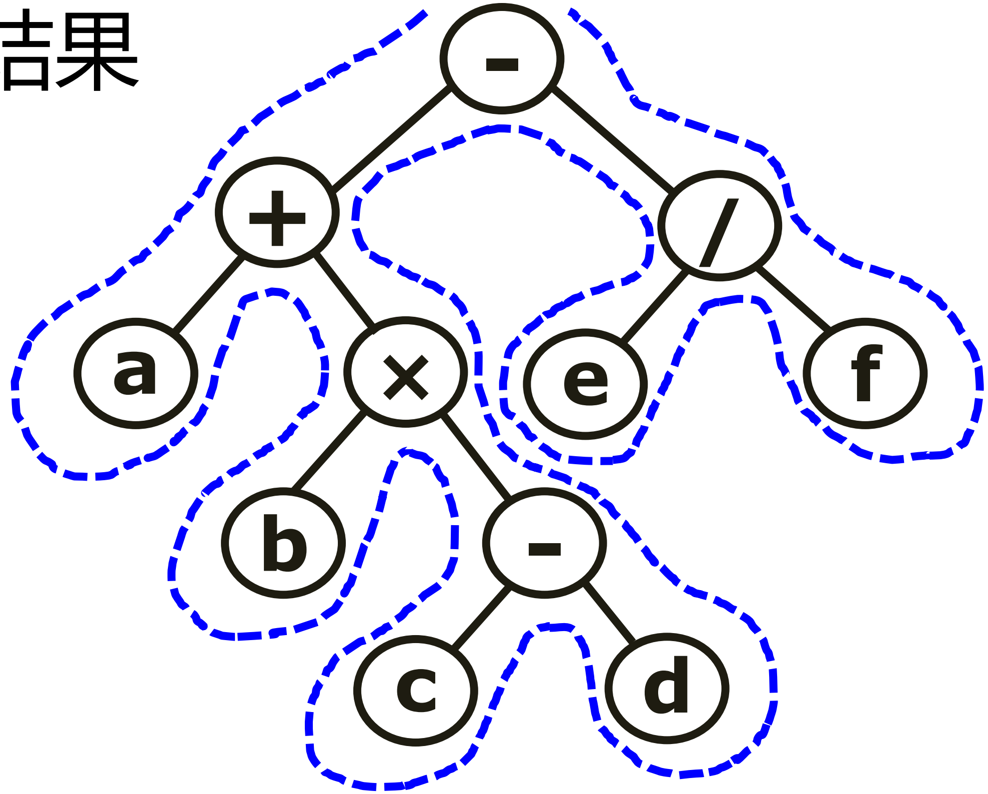
层次遍历序列：A　B　C　D

  - 首先将根结点的指针入队
  - 循环取队首元素，执行如下操作，直至队空为止
    - 访问该元素（结点）的数据部分
    - 若该结点有左孩子，则将其入队
    - 若该节点有右孩子，则将其入队

代码示例：二叉树的层次遍历算法

```
void level_order(PBT pbt) {              // 层序遍历
    PBT p;  PQue que = init_que(MAXSIZE);
    if(pbt){                                  // 二叉树非空
        enque(que, pbt);                      // 根节点（指针）入队
        while (! is_empty(que)){
            p = deque(que);                   // 队首元素出队
            printf("%c", p->data);
            if (p->lchild != NULL) { enque(que, p->lchild); }
            if (p->rchild != NULL) { enque(que, p->rchild); }
        }
    }
    destroy_que(&que);  // 释放为队列分配的内存
}
```

请给出遍历结果



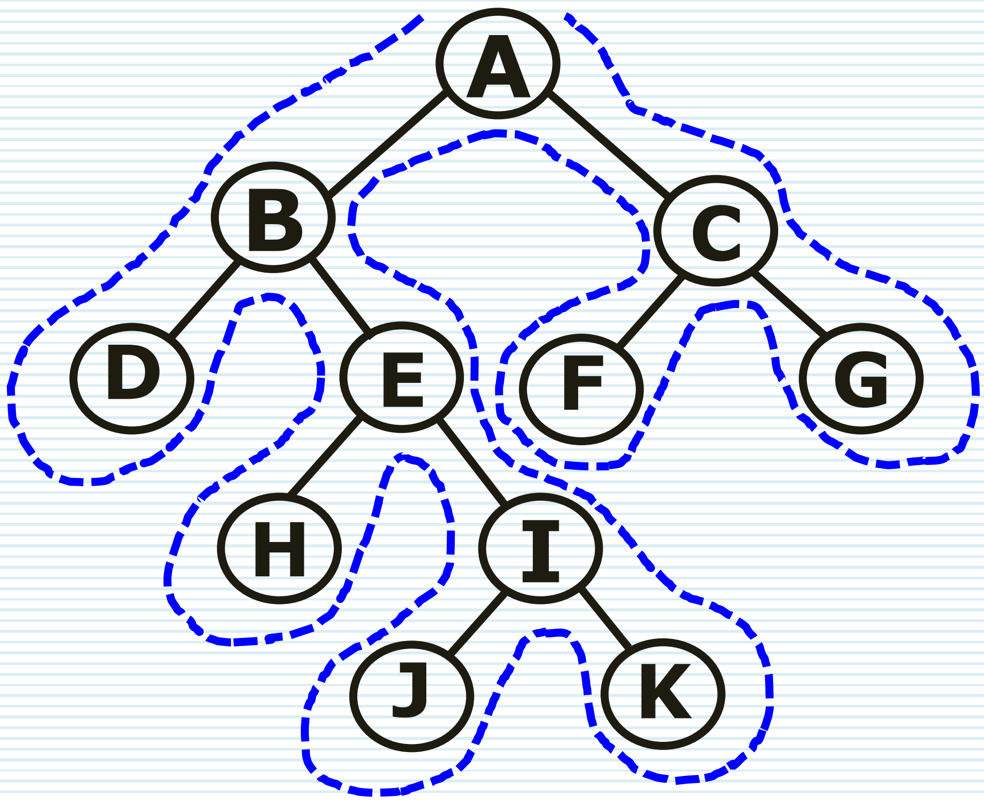
前序遍历： - + a × b - c d / e f

中序遍历： a + b × c - d - e / f

后序遍历： a b c d - × + e f / -

层次遍历： - + / a × e f b - c d

二叉树的非递归遍历



- 二叉树的前序、中序和后序遍历都是沿着图中路线进行
  - 前序遍历：遇到结点就访问
  - 中序遍历：左子树返回时访问
  - 后序遍历：右子树返回时访问

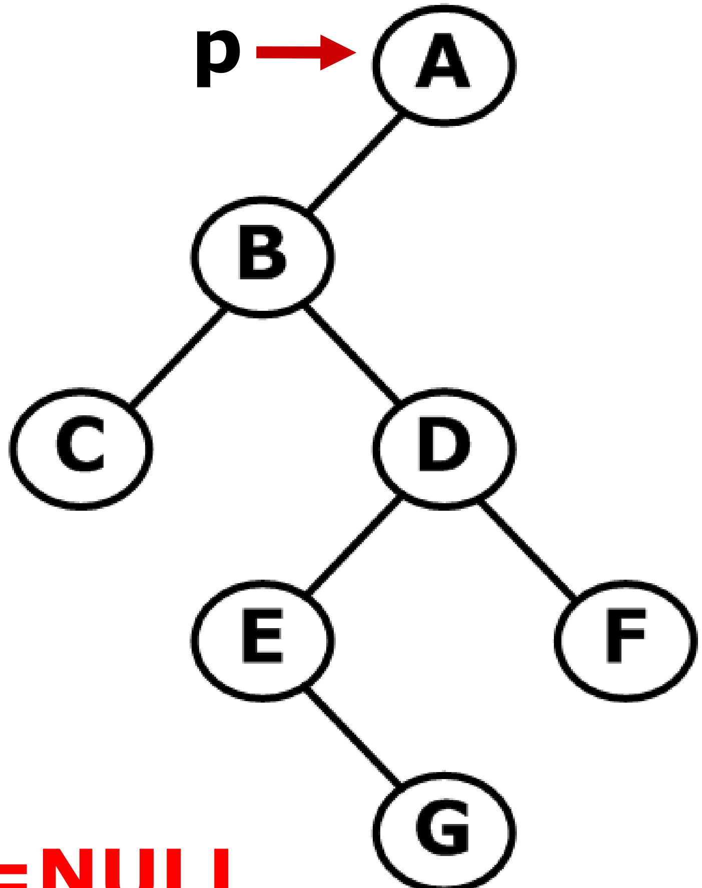
本质上是深度优先遍历

可用**栈**实现二叉树的遍历

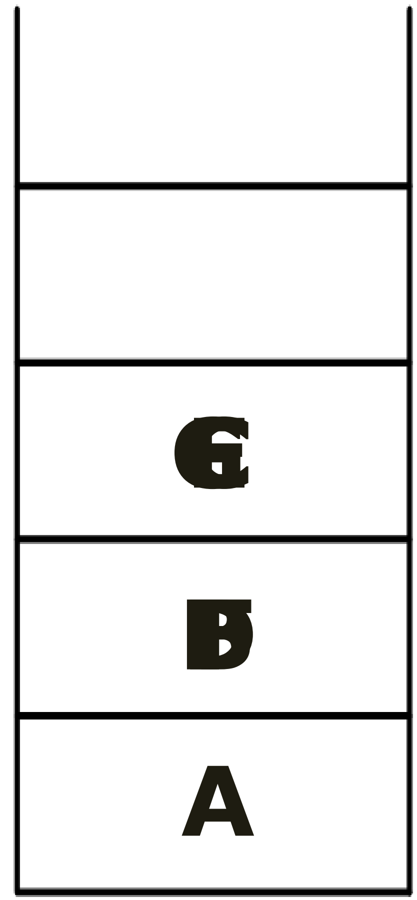
前序遍历的非递归算法

```
void preorder(PBT pbt){
   PStack ps = init_stack(MAXSIZE);
   while ((pbt != NULL) || ! is_empty(ps)){
       if (pbt !=NULL){
             printf ("%c", pbt->data);  // 访问当前结点
             push_stack(pb, pbt);        // 将pbt压栈
             pbt = pbt->lchild;          // 将pbt指向其左子树
       }
       else{
             pbt = pop_stack(ps);     // 栈顶元素退栈
             pbt = pbt->rchild;          // 将pbt指向其右子树
       }
   } destroy_stack(&ps);  // 释放为栈分配的内存
}
```

中序遍历的非递归算法



p==NULL



辅助栈

中序序列： **C B E G D F A**

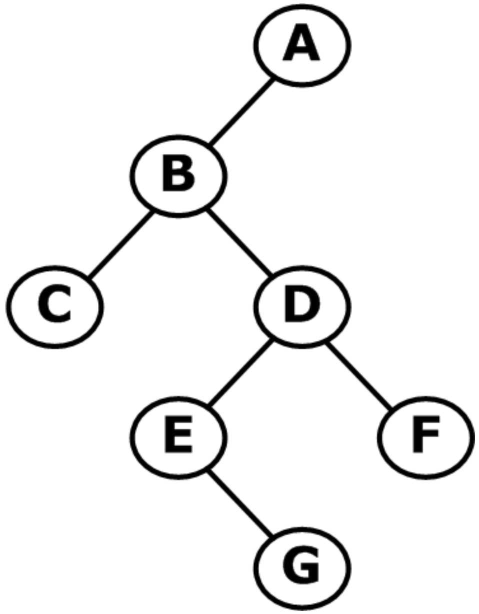
中序遍历的非递归算法

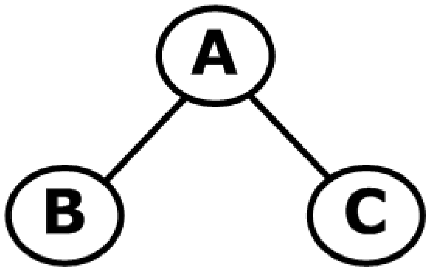
```
void inorder(PBT pbt){
    PStack ps = init_stack(MAXSIZE);
    while ((pbt != NULL) || ! is_empty(ps)){
        if (pbt !=NULL){
            push_stack(ps, pbt);        // 将pbt压栈
            pbt = pbt->lchild;          // 将pbt指向其左子树
        }
        else{
            pbt = pop_stack(ps);             // 栈顶元素退栈
            printf (“%c”, pbt->data);   // 访问当前结点
            pbt = pbt->rchild;           // 将pbt指向其右子树
        }
    } destroy_stack(&ps);  // 释放为栈分配的内存
}
```

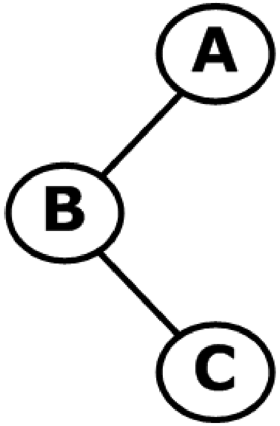
后序遍历的非递归算法

```
void postorder(PBT pbt){
   PStack ps = init_stack(MAXSIZE);
   while ( pbt || ! is_empty(ps)){
       if (pbt ){
           if(!pbt->visited) push_stack(ps, pbt);
           pbt = pbt->lchild;
       }
       else{
           pbt = pop_stack(ps);
           if(pbt->visited) printf ("%c", pbt->data);
           else{ pbt->visited = 1;
                 push_stack(pb, pbt); }
           pbt = pbt->rchild;
       }
   } destroy_stack(&ps);  // 释放为栈分配的内存
}
```

二叉树遍历算法的应用

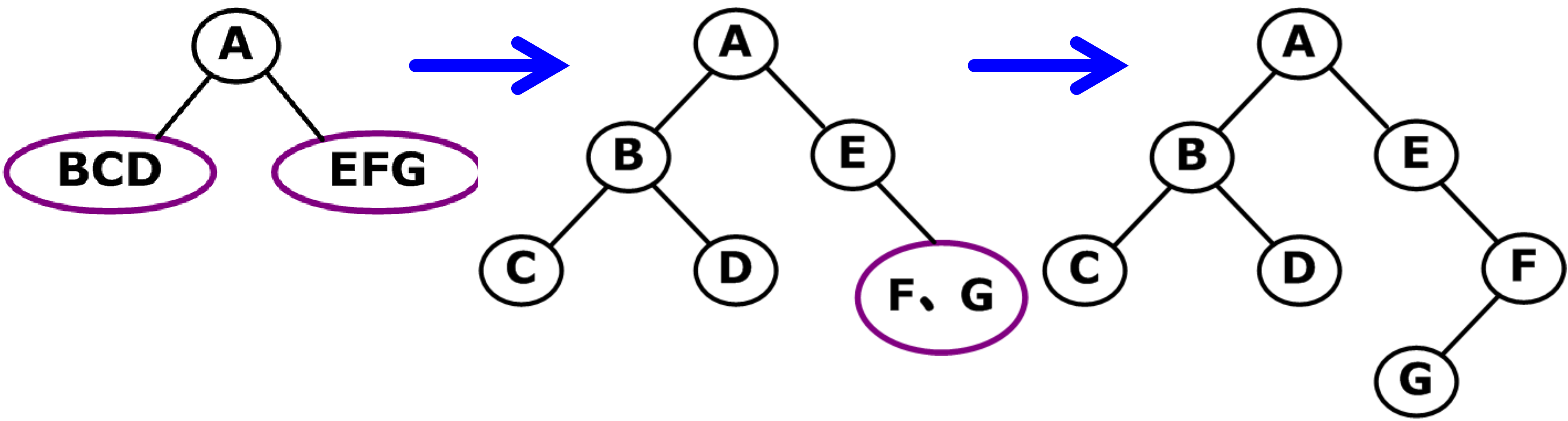






- 按前序遍历的结果序列建立二叉树的二叉链表
  - 已知前序遍历序列：A B C D E G F
  - 问题：能否由此唯一确定一棵二叉树?
    - 不能！反例如图所示（两者的前序遍历序列均为ABC）

二叉树遍历算法的应用

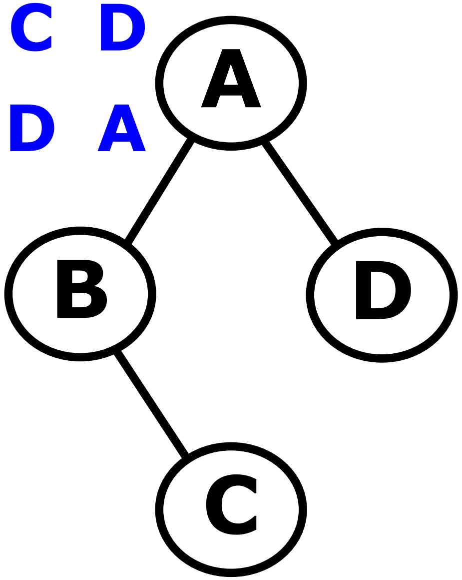


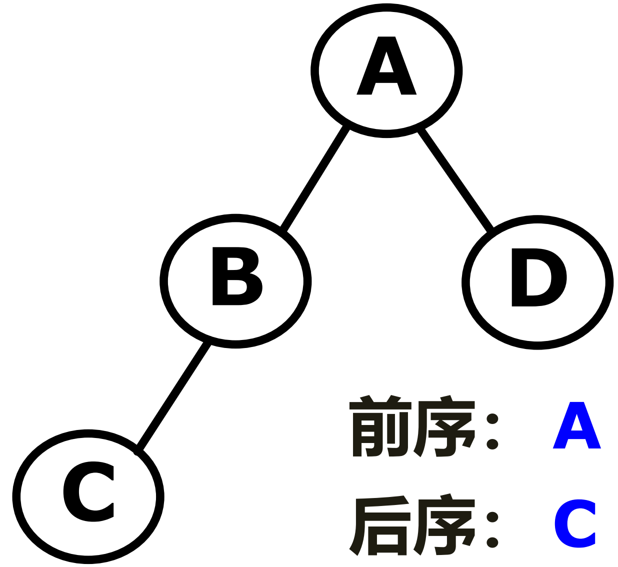
- 问题：已知**前序**和**中序**序列，能否唯一确定一棵二叉树?
  - 可以！根据**前序**可得**根结点**；根据**中序**可区分**左右子树**
    - 已知前序序列为：A B C D E F G
    - 已知中序序列为：C B D A E G F

二叉树遍历算法的应用

前序： A B C D

后序： C B D A





前序： A B C D

后序： C B D A

- 思考：已知**前序**和**后序**序列，能否唯一确定一棵二叉树?
  - 不可以！ …… 试举出反例
- 思考：已知**中序**和**后序**序列，能否唯一确定一棵二叉树?
  - 可以！根据**后序**可得**根结点**；根据**中序**可区分**左右子树**

统计二叉树中叶结点个数

```
int count_leavs(PBT pbt){
    if(! pbt){                  // pbt == NULL
        return 0;               // 空树的叶节点数为零
    }
    if((!pbt->lchild) && (!pbt->rchild)){
        return 1;               // 左右子树均为空，则为叶节点
    }
    // 存在左子树或右子树时，递归调用结点统计函数
    return  count_leavs(pbt->lchild) +
            count_leavs (pbt->rchild);
}
```

求二叉树深度

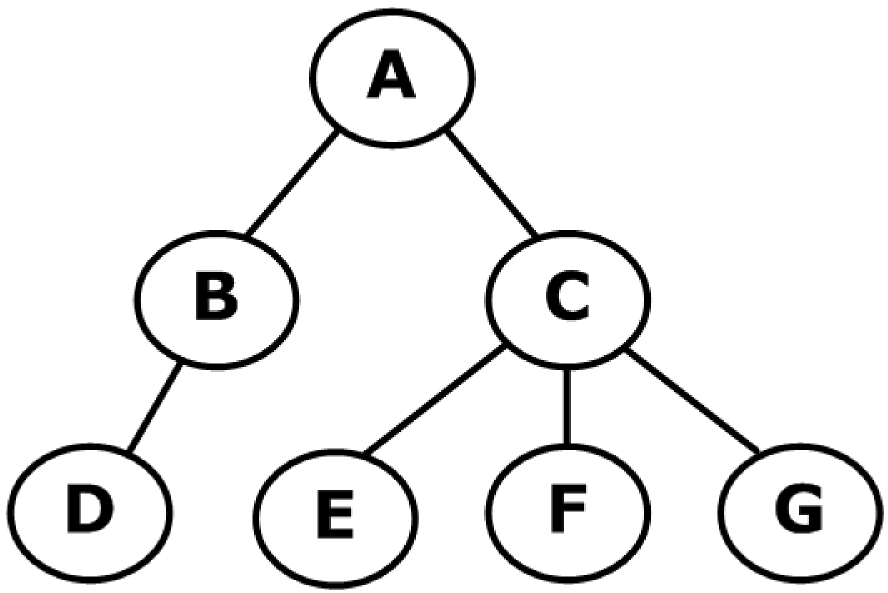
```
int get_depth(PBT  pbt){
    int dL = 0,  dR = 0
    if( pbt == NULL )   return 0;   // 空树深度为0
    if((!pbt->lchild) && (!pbt->rchild)){
        return 1;                   // 叶结点深度为1
    }
    dL = get_depth (pbt->lchild);
    dR = get_depth (pbt->rchild);
    // 二叉树的深度为根结点左、右子树的深度值较大者加一
    return 1 + ((dL > dR) ? dL : dR);
}
```

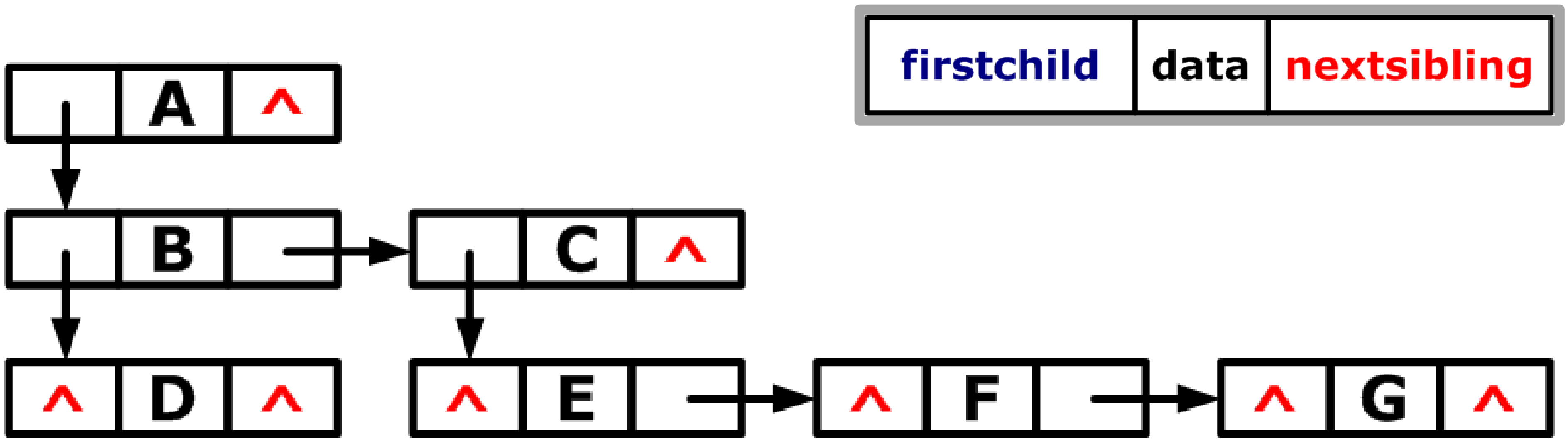
3. 树和森林

- 树的存储结构
- 森林、树、二叉树的相互转化
- 树和森林的遍历

树的存储结构

树的存储结构示例：孩子兄弟表示法





树的存储结构：孩子兄弟链表

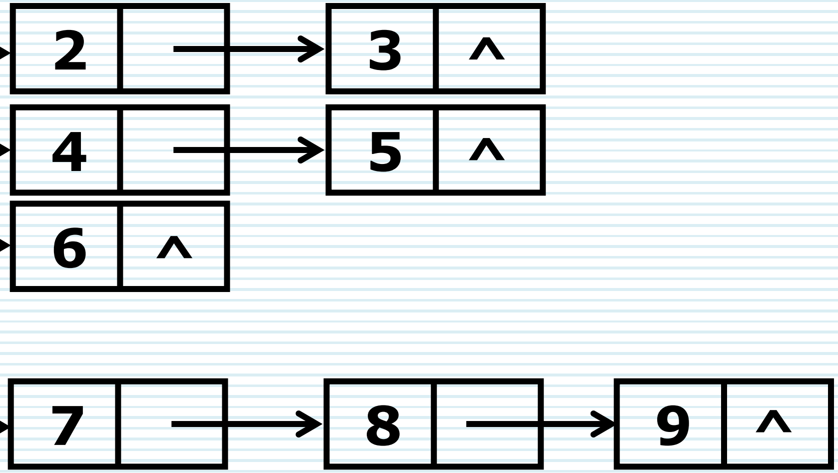
```
typedef struct node{
    ElemType data;
    struct node * firstchild, * nextsibling;
}TNode, *PTree;
```

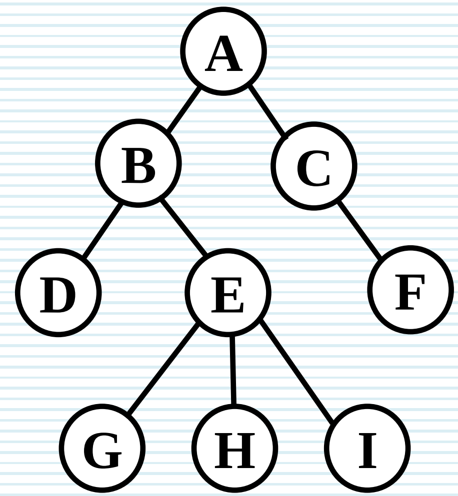
firstchild	data	nextsibling

- 孩子兄弟链表表示法
  - 以二叉链表作为树的存储结构
  - 与二叉树的二叉链表的区别在于
    - 取消了左右孩子指针的定义
    - **firstchild**指针：指向该节点第一个孩子节点
    - **nextsibling**指针：指向下一个兄弟节点

树的双亲孩子表示法

	data	parent	firstchild
1	A	0	
2	B	1	
3	C	1	
4	D	2	^
5	E	2	
6	F	3	^
7	G	5	^
8	H	5	^
9	I	5	^

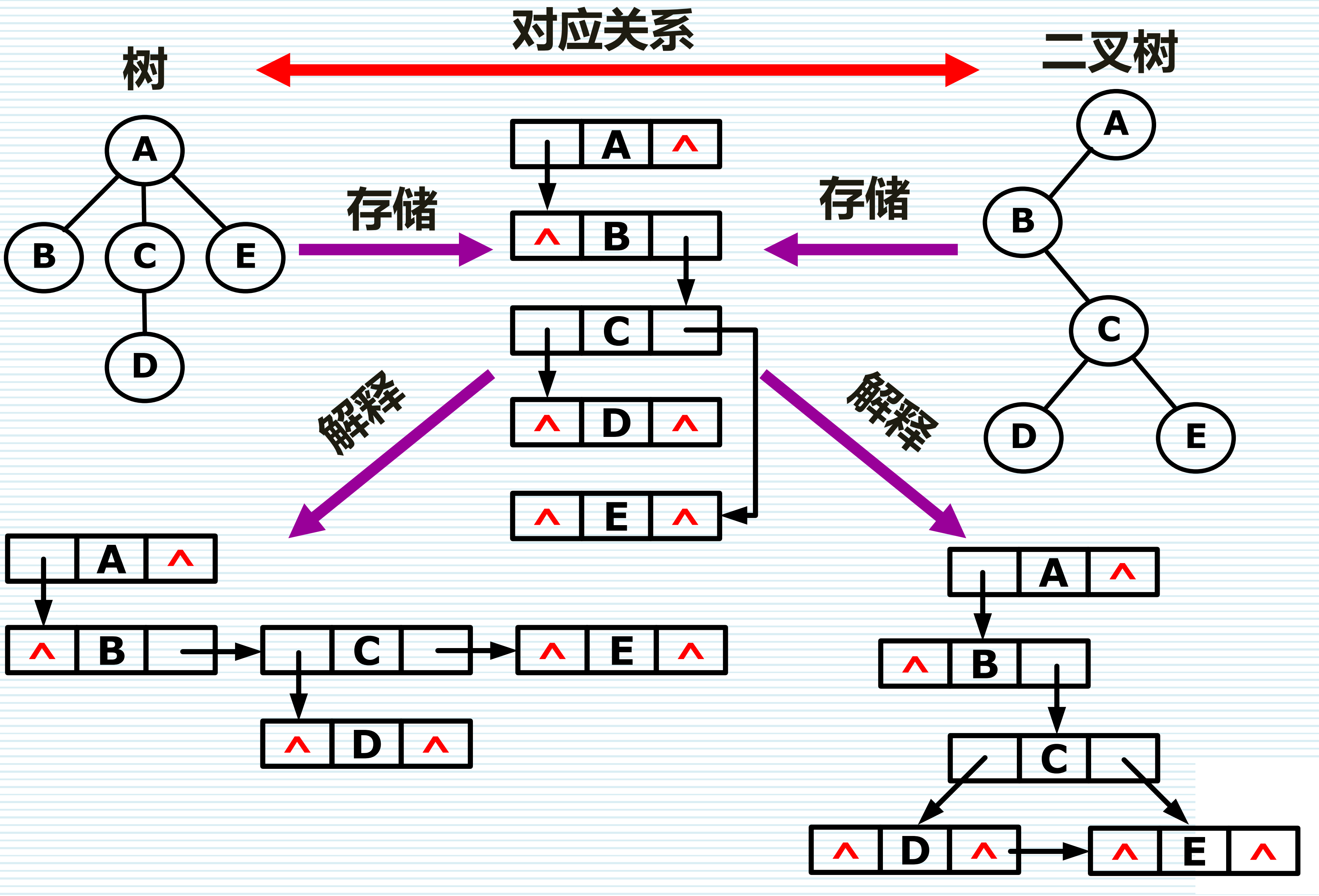




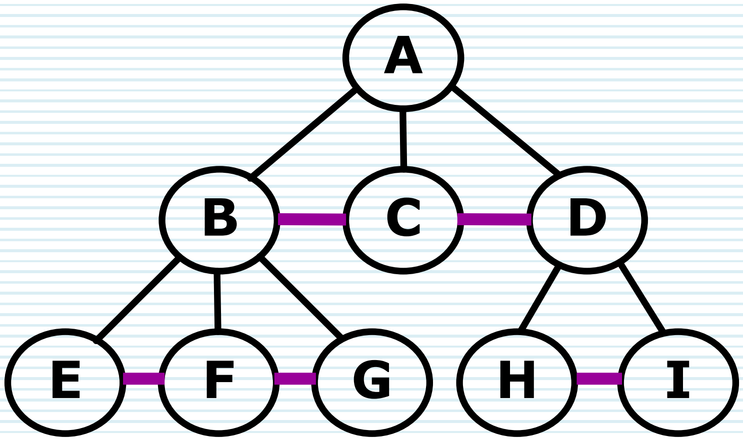
树与二叉树的相互转换

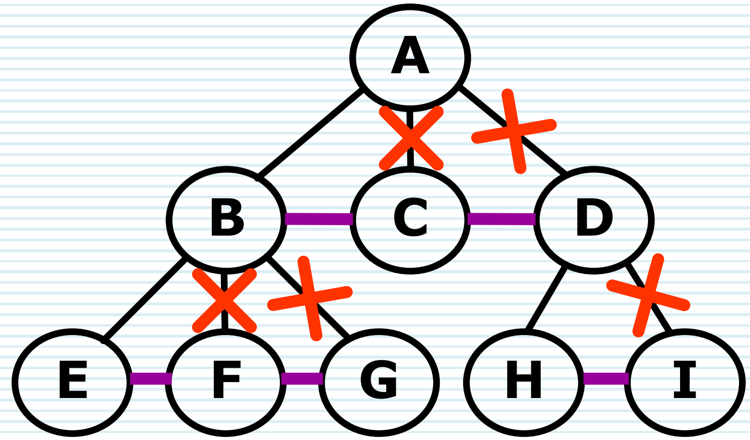
- 二叉树和树均可采用二叉链表作为存储结构
  - 可利用二叉链表导出：树与二叉树的对应关系
  - 即：对一棵采用孩子兄弟链表表示法存储的树
    - 可找到一棵二叉树的二叉链表与之相对应
    - 二者的物理存储方式一致，但解释不同

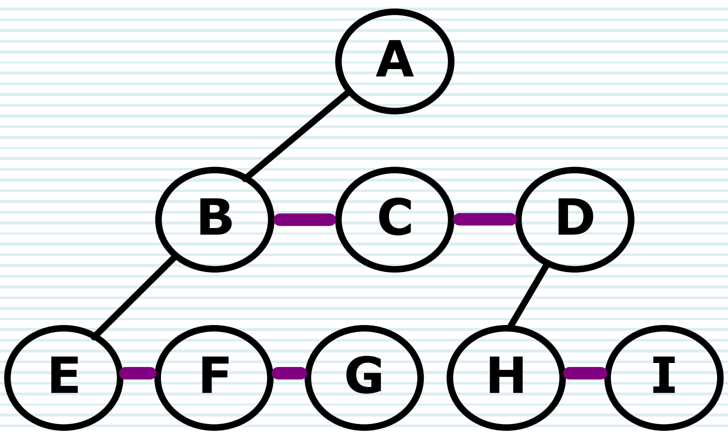
树与二叉树的相互转换

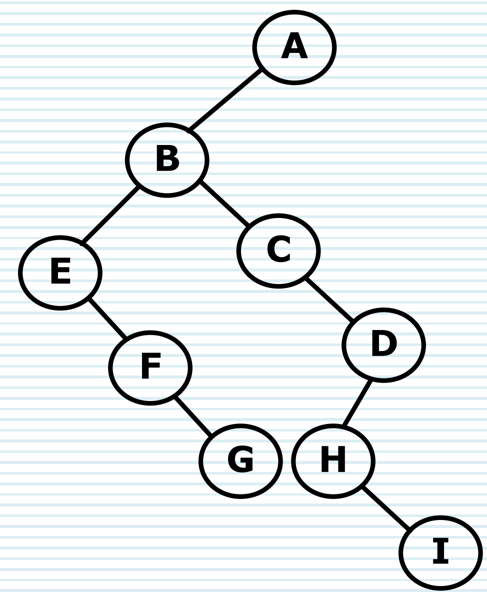


将树转换成二叉树





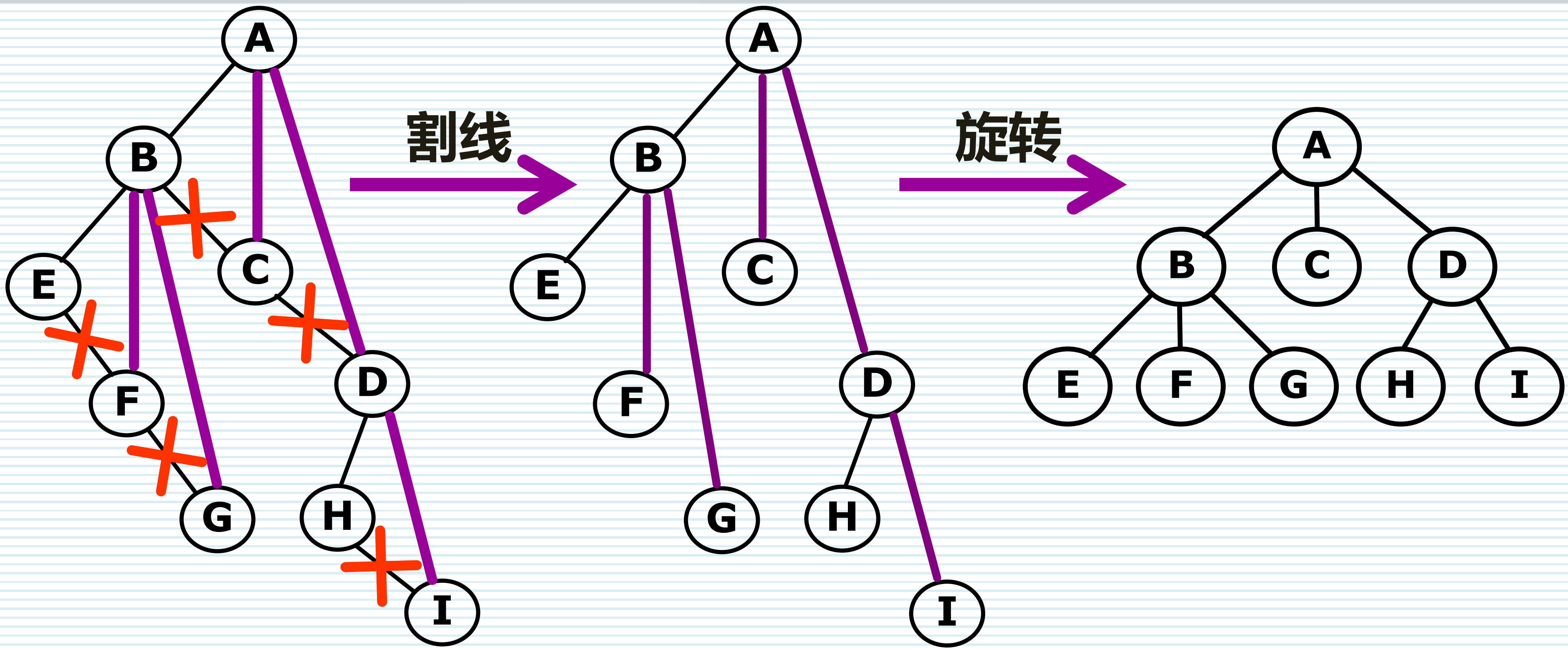




特点：树转换成的二叉树其右子树一定为空

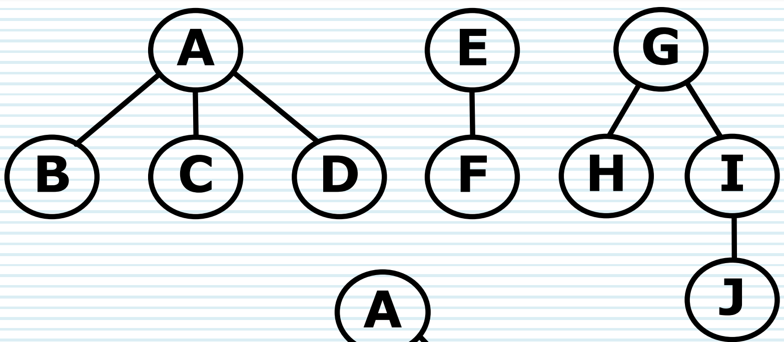
1. 加线：在所有兄弟节点之间加一条连线
2. 割线：对每个结点，去掉**除左孩子外**它和其余孩子间的连线
3. 旋转：以树的根结点为轴心，将整棵树顺时针转45°

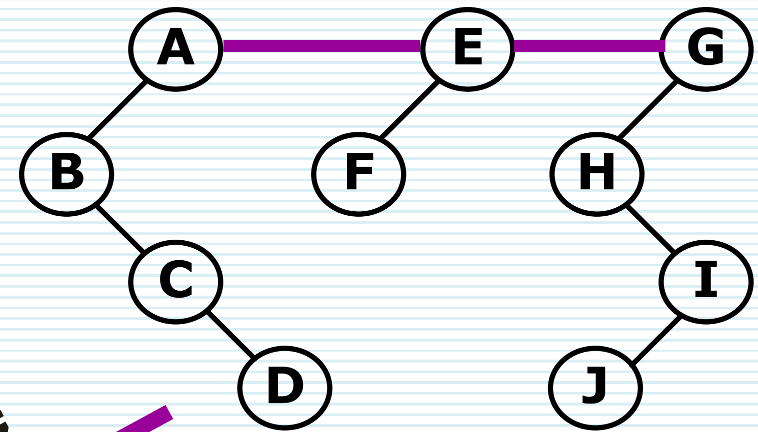
将二叉树转换成树

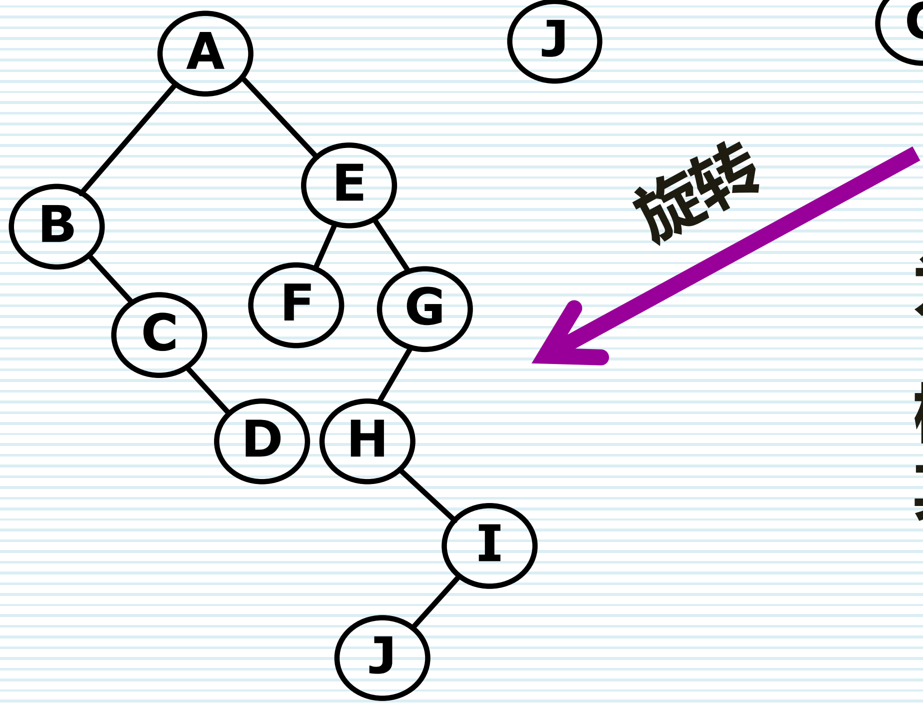


1. 加线：若结点c是父结点p的左孩子，则将p的右孩子，右孩子的右孩子(……所有右孩子)，都与父结点p进行连线
2. 割线：去掉原二叉树中父结点与右孩子之间的连线
3. 调整：将结点按层次排列，形成树形结构

森林转换成二叉树





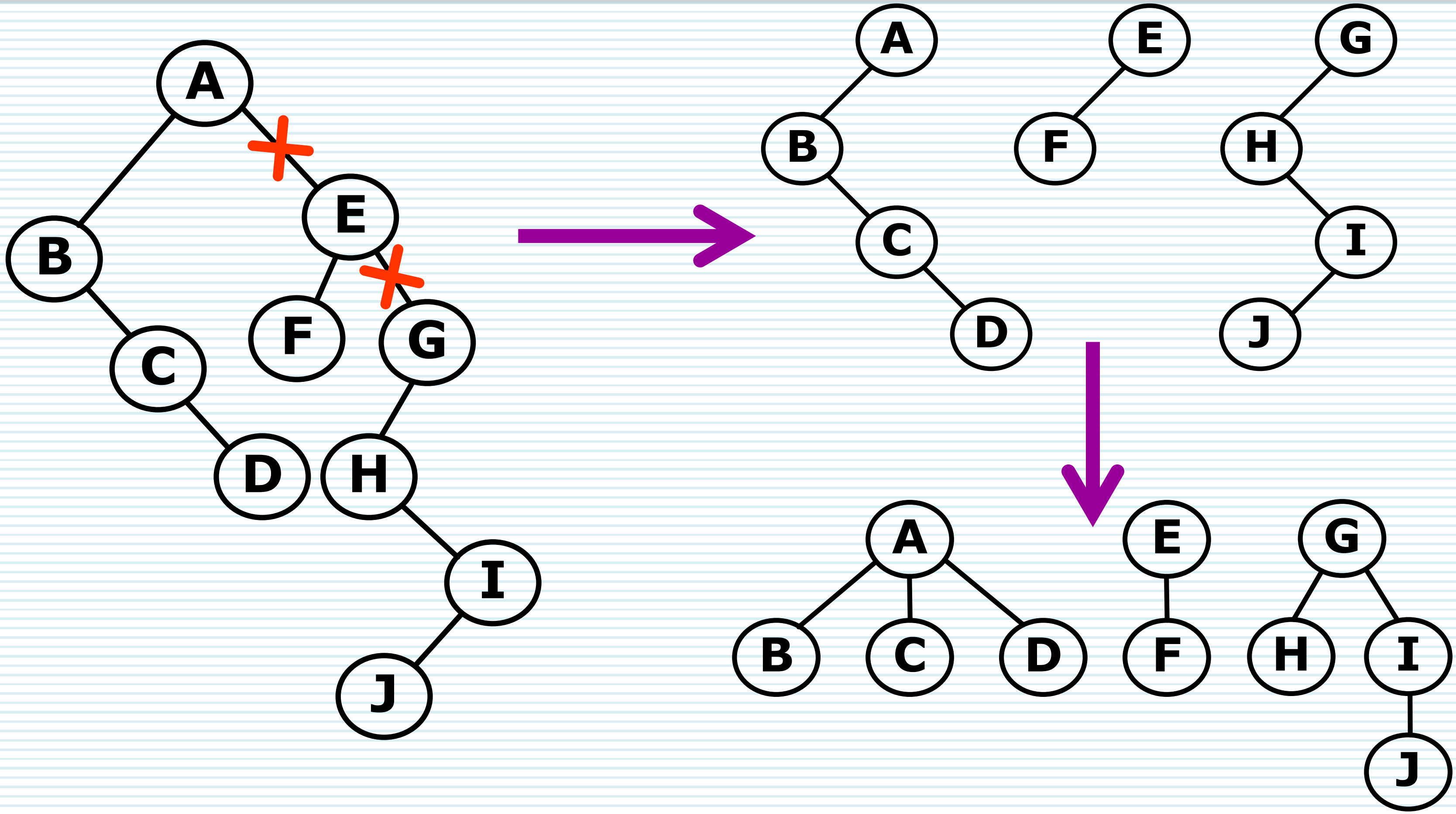


为什么这样连接?

树转换成的二叉树其右子树一定为空

1. 将各棵树分别转换成二叉树，将每棵树的根结点用线相连
2. 以第一棵树根结点为二叉树的根
   - 再以根结点为轴心，顺时针旋转，构成二叉树型结构

二叉树转换成森林

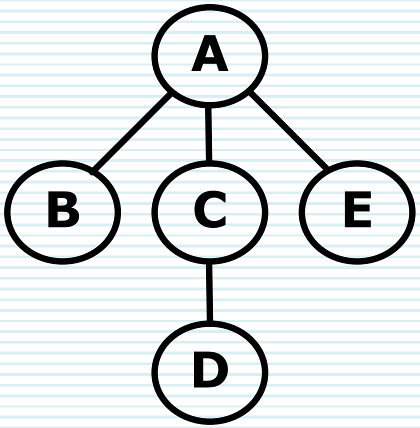


1. 割线：将二叉树中根结点与其右孩子、及沿右分支搜索到的所有右孩子间的连线全部去掉，使之变成孤立的二叉树
2. 还原：将孤立的二叉树还原成树

树和森林的遍历

- 树的遍历：无遗漏地访问树的结点，使每一结点仅被访问一次
  - 即：找出一个完整而有规律的走法
  - 得到树中所有结点的一个线性排列
- 常用的遍历方法
  - 先根次序遍历：先访问树的根结点
    - 然后依次先根遍历根的每棵子树(从左到右)
  - 后根次序遍历：先依次后根遍历每棵子树(从左到右)
    - 最后访问根结点
  - 层次遍历：先访问树的第一层上的结点，然后依次遍历第二层……直到第n层的结点，每层遍历顺序为从左到右

树的遍历

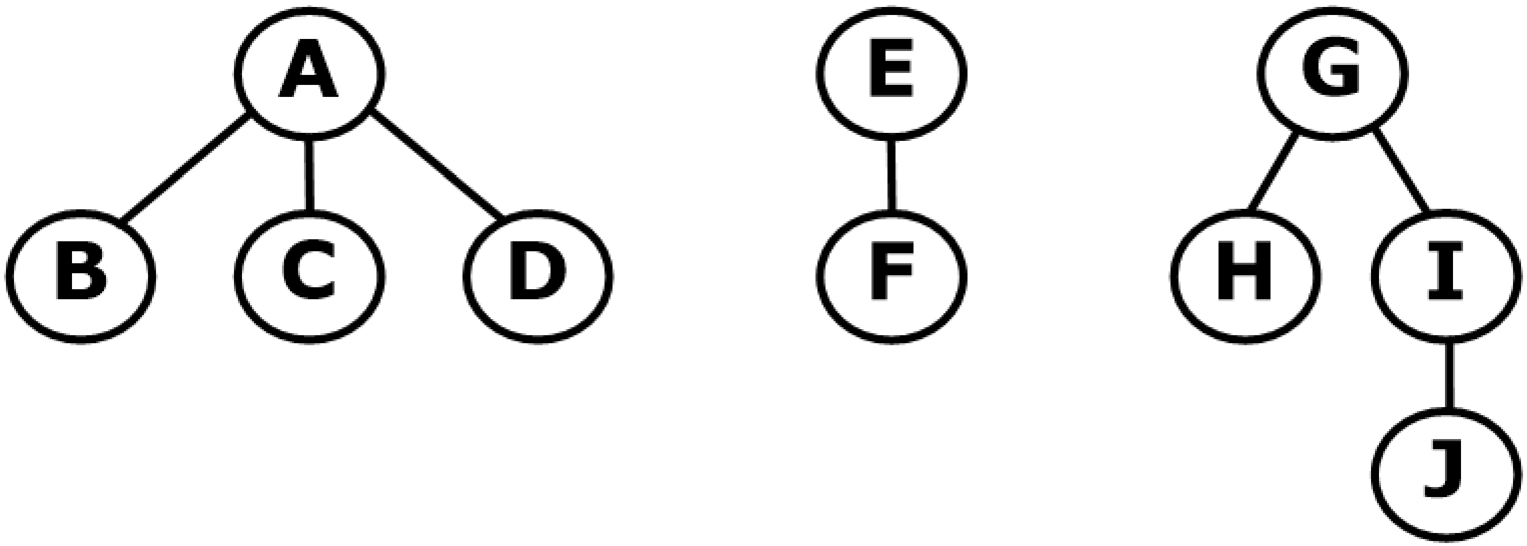


前序遍历结果： A B C D E

后序遍历结果： B D C E A

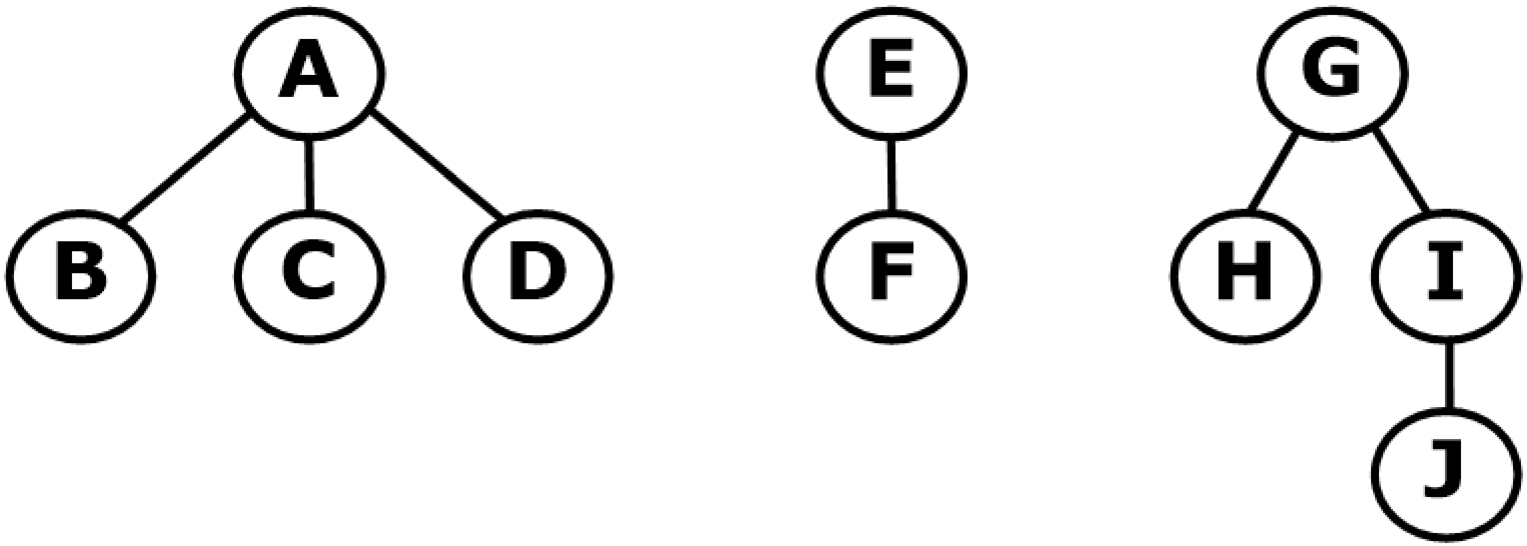
- 树的**前序**遍历与将树转换成二叉树后对其**前序**遍历的结果相同
- 树的**后序**遍历与将树转换成二叉树后对其**中序**遍历的结果相同
- 因此：当以二叉链表作为树的存储结构时
  - 对树执行前序遍历和后序遍历操作
  - 可借用二叉树的前序遍历和中序遍历的算法实现

森林的遍历



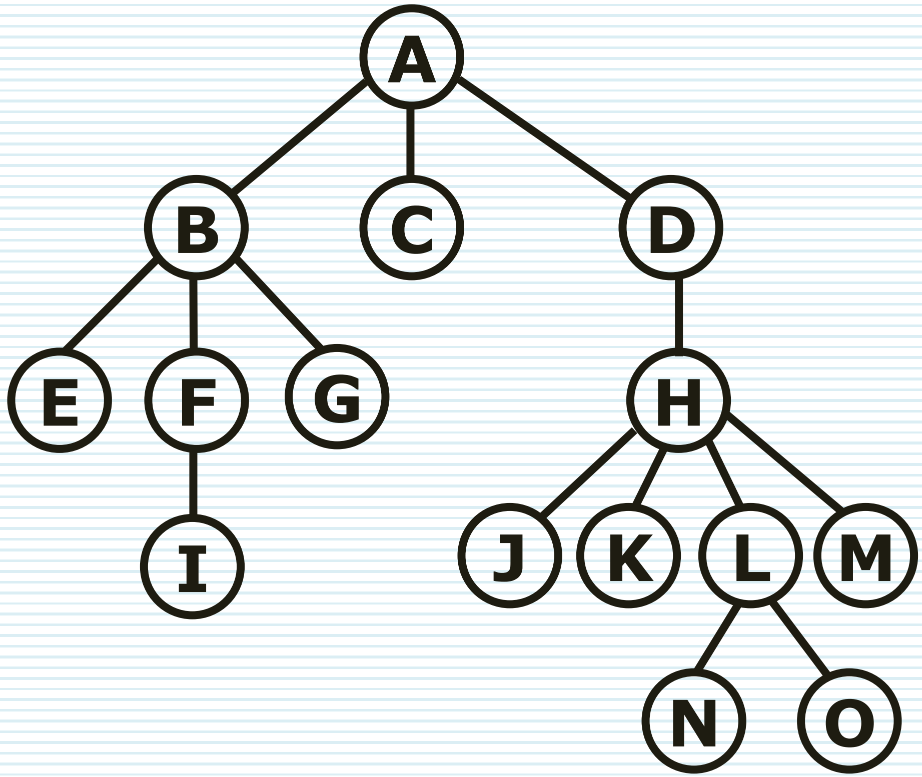
- 前序遍历森林的方法：依次前序遍历森林中的每棵树
  - 访问森林中第一棵树的根节点
  - 前序遍历第一棵树中根节点的子树森林
  - 前序遍历森林中剩余的树
- 上图的前序遍历结果：**A B C D E F G H I J**

森林的遍历



- 中序遍历森林的方法：依次**后序**遍历森林中的每棵树
  - 中序遍历森林中第一棵树的根节点的子树森林
  - 访问第一棵树的根节点
  - 中序遍历剩余的树构成的森林
- 上图的中序遍历结果：**B C D A F E H J I G**

树的遍历



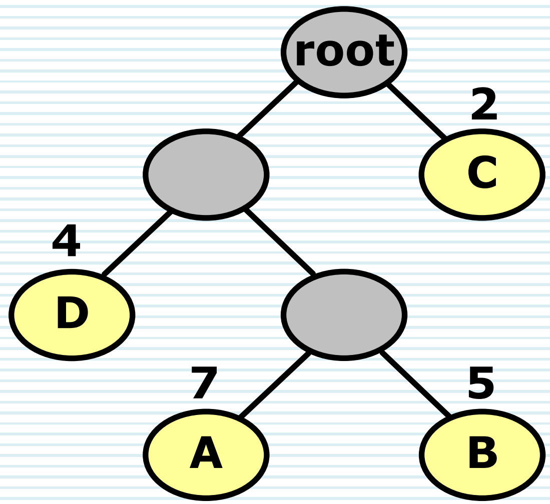
前序遍历： A B E F I G C D H J K L N O M

后序遍历： E I F G B C J K N O L M H D A

层次遍历： A B C D E F G H I J K L M N O

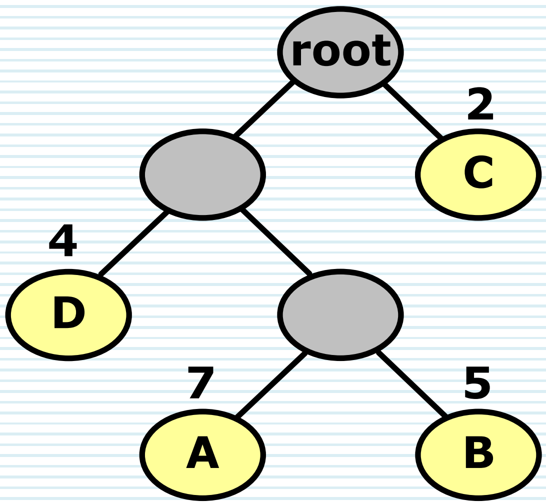
4. 哈夫曼树与哈夫曼编码

哈夫曼树（Huffman Tree）



- 二叉树的几个基本概念
  - 路径：从树中一个上层结点到一个下层结点之间经过的分支
  - 路径长度：路径中的分支数
  - 树的路径长度：从树根到树中每一个结点的路径长度之和
  - 结点的带权路径长度（若结点含权）
    - 结点到根的路径长度与结点权值的乘积

哈夫曼树（Huffman Tree）



WPL = 4×2+7×3+5×3+2×1 = 46

- 二叉树的几个基本概念
  - 树的带权路径长度：树中所有含权结点的路径长度之和

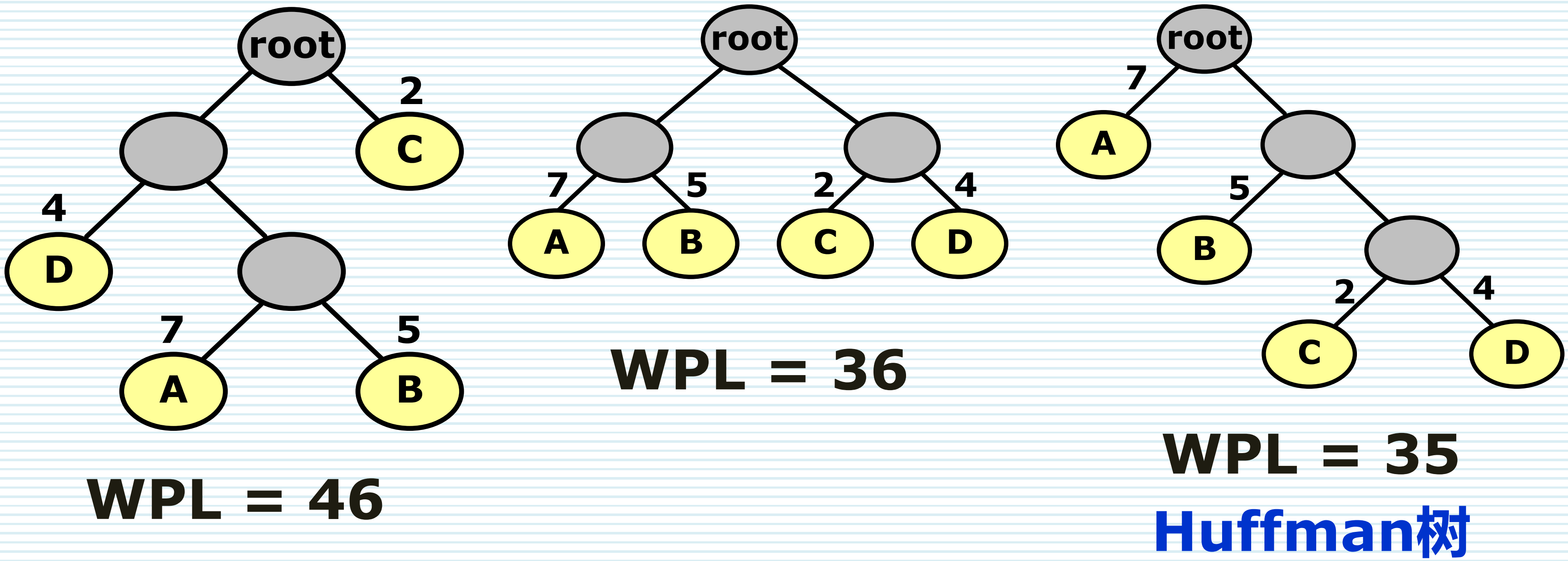
记为：

$$\mathrm{WPL} = \sum_{k=1}^{n} \mathrm{w}_k \cdot \mathrm{s}_k$$

其中：$\mathbf{w_k}$ 表示结点**k**的权值

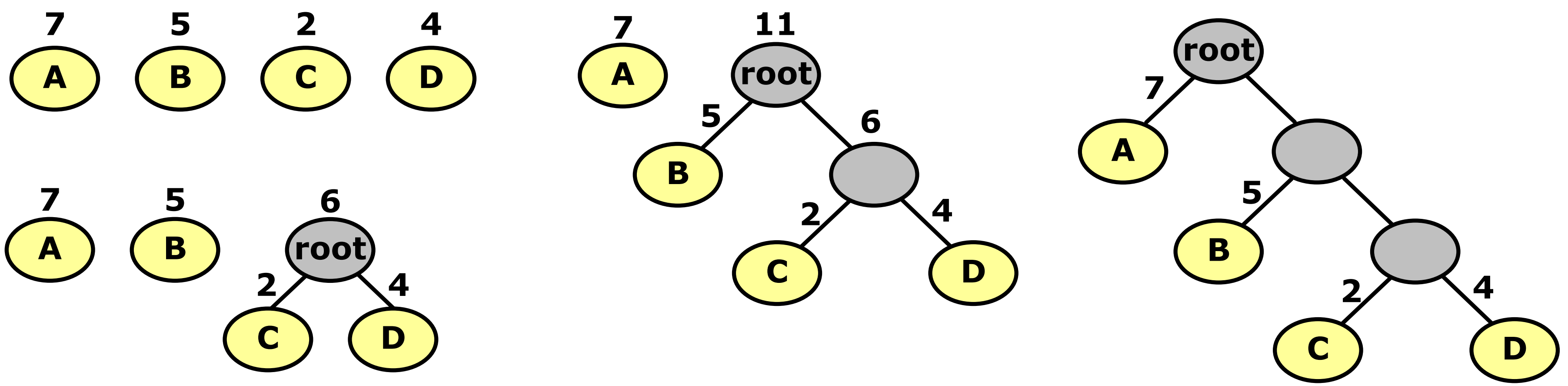
$\mathbf{s_k}$ 表示结点**k**到根结点的路径长度

哈夫曼树（Huffman Tree）



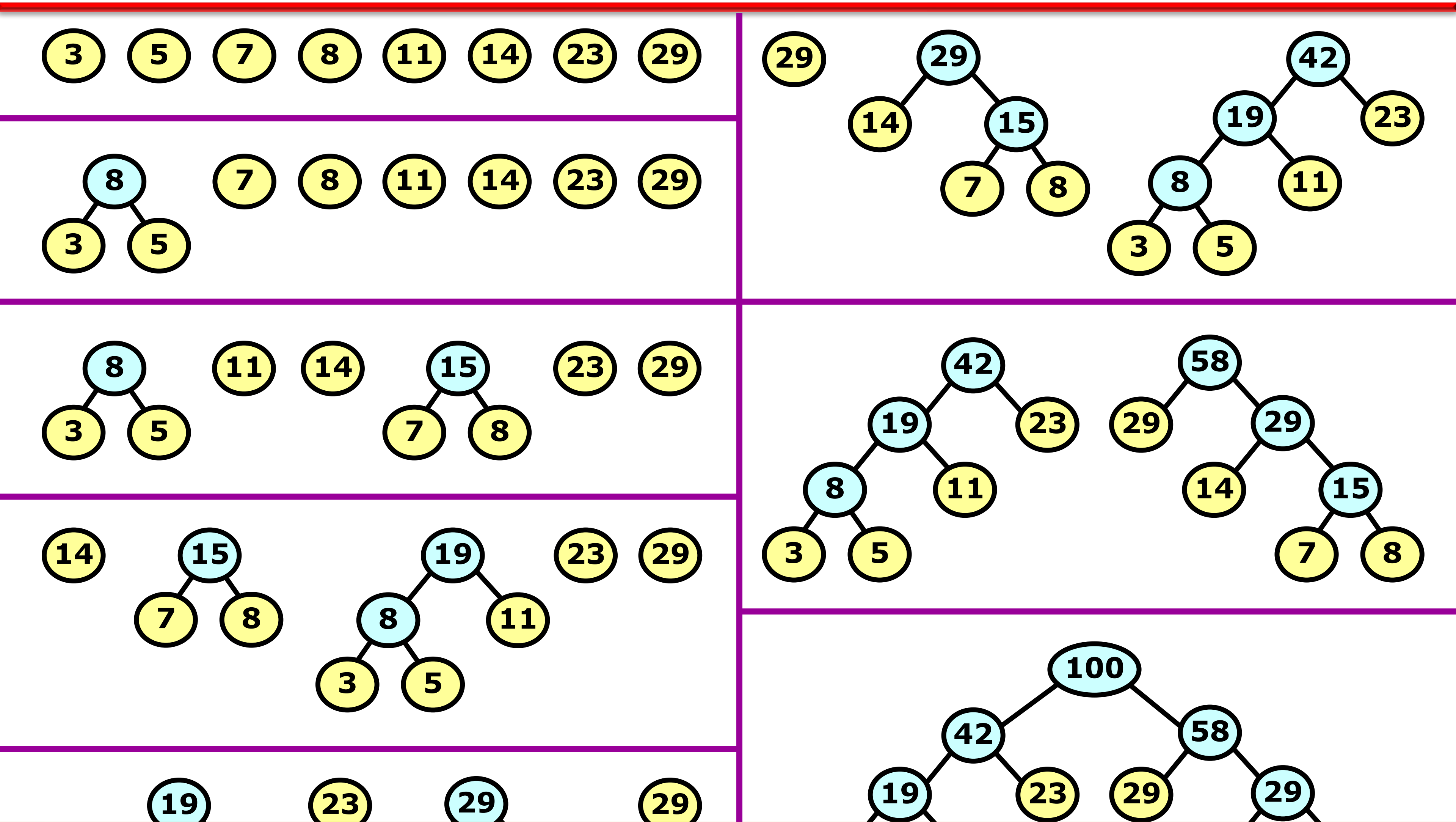
- 哈夫曼树：带权路径长度最短的二叉树（最优二叉树）
  - 设：给定n个权值：{ w_1 , …, w_n }
  - 据此构造一棵有n个叶结点的二叉树（每个叶结点权值为w_i）
  - 其中：WPL最小的二叉树称为Huffman树

Huffman树的构造方法：Huffman算法



1. 根据给定的n个权值：{w_1,w_2,……w_n}，构造n棵只含根结点的二叉树，令每棵树的权值为相应的结点权值（w_j）
2. 在森林中选取两棵根结点权值最小的树作为左右子树，构造一棵新的二叉树，新树根节点权值为其左右子树根结点权值之和
3. 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
4. 重复上述两步直到森林中只含一棵树为止，这棵树即哈夫曼树

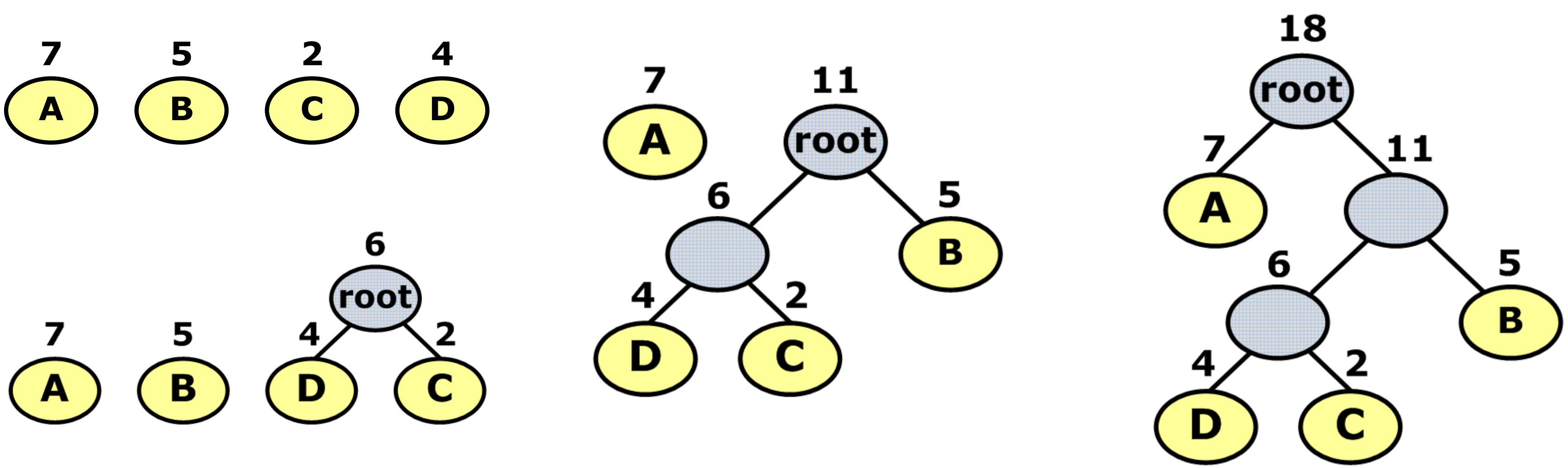
Huffman算法示例：w={5, 29, 7, 8, 14, 23, 3, 11}



问题：如此建立的哈夫曼树是否唯一？

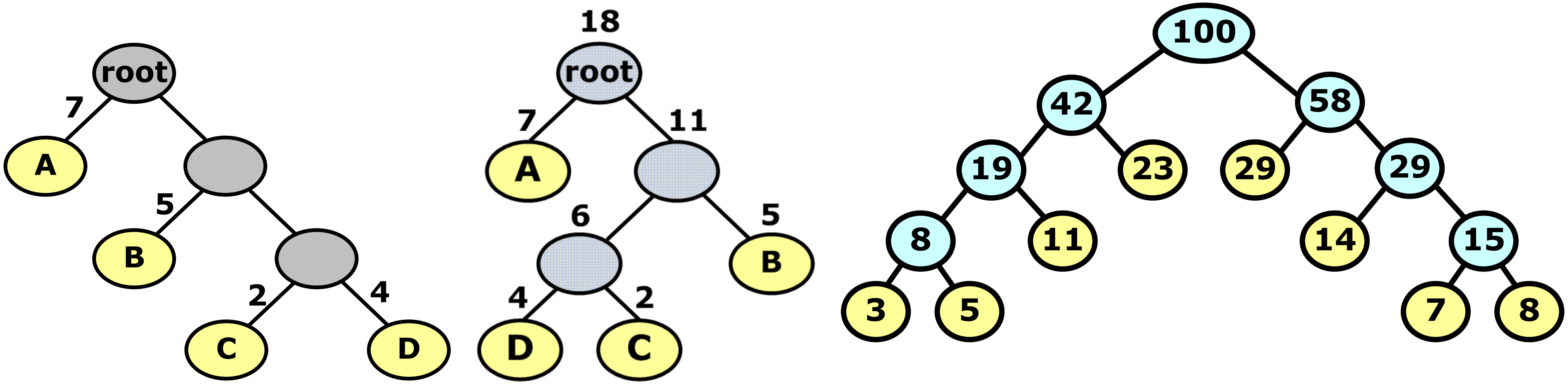
Huffman算法的结果是否唯一？

- 解决方案：为了规范Huffman树的构造算法，规定如下
  1. 设当前森林为：$F=\{T_1, T_2, \cdots, T_n\}$，构造时选择：
  2. 权值小的二叉树作为新构造的二叉树的左子树
  3. 权值大的二叉树作为新构造的二叉树的右子树
  4. 在权值相等时，深度小的二叉树作为新构造的二叉树的左子树，深度大的二叉树作为新构造的二叉树的右子树



WPL = 4×3+2×3+5×2+7×1 = 35

哈夫曼树的特点



- n个叶子的哈夫曼树的形态一般不唯一
  - 但带权路径长度（WPL）是相同的
- n个叶子的哈夫曼树共有2n-1个结点
  - 权值大的结点离根结点近
  - 哈夫曼树只有度为0和2的结点，无度为1的结点

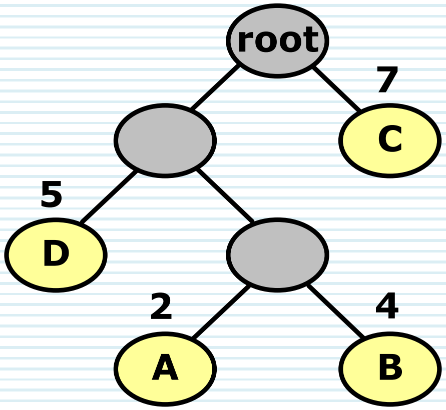
Huffman算法的实现

```
typedef struct{
    int weight, parent;
    int lchild, rchild;
}TNode, *PHT;
```

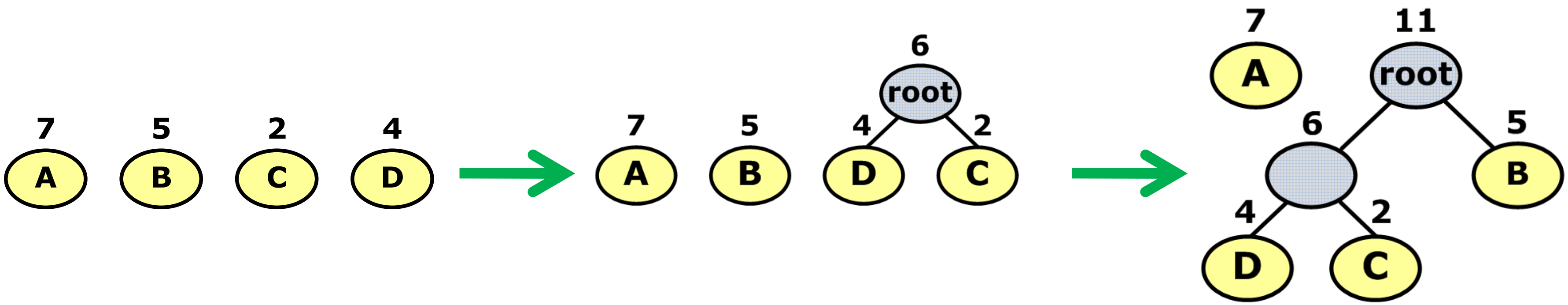
Huffman树的结点定义

weight	lchild	rchild	parent

- 数据结构设计
  - Huffman树的结点构成要素
    - 左右孩子指针
    - 父指针：记录当前结点隶属于哪个中间结点
    - 权值：叶节点权值已知，中间结点的权值计算得到



Huffman算法的实现



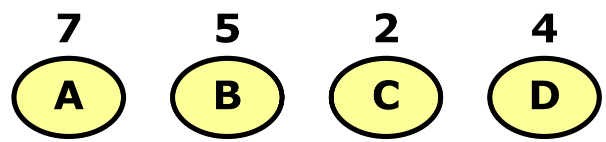
- 数据结构设计：采用顺序存储还是链式存储?
  - 提示：Huffman树的构造是反复从森林中选择子树进行合并
    - 子树的选择：父指针为空，根权重最小
    - 子树的合并：新增一个中间结点（总共新增n-1个）
  - 设置一个大小为2n-1的结构数组存储Huffman树的结点

Huffman算法

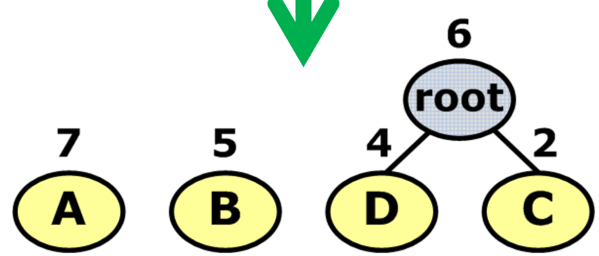
- Huffman算法流程
  - 首先将n个叶结点存入数组
    - 父指针置为-1表示根节点
  - 循环处理（直至数组被填满）
    - 从现有子树的根结点中选择两个权重最小者
    - 构造新子树的根结点加入数组
    - 修改选中结点的父指针指向新的根结点
    - 同时更新该新根节点的孩子指针和权重值

Huffman算法的实现

data	parent
lchild	rchild



7	−1	5	−1	2	−1	4	−1	......			−1
−1	−1	−1	−1	−1	−1	−1	−1			−1	−1
0		1		2		3		......		2n-2	



7	−1	5	−1	2	4	4	4	6	−1	......	
−1	−1	−1	−1	−1	−1	−1	−1	3	2		
0		1		2		3		4		......	

Huffman：子树选择算法

```
void select_subtree(PHT pht, int n, int *pa, int *pb){
    int  id,  ida = -1, idb= -1;            // ida存放权重最小的结点下标
    int  wa = INT_MAX,  wb = INT_MAX;    // wa最小值 wb次小值
    for(id = 0; id <= n; id++){
        if(pht[id].parent == -1){
            if( pht[id].weight < wa ){
                idb = ida;  wb = wa;
                ida = id;  wa = pht[id].weight;  }
            else if(pht[id].weight < wb ){
                idb = id;  wb = pht[id].weight;  }
        }
    }
    *pa = ida;    *pb = idb;     return;
}
```

构建Huffman树

```
PTH create_htree( int weights[],  int n ){
    PHT  pht;   int i,  lc,  rc,  ntotal = 0;
    ntotal = (2 * n) - 1;                    // Huffman树的结点总数
    pht = (PHT) malloc( sizeof( TNode ) * ntotal );
    for( i = 0; i < ntotal; ++i ){           // Huffman树结点初始化
        pht[i].weight = (i < n) ? weights[i] : 0;
        pht[i].lchild = -1; pht[i].rchild = -1; pht[i].parent = -1;
    }
    for( i = n; i < ntotal; ++i ){           // 构建Huffman树
        select_subtree( pht,  (i-1),  &lc,  &rc );
        pht[lc].parent = i;    pht[rc].parent = i;
        pht[i].lchild = lc;    pht[i].rchild = rc;
        pht[i].weight = pht[lc].weight + pht[rc].weight;
    }
    return pht;
}
```

哈夫曼编码

哈夫曼编码

- 编码：将文件字符转换为二进制位串（数据压缩）
- 解码：将二进制位串转换为文件字符（数据解压）
- 编码方式：等长编码和变长编码
- 哈夫曼编码（压缩率通常在20%~90%之间）
  - 是广泛应用于数据文件压缩的一种十分有效的编码方法
- 哈夫曼编码算法的基本思路
  - 使用字符在文件中出现的频率表作为输入
  - 目标是：构建一个用0/1位串表示各字符的最优表示方式
    - 为出现频率较高的字符赋予较短的编码
    - 为出现频率较低的字符赋予较长的编码
    - 由此实现对文件总编码长度的压缩

等长编码

例如：需将文字“ABACCDA”转换成电文

分析：文字中有四种字符，用2位二进制便可分辨

编码方案	A	B	C	D
等长编码	00	01	10	11

则上述文字的电文为：00010010101100　　共14位

译码时：只需每2位一译即可

特点：等长等频率编码，译码容易，但电文不一定最短

不等长编码

例如：需将文字“ABACCDA”转换成电文

编码方案2

不等长编码

A	B	C	D
0	00	1	01

采用不等长编码，让出现次数多的字符用短码

则ABACCDA文字的电文为：000011010　　共9位

但无法译码：既可译为BBCCACA，也可译为AAAACCDA等

前缀码

例如：需将文字“ABACCDA”转换成电文

编码方案3

前缀码

A	B	C	D
0	110	10	111

采用不等长编码

- 出现次数多的字符用短码
- 且任一编码不能是另一编码的前缀

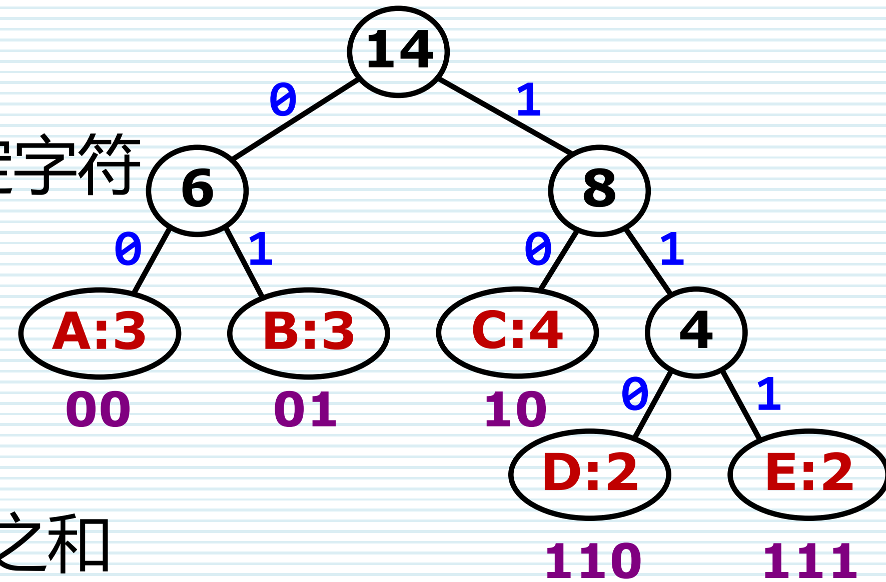
则ABACCDA文字的电文为：0110010101110　共13位

前缀码（prefix code）

- 前缀码：对每一个字符规定一个0/1串作为其代码
  - 要求：**任一字符的代码都不是其他字符代码的前缀**
  - 这种编码称为前缀码（标准书面语， prefix-free code）
- 为什么要关注前缀码
  - 已经证明：通过字符编码获得的最优数据压缩方式总可用某种前缀编码来表达，因此算法设计时考虑前缀码不失一般性
  - 编码的前缀性质可以简化编解码方式
    - 编码：只要将文件中表示每个字符的编码并置起来即可
    - 解码：只需对第一个编码进行解码，然后迭代进行解码
      - 由于是前缀码，因此被编码文件的起始编码是确定的

前缀码的二叉树表示

- 前缀码可以采用二叉树进行表示
  - 利用二叉树的性质，可以很方便地对前缀码进行解码
- 前缀码二叉树的数据结构
  - 二叉树的叶节点表示一个特定字符
    - 出现的频率（即权重）
  - 二叉树的内节点表示
    - 其子树中所有叶子的频率之和
  - 字符的编码为从根至该字符的路径
    - 路径上的字符0表示：转向左子节点
    - 路径上的字符1表示：转向右子节点



最优前缀码

- 平均编码长度
  - 设：字母表A中的某个字符c在文件中出现的频率为：f(c)
  - 对于给定的编码方案，设对应的二叉树表示为T
  - 则：字符c在T中的深度 $d_T(c)$ 就是该字符的编码长度
  - 该编码方案的平均码长定义为：$B(T)=\sum_{c\in A} f(c)\cdot d_T(c)$
  - 即：编码该文件需要的位（bit）数，也称为树T的代价
- 最优前缀码
  - 使平均编码长度达到最小的前缀编码方案
  - 称为给定字符集A的最优前缀码

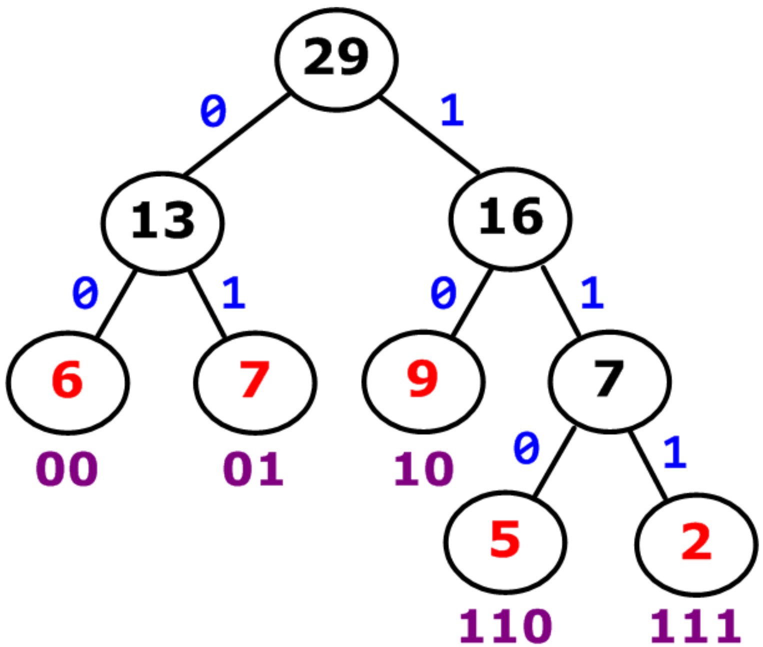
最优前缀码的性质

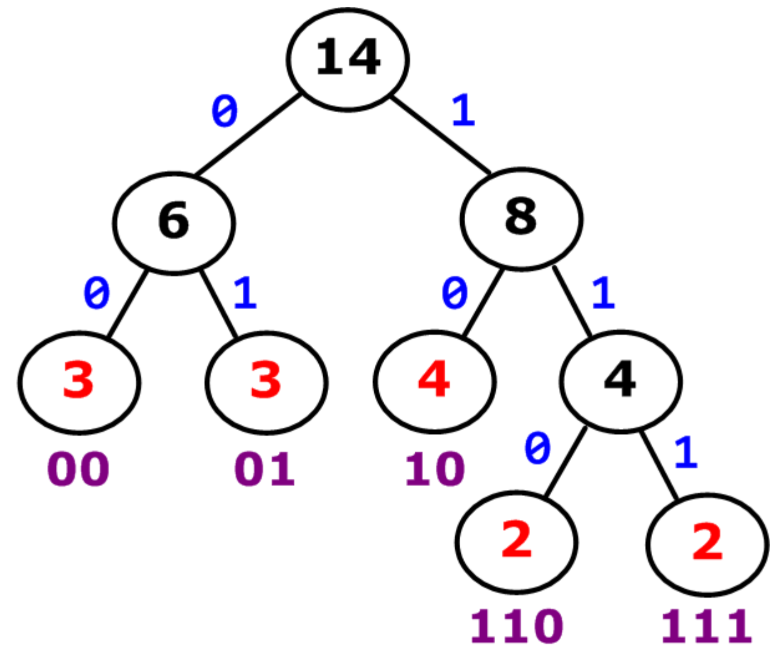
- 表示最优前缀码的二叉树总是一棵完全二叉树
  - 即：树中任何一个内节点都有2个子节点
- 如果A是包含待编码字符的字母表
  - 则：表示最优前缀编码的树T中恰有|A|片叶子
    - 每个叶节点表示字母表中的一个字母
  - 表示最优前缀编码的树T中共有|A|-1个内节点
- 哈夫曼提出了一种构造最优前缀码的贪心算法
  - 由此产生的编码方案称为哈夫曼编码
  - 编码一个文件所需要的位数即哈夫曼树的带权路径长度

Huffman编码

- 哈夫曼编码方法
  - 设：有n种字符（每种字符出现的次数为 w_i ）
    - 设每种字符的编码长度为 d_i
    - 则整个电文总长度为：$\sum w_i d_i$ ，
  - 要得到最短的电文（即使得$\sum w_i d_i$ 最小）：
    - 以字符出现的次数为权值构造一棵Huffman树
    - 规定：左分支编码为0，右分支编码为1
    - 则字符的编码为：从根到该字符所在的叶结点的路径上的分支编号构成的序列
  - 用Huffman树编出来的码称为Huffman编码

Huffman编码





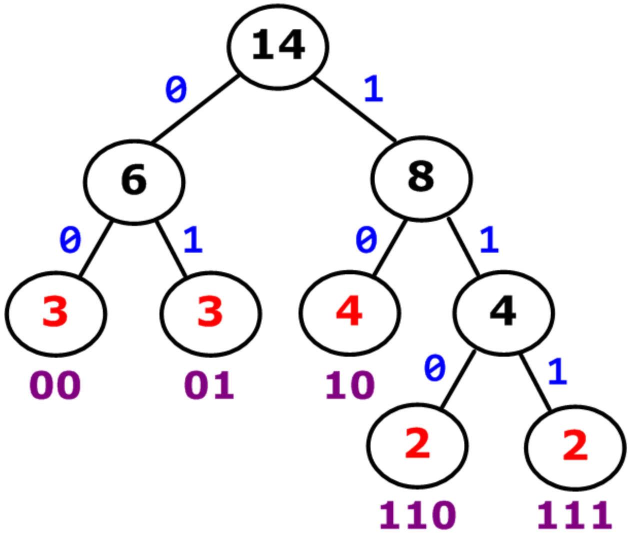
A:9	10
B:7	01
C:6	00
D:5	110
E:2	111

A:4	10
B:3	01
C:3	00
D:2	110
E:2	111

哈夫曼编码方法

- 通过回溯生成字符的哈夫曼编码（编码本）
  1. 选择哈夫曼树的某个叶结点（设其下标为 idxa）
  2. 利用parent指针找到其父结点（设其下标为 idxb ）
  3. 利用父结点的孩子指针域判断该结点是左孩子还是右孩子
     - 若该结点是左孩子（ lchild== idxa），则生成代码0
     - 若该结点是右孩子（ rchild== idxb），则生成代码1
  4. 重复步骤(2)~(3) 直至回溯到根节点，得到一个0/1序列
     - 思考：这个0/1序列是否为该字符的Huffman编码?
     - 该序列是Huffman编码的逆序：将其反序得到字符编码
  5. 重复步骤(1)~(4)，实现对全部叶节点的编码

哈夫曼编码的存储结构



A:4	10	0	'A'	"10"
B:3	01	1	'B'	"01"
C:3	00	2	'C'	"00"
D:2	110	3	'D'	"110"
E:2	111	4	'E'	"111"

```
#define LEN 100              // 待编码字符个数
typedef struct{
      char ch;               // 存储字符
      char code[LEN];        // 存放编码
}TCode;
TCode CodeBook[LEN];   // 编码本
```

根据Huffman树求字符编码表

```
void encoding (PHT pht, TCode *book, int n){
    char *str = (char *)malloc(n+1); // 临时存放编码
    str[n] ='\0';    int i, j, idx, p;
    for(i = 0, i < n, i++){          // 依次求叶子pht[i]的编码
        book[i].ch = pht[i].ch;  idx = i;  j = n;
        while( p = pht[idx].parent > 0){
            if(pht[p].lchild == idx){
                j--;  str[j]='0';
            }
            else { j--;  str[j] ='1'; }
            idx = p;
        }
        strcpy(book[i].code, &cd[j]); // 复制编码位串
    }
}
```

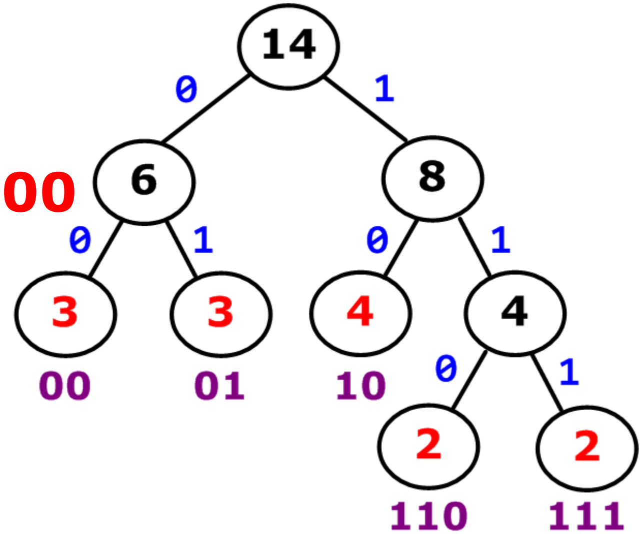
Huffman编码的译码操作

明文是：CAS;CAT

编码为：11010111011101000

密文是：1101000

译文为：CAT



T : 3	00
; : 3	01
A : 4	10
C : 2	110
S : 2	111

- 从待译码电文中逐位读取编码
  - 从Huffman树根开始
    - 若编码是'0'： 则沿lchild下行
    - 若编码是'1'： 则沿rchild下行
  - 若到达叶结点：则译出一个字符
  - 重复上述步骤，直到电文结束

哈夫曼解码算法

```
void decoding(PHT pht, char* codes, int n){
    int  i = 0, p = 2*n - 2;        // 从根结点开始
    while(codes[i]!='\0'){       // 当要解码的串没有结束时
        while(pht[p].lchild != -1 && pht[p].rchild != -1){
            if (codes[i]=='0') p = pht[p].lchild;
            else p = pht[p].rchild;
            i++;
        }
        printf("%c", pht[p].ch);  p = 2*n-2;
    }
    printf("\n");
}
```

树的应用：堆排序

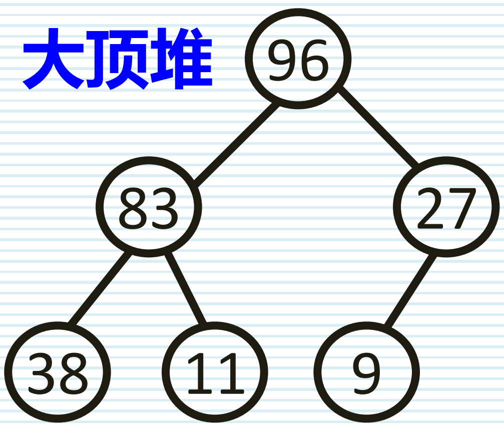
堆排序

- n个元素（k_i）的序列，当且仅当满足下列关系时，称之为**堆**

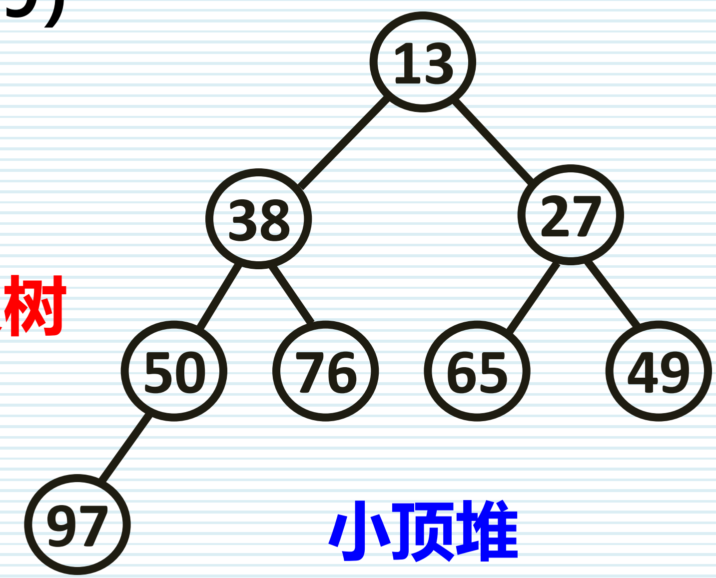
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad (i=1,2,\ldots\ldots\lfloor n/2 \rfloor)$$

- 例　(96，83，27，38，11，9)

大顶堆



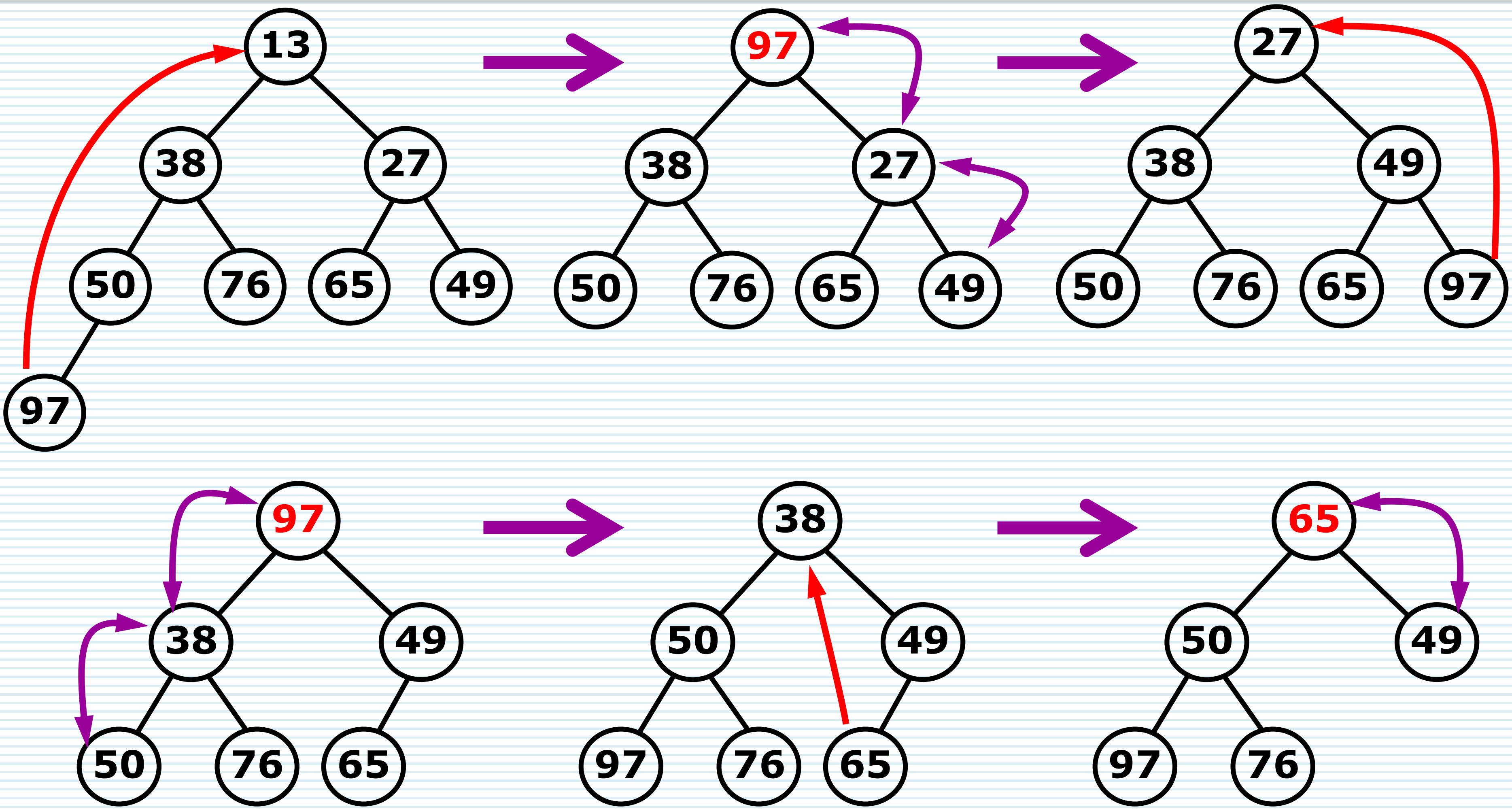
完全二叉树



小顶堆

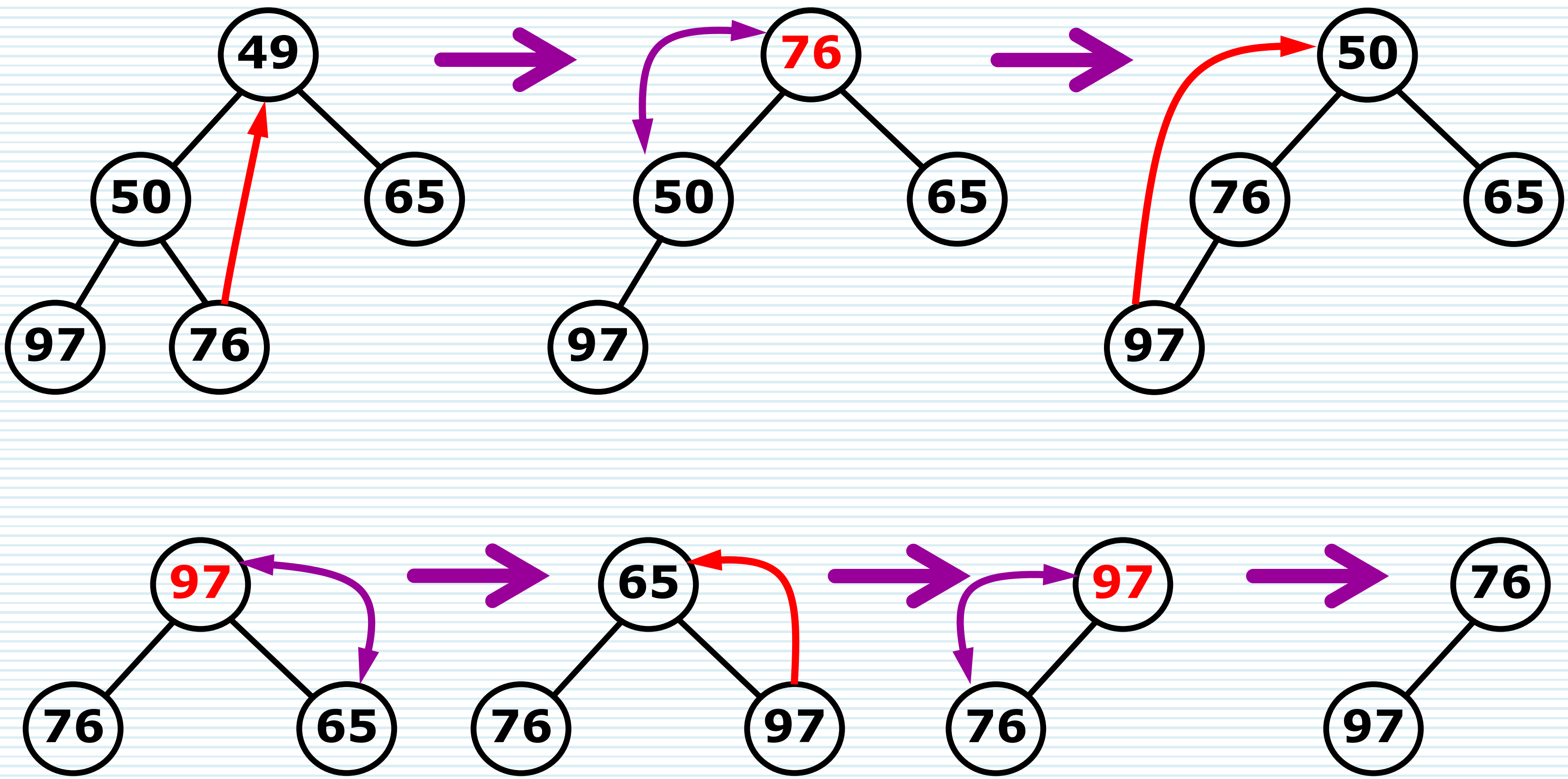
- 例：(13，38，27，50，76，65，49，97)

堆排序示例



输出: 13 27 38 ……

堆排序示例

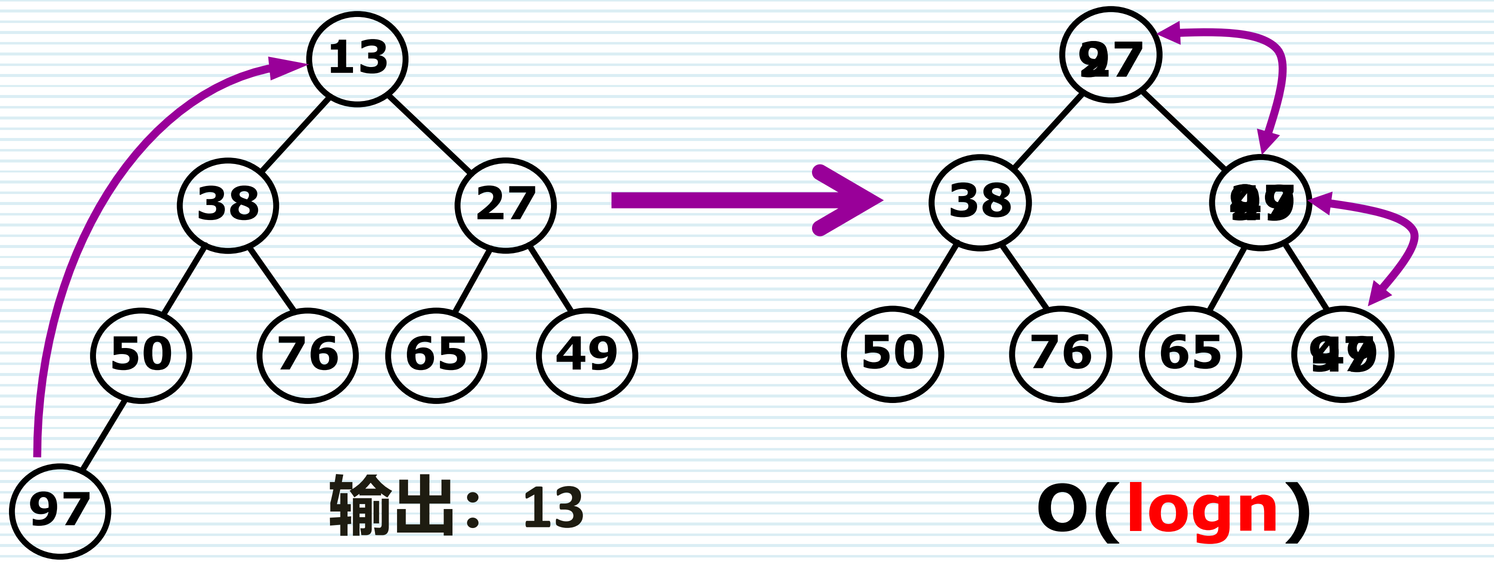


输出： 13 27 38 49 50 65 76 97

堆排序算法

- 堆排序算法
  1. 首先将n个元素构成的无序序列构造成一个堆
  2. 通过堆顶得到堆中元素的最小（或最大）值
  3. 取出堆顶元素，将剩余的n-1个元素重构为一个堆
  4. 通过堆顶可以得到n个元素的次小（或次大）值
  5. 重复执行得到一个有序序列，这个过程叫堆排序
- 堆排序需解决的两个问题?
  - 如何由一个无序序列建成一个堆?
  - 如何在输出堆顶之后调整剩余元素使之成为一个新堆?

输出堆顶元素后调整剩余元素成为一个新堆



- 解决方案（以小顶堆为例）
  - 输出堆顶元素之后，以堆中最后一个元素替代之
  - 比较根结点与左右子树根结点的值并与其中小者进行交换
  - 重复上述操作直至叶结点，将得到新的堆
- 称这个从堆顶至叶结点的调整过程为“筛选” （**Sift**）

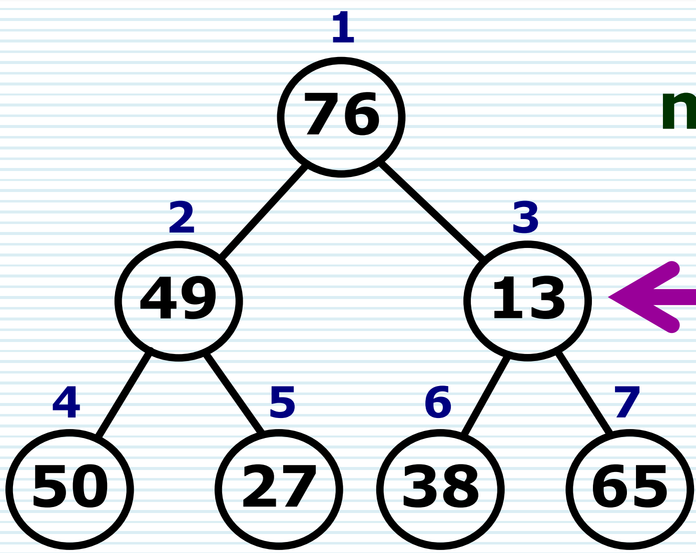
堆排序的筛选算法

```
// p是长度为n+1的数组（p[1:n]为堆元素序列）
void sift (int *p, int r, int n){  // r为指定的堆顶元素下标
   int k = 2 * r; p[0]= p[r];
   while ( k <= n ){
      if ( ( k < n) && p[k + 1] < p[k])  k++;
      if ( p[k] >= p[0]) { break; }
      p[r] = p[k];   r = k;
      k = 2 * r;
   }
   p[r] = p[0];     return;
}
```

时间复杂度

$T(n) = O(\log n)$

堆排序算法



$n=7 \quad \lfloor n/2 \rfloor = 3$

从此处开始筛选

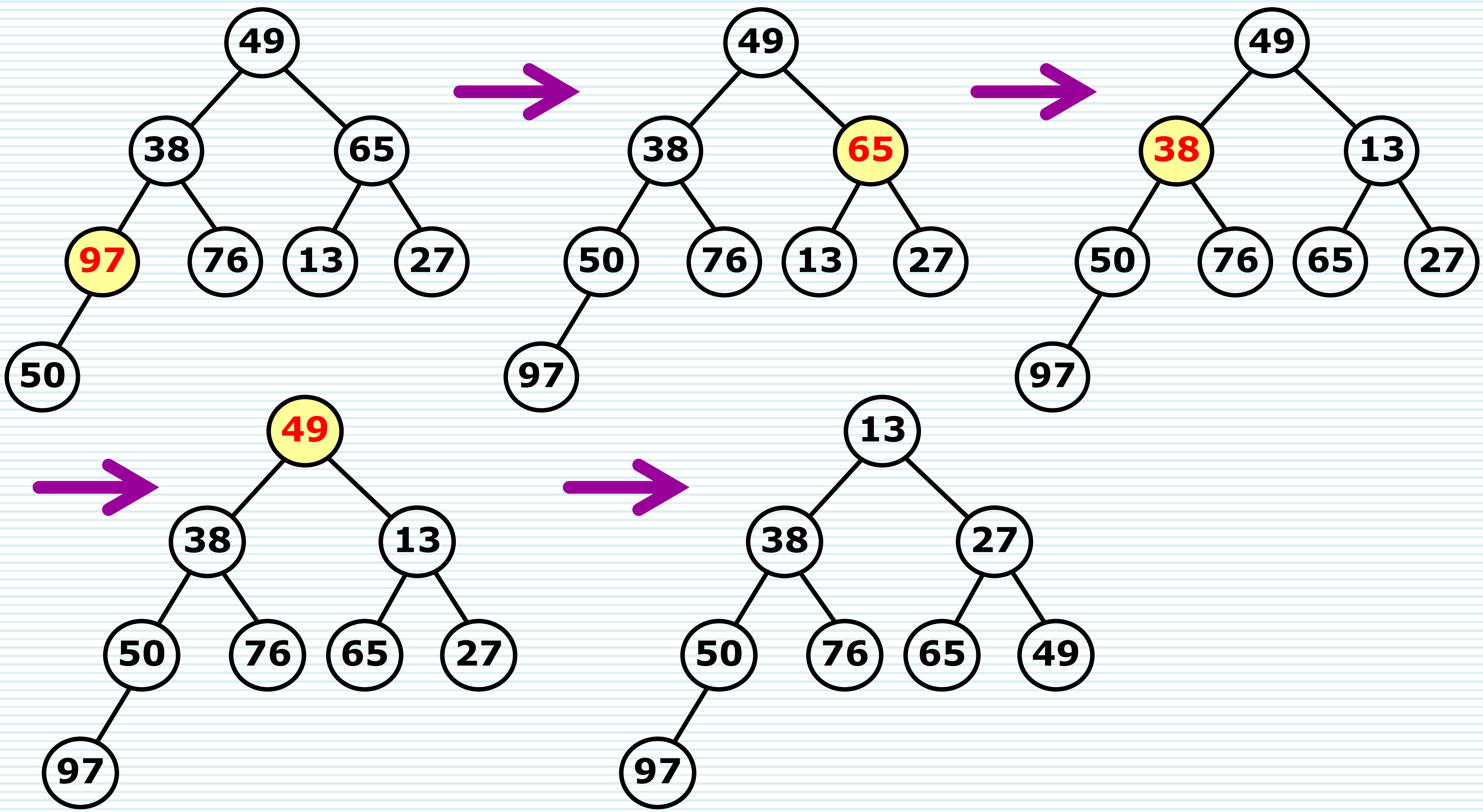
筛选顺序：3 → 2 → 1

- 如何由n个元素构成的无序序列构建一个堆?
  - 从无序序列的第 $\lfloor n/2 \rfloor$ 个元素起
  - 至第一个元素止，进行反复筛选
- 无序序列的第 $\lfloor n/2 \rfloor$ 个元素是什么意思?
  - 即：该序列对应的完全二叉树的最后一个非叶结点

堆的构建算法示例

例：对由如下8个元素构成的无序序列进行建堆操作

{ 49，38，65，97，76，13，27，50 }　$\lfloor n/2 \rfloor = 4$



堆排序的建堆算法

```
// p是长度为n+1的数组（p[1:n]为堆元素序列）
void build_heap (int *p, int n) {
    int i = 0;
    for( i = n/2;  i>=1;  --i){
        sift (p, i, n);
    }
}
```

时间复杂度

$$T(n) = O(n\log n)$$

堆排序算法

```
void heap_sort(int *p, int n) {
    int i;
    for( i = n;  i>=2;  --i){
        p[0] = p[1];        // 保存堆顶元素
        p[1] = p[i];        // 将队尾元素交换到堆顶
        p[i] = p[0];        // p[i] 用于保存排序结果
        sift (p, 1, i-1);
    }
}
```

电子科技大学

UESTC

1956