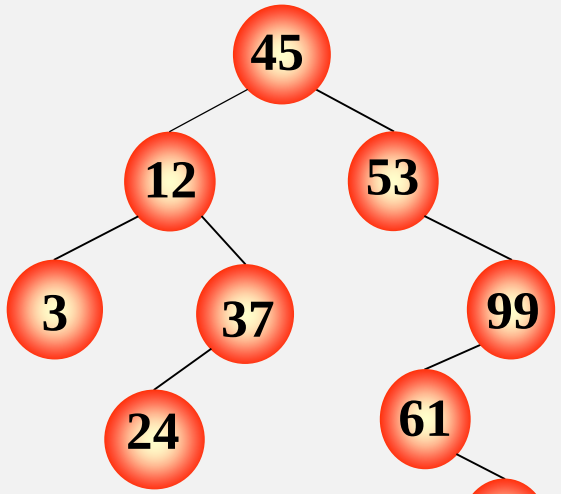
4.4.1 二叉搜索树（二叉排序树）

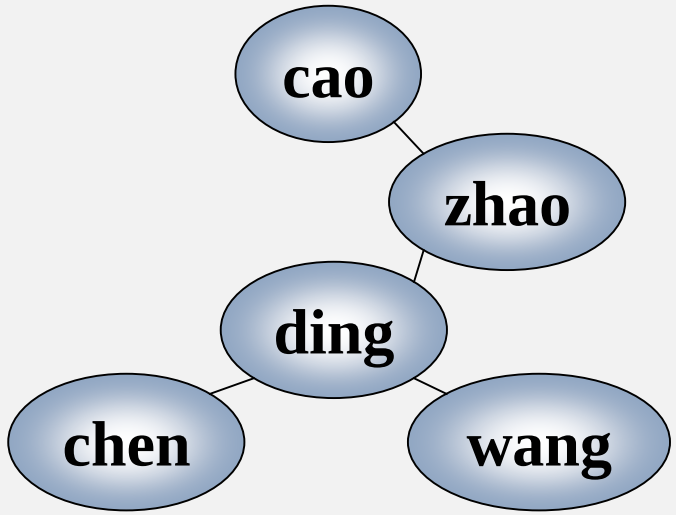
二叉搜索树或者是一棵空树；或者是具有如下特性的二叉树：

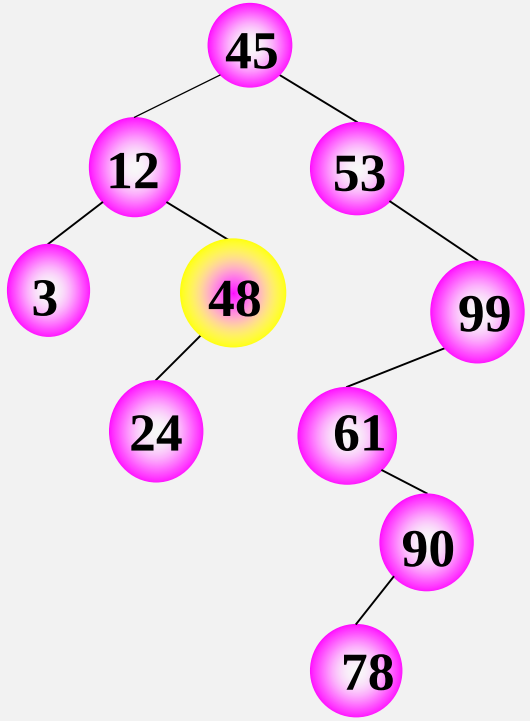
- **1）若它的左子树不空，则左子树上所有结点的值均小于根结点的值；**
- **2）若它的右子树不空，则右子树上所有结点的值均大于根结点的值；**
- **3）它的左、右子树也都分别是二叉搜索树。**

例：

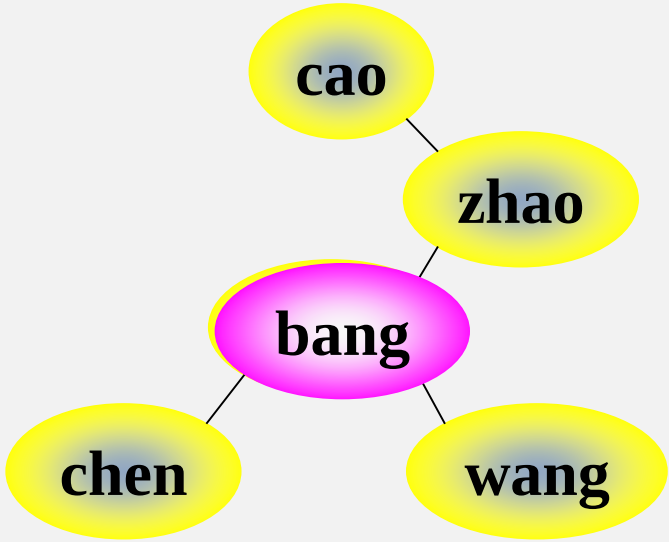


二叉搜索树





非二叉搜索树



结点类型定义如下：

二叉搜索树是二叉树的特殊情形，它继承了二叉树的结构：

```
typedef   struct   TreeNode *BinTree;
typedef   BinTree Position;
struct   TreeNode{
      ElementType   Data;
      BinTree   Left;
      BinTree   Right;
};
```

二叉搜索树增加了自己的特性，对数据的存放增加了约束

➢ 二叉搜索树的操作集中增加下列的函数：

- **Position Find(ElementType X, BinTree BST)**：从二叉搜索树BST中查找元素X，返回其所在结点的地址
- **Position FindMin(BinTree BST)**：从二叉搜索树BST中查找并返回最小元素所在结点的地址
- **Position FindMax(BinTree BST)**：从二叉搜索树BST中查找并返回最大元素所在结点的地址
- **BinTree Insert(ElementType X, BinTree BST)**：插入结点
- **BinTree Delete(ElementType X, BinTree BST)**：删除结点

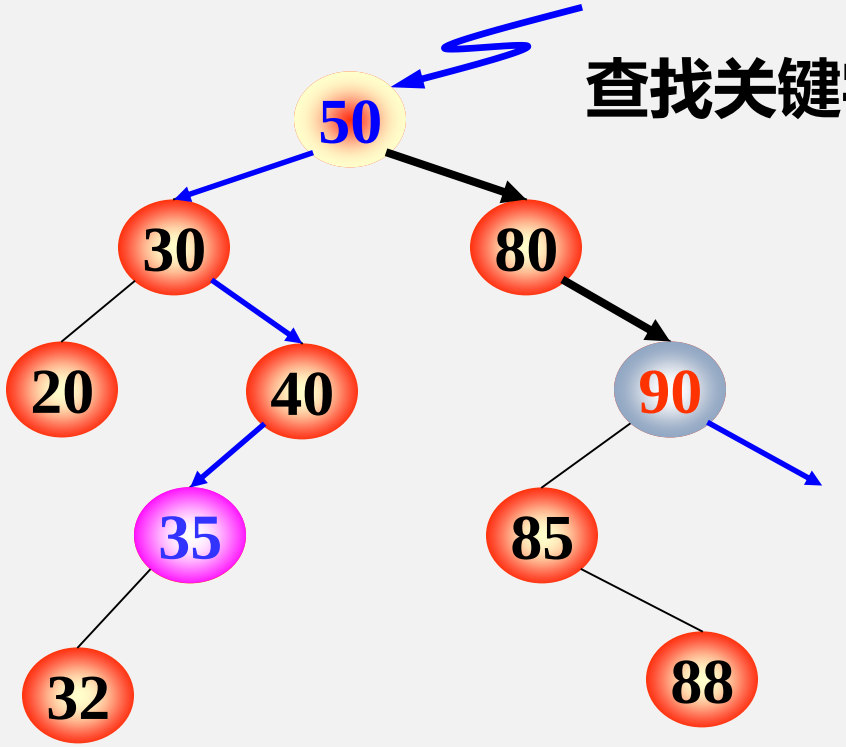
4.4.2 二叉搜索树的查找算法

若二叉搜索树为空，则查找不成功；否则

- 1) 若给定值等于根结点的关键字，则查找成功；
- 2) 若给定值小于根结点的关键字，则继续在左子树上进行查找；
- 3) 若给定值大于根结点的关键字，则继续在右子树上进行查找。

【例】

查找关键字：50，35，90，95



从根结点 出发，沿着左分支或右分支逐层向下直至关键字等于给 定值的结点。

——查找成功

从根结点出 发，沿着左分支 或右分支逐层向 下直至指针指向 空树为止。

——查找失败

1、递归查找 (用二叉链表作二叉搜索树的存储结构)：

```
Position Find( ElementType X, BinTree BST )
{
    if(BST==NULL) return NULL; /*查找失败*/
    if( X > BST->Data )
        return Find( X, BST->Right ); /*在右子树中继续查找*/
    Else if( X < BST->Data )
        return Find( X, BST->Left ); /*在左子树中继续查找*/
    else /* X == BST->Data */
        return BST; /*查找成功，返回找到的结点的地址*/
}
```

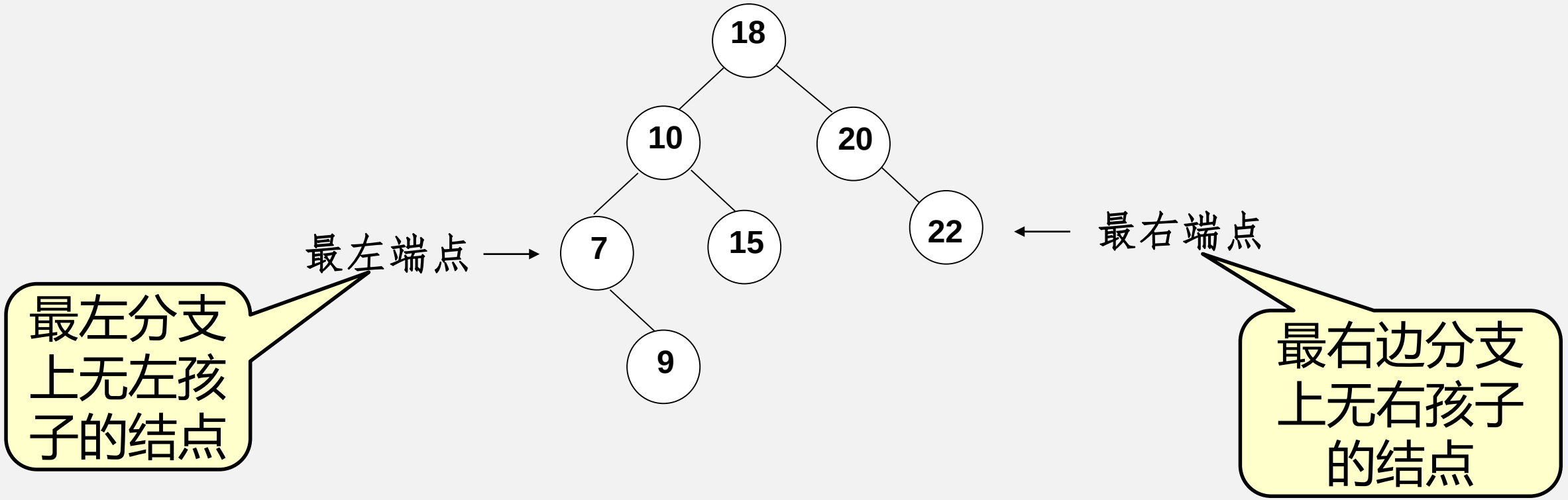
2、非递归查找函数

➢ 由于非递归函数的执行效率高，一般采用非递归的迭代来实现查找。

```
Position IterFind( ElementType X, BinTree BST )
{
    while( BST ) {
        if( X > BST->Data )
            BST = BST->Right; /*向右子树中移动，继续查找*/
        else if( X < BST->Data )
            BST = BST->Left;  /*向左子树中移动，继续查找*/
        else /* X == BST->Data */
            return BST; /*查找成功，返回找到的结点的地址*/
    }
    return NULL; /*查找失败*/
}
```

3、查找最大和最小元素

- **最大元素一定是在树的最右分枝的端结点上**
- **最小元素一定是在树的最左分枝的端结点上**



查找最小元素的递归函数

```
Position FindMin( BinTree BST )
{
    if( BST==NULL ) return NULL; /*空的二叉搜索树，返回NULL*/
     else if(BST->Left==NULL )
          return BST;   /*找到最左叶结点并返回*/
     else
          return FindMin( BST->Left ); /*沿左分支继续查找*/
}
```

查找最大元素的迭代函数

```
Position FindMax( BinTree BST )
{
     if( BST )
          while( BST->Right )  BST = BST->Right;
                /*沿右分支继续查找，直到最右叶结点*/
     return BST;
}
```

4.4.3 二叉搜索树的插入

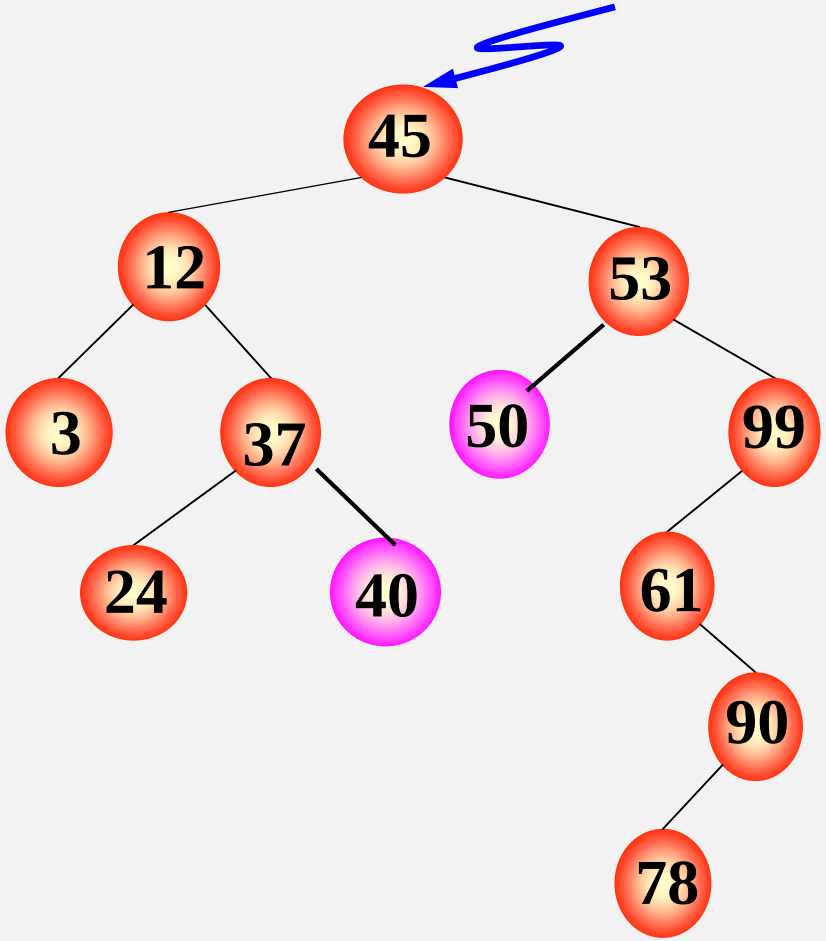
1)若二叉搜索树为空树，则新插入的结点为根结点；

2)若二叉搜索树非空，则新插入的结点必为一个新的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。

①若X<BST->data：结点X插入到BST的左子树中；

② 若X>BST->data：结点X插入到BST的右子树中。

【例】关键是找到元素应该插入的位置，可以采用与Find类似的方法



插入 40，是 37 的右孩子。

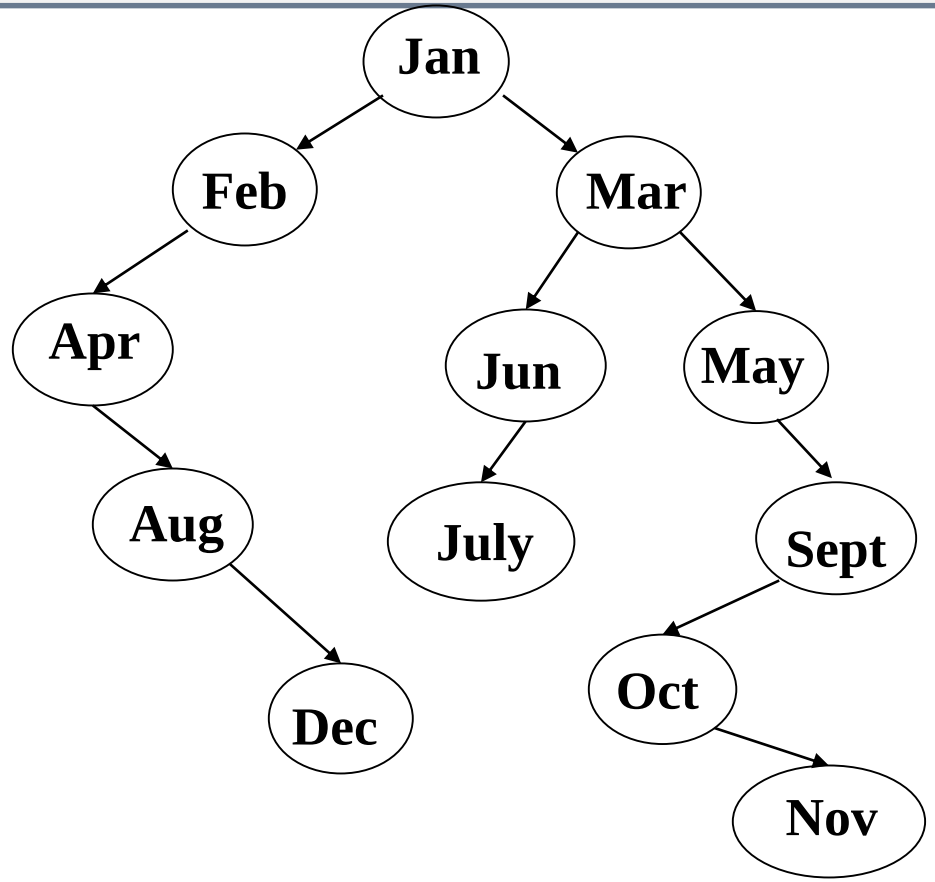
插入 50，是 53 的左孩子。

插入算法：(递归)

```
BinTree Insert( ElementType X, BinTree BST )
{
    if(BST==NULL ){/*原树为空*/
            /*生成新结点，并返回一个结点的二叉搜索树*/
        BST = malloc(sizeof(struct TreeNode));
        BST->Data = X;
        BST->Left = BST->Right = NULL;
    }
    else  /*原树不空*/
        if( X < BST->Data )
            BST->Left = Insert( X, BST->Left);
                        /*递归插入左子树*/
        else  if( X > BST->Data )
            BST->Right = Insert( X, BST->Right);
                        /*递归插入右子树*/
        /* else X已经存在，什么都不做 */
    return BST;
}
```

二叉搜索树生成：从空树出发，经过一系列的查找、插入操作之后，可生成一棵二叉搜索树。

【例】以一年十二个月的英文缩写为键值，按从一月到十二月顺序输入它们，即输入序列为（Jan, Feb, Mar, Apr, May, Jun, July, Aug, Sep, Oct, Nov, Dec），生成一棵二叉搜索树。



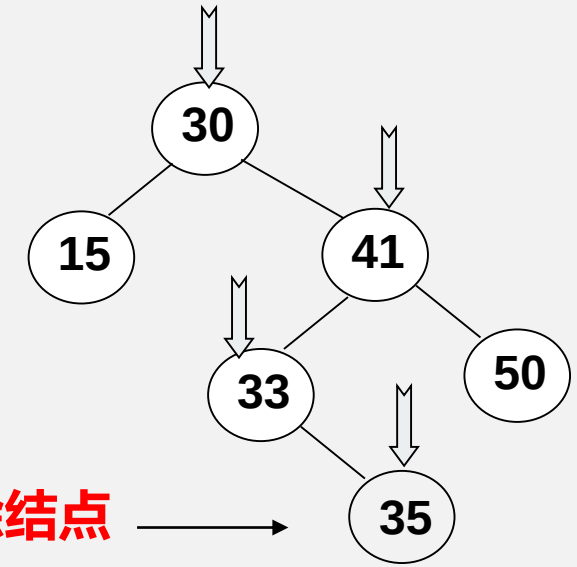
4.4.4 二叉搜索树的删除

要求：在删除某个结点之后，仍然保持二叉排序树的特性

删除二叉搜索树中的 *p 结点，分三种情况讨论：

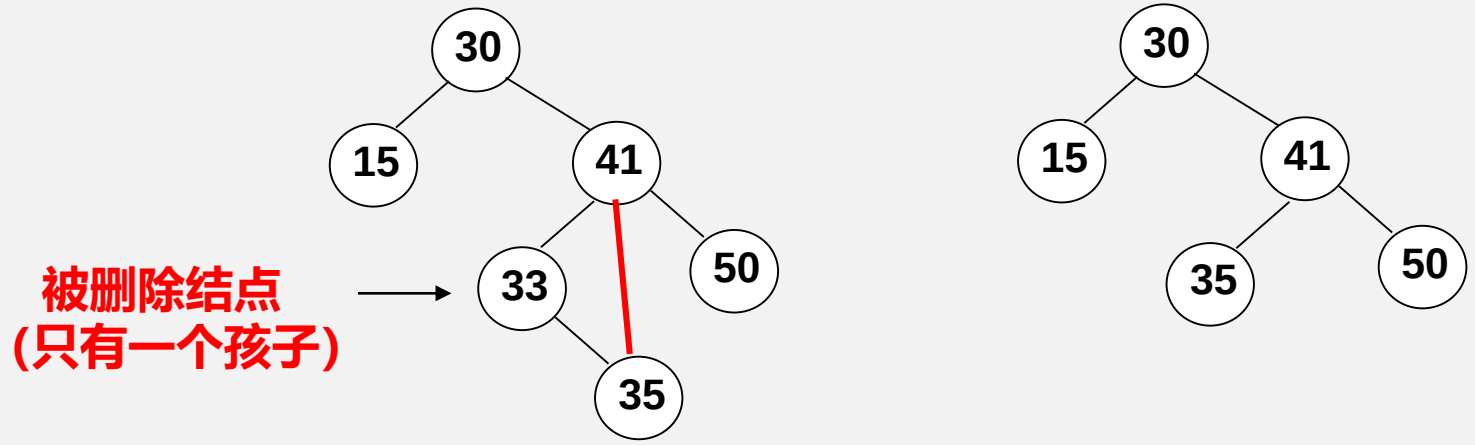
（1）删除叶结点——可以直接删除，然后再修改其父结点的指针。

【例】删除 35



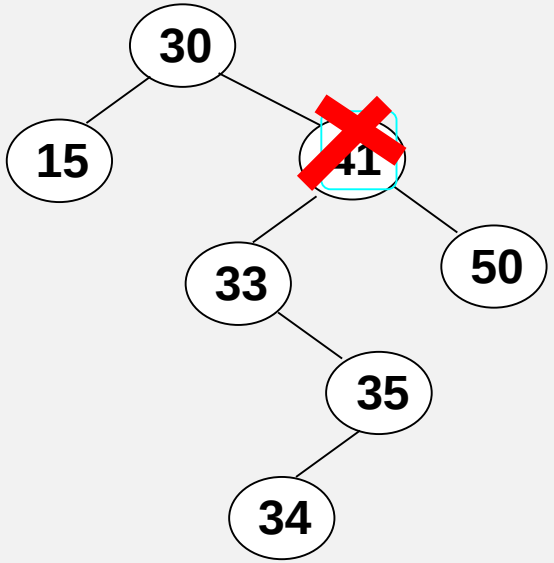
(2) 删除只有一个孩子结点——删除之前需要改变其父结点的指针，指向被删除结点的子结点。

【例】删除 33

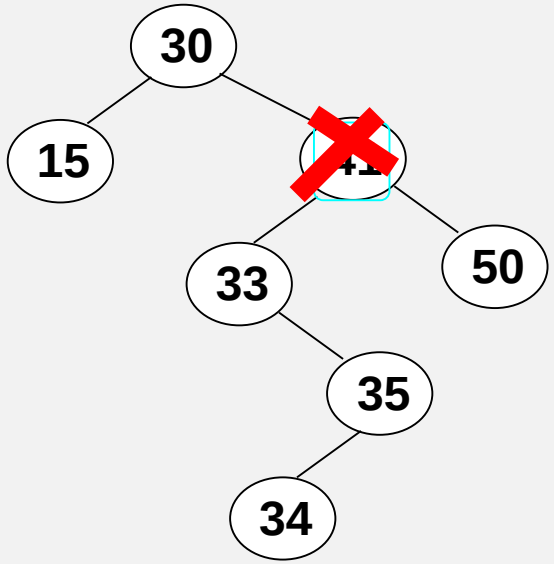


(3) 被删除的结点有左、右两棵子树——为了保持二叉搜索树的有序性，替代被删除的元素的位置可以有两种选择：一种是取其右子树中的最小元素；另一个是取其左子树中的最大元素。

【例】删除 41



(a) 取右子树中的最小元素替代



(b) 取左子树中的最大元素替代

```
BinTree Delete( ElementType X, BinTree BST )
{    Position Tmp;
     if( BST==NULL ) printf("要删除的元素未找到");
     else if( X < BST->Data )
              BST->Left = Delete( X, BST->Left); /* 左子树递归删除 */
     else if( X > BST->Data )
              BST->Right = Delete( X, BST->Right); /* 右子树递归删除 */
     else
          /*找到要删除的结点 */
          if( BST->Left && BST->Right ) { /*被删除结点有左右两个子结点 */
              Tmp = FindMin( BST->Right );
                              /*在右子树中找最小的元素填充删除结点*/
              BST->Data = Tmp->Data;
              BST->Right = Delete( BST->Data, BST->Right);
                            /*在删除结点的右子树中删除最小元素*/
          } else {
              Tmp = BST;
              if(BST->Right!=NULL ) /* 有右孩子或无子结点*/
                   BST = BST->Right;
              else /*有左孩子或无子结点*/
                   BST = BST->Left;
              free( Tmp );          }
     return BST;}
```

递归查找被删除结点

被删除结点有左右两个子结点

被删除结点只有一个子结点或无子结点

4.4.5 二叉搜索树的查找分析

二叉搜索树上查找某关键字等于给定值的结点过程，其实就是走了一条从根到该结点的路径。

比较的关键字次数

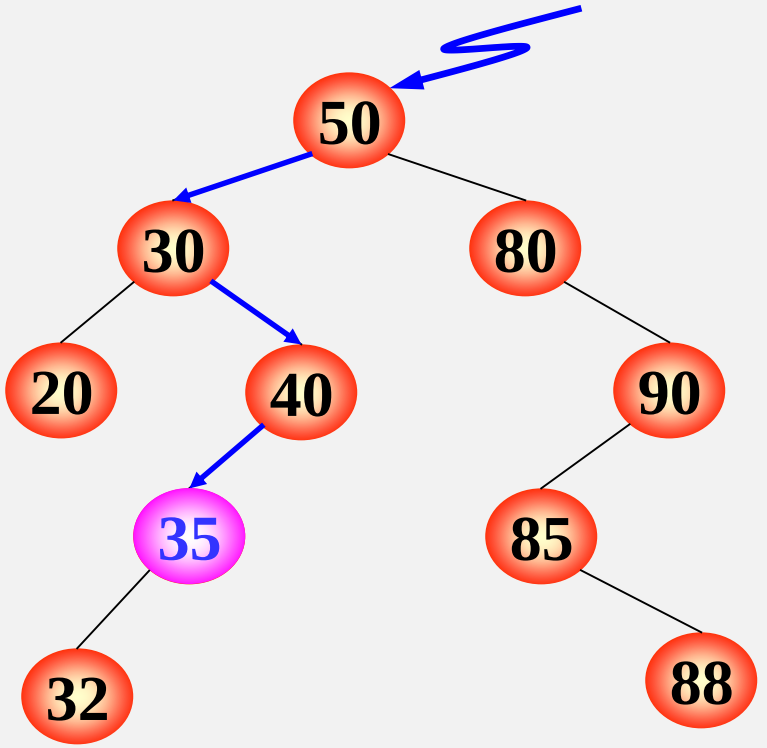
=

此结点所在层次数

最多的比较次数

=

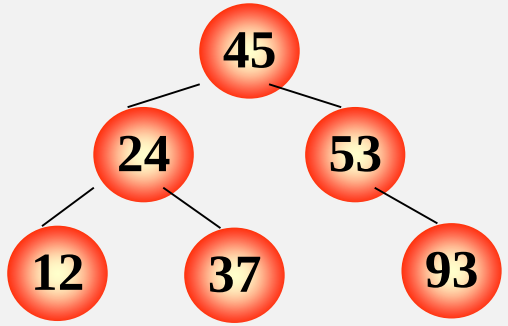
树的深度



查找关键字：35

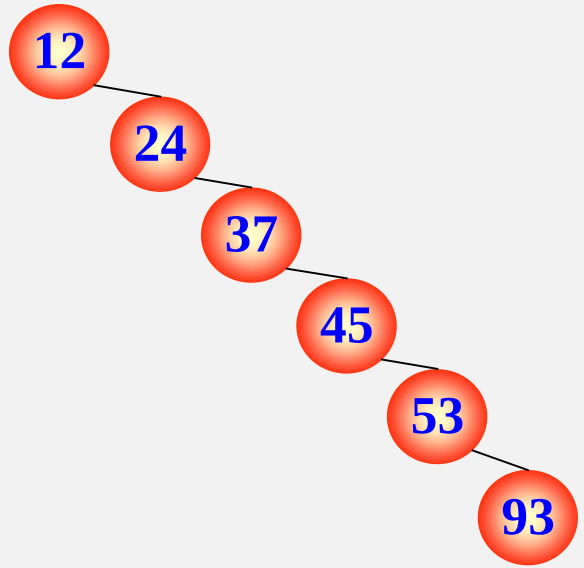
二叉搜索树的平均查找长度：

(45, 24, 53, 12, 37, 93)



$$ASL=\frac{1}{6}(1+2+2+3+3+3)=\frac{14}{6}$$

(12, 24, 37, 45, 53, 93)



$$ASL=\frac{1}{6}(1+2+3+4+5+6)=\frac{21}{6}$$

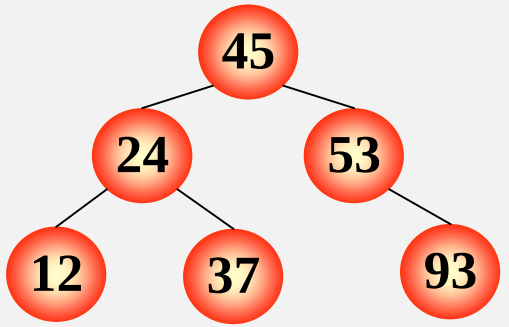
含有 n 个结点的二叉搜索树的平均查找长度和树的形态有关

最好情况： $ASL=\log_2(n+1)-1$;

树的深度为：$\lfloor \log_2 n \rfloor + 1$;

与折半查找中的判定树相同。

（形态比较均衡）。

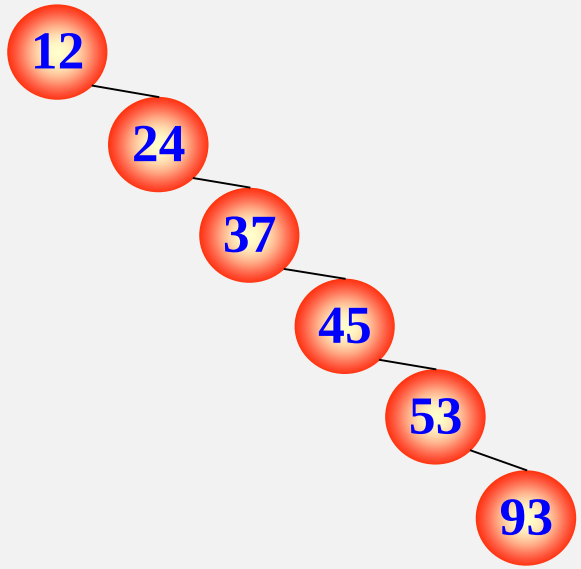


最坏情况：插入的 n 个元素从一开始就有序，

—— 变成单支树的形态！

此时树的深度为 n； $ASL = (n+1)/2$

查找效率与顺序查找情况相同。



【分析】有 n 个关键字的二叉搜索树的平均查找长度

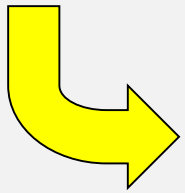
设每种树态出现概率相同，查找每个关键字也是等概率的，

则二叉搜索树的 $ASL \le 2(1+\frac{1}{n})\log n$

由此可见，在随机的情况下，二叉搜索树的 ASL 和 $\log n$ 是 等数量级的。

问题：如何提高形态不均衡的二叉搜索树的查找效率?

解决办法：做“平衡化”处理，即尽量让二叉树的形状均衡!



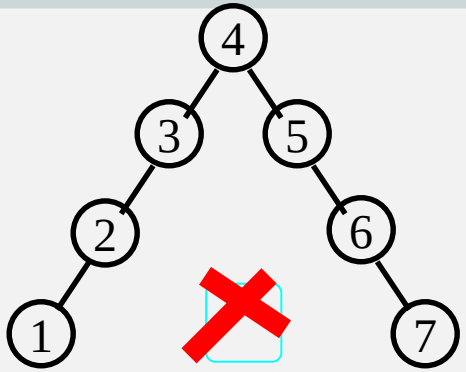
平衡二叉树

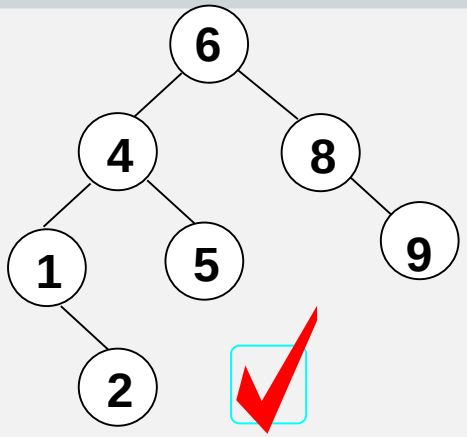
4.5.1 平衡二叉树的定义

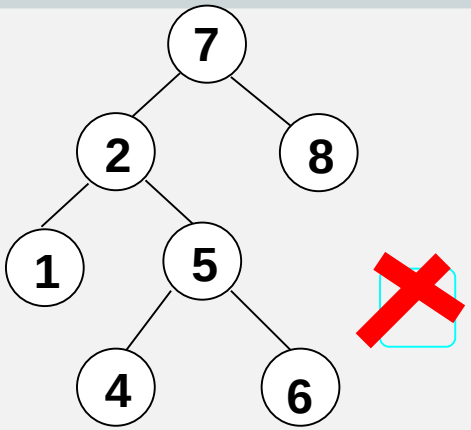
【定义】对于二叉树中任一结点T，其**平衡因子**（Balance Factor，简称BF）定义为$BF(T) = h_L - h_R$，其中h_L和h_R分别为T的左、右子树的高度。

“**平衡二叉树**（Balanced Binary Tree）”又称为“**AVL树**”。AVL树或者是一棵空树，或者是具有下列性质的非空二叉搜索树：

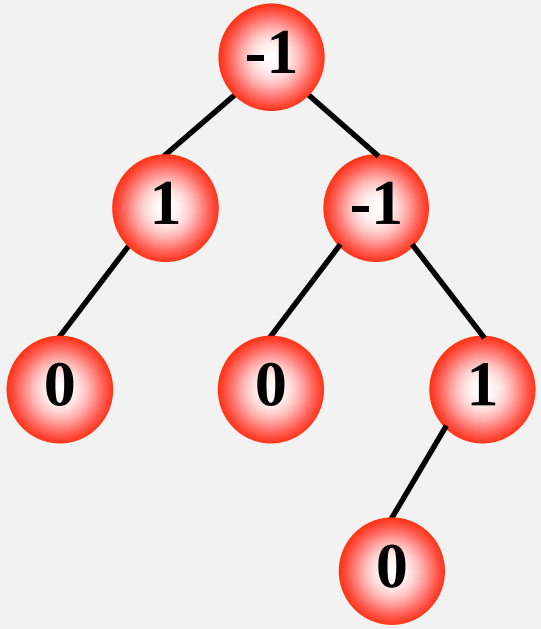
“任一结点左、右子树高度差的绝对值不超过1。”即|BF(T)|≤ 1。



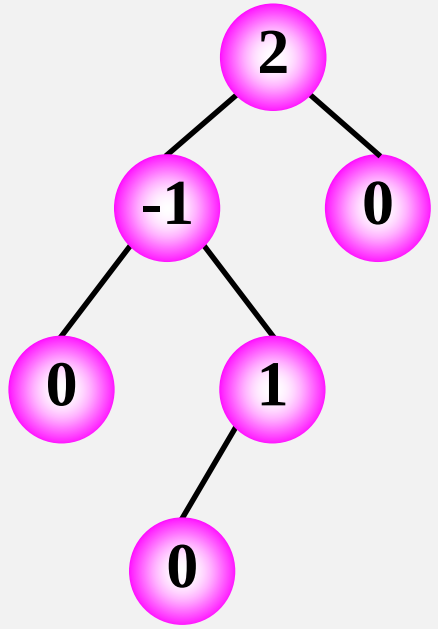




【例】判断下列二叉树是否 AVL 树?



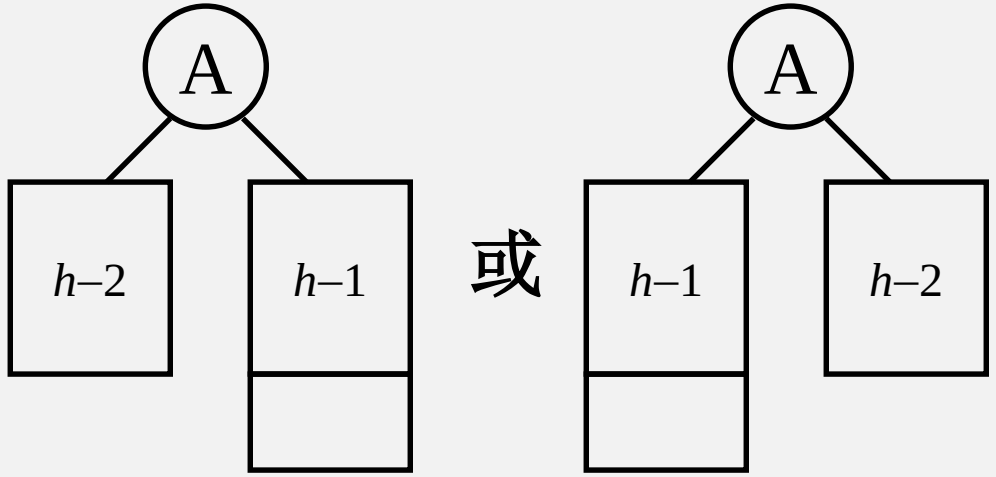
平衡二叉树



非平衡二叉树

平衡二叉树的高度为什么能达到 $\log_2 n$?

设 n_h 为高度为h的平衡二叉树的最小结点数。二叉树看起来应该是如下结构形式：



$\Rightarrow \quad n_h = n_{h-1} + n_{h-2} + 1$

斐波那契序列：

$$F_0 = 1,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1$$

设 n_h 是高度为h的平衡二叉树的最小结点数 $\Rightarrow$ $n_h = n_{h-1} + n_{h-2} + 1$

高度h	结点数n_h	斐波那契F_h
0	1	1
1	2	1
2	4	2
3	7	3
4	12	5
5	20	8
6	33	13
7	54	21
8	88	34
9	……	

$\Rightarrow$ $n_h = F_{h+2} - 1$, （对 $h \geq 0$）

$$\Rightarrow \quad n_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 1$$

$$\Rightarrow \quad h = O(\log_2 n)$$

- 给定n个结点的AVL树的最大高度为$O(\log_2 n)$
- 保证AVL树的查找时间性能为$O(\log_2 n)$

4.5.2 平衡化旋转

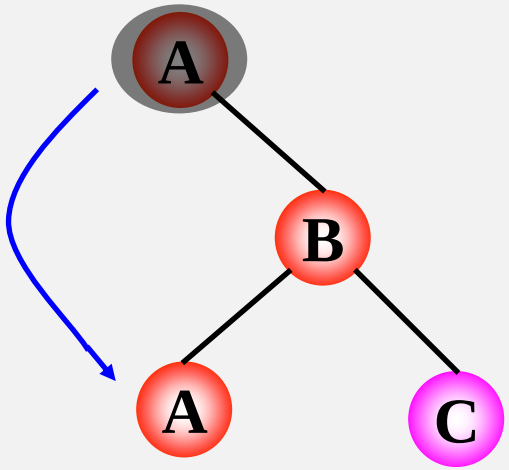
如果在一棵 AVL 树中插入一个新结点后造成失衡，则必须**重新调整树的结构**，使之恢复平衡。

我们称此调整平衡的过程为**平衡旋转**。

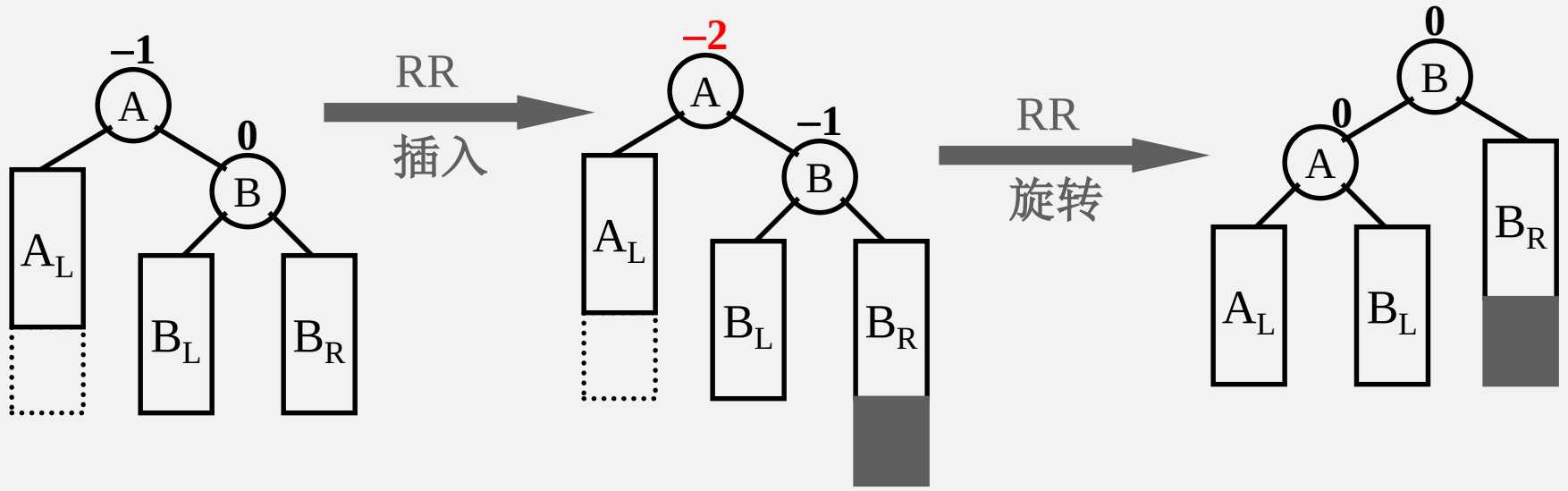
平衡旋转的类别
- LL 平衡旋转
- RR 平衡旋转
- LR 平衡旋转
- RL 平衡旋转

1) RR 平衡旋转：

若在 A 的右子树的右子树上插入结点，使 A 的平衡因子从 -1 改变为 -2，需要进行一次逆时针旋转。(以 B 为旋转轴)

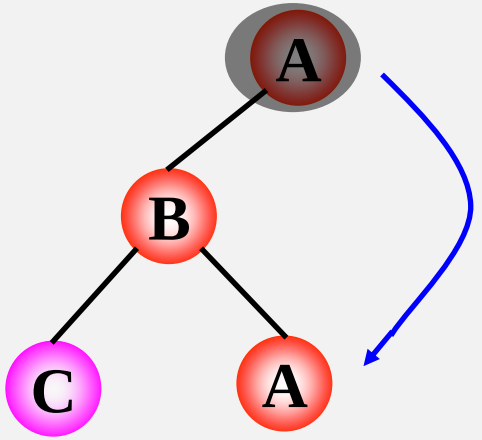


- 首个不平衡的“发现者”是A结点，它是调整起点位置。而“麻烦结点”在发现者右子树的右边，因而叫 RR 插入，需要RR 旋转（右单旋）；一般情况的调整方式如下:

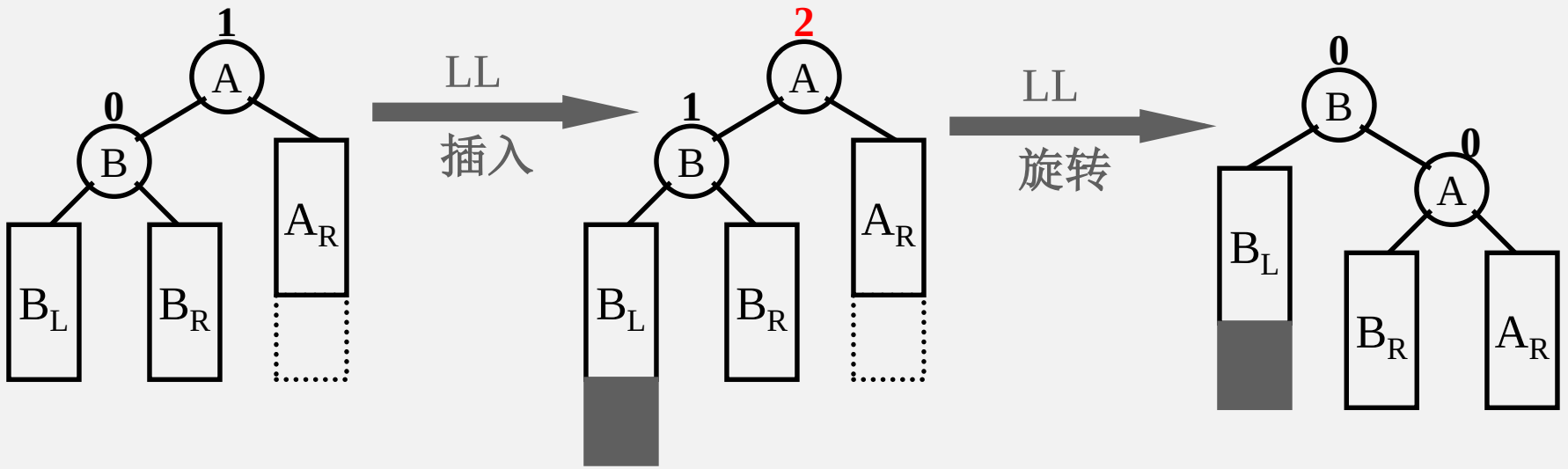


2） LL 平衡旋转：

若在 A 的左子树的左子树上插入结点，使 A 的平衡因子从 1 增加至 2，需要进行一次顺时针旋转。(以 B 为旋转轴)

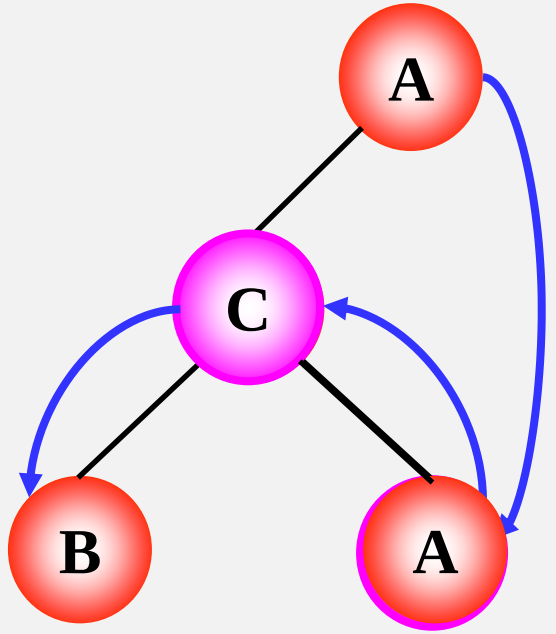


- 首个“发现者”是A；“麻烦结点”在发现者左子树的左边，因而叫 LL 插入；一般情况的调整方式如下:

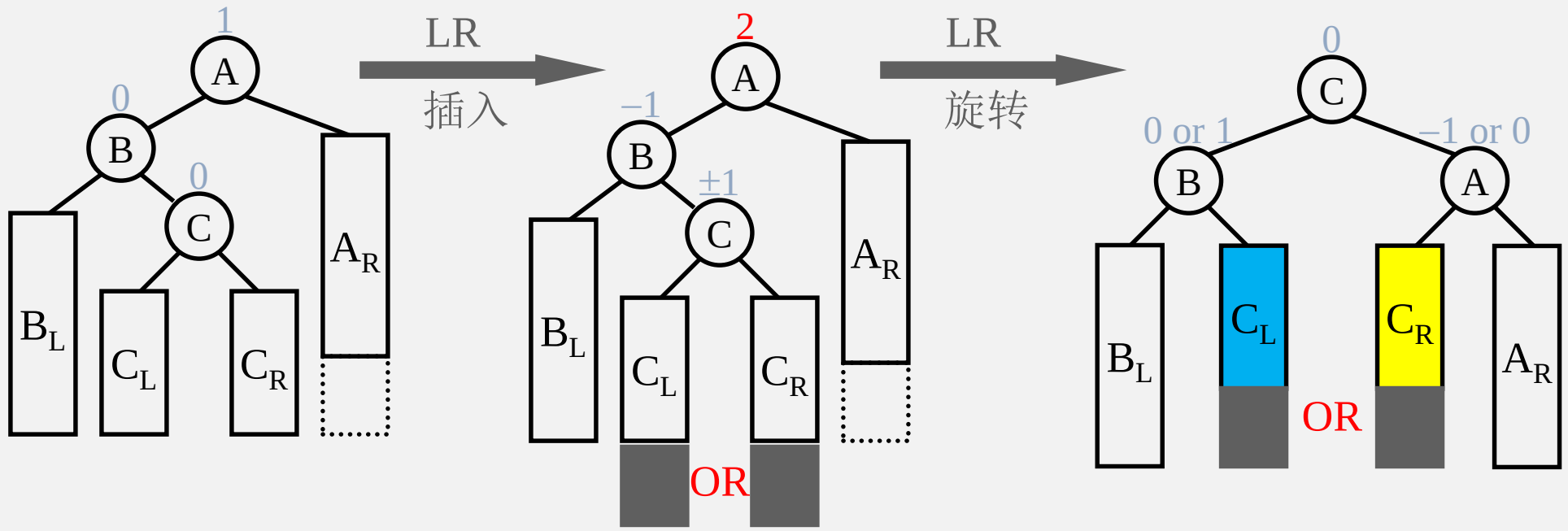


3) LR 平衡旋转：

若在 A 的左子树的右子树上插入 结点，使 A 的平衡因子从 1 增加 至 2， 需要先进行逆时针旋转， 再顺时针旋转。 (以插入的结点 B 为旋转轴)



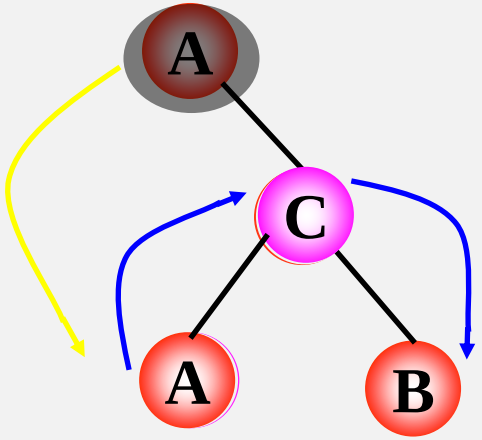
- 首个“发现者”是A，“麻烦结点”在左子树的右边，因而叫 LR 插入；一般情况的调整如下:



4) RL 平衡旋转：

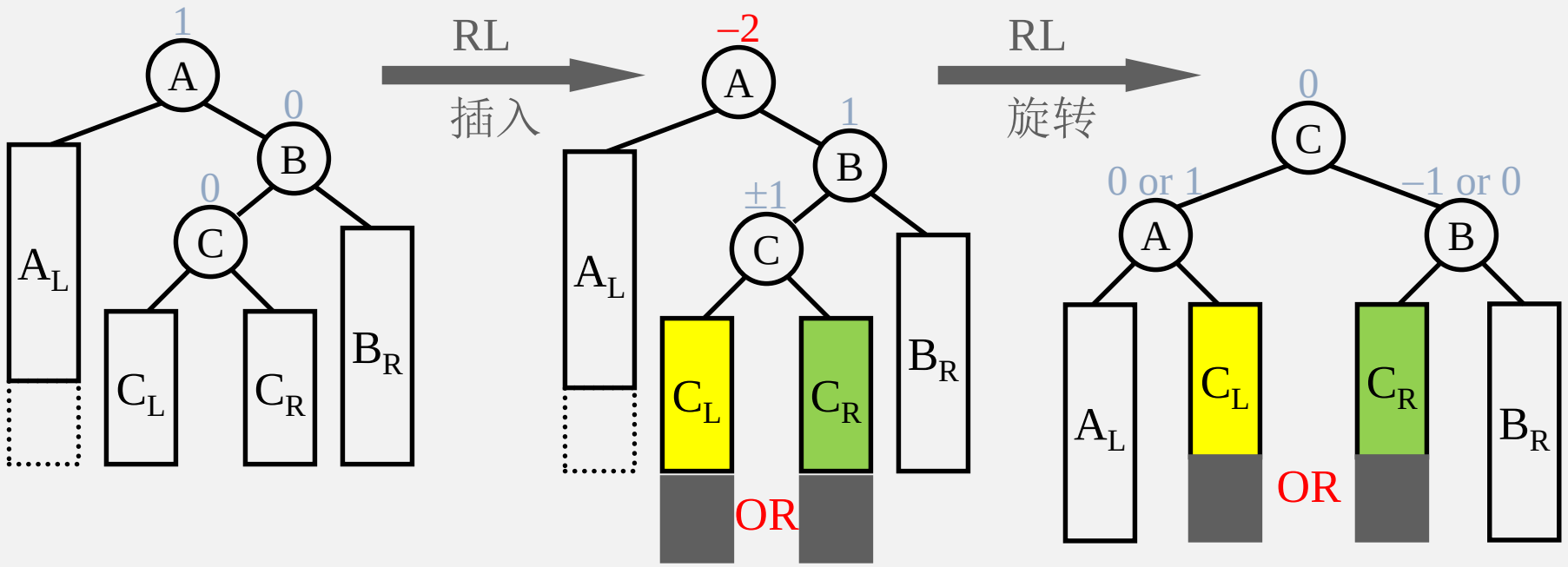
若在 A 的右子树的左子树上插入 结点，使 A 的平衡因子从-1 改变 为-2，需要先进行顺时针旋转，再逆时针旋转。

(以插入的结点 B 为旋转轴)



调整必须保证二叉排序树的特性不变

➢ 一般情况调整如下:


1
A
0
B
0
C
A_L
C_L
C_R
B_R
RL
插入
−2
A
1
B
±1
C
A_L
C_L
C_R
B_R
OR
RL
旋转
0
C
0 or 1
−1 or 0
A
B
A_L
C_L
C_R
B_R
OR

4.5.3 平衡二叉树的插入

- **平衡二叉树插入新结点**

  1)若平衡二叉树为空树，则新插入的结点为根结点；

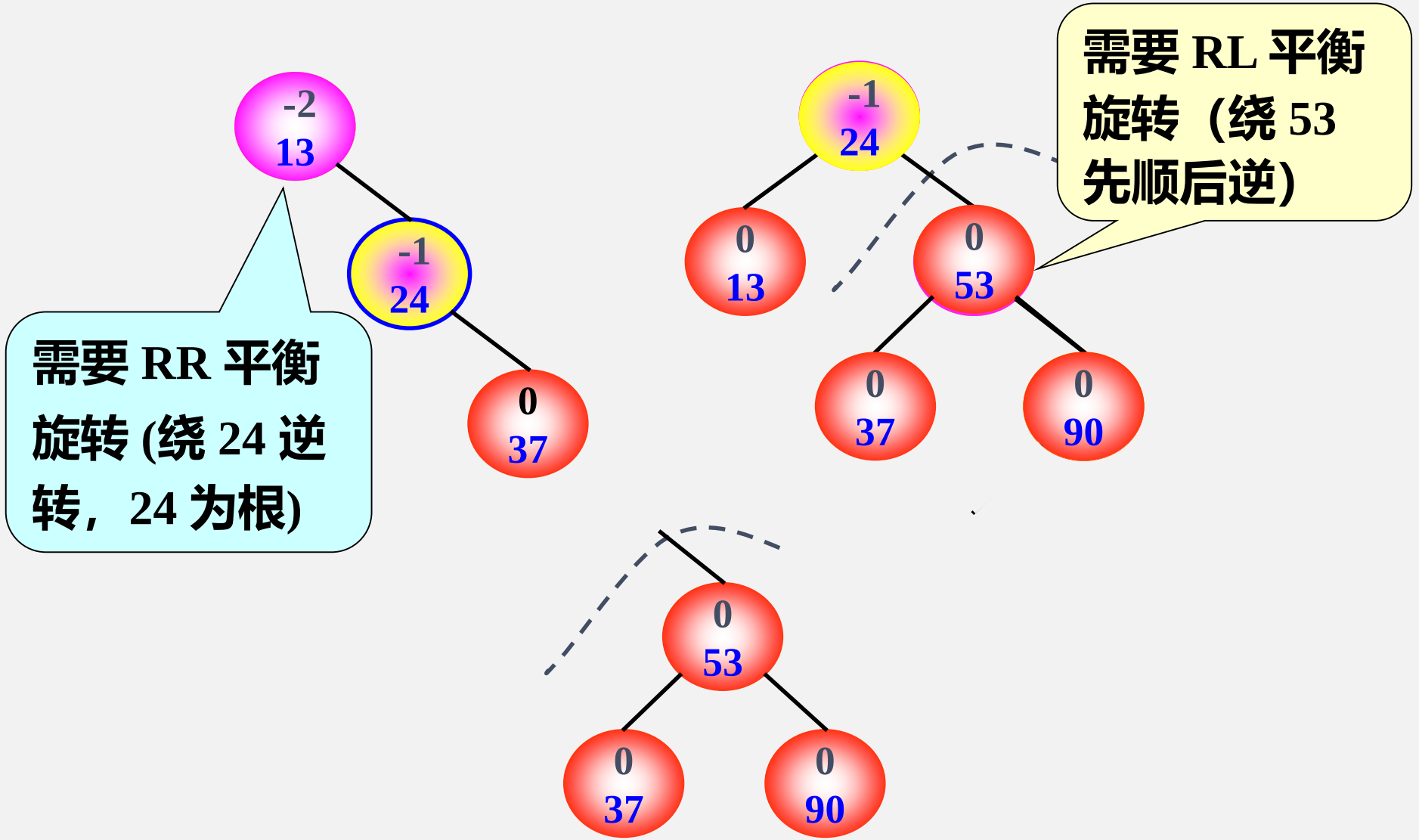
  2)若平衡二叉树非空，则新插入的结点必为一个新的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。

  ①若X<BST->data：结点X插入到平衡二叉树的左子树中；

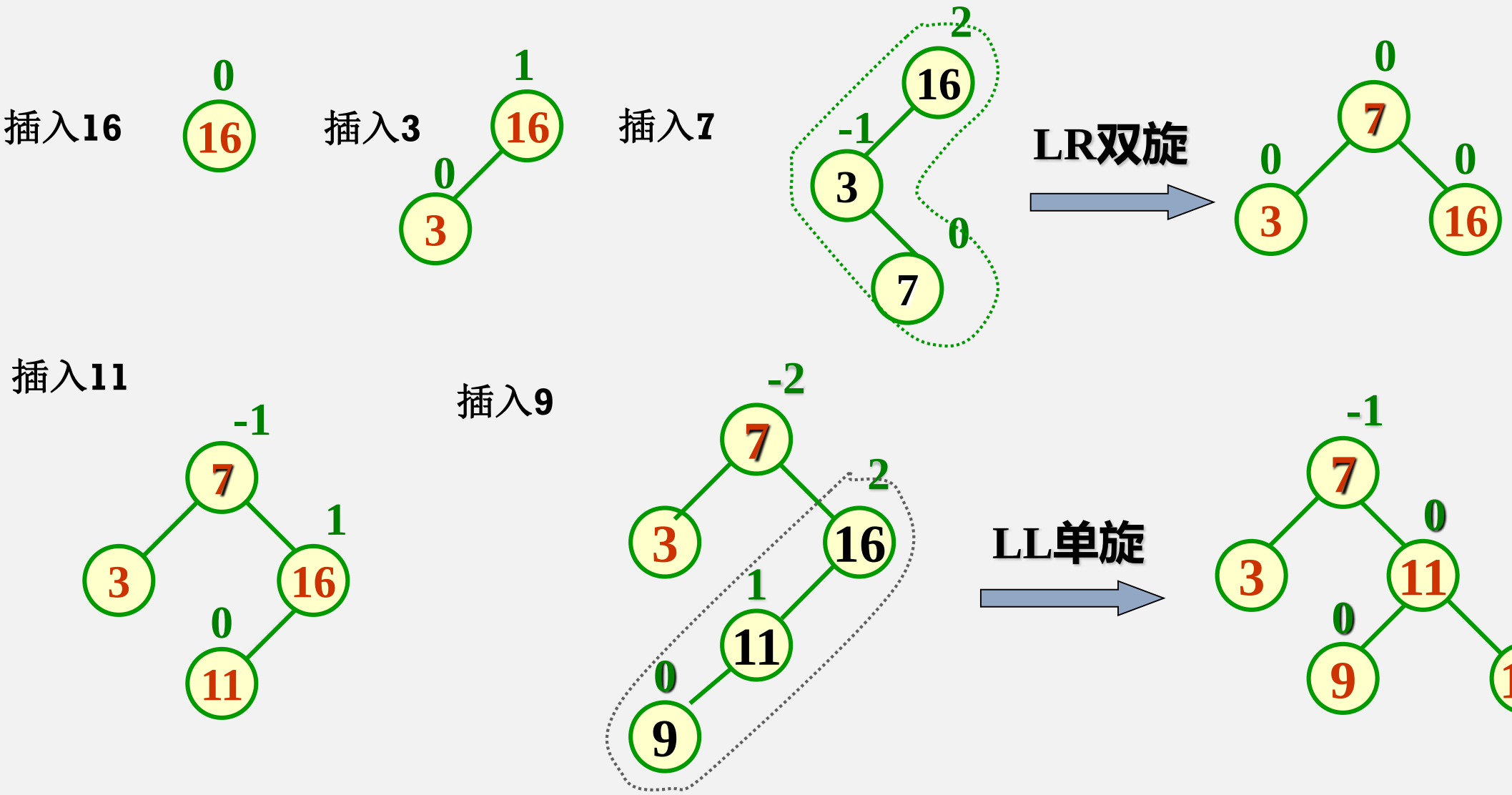
  ② 若X>BST->data：结点X插入到平衡二叉树的右子树中。

- **计算平衡因子，进行平衡化调整**

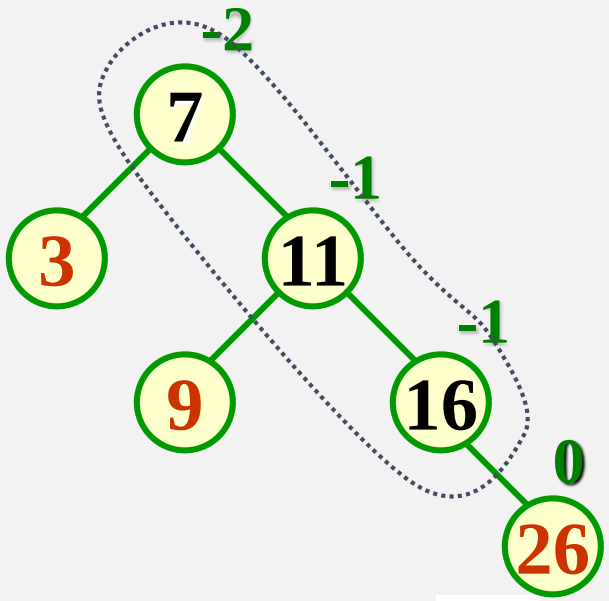
【例】请将序列(13, 24, 37, 90, 53)构成一棵平衡二叉排序树



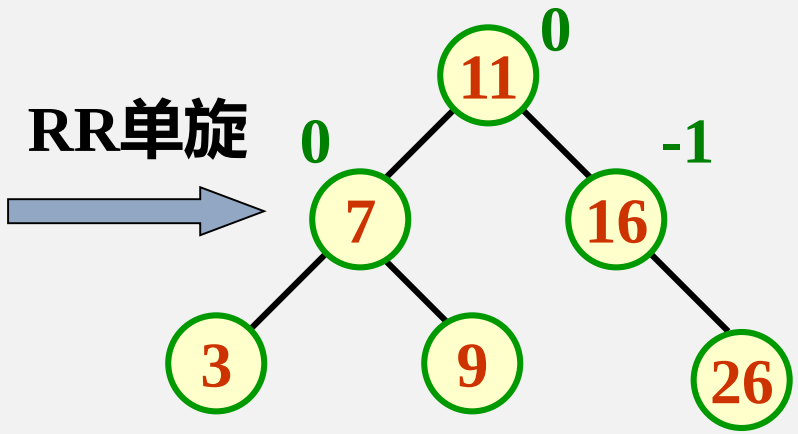
【例】输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，构成一棵平衡二叉排序树。



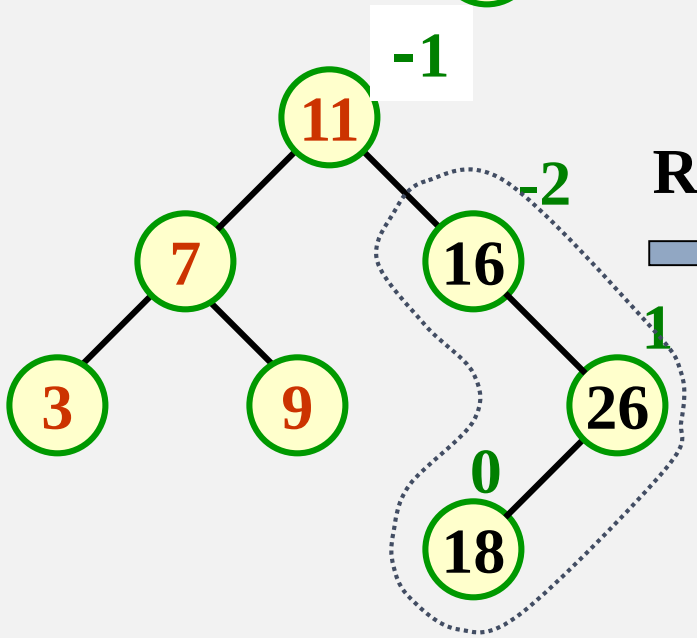
插入26



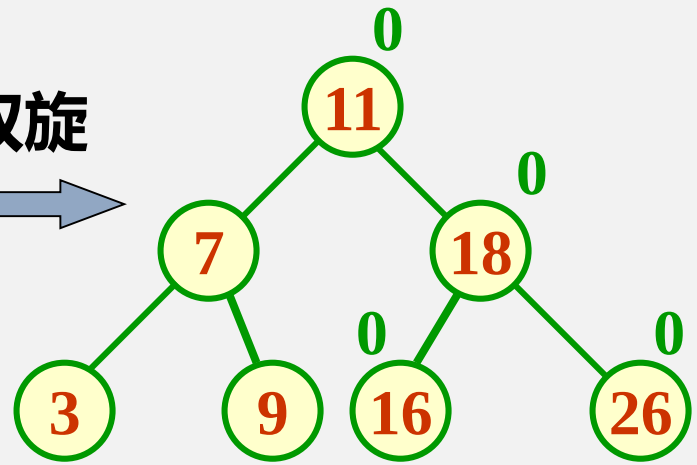
RR单旋



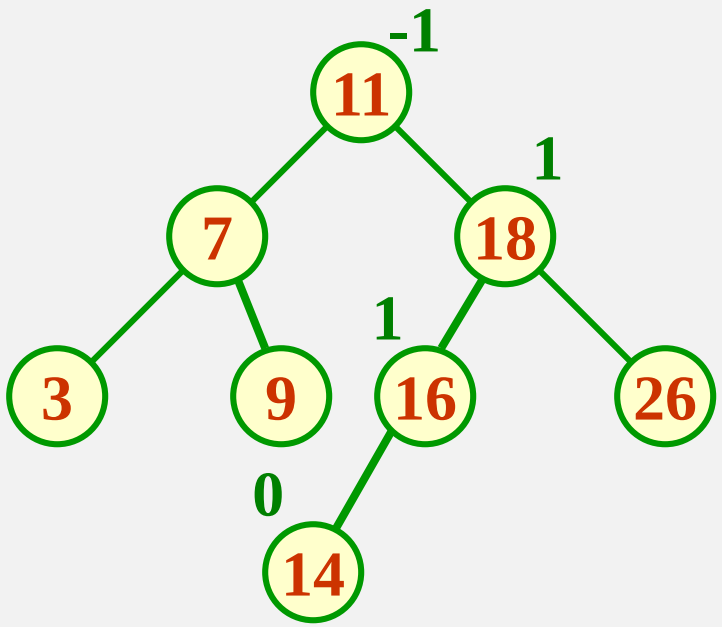
插入18



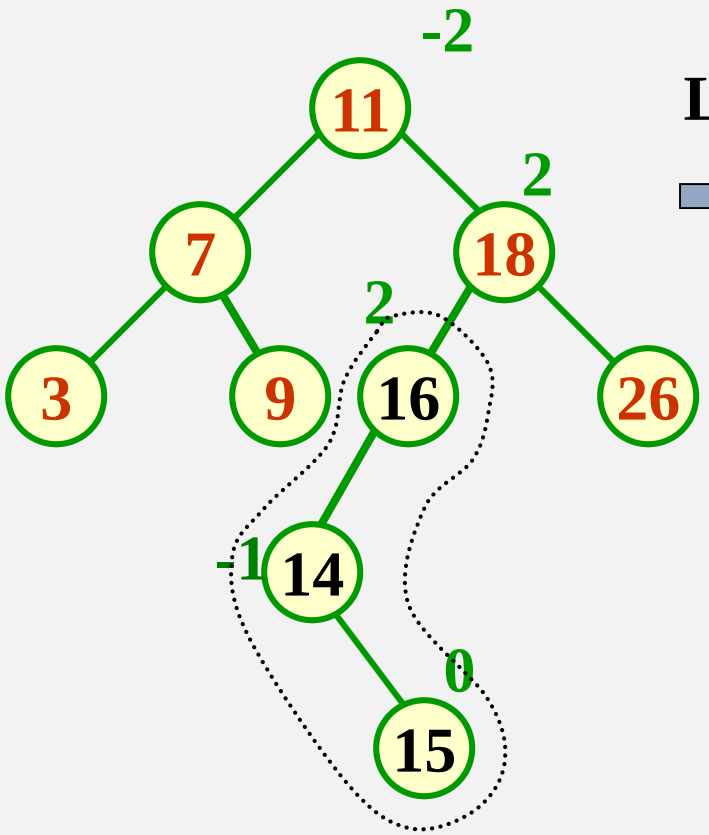
RL双旋



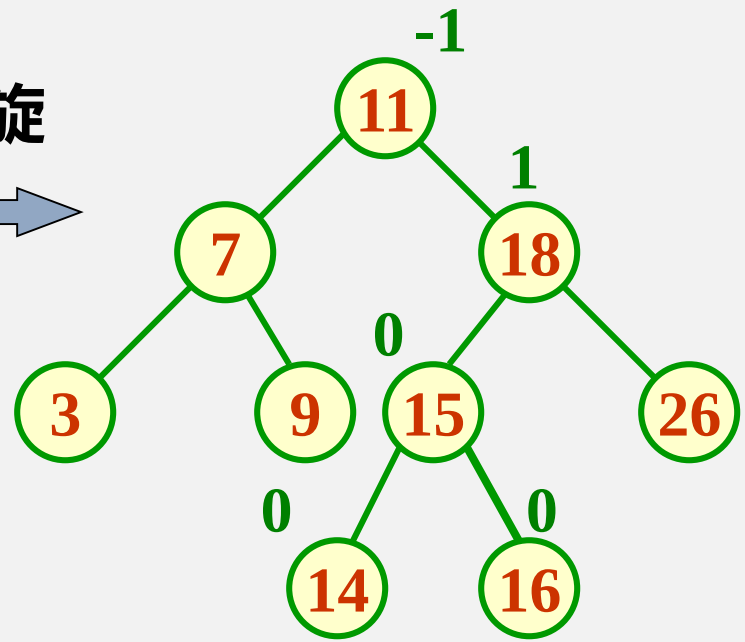
插入14



插入15



LR双旋



4.5.4 平衡二叉树的删除

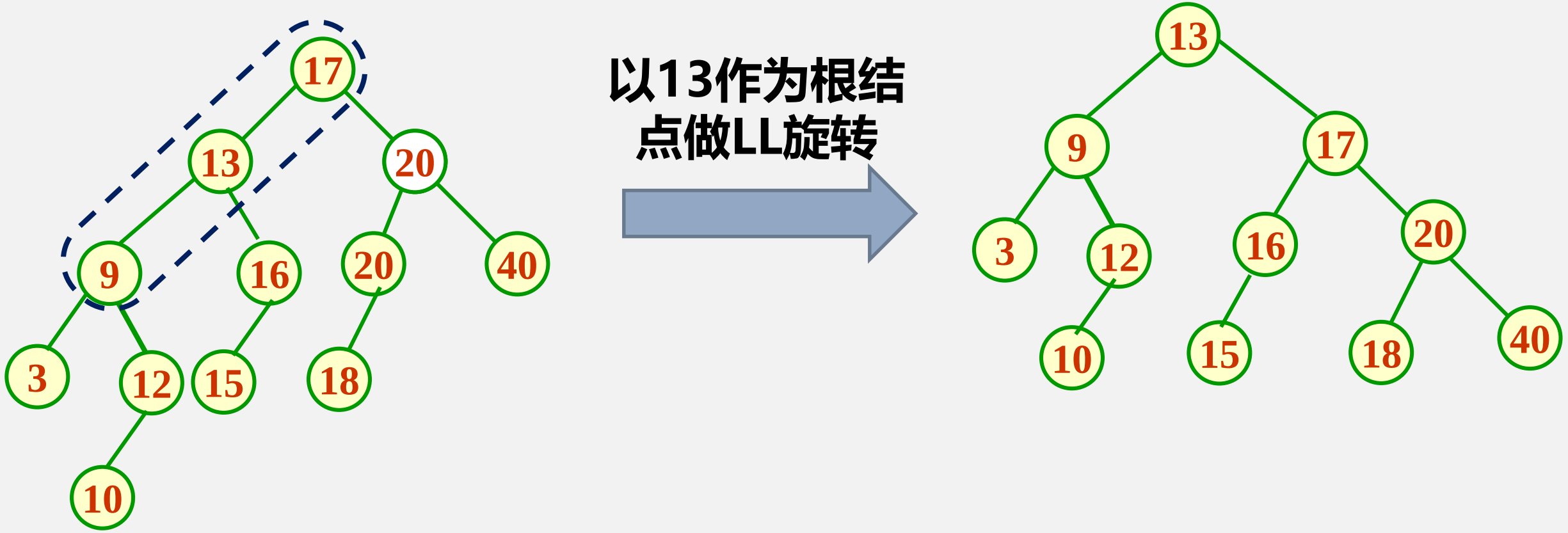
1. 如果被删结点x最多只有一个子女
   - 该节点没有左、右孩子，直接删除该节点=NULL。
   - 该节点只有左孩子，将其左孩子的值复制到该节点（仅移动值即可），直接删除其左孩子。
   - 该节点只有右孩子，将其右孩子的值复制到该节点（仅移动值即可），直接删除其右孩子。

4.5.4 平衡二叉树的删除

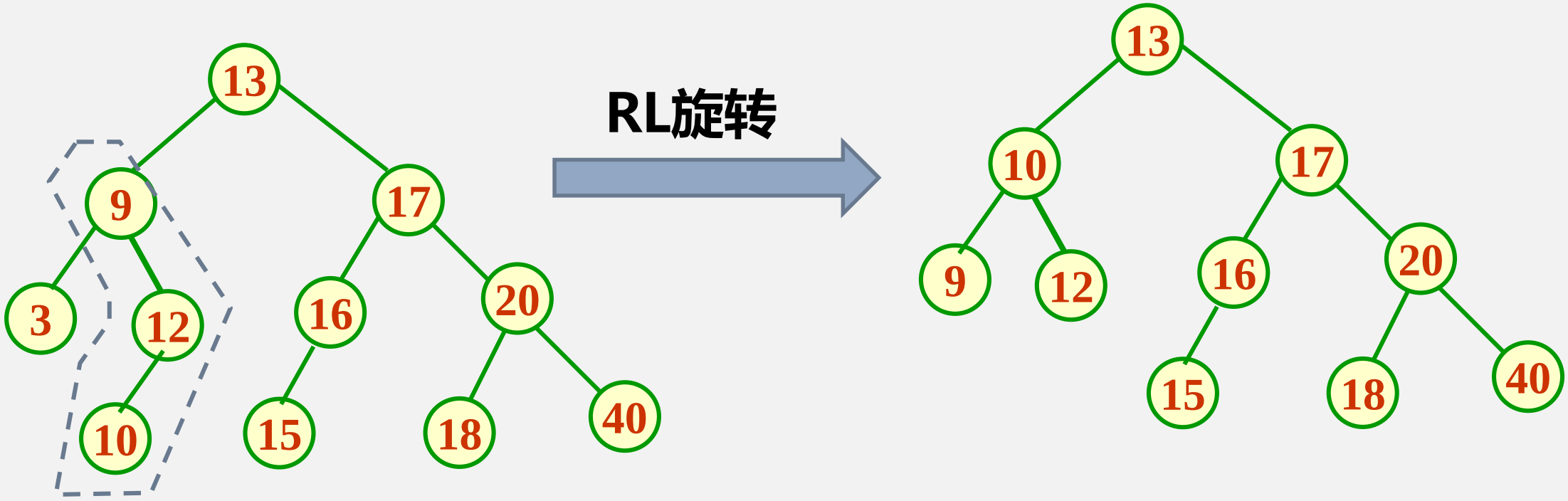
2. 如果被删结点x有两个子女:
   - 查找结点x在中序次序下的直接前驱结点y (同样可以找直接后继)。
   - 把结点y的内容复制给结点x，把结点y当作被删结点x。
   - 结点y最多有一个子女，使用1.给出的方法进行删除。

- 在执行结点x的删除之后，需要从结点x开始，沿通向根的路径反向追踪高度的变化对路径上各个结点平衡因子的影响。

例如：平衡二叉树中删除结点22



例如：平衡二叉树中删除结点3



4.6 树的应用

- 堆及其操作
- 哈夫曼树
- 集合的应用

4.6.1 堆

【例】假设现在10 亿个搜索关键词，如何能快速获取到热门榜 Top 10 的搜索关键词呢?

	数组	有序数组（降序）	链表	有序链表（降序）
插入	将新元素存入数组末端，时间复杂度O(1)	需要O(log n)时间找合适的插入位置，移动的时间复杂度O(N)	新元素插入链头 时间复杂度O(1)	遍历全部元素找到插入位置，时间复杂度O(N)
删除	遍历全部元素，位置移动时间复杂度O(N)	删除最后一个元素 时间复杂度O(1)	遍历全部元素 时间复杂度O(N)	删除最后一个元素 时间复杂度O(1)
搜索 (排名前10)	必须遍历全部元素 时间复杂度O(N)	直接读取前10个元素 时间复杂度O(1)	必须遍历全部元素 时间复杂度O(N)	直接从链头读取元素 时间复杂度O(1)

利用树型结构实现代价为O(log n)的搜索

➢ 1、堆的定义

【定义】n 个元素的序列 $(k_1, k_2, \ldots, k_n)$，当且仅当满足下列关系时，称之为**堆**。

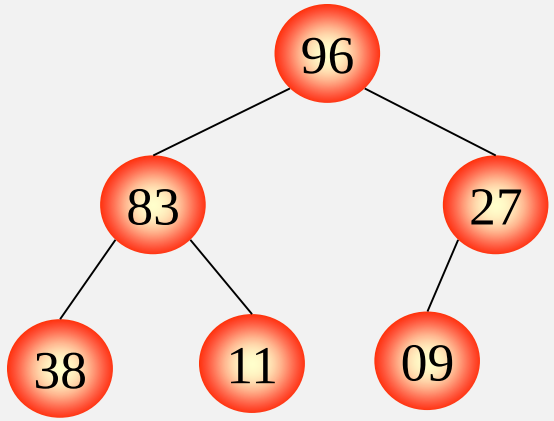
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad (i = 1, 2, \ldots, \lfloor n/2 \rfloor)$$

若按顺序组成一棵完全二叉树，则

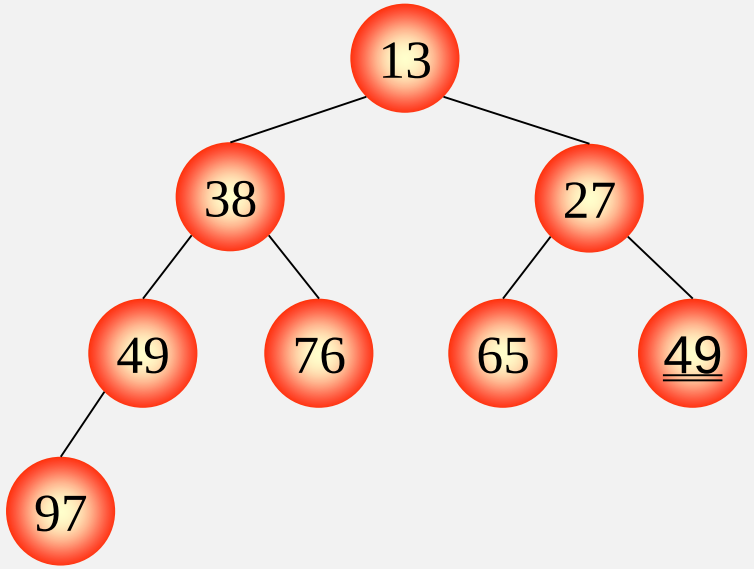
(1) **最小堆**：二叉树的所有根结点值小于或等于左右孩子的值

(2) **最大堆**：二叉树的所有根结点值大于或等于左右孩子的值。

例1： (96, 83, 27, 38, 11, 09)



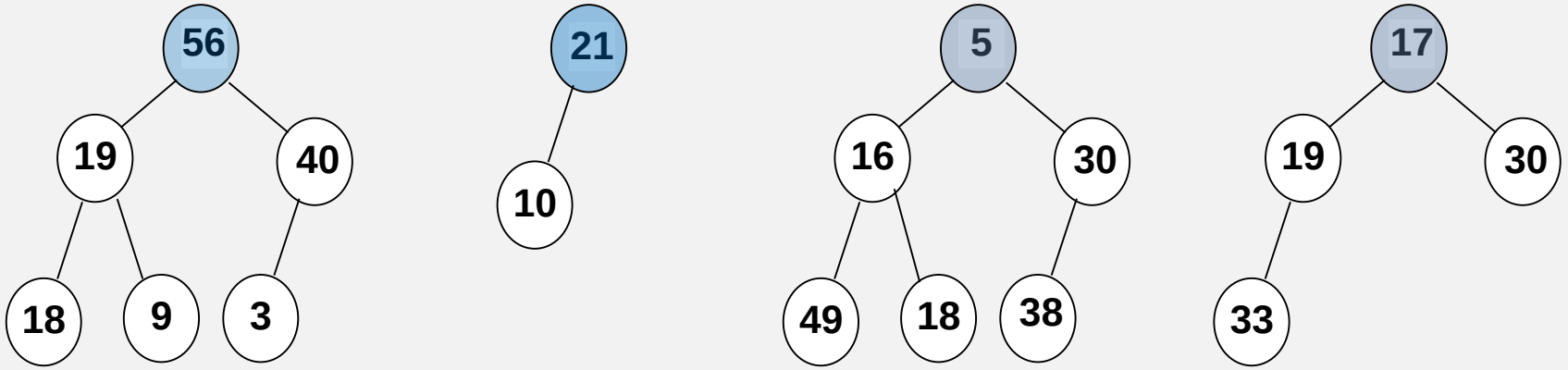
例2： (13, 38, 27, 49, 76, 65, 49, 97)



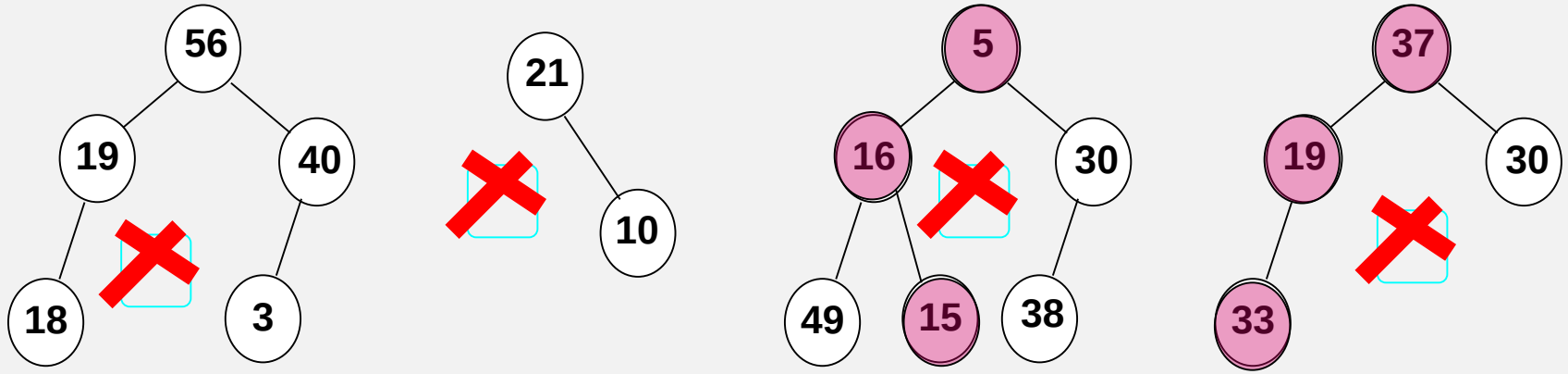
可将堆序列看成**完全二叉树**，则：

- k_{2i} 是 k_i 的左孩子； k_{2i+1} 是 k_i 的右孩子。
- 所有非终端结点的值均不大（小）于其左右孩子结点的值。
- 堆顶元素必为序列中 n 个元素的最小值或最大值。

【例】最大堆和最小堆



【例】不是堆



➢ 2、最大堆的抽象数据类型

类 型 名 称：最大堆（MaxHeap）

数据对象集：一个有N>0个元素的最大堆H是一棵完全二叉树，每个结点上的元素值不小于其子结点元素的值。

操　作　集：

- **MaxHeap Create(int MaxSize)：创建一个空的最大堆。**
- **Boolean IsFull(MaxHeap H)：判断最大堆H是否已满。**
- **Insert(MaxHeap H, ElementType item)：将元素item插入最大堆H。**
- **Boolean IsEmpty(MaxHeap H)：判断最大堆H是否为空。**
- **ElementType DeleteMax(MaxHeap H)：返回H中最大元素**

3、最大堆的插入

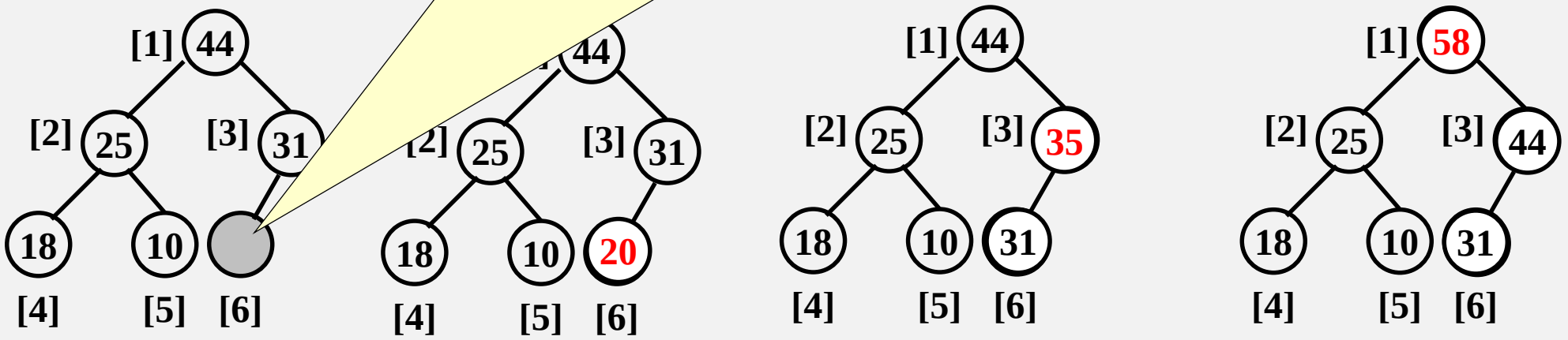
```
typedef  struct  HeapStruct  *MaxHeap;
struct  HeapStruct {
        ElementType *Elements;   /* 存储堆元素的数组 */
        int Size;                /* 堆的当前元素个数 */
        int Capacity;            /* 堆的最大容量     */
};
```

```
MaxHeap Create( int MaxSize )
{          /* 创建容量为MaxSize的空的最大堆 */
MaxHeap H = malloc( sizeof( struct HeapStruct ) );
H->Elements = malloc( (MaxSize+1) * sizeof(ElementType));
H->Size = 0;
H->Capacity = MaxSize;
H->Elements[0] = MaxData;
    /* 定义“哨兵”为大于堆中所有可能元素的值，便于以后更快操作 */
return H;
}
```

MaxData换成小于堆中所有元素的MinData，同样适用于创建**最小堆**。

【例】最大堆的插入新结点的位置必须在这里。

从堆的完全二叉树结构要求来说，



Case 1 : new_item = 20 20 < 31 ✓

Case 2 : new_item = 35 35 > 31 35 < 44 ✓

Case 3 : new_item = 58 58 > 31 58 > 44 58 < MaxData ✓

算法可以概括为：从新增的最后一个结点的父结点开始，用要插入元素**向下过滤上层结点**（相当于要插入的元素**向上渗透**）。

```
void Insert( MaxHeap H, ElementType item )
{ /* 将元素item 插入最大堆H，其中H->Elements[0]已经定义为哨兵 */
    int i;
    if ( IsFull(H) ) {
        printf("最大堆已满");
        return;
    }
  i = ++H->Size; /* i指向插入后堆中的最后一个元素的位置 */
  for ( ; H->Elements[i/2] < item; i/=2 )
        H->Elements[i] = H->Elements[i/2]; /* 向下过滤结点 */
  H->Elements[i] = item; /* 将item 插入 */
}
```

由于H->Element[0]是**哨兵**元素，当它**不小于**堆中的最大元素，控制顺环结束

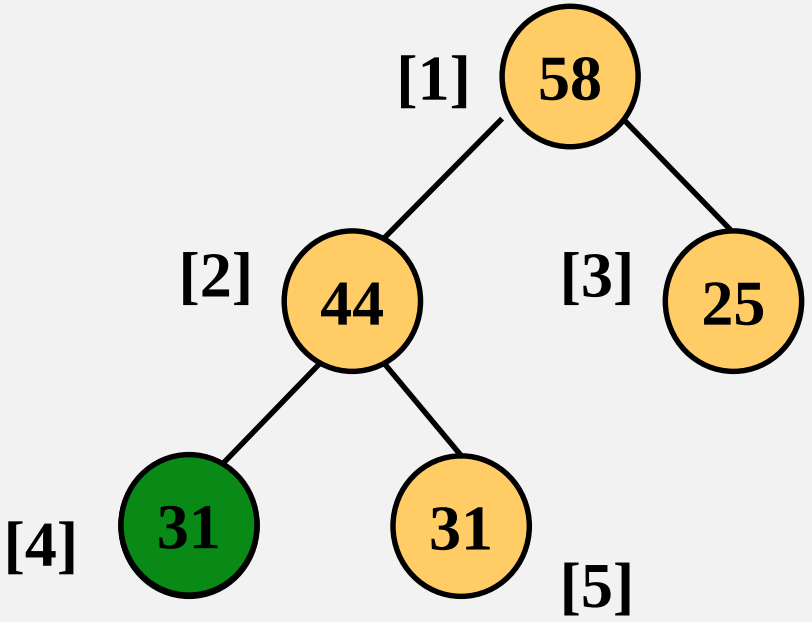
父结点数据向下过滤，比交换数据要快

算法的时间复杂度为：$T(N) = O(\log N)$

➢ 4、最大堆的删除

- 取出根结点（最大值）元素，同时删除堆的一个结点。
  - 输出堆顶元素后，以堆中最后一个元素**替代**之；
  - 然后将根结点值与左、右子树的根结点值进行比较，并与其中小（大）者进行**赋值**；
  - 重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“**筛选**”。

【例】删除最大堆的堆顶元素58



① 把最后一个元素 31 移至根

② 找出31的较大的孩子，筛选

44 > 25

31 < 35

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

```
ElementType DeleteMax( MaxHeap H )
{   /* 从最大堆H中取出键值为最大的元素，并删除一个结点 */
    int Parent, Child;
    ElementType MaxItem, temp;
    if ( IsEmpty(H) ) {
        printf("最大堆已为空");
        return;
    }
    MaxItem = H->Elements[1]; /* 取出根结点最大值 */
    /* 用最大堆中最后一个元素从根结点开始向上过滤下层结点 */
    temp = H->Elements[H->Size--];
    for( Parent=1; Parent*2<=H->Size; Parent=Child ) {
        Child = Parent * 2;
        if( (Child!= H->Size) &&
            (H->Elements[Child] < H->Elements[Child+1]) )
           Child++;  /* Child指向左右子结点的较大者 */
        if( temp >= H->Elements[Child] ) break;
        else  /* 移动temp元素到下一层 */
           H->Elements[Parent] = H->Elements[Child];
    }
    H->Elements[Parent] = temp;
    return MaxItem;
}
```

➢ 5、最大堆的建立

“建立最大堆”是指如何将已经存在的N个元素按最大堆的要求存放在一个一维数组中。

- 通过最大堆的插入操作，将N个元素相继插入到一个初始为空的堆中去，其时间代价最大为$O(N\ logN)$。

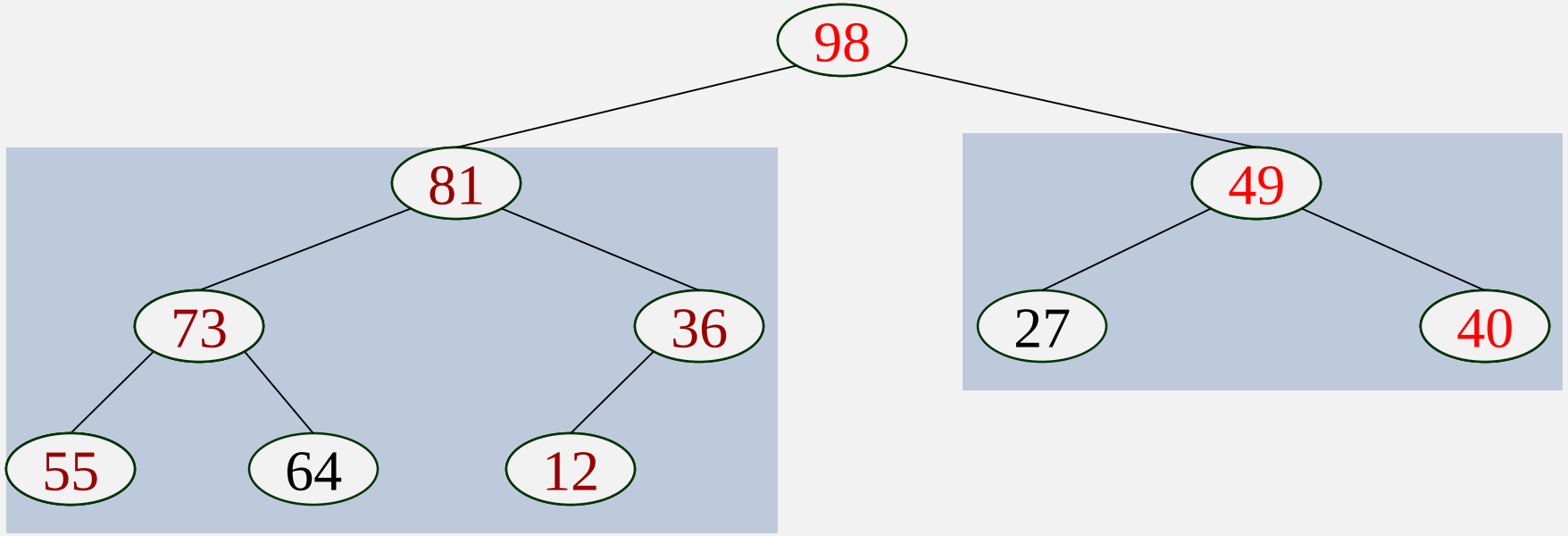
- 在线性时间复杂度下建立最大堆。具体分两步进行：

  ①将N个元素按输入顺序存入二叉树中，这一步只要求满足完全二叉树的结构特性，而不管其有序性。

  ②调整各结点元素，以满足最大堆的有序特性。

从无序序列的第⌊*n*/2⌋个元素（即无序序列对应的完全二叉树的最后一个内部结点）起，至第一个元素止，进行反复筛选。

【例】将如下待排序的序列构建大根堆。



现在，左/右子树都已经调整为堆，最后只要调整根结点，使整个二叉树是个“堆”即可。 建堆是一个从下往上进行“筛选”的过程。

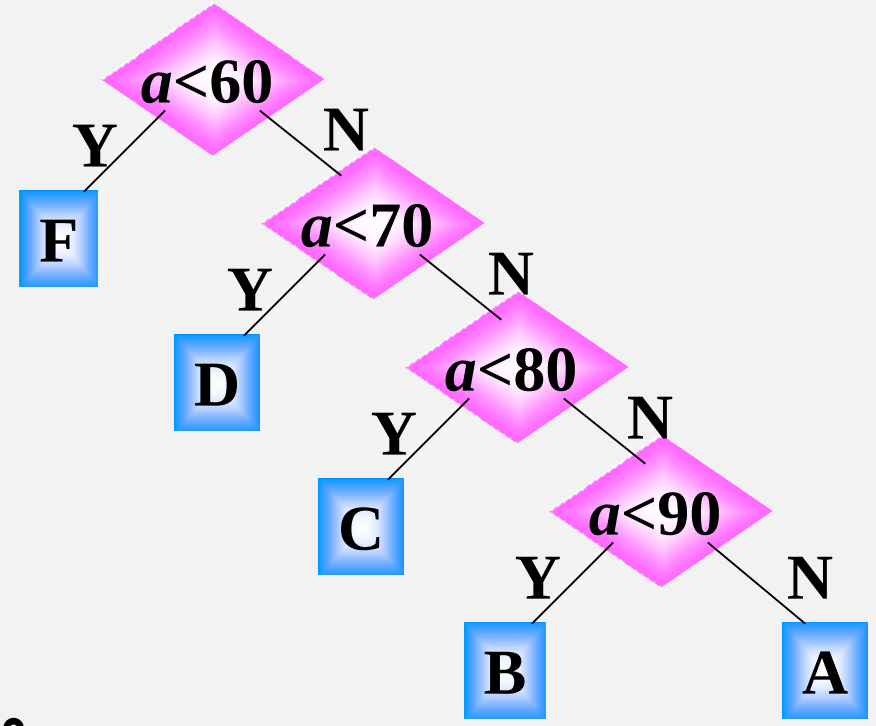
```
MaxHeap BuildMaxHeap( MaxHeap H )
{    /* 这里假设所有H->Size个元素已经存在H->Elements[]中      */
     /* 本函数将H->Elements[]中的元素调整，使满足最大堆的有序性 */
   int Parent, Child, i;
   ElementType temp;
   for( i = H->Size/2; i>0; i-- ){ /*从最后一个结点的父结点开始 */
      temp = H->Elements[i];
      for( Parent=i; Parent*2<=H->Size; Parent=Child ) {
         /* 向下过滤 */
         Child = Parent * 2;
         if( (Child!= H->Size) &&
              (H->Elements[Child] < H->Elements[Child+1]) )
             Child++;  /* Child指向左右子结点的较大者 */
         if( temp >= H->Elements[Child] ) break;
          else  /* 移动temp元素到下一层 */
             H->Elements[Parent] = H->Elements[Child];
       } /* 结束内部for循环对以H->Elements[i]为根的子树的调整 */
       H->Elements[Parent] = temp;
   }
   return H;
}
```

4.6.2 哈夫曼（Huffman）树及其应用

【例】编写程序将学生的百分制成绩转换为五分制成绩：≥90 分: A，80～89分: B，70～79分: C，60～69分: D，＜60分: F。

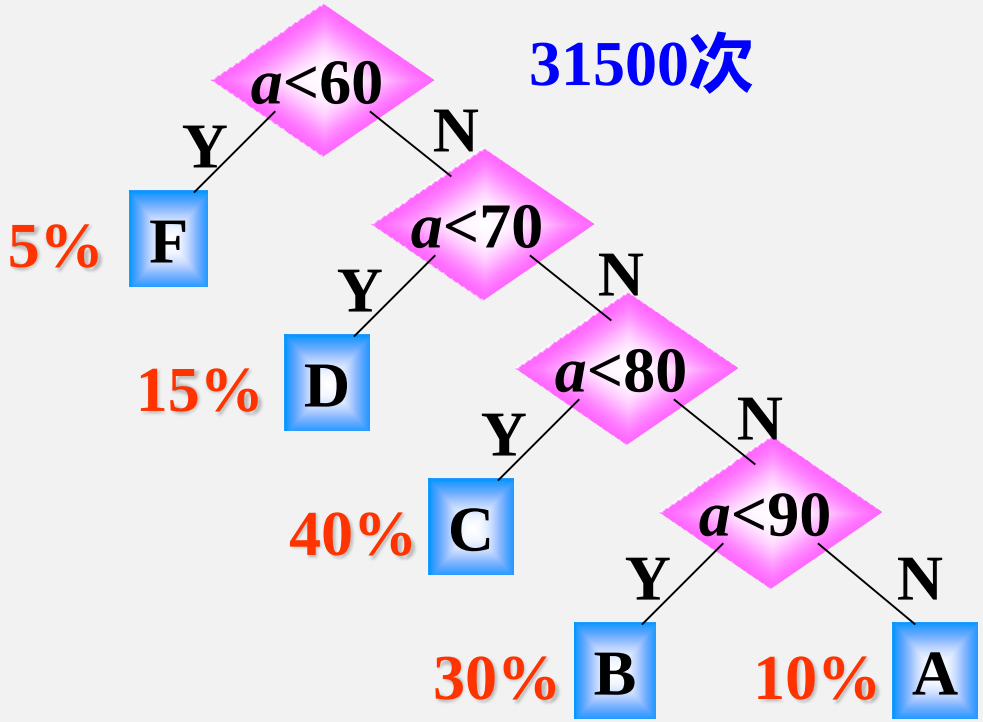
转换程序可用条件语句实现：

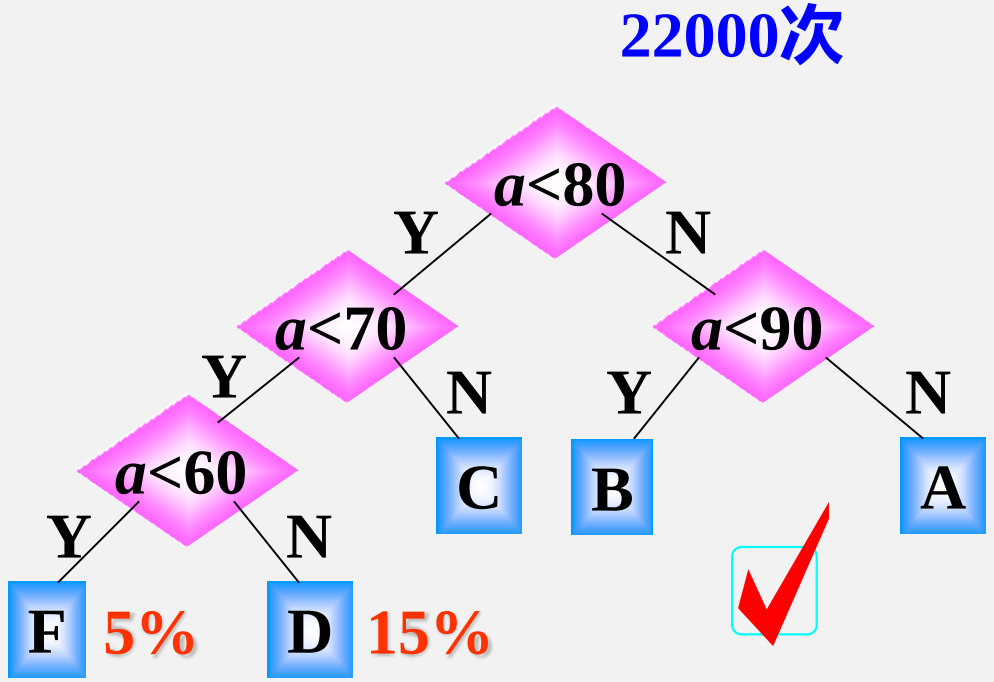
```
if (a<60) b=='F';
else if (a<70) b=='D';
        else if (a<80) b=='C';
                else if (a<90) b=='B';
                        else b=='A';
```



判定树： 用于描述分类过程的二叉树。

【例】若学生的成绩数据共10000个，考虑程序下列两种情况的操作时间。





分数段	<60	60~69	70~79	80~89	≥90
百分比	5%	15%	40%	30%	10%

情况1：10000* (5%＋2×15%＋3×40%＋4×40%) = 31500次

情况2：10000* (3×20%＋2×80%) = 22000次

最优二叉树（哈夫曼树）定义

权： 将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。

结点的带权路径长度： 从根结点到该结点之间的路径长度与该结点所带权值的乘积。

树的带权路径长度： 树中所有叶子结点的带权路径长度之和。

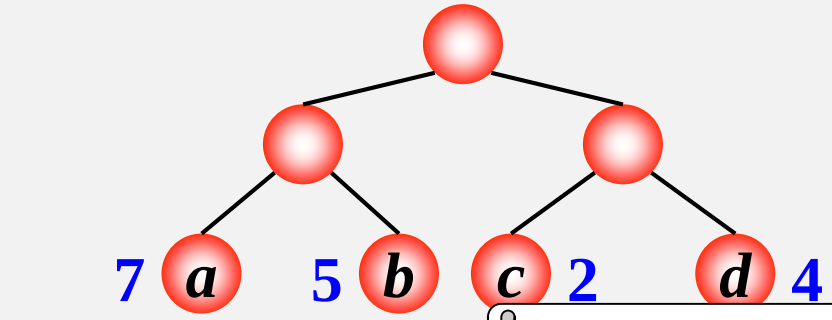
哈夫曼树：

最优二叉树

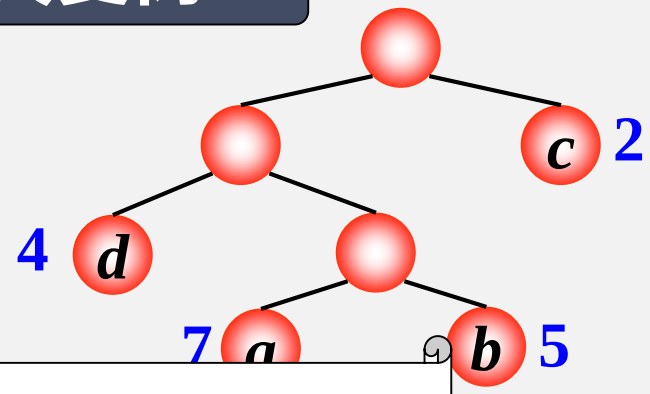
带权路径长度 (WPL) 最短的二叉树

【例】有4 个结点 a [illegible] 构造以此 4 个结点为叶子结点的二叉树。

满二叉树不一定是哈夫曼树



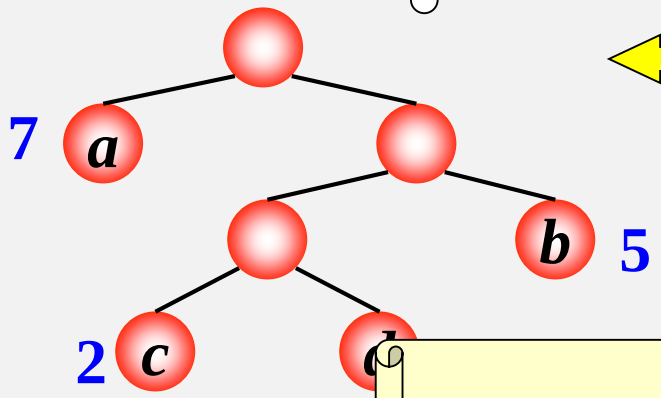
WPL=7×2+5×2+2



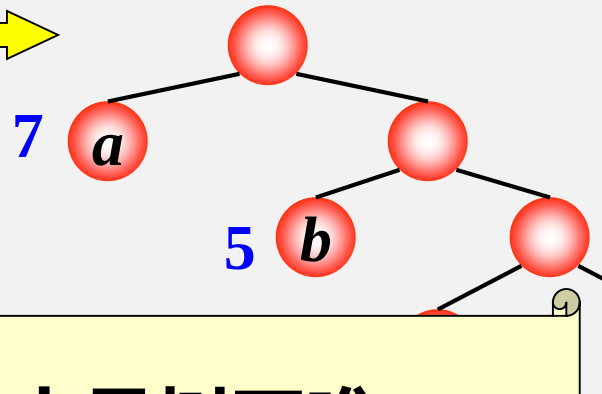
[illegible]×1+4×2= 46

哈夫曼树中权越大的叶子离根越近

哈夫曼树



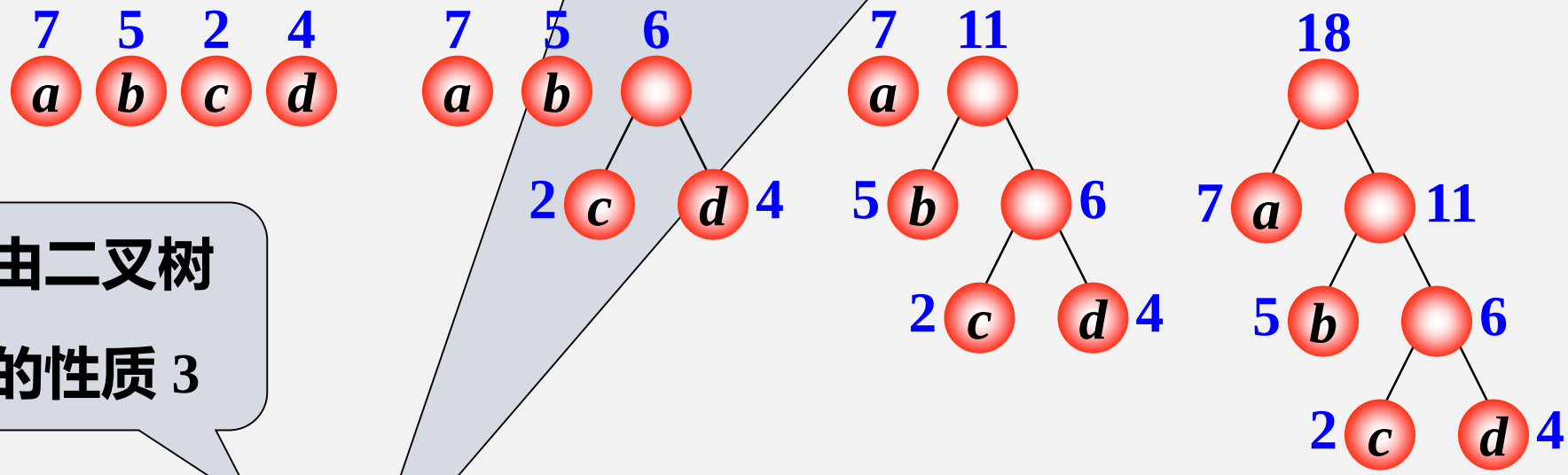
WPL=7×1+5×2+



[illegible]×3+4×3= 35

具有相同带权结点的哈夫曼树不唯一

【例】有4 个结点 a, 包含 n 棵树的森林要经过 n–1 次合并才能形成哈夫曼树，共产生 n–1 个新结点 夫曼树。



由二叉树的性质 3

包含 n 个叶子结点的哈夫曼树中共有 $2n-1$ 个结点。

哈夫曼树的结点的度数为 0 或 2， 没有度为 1 的结点。

哈夫曼算法（构造哈夫曼树的方法）

(1)、根据 n 个给定的权值 $\{w_1, w_2, \cdots, w_n\}$ 构成 n 棵二叉树的森林 $F=\{T_1, T_2, \cdots, T_n\}$，其中 T_i 只有一个带权为 w_i 的根结点。

(2)、在 F 中选取两棵根结点的**权值最小的树**作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左右子树上根结点的权值之和。

(3)、在 F 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 (2) 和 (3)，直到森林中只有一棵树为止，即为哈夫曼树。

```
typedef struct TreeNode *HuffmanTree;
struct TreeNode{
  int Weight;
  HuffmanTree Left, Right;
}
HuffmanTree Huffman( MinHeap H )
{    /* 假设H->Size个权值已经存在H->Elements[]->Weight里 */
     int i;  HuffmanTree T;
     BuildMinHeap(H); /*将H->Elements[]按权值调整为最小堆*/
     for (i = 1; i < H->Size; i++) { /*做H->Size-1次合并*/
          T = malloc( sizeof( struct TreeNode) ); /*建立新结点*/
          T->Left = DeleteMin(H);
                         /*从最小堆中删除一个结点，作为新T的左子结点*/
          T->Right = DeleteMin(H);
                        /*从最小堆中删除一个结点，作为新T的右子结点*/
          T->Weight = T->Left->Weight+T->Right->Weight;
                        /*计算新权值*/
          Insert( H, T ); /*将新T插入最小堆*/
     }
     T = DeleteMin(H);
     return T;
}
```

利用最小堆选取两个权值最小的两个结点，作为哈夫曼树的左右子结点

复杂度的组成：

1) 调整最小堆：O(N)

2)) 2(n-1)+1个删除结点：O(Nlog N)

3) N-1个插入结点：O(Nlog N)

整体复杂度为O(N logN)

哈夫曼编码

【例】如果需传送的电文为 ‘ABACCDA’，如何进行编码?

【分析】它只用到四种字符，用两位二进制编码便可分辨。

- 编码：假设 A, B, C, D 的编码分别为 00, 01, 10, 11，则上述电文便为 ‘00010010101100’ (共 14 位)，
- 译码：译码员按两位进行分组译码，便可恢复原来的电文。

- 为了获得更好的压缩效果，考虑实际应用中各字符的出现频度不相同，采用如下编码方式： 短（长）编码表示频率大（小）的字符

【例】若假设 A，B，C，D 的编码分别为 0，00，1，01，则电文‘ABACCDA’ 便为 ‘000011010’（共 9 位）。该方法“000011010”译码时存在多义性，例如可译为 ‘BBCCDA’、‘ABACCDA’、‘AAAACCACA’ 等。

- 因此要求任一字符的编码都不能是另一字符编码的前缀！

用哈夫曼树设计总长最短的二进制前缀编码

例：假设用于通信的电文仅由8个字母 {a, b, c, d, e, f, g, h} 构成，它们在电文中出现的概率分别为{ 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10}，试为这8个字母设计哈夫曼编码。

解：先将概率放大100倍，以方便构造哈夫曼树。

权值集合 w={7, 19, 2, 6, 32, 3, 21, 10}，

按哈夫曼树构造规则（合并、删除、替换），可得到哈夫曼树。

为清晰起见，重新排序为:

w={2, 3, 6, 7, 10, 19, 21, 32}

w1={5, 6, 7, 10, 19, 21, 32}

w2={7, 10, 11, 19, 21, 32}

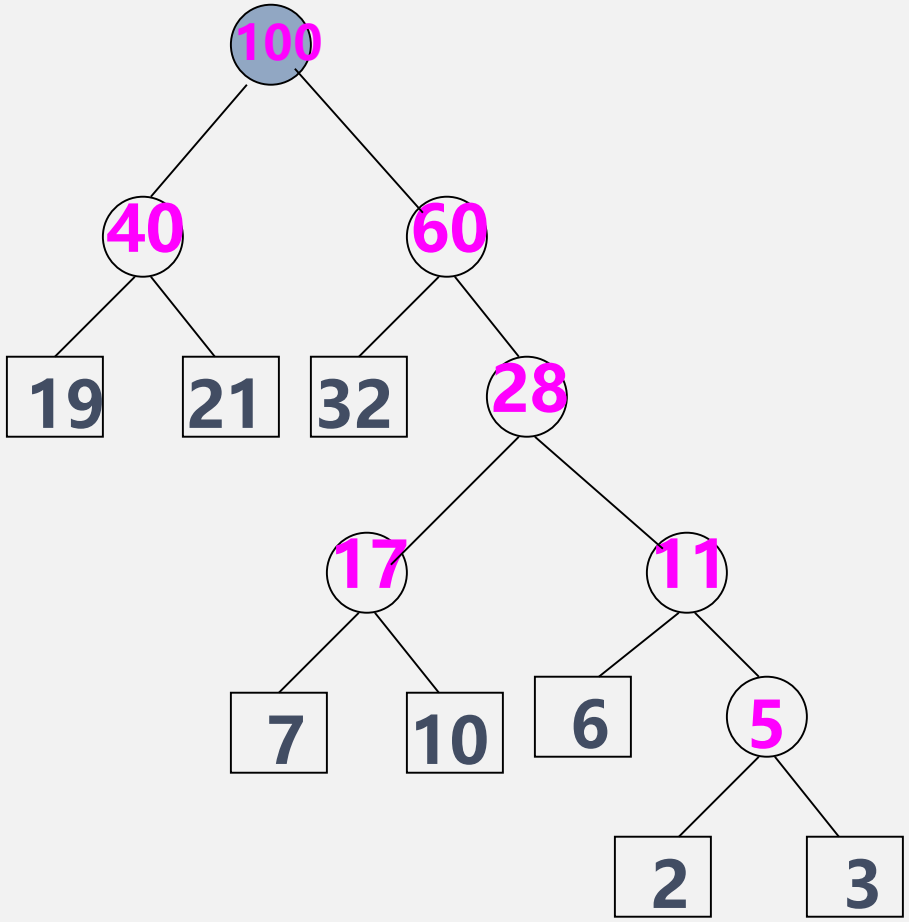
w3={11, 17, 19, 21, 32}

w4={19, 21, 28, 32}

w5={28,32,40}

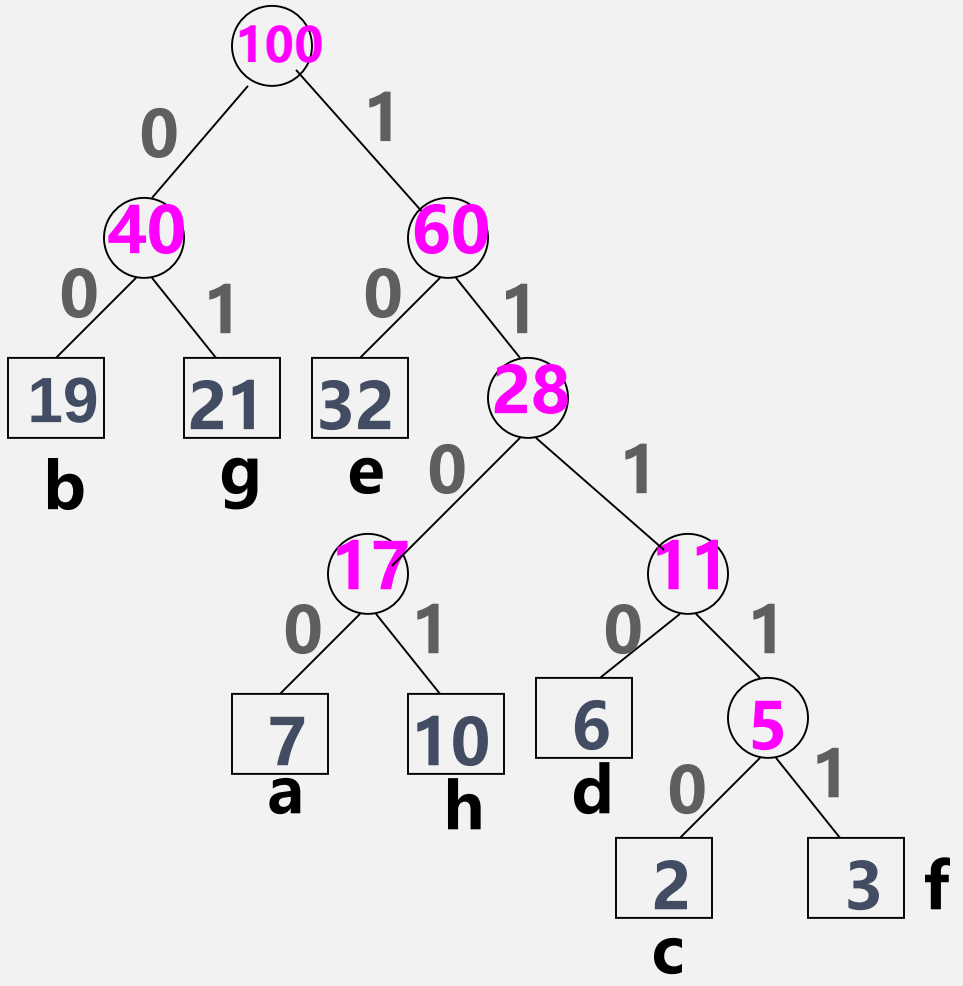
w6={40,60}

w7={100}



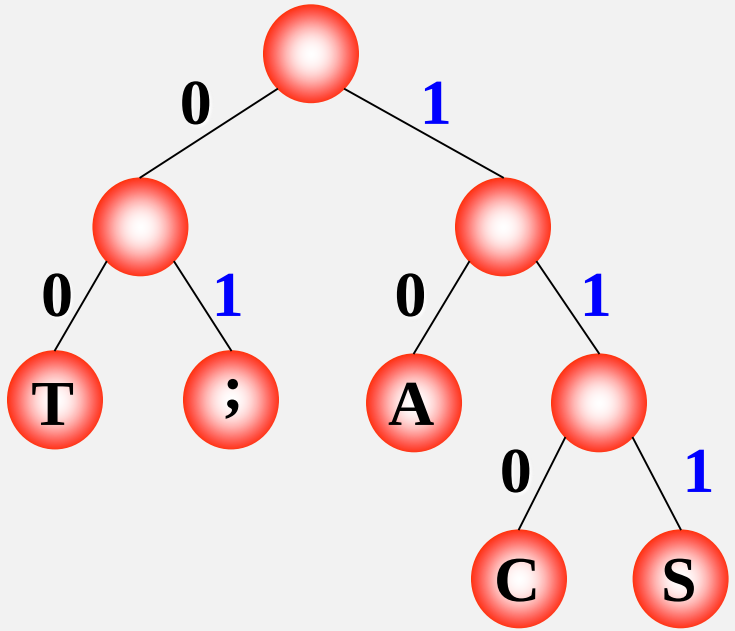
对应的哈夫曼编码（左分支记为0，右分支记为1）：

符	编码	频率
a	1100	0.07
b	00	0.19
c	11110	0.02
d	1110	0.06
e	10	0.32
f	11111	0.03
g	01	0.21
h	1101	0.10



译码

从哈夫曼树根开始，对待译码电文逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束。



电文为 “1101000”

译文只能是“CAT”

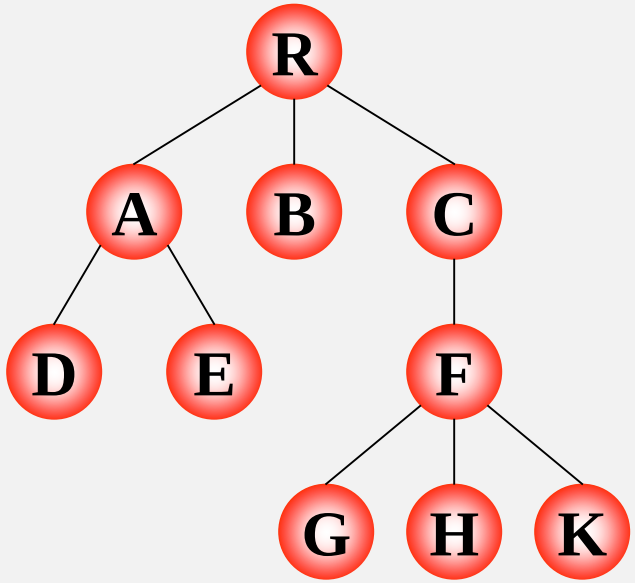
1

树的存储结构

CONTENTS

目录

1 双亲表示法



r = 0
n = 10

数组下标	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

实现：定义结构数组存放树的结点，每个结点含两个域：

- **数据域：存放结点本身信息。**
- **双亲域：指示本结点的双亲结点在数组中的位置。**

结点结构：

```
typedef struct PTNode {
   TElemType  data;
   int    parent;   // 双亲位置域
} PTNode;
```

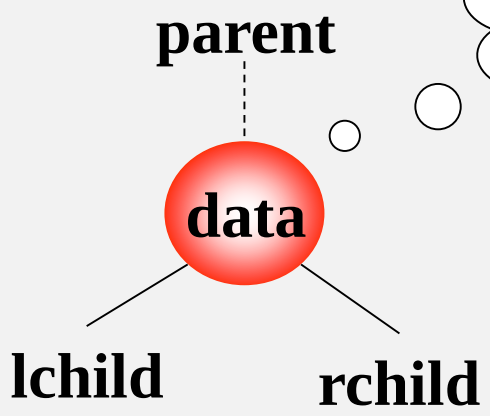
树结构：

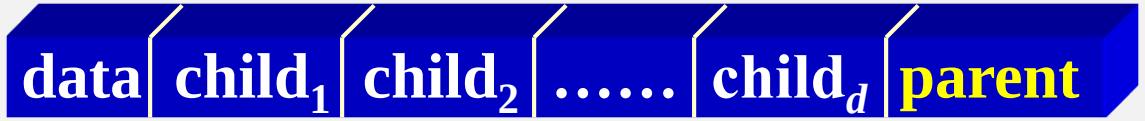
```
#define MAXSIZE  100
typedef struct {
     PTNode  nodes[MAXSIZE];
     int    n,r;     // 结点个数,根结点位置
   } PTree;
```

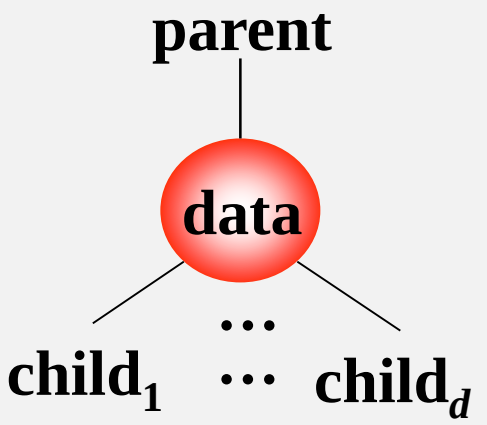
2 孩子表示法（树的链式存储结构）

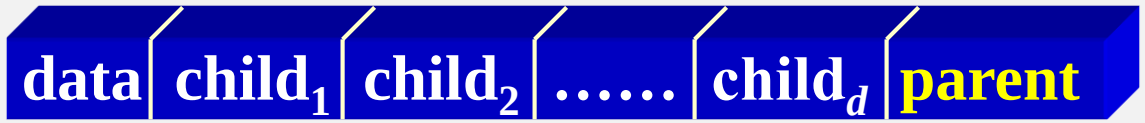
类似二叉树的结点结构

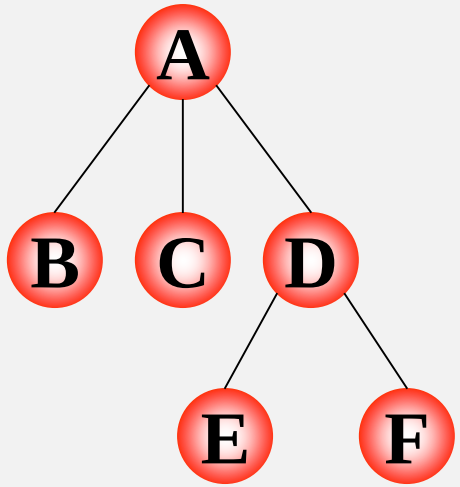
1）多重链表

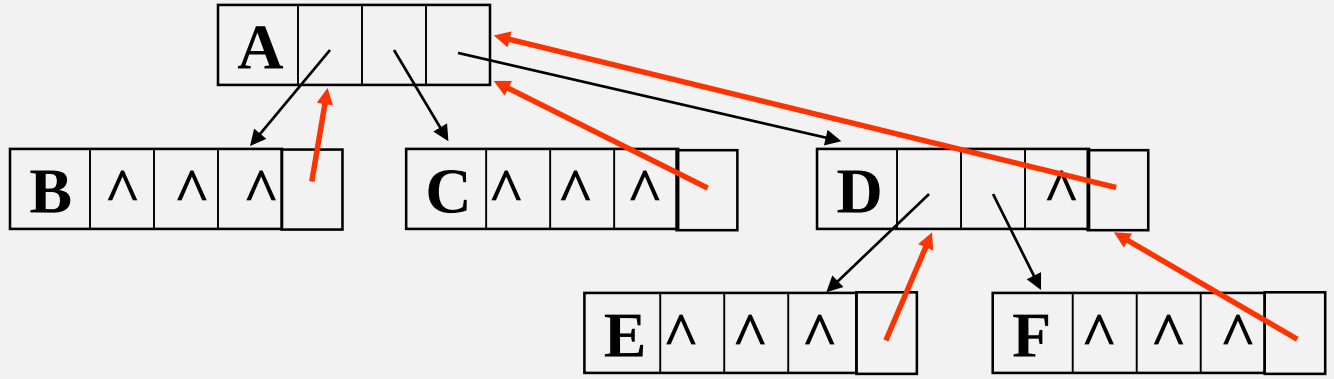








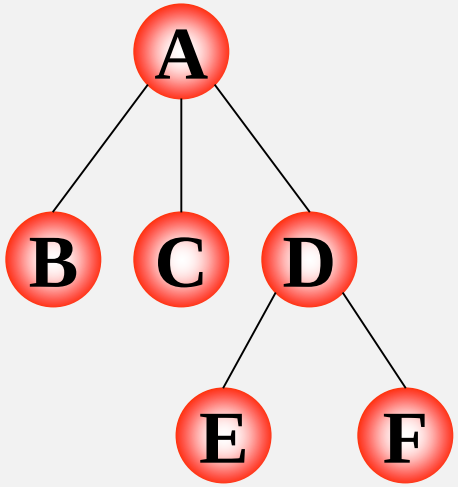


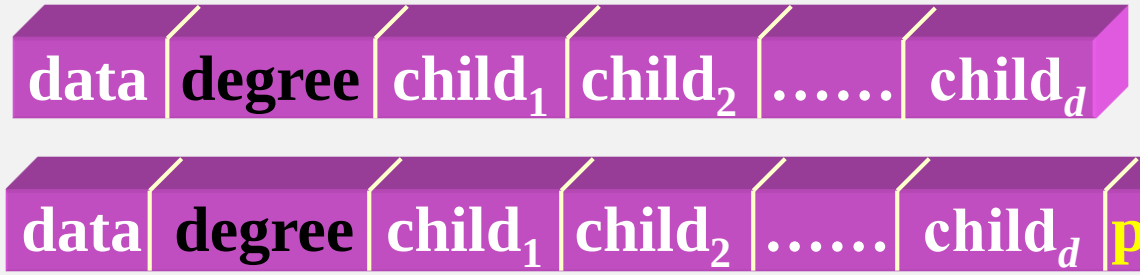


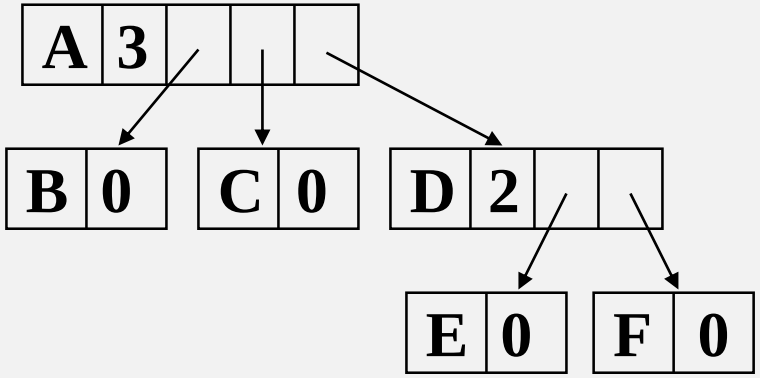
在有 n 个结点、度为 d 的树的 d 叉链表中，有 $n\times(d-1)+1$ 个空链域。

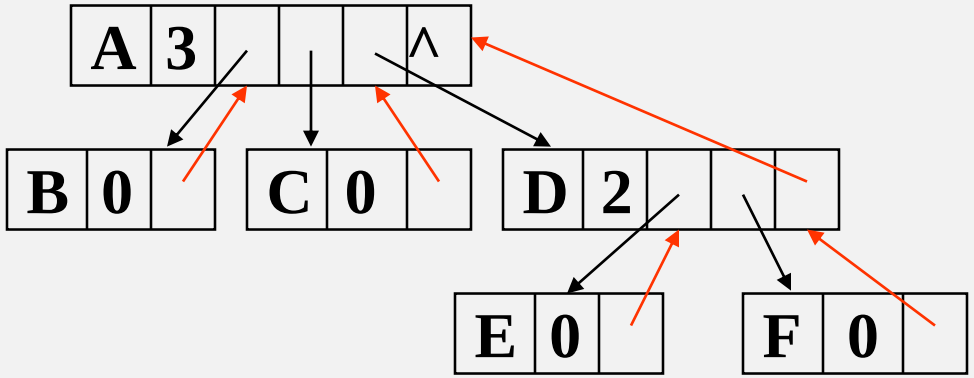
结点同构：

结点的指针个数相等，为树的度 d 。









节约存储空间

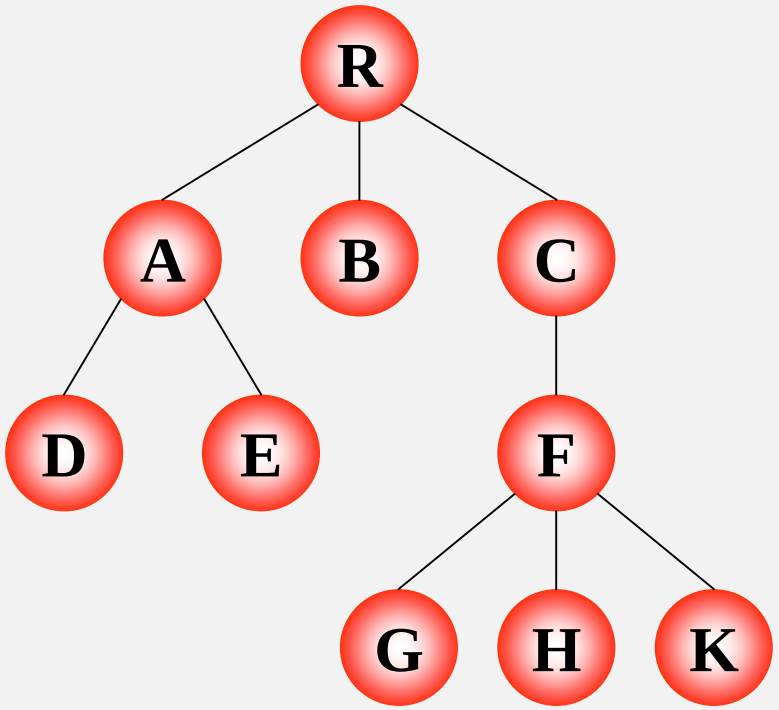
但操作不方便

结点不同构：

结点的指针个数不相等，

为该结点的度 degree。

- **2)孩子链表：**
  - n 个结点组成一个线性表，用顺序表（含 n 个元素的结构数组）存储，每个结点包括：数据域，指向孩子结点链表的指针
  - 把每个结点的孩子结点排列起来，看成是一个线性表，用单链表存储，则 n 个结点有 n 个孩子链表（叶子的孩子链表为空表）。



r=4

n=10

数组下标	data	firstchild
0	4	A
1	4	B
2	4	C
3	0	D
4	-1	R
5	0	E
6	2	F
7	6	G
8	6	H
9	6	K

孩子链表

带双亲的孩子链表

孩子结点结构：

```
typedef struct CTNode {
    int          child;
    struct CTNode   *next;
} *ChildPtr;
```

child	next

结点结构：

```
typedef struct {
    TElemType    data;
    ChildPtr  firstchild;
                  // 孩子链表头指针
} CTBox;
```

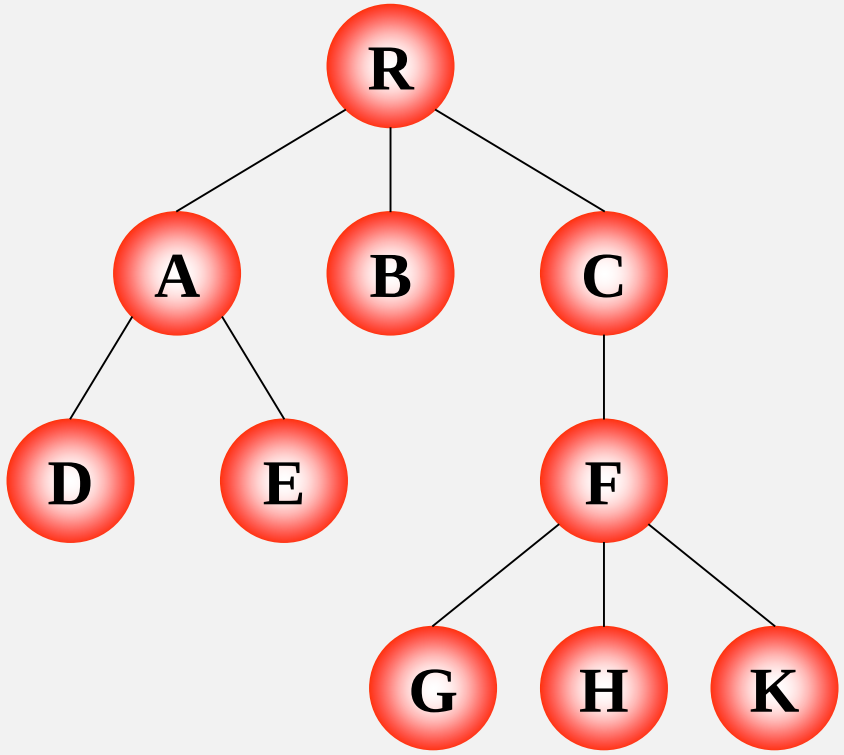
data	firstchild

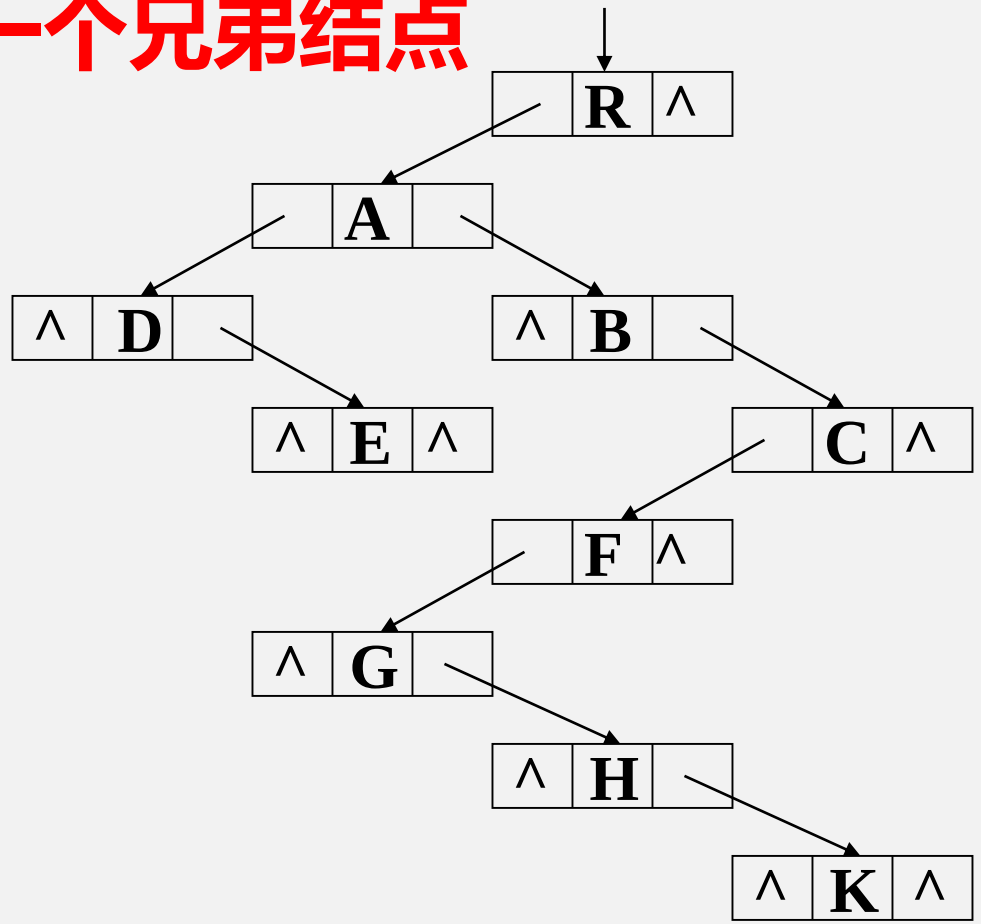
树结构：

```
typedef struct {
    CTBox  nodes[MAXSIZE];
    int    n,r;   // 结点数,根结点位置
} CTree;
```

3 孩子兄弟表示法（二叉树表示法，二叉链表表示法）

用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点





孩子兄弟链表的结构形式与二叉链表完全相同，但存储结点中指针的含义不同：

1、二叉链表中结点的左右指针分别指向该结点的左右孩子；

2、孩子兄弟链表结点的左右指针分别指向它的“长子” 和“大弟”。

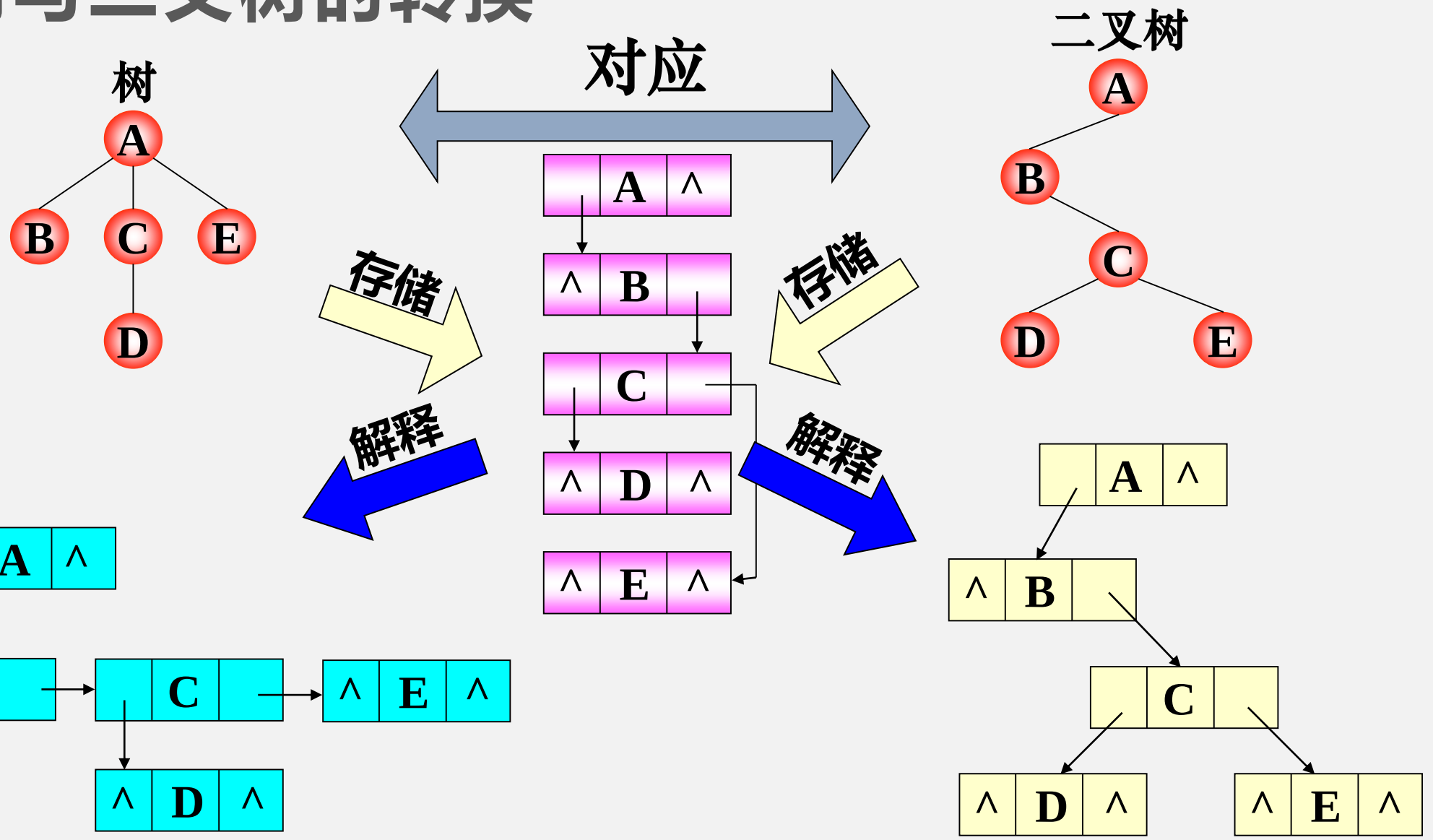
这种解释上的不同正是树与二叉树相互转化的内在基础

2 森林与二叉树的转换

CONTENTS

目录

树与二叉树的转换



树与二叉树的转换——树→二叉树

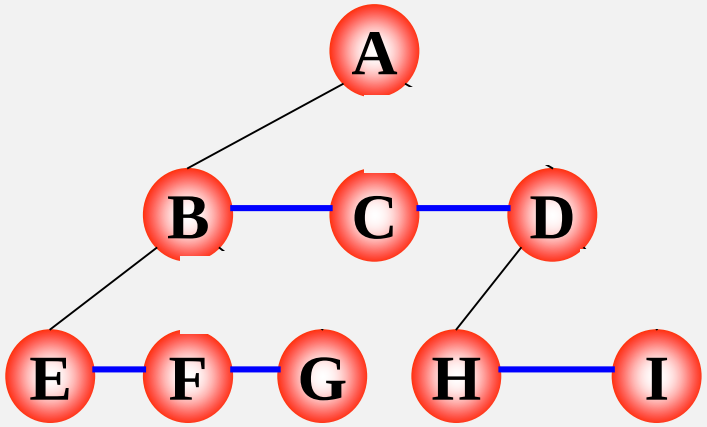
转换步骤：

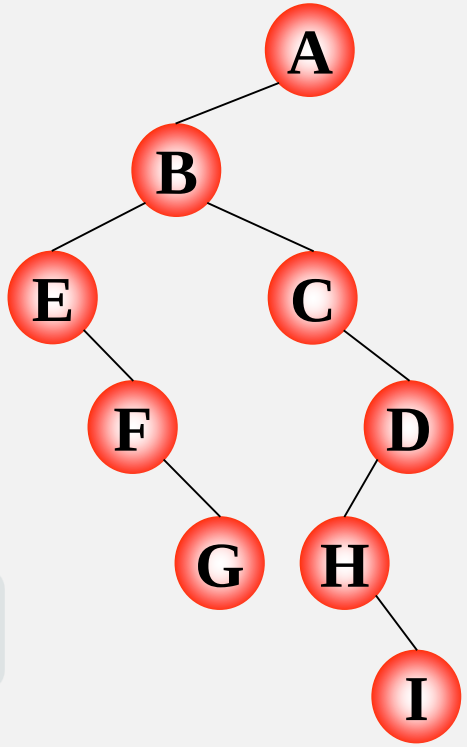
- 将树中同一结点的兄弟相连；
- 保留结点的最左孩子连线，删除其它孩子连线；
- 将同一孩子的连线绕左孩子旋转45度角。

加线

抹线

旋转





树变二叉树：兄弟相连留长子。

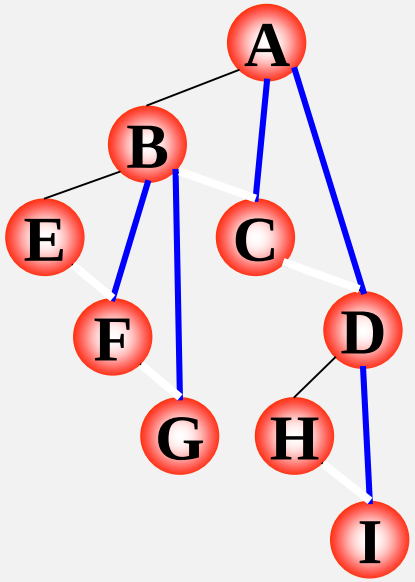
树转换成的二叉树其右子树一定为空。

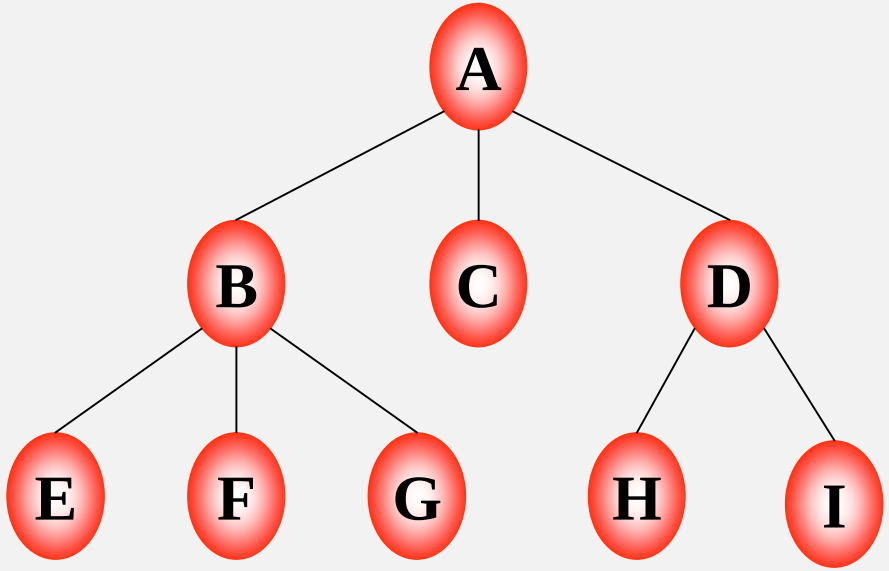
树与二叉树的转换——二叉树→树

加线：若 p 结点是左孩子，则将 p 的右孩子、右孩子的右孩子、…沿分支找到的所有右孩子，都与 p 的双亲用线连起来。

抹线：抹掉原二叉树中双亲与右孩子之间的连线。

调整：将结点按层次排列，形成树结构。





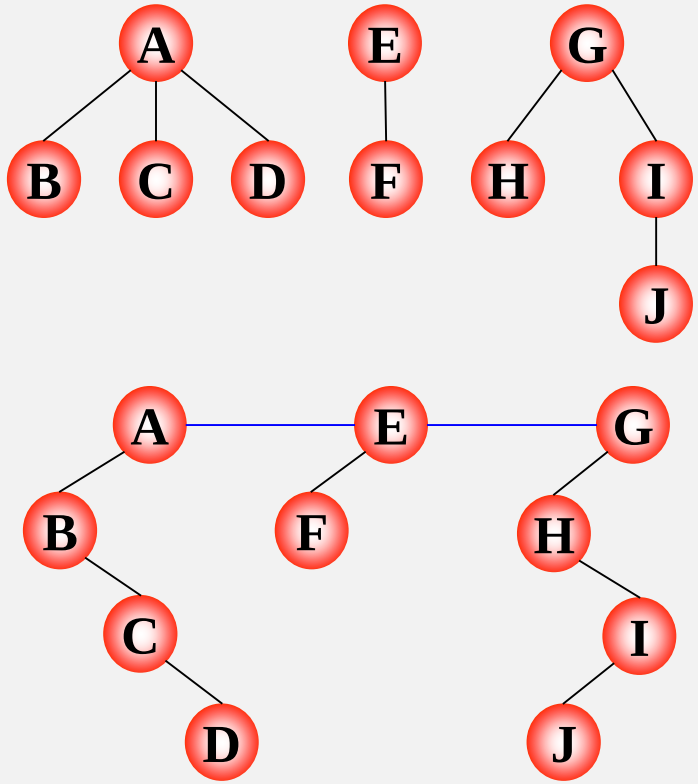
二叉树变树：左孩右右连双亲，去掉原来右孩线。

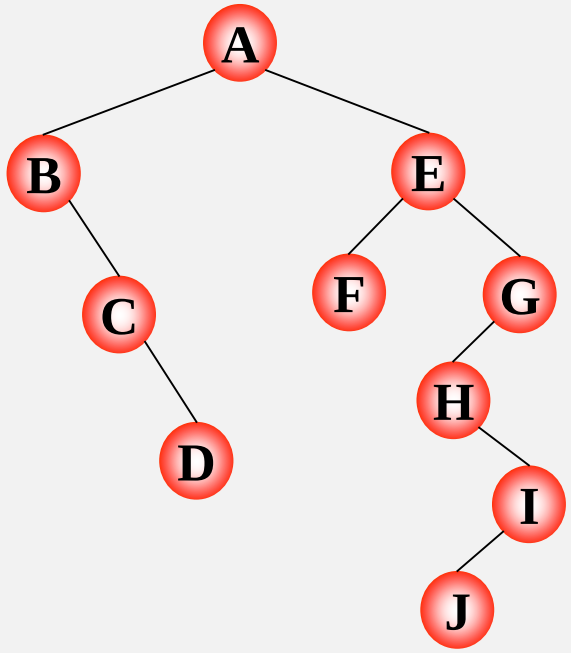
森林与二叉树的转换——森林→二叉树

1、将各棵树分别转换成二叉树。

2、将每棵二叉树的根结点用线相连。

3、以第一棵二叉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。



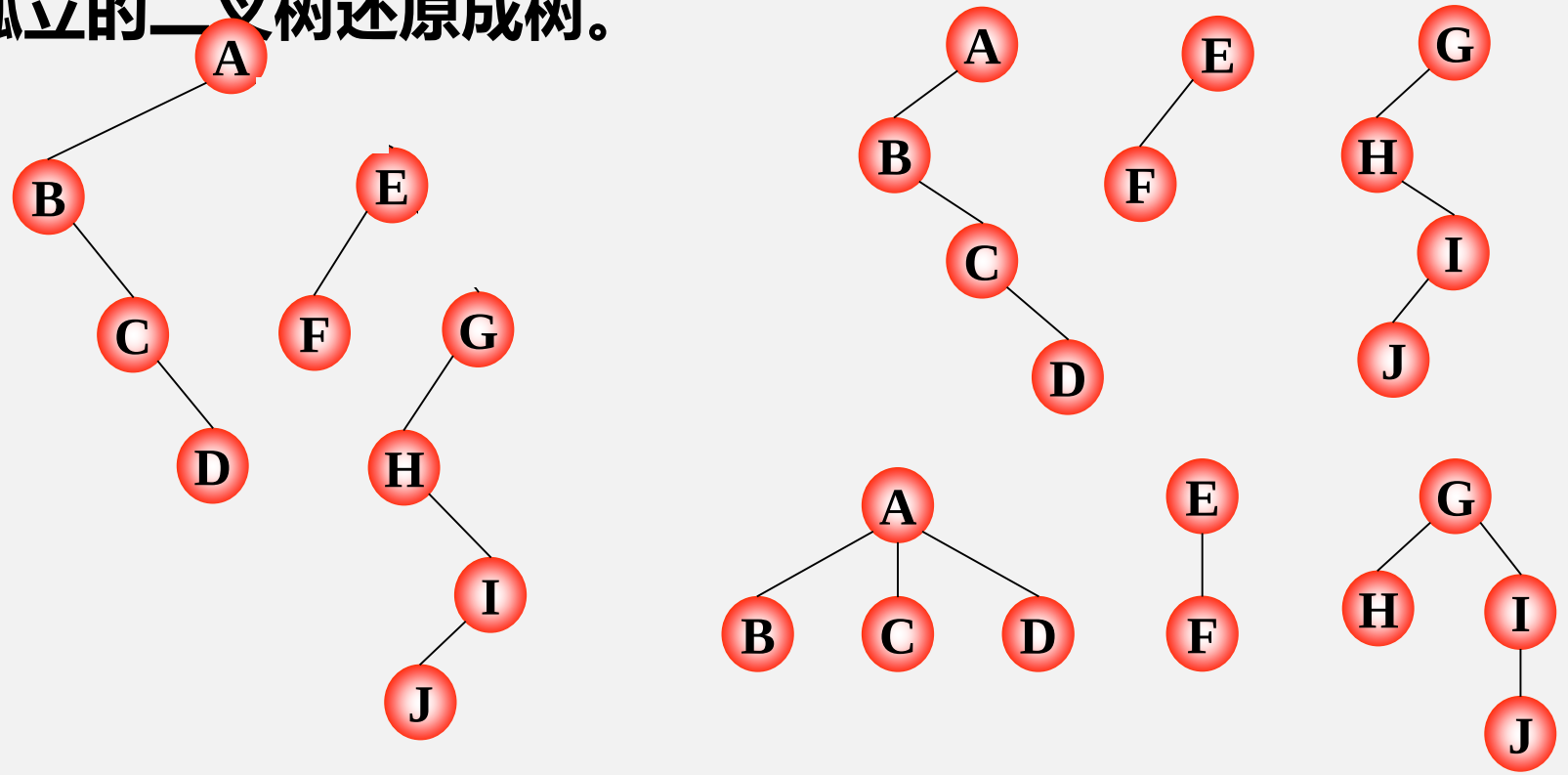


森林变二叉树：树变二叉根相连。

森林与二叉树的转换——二叉树→森林

抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树。

还原：将孤立的二叉树还原成树。



二叉树变森林：去掉全部右孩线，孤立二叉再还原。

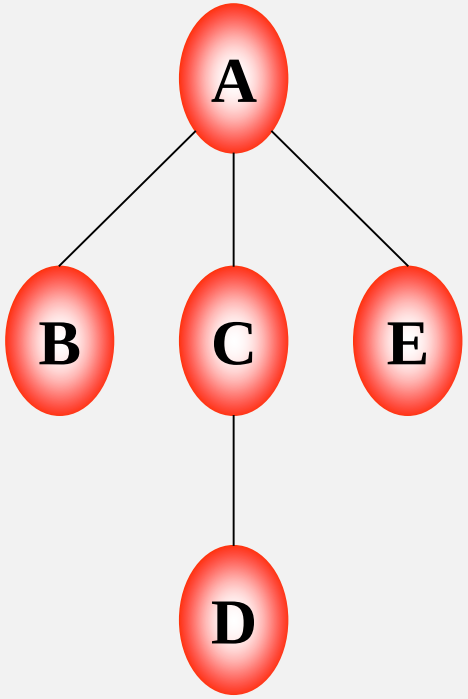
树与森林的遍历

- **树的遍历**

  1、先根（次序）遍历:
  若树不空，则先访问根结点，然后依次先根遍历各棵子树。

  2、后根（次序）遍历:
  若树不空，则先依次后根遍历各棵子树，然后访问根结点。

  3、按层次遍历:
  若树不空，则自上而下自左至右访问树中每个结点。



遍历结果：

先根遍历： A B C D E

后根遍历： B D C E A

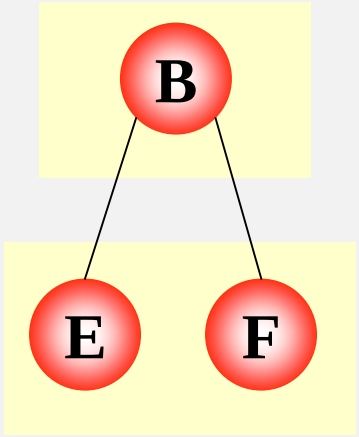
按层次遍历： A B C E D

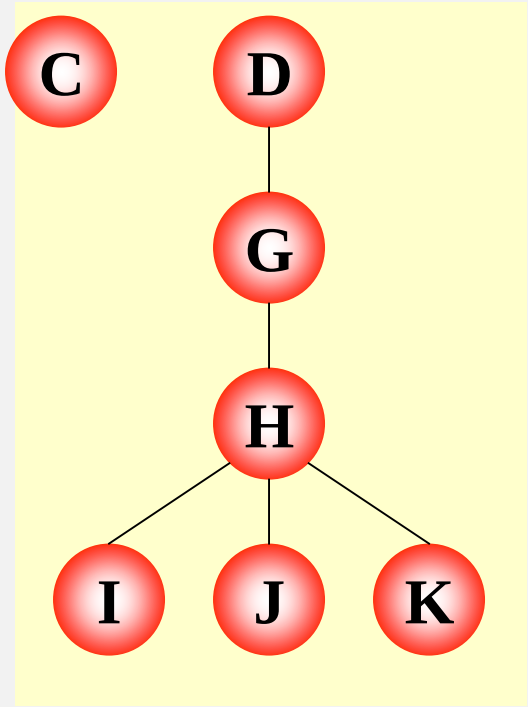
树与森林的遍历

森林的遍历

森林由三部分构成：

1、森林中第一棵树的根结点；
2．森林中第一棵树的子树森林；
3．森林中其它树构成的森林。





- 先序遍历：若森林不空，则

1.访问森林中第一棵树的根结点；

2.先序遍历森林中第一棵树的子树森林；

3.先序遍历森林中（除第一棵树之外）其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行先根遍历。

- **中序遍历：若森林不空，则**

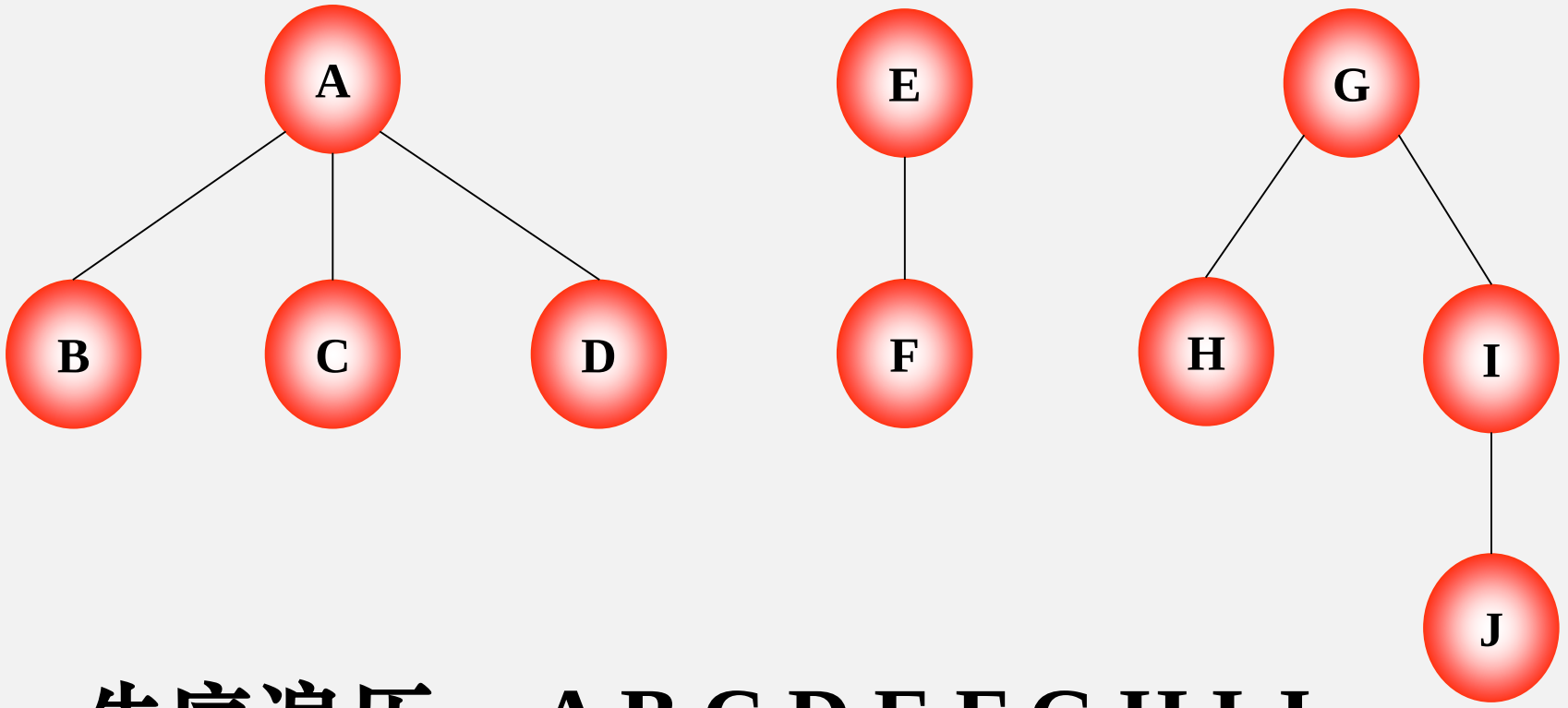
1.中序遍历森林中第一棵树的子树森林；

2.访问森林中第一棵树的根结点；

3.中序遍历森林中（除第一棵树之外）其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行后根遍历。

【例】先序和中序遍历如下森林



先序遍历： A B C D E F G H I J

中序遍历： B C D A F E H J I G