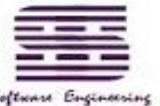


# 实用算法设计——查找

主讲：娄文启

[louwenqi@ustc.edu.cn](mailto:louwenqi@ustc.edu.cn)



# 5 查找

**假定：待查找的数据已在内存中。**

**重点：**理解二叉查找树、三类平衡二叉树（AVL树、红黑树、伸展树）的相关概念以及特点。

**难点：**理解三类平衡二叉树（AVL树、红黑树、伸展树）的基本操作的设计思想。

**基础：**C语言编程；树形结构的定义及基本操作的实现。



# 5 查找

## 5.1 基于Hash表的查找

## 5.2 蛮力查找（顺序查找）

## 5.3 基于有序表的二分查找

## 5.4 字符串的查找

## 5.5 基于树的查找



2024/12/9



3

# 5.5 基于树的查找

- 基本概念：树形结构、树、二叉树
  - 二叉树的定义、性质、实现
- 二叉查找（排序）树
  - 二叉查找树的定义、特点、基本操作的实现（结合源码）、性能分析
- 如何解决二叉查找（排序）树的失衡？
  - AVL树
  - 红黑树
  - 伸展树

# 回顾：已学过的查找算法

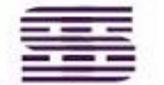
- 已学过的查找算法（按内容查找）：
  - 蛮力查找（顺序查找）
  - 基于有序表的二分查找：  $T(N) = O(\log_2 N)$
  - 基于Hash表的查找：  $T(N) = T_{\text{Slot}} + T_{\text{node}}$
  - 字符串查找：
    - Boyer-Moore查找：  $T(N) = ?$
    - KMP查找：  $T(N) = ?$
- 其中，基于有序表的二分查找：
  - 性能最好：  $O(\log_2 N)$
  - 属于静态查找算法：仅做查询和检索操作。
  - 查询某个特定的数据项是否在待查找的集合中；
  - 检索某个特定的数据项的各种属性。

按内容快速查找

- 二叉查找树（二叉排序树）：
  - 可维持最好性能：  $T(N)=O(\log_2 N)$
  - 属于动态查找算法：在查询过程中，同时插入待查集合中不存在的元素，或者删除已存在的某个元素。



2024/12/9



Software Engineering

6

# 树形结构

直接前驱/后继在不同树中的含义不同

- 四类数据结构 (D+S) :

- 集合结构:

- 元素间无任何关系，即关系结合是空集：  $D=\emptyset$

- 线性结构: 如, 线性表, 字符串。

- 元素间的关系是1:1

- 除头结点外，所有结点有且仅有一个直接前驱；

- 除尾结点外，所有结点有且仅有一个直接后继。

- 树形结构: 如, 一般树、二叉树、森林

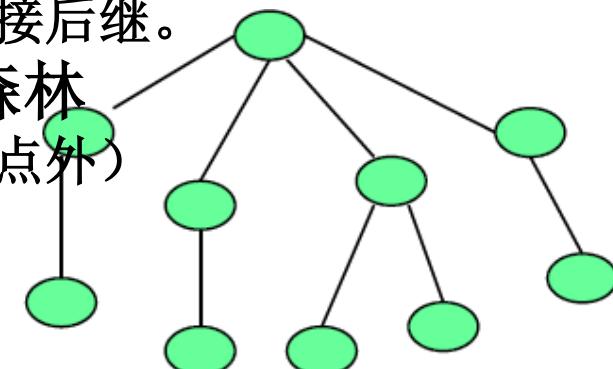
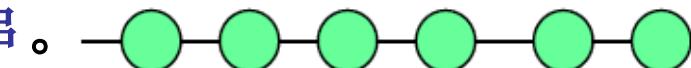
- 一个结点可有多个直接后继（除叶子结点外）

- 但只有一个直接前驱（除根结点外）；

- 图形结构:

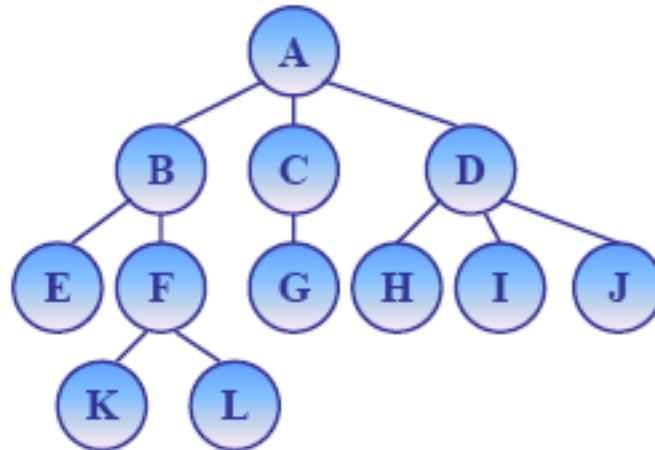
- 元素间的关系是m:n;

- 一个结点可以有多个直接后继，也有多个直接前驱。

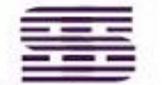


# 树

- 树：  $n(\geq 0)$  个有限节点的集合（递归定义）
  - 结点数  $n=0$  时，是空树
  - 结点数  $n>0$  时，有且仅有一个根结点、  $m$  个互不相交的有限结点集—— $m$  棵子树



2024/12/9



Software Engineering

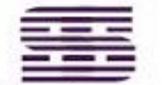
# 树

## 树的特点

- 每个节点都只有有限个子节点或无子节点
- 没有父节点的节点称为根节点；每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；
- 树里面没有环路，意思就是从一个节点出发，除非往返，否则不能回到起点。



2024/12/9



Software Engineering

9

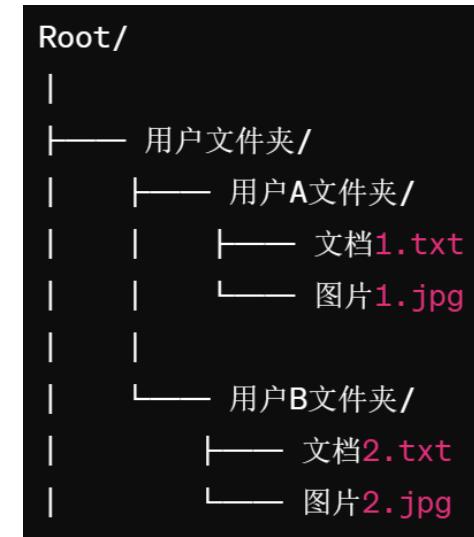
# 树

## 有序树

- 每个节点的子节点之间有明确的顺序或者位置关系；
- 子节点的顺序通常由它们在父节点中的位置来确定
- 有序树的典型例子是二叉树

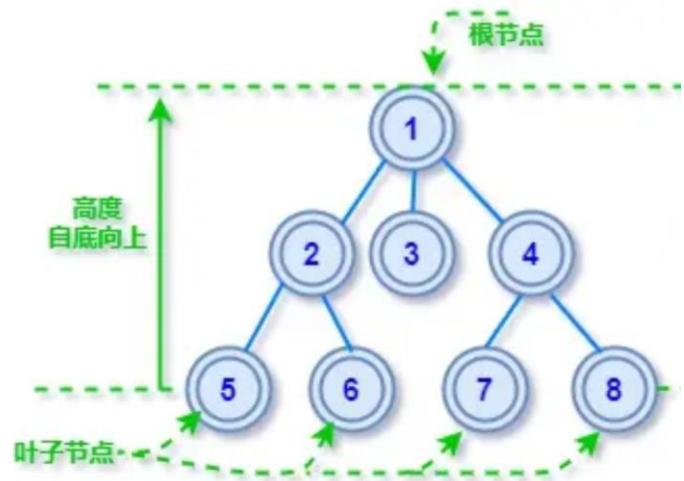
## 无序树

- 节点之间没有明确的顺序关系
- 子节点之间可以任意排序，没有左右之分
- 典型例子是普通的树结构，例如文件系统



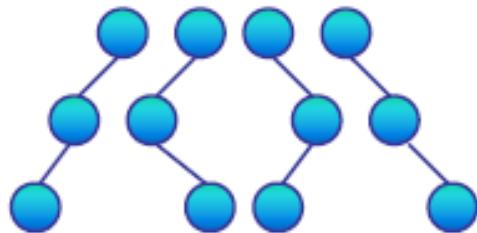
2024/12/9

- 树的相关术语：
  - 结点：叶子(终端结点)、根、内部结点（非终端结点/分支结点）；
  - 树的规模：结点的度、树的度、结点的层次、**树的高度(深度)**；
  - 结点间的关系：双亲(1)—孩子( $m$ )，祖先—子孙，兄弟，堂兄弟；

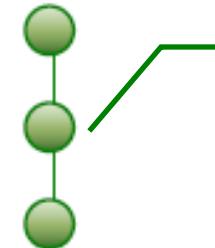


# 二叉树的定义

- 特点：
  - 树的度 $\in [0,2]$ ，即每个节点至多只有两棵子树；
  - 子树有左右之分。
- Note: 二叉树 vs. 度不大于2的有序树



二叉树



有序树

当某个结点只有一棵子树时，不存在序的概念



# 二叉树的性质

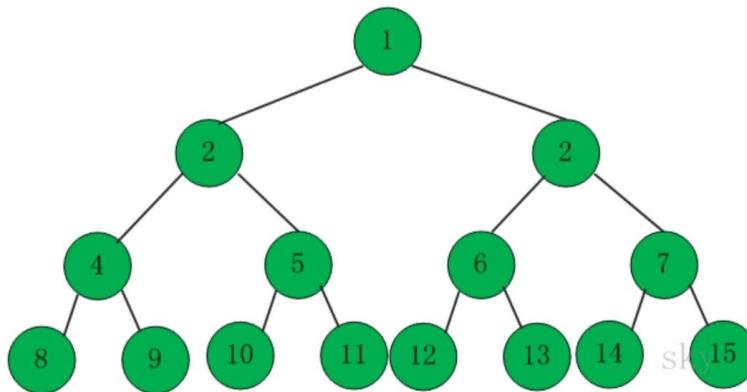
- **性质1：**第*i*层至多有 $2^{i-1}$ 个节点；
  - 因为每个节点最多只有2个孩子
- **性质2：**深度为*k*的二叉树至多有 $2^k-1$ 个节点( $k \geq 0$ )；
  - 由性质1，将各层最多的节点数累加，再结合等比数列的求和得出
- **性质3：**  $n_0 = n_2 + 1$  ( $n_i$ 表示二叉树中度为*i*的节点个数)
  - 二叉树中节点的构成（根据度）  $n = n_0 + n_1 + n_2$
  - 二叉树中充当其余节点的孩子的节点数  $n-1$ (去掉根) =  $n_1 + 2 \times n_2$



# 二叉树的性质

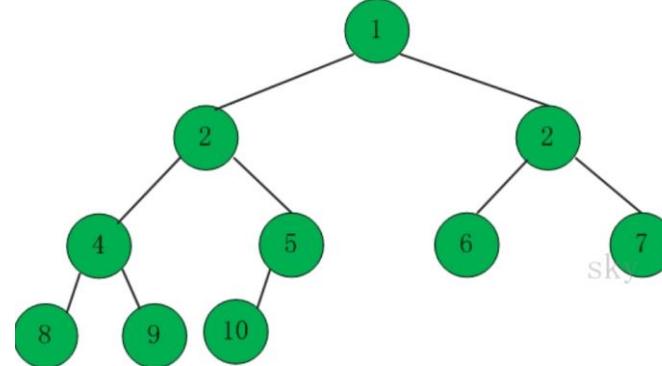
- **性质4：**包含n个节点的二叉树的高度至少为 $\log_2(n+1)$ ；
  - 因为每个节点最多只有2个孩子(和性质2类似)

满二叉树



每一层的结点数都达到最大

完全二叉树



仅最下面两层节点的度可以小于2，且靠左



# 二叉树的性质

- 例题：若一颗完全二叉树有**768**个结点，则该二叉树中叶结点的个数是？

$$N = n_0 + n_1 + n_2 = 2n_0 - 1 + n_1 = 768$$

$$n_1 = 1 \text{ or } 0$$

$$\text{故 } n_0 = 384$$



2024/12/9

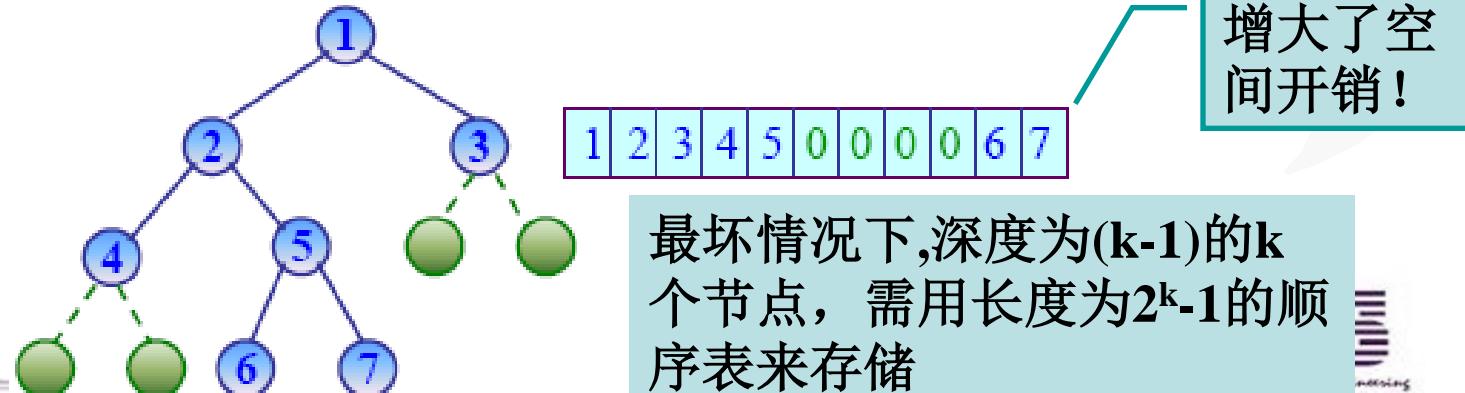
# 二叉树的实现

- 逻辑特征：树形结构
- 存储结构：顺序存储和链式存储两种方式
- 基本操作：
  - 创建空的二叉树
  - 销毁已有二叉树
  - 遍历二叉树：如何确保在遍历过程中，对于树中每个节点只访问一次（将树形输入转变为线性输出）
  - 查找结点
  - 插入结点
  - 删除结点



# 二叉树的顺序存储结构

- 适用于完全二叉树：
  - 完全二叉树：二叉树中，除最底层外，其它各层的结点数都达到最大个数，且最底层所有的节点都连续集中在最左边。（特殊的二叉树，由满二叉树演变而来）
- 对于一般二叉树：
  - 方法：二叉树→补虚结点形成完全二叉树→自上而下、自左至右存储
  - 例：



# 二叉树的顺序存储结构——定义

定义1: (须引入特殊符号表示虚结点的值)

```
#define MAX_TREE_SIZE 100 /* 二叉树的最大结点数 */  
ElemType SqBiTree[MAX_TREE_SIZE]; /*下标为0的数组元  
素存储根结点 */
```

定义2:

```
#define MAX_TREE_SIZE 100 /* 二叉树的最大结点数 */  
typedef struct{  
    ElemType elem[MAX_TREE_SIZE+1]; /*下标为1的数组  
元素存储根结点 */  
    int length;  
}SqBiTree;
```

左子节点 $2*i$ , 右子节点的位置是  $2*i+1$ ,  
父节点的位置是  $i/2$ 。



# 二叉树的链式存储结构

- 采用链式存储时，二叉树由一连串的结点组成，结点之间的关系用链（指针）来表示。
- 二叉链表：
  - 结点中包含两个指针（分别指向该结点的两个孩子结点）和数据域。
- 三叉链表：
  - 结点中包含三个指针（分别指向该结点的两个孩子结点和双亲结点）和数据域。



# 二叉链表的定义

```
typedef struct BiTNode{  
    ELEM_TYPE data;  
    struct BiTNode *lchild, *rchild; /* 左右孩子指针 */  
}BiTNode, *BiTree;
```

或者Left, right, p， 分别指向左、右孩子以及父母

前、中、后序遍历

前序(根左右)， 中序(左根右)， 后序(左右根)

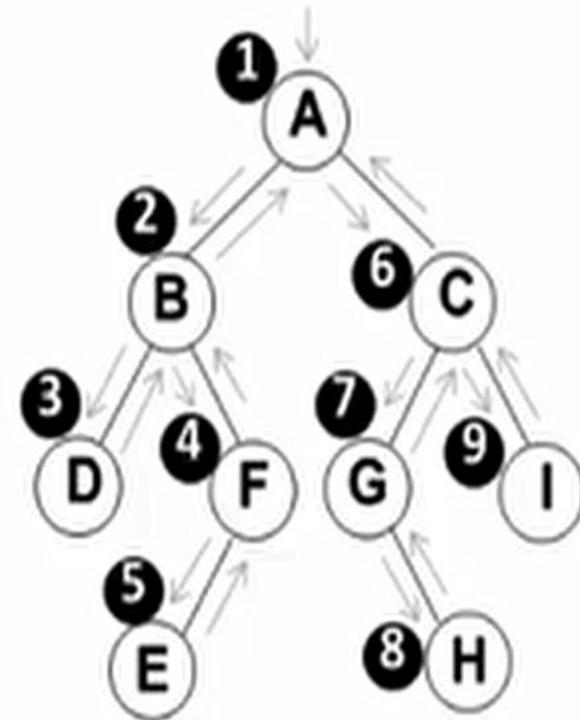


# 前序遍历 (根左右)

Preorder-Tree-Walk(x)

1. if  $x \neq \text{nil}$
2. print  $x.\text{key}$
3. Preorder-Tree-Walk( $x.\text{left}$ )
4. Preorder-Tree-Walk( $x.\text{right}$ )

A (B D F E ) (C G H I)



先序遍历=> A B D F E C G H I

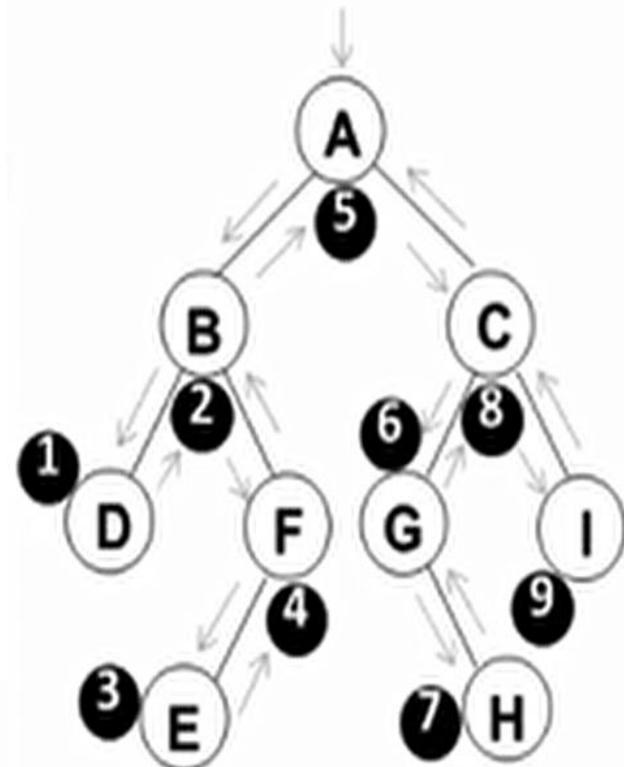


# 中序遍历 (左根右)

Inorder-Tree-Walk(x)

1. if  $x \neq \text{nil}$
2. Inorder-Tree-Walk( $x.\text{left}$ )
3. print  $x.\text{key}$
3. Inorder-Tree-Walk( $x.\text{right}$ )

(D B E F) A (G H C I)



中序遍历=> D B E F A G H C I



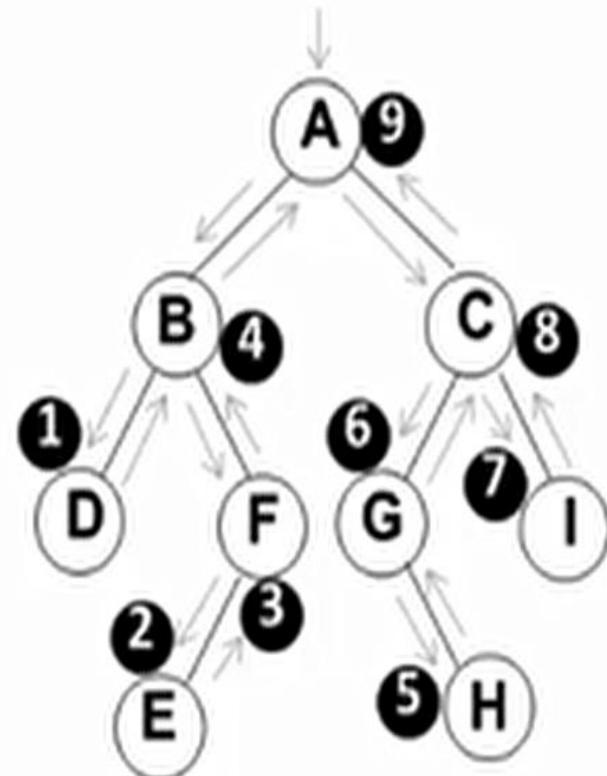
# 后序遍历 (左右根)

Postorder-Tree-Walk(x)

1. if  $x \neq \text{nil}$
2. Postorder-Tree-Walk( $x.\text{left}$ )
3. Postorder-Tree-Walk( $x.\text{right}$ )
4. print  $x.\text{key}$

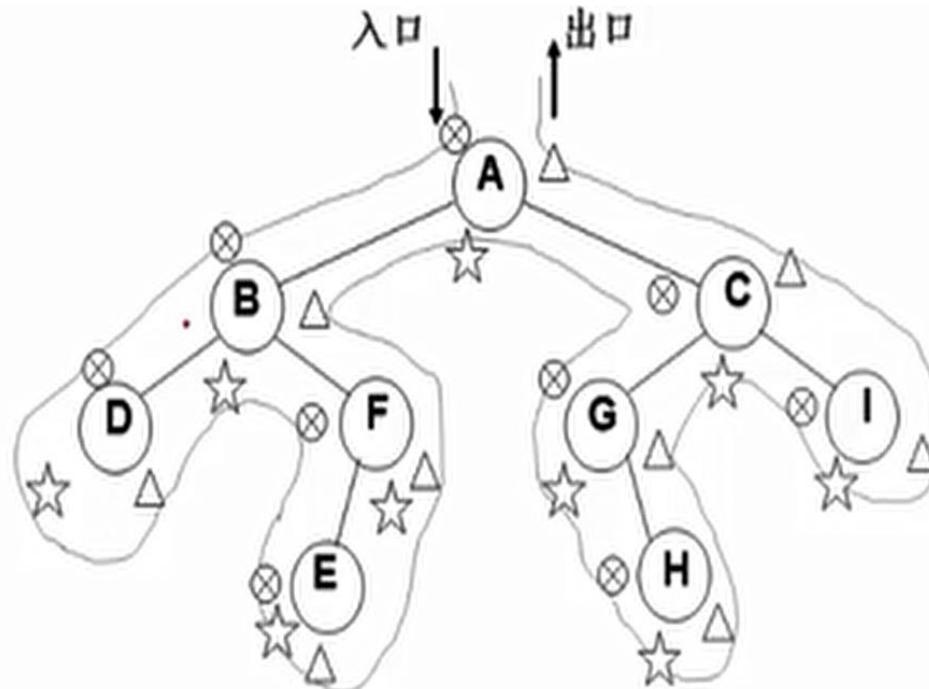
(DEFB ) (HGIC) A

后序遍历=> D E F B H G I C A



# 前/中/后序遍历规律

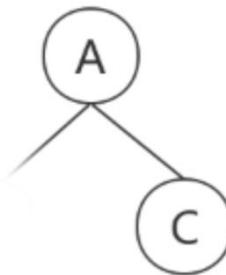
- 先序、中序和后序遍历过程：遍历过程中经过结点的路线一样，只是访问各结点的时机不同。
- 图中在从入口到出口的曲线上用 $\otimes$ 、 $\star$ 和 $\triangle$ 三种符号分别标记出了先序、中序和后序访问各结点的时刻。



# 二叉树遍历-例题

已知某二叉树的前序遍历为A-B-D-F-G-H-I-E-C,中序遍历为F-D-H-G-I-B-E-A-C,请还原这颗二叉树

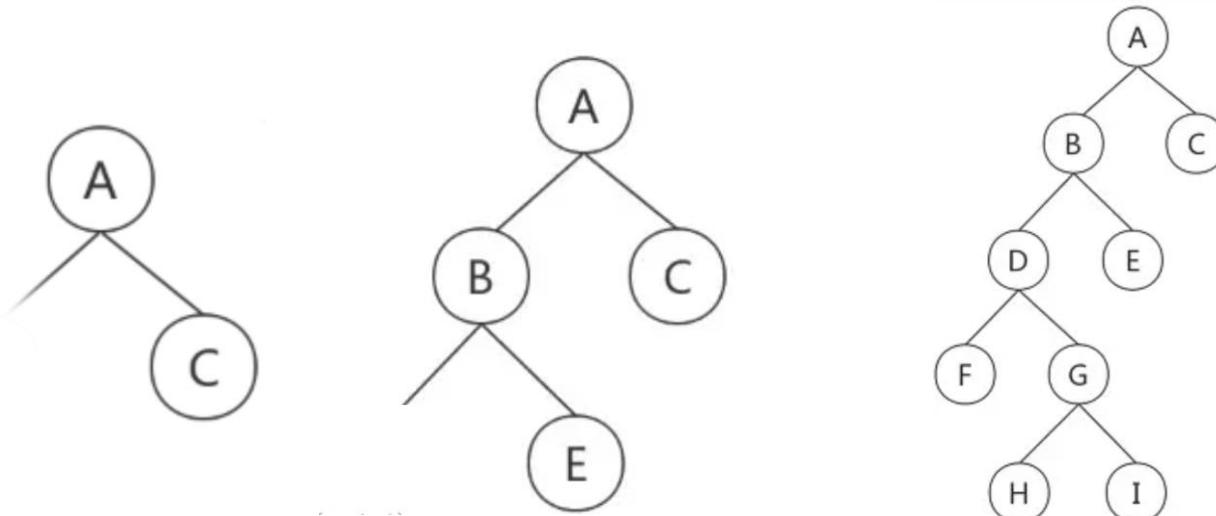
从前序遍历中，我们确定了根结点为A，在从中序遍历中得出 F-D-H-G-I-B-E在根结点的左边，C在根结点的右边，那么我们就可以构建我们的二叉树的雏形



# 二叉树遍历-例题

已知某二叉树的前序遍历为A-B-D-F-G-H-I-E-C, 中序遍历为F-D-H-G-I-B-E-A-C, 请还原这颗二叉树

剩下的前序遍历为B-D-F-G-H-I-E, 中序遍历为F-D-H-G-I-B-E, **B**就是我们新的“根结点”，从中序遍历中得出F-D-H-G-I在B的左边，E在B的右边，继续构建



# 二叉树遍历-例题

已知某二叉树的后序遍历为D-A-B-E-C,中序遍历为D-E-B-A-C,请还原这颗二叉树。



# 遍历复杂度

If  $x$  is the root of an  $n$ -node subtree, then the call INORDER-TREE-WALK( $x$ ) takes  $\Theta(n)$  time.

考虑基本二叉平衡的时候

```
INORDER-TREE-WALK(x)
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```

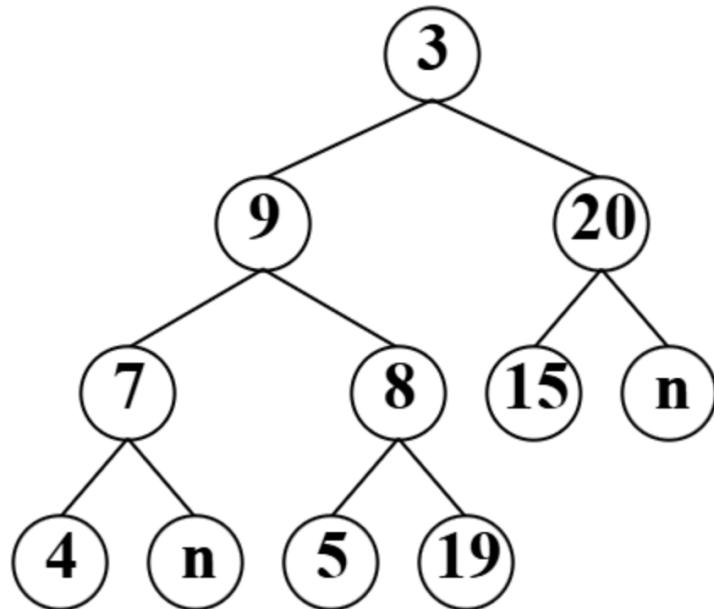
$$T(0) = c \quad T(n) \leq (c + d)n + c$$

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c , \end{aligned}$$



# 层序遍历

给你二叉树的根节点 root，返回其节点值的 层序遍历。（即逐层地，从左到右访问所有节点）。



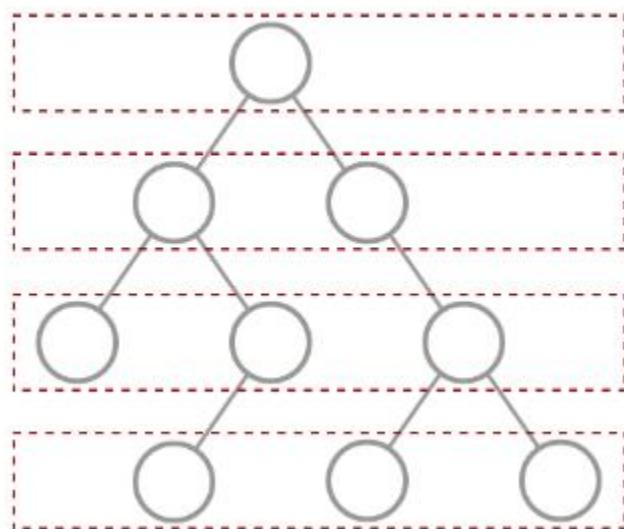
输入： root =  
[3,9,20,7,8,15,null,4,null,5,19]

输出：  
[[3],[9,20],[7,8,15],[4,5,19]]



# 层序遍历 (BFS)

给你二叉树的根节点 root，返回其节点值的 层序遍历。（即逐层地，从左到右访问所有节点）。

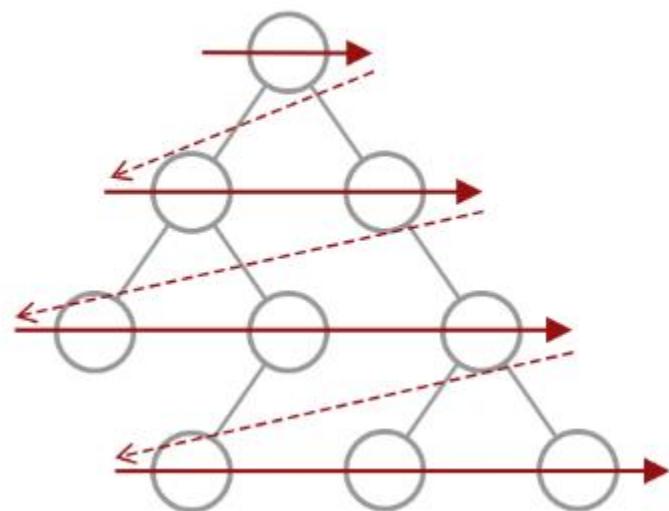


第 0 层

第 1 层

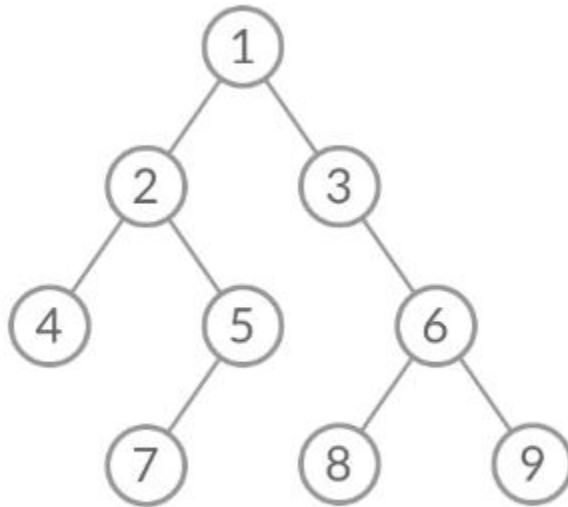
第 2 层

第 3 层



# 层序遍历 (BFS)

层序遍历要求我们区分每一层，也就是返回一个二维数组。  
而 BFS 的遍历结果是一个一维数组



BFS 遍历结果：一维数组

[1 2 3 4 5 6 7 8 9]

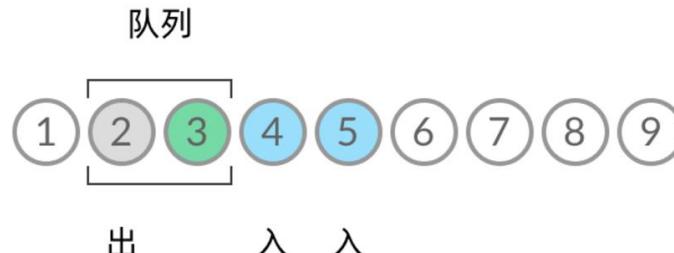
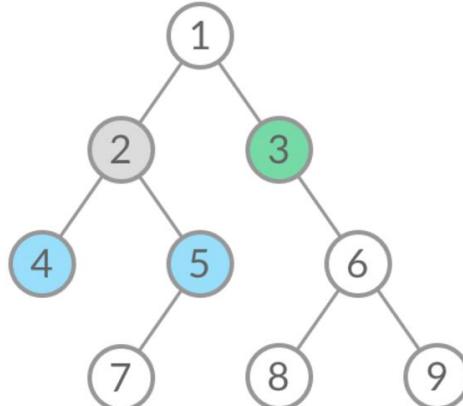
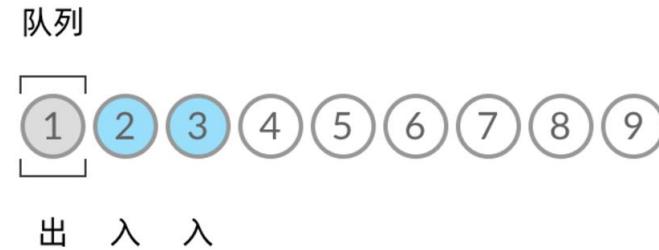
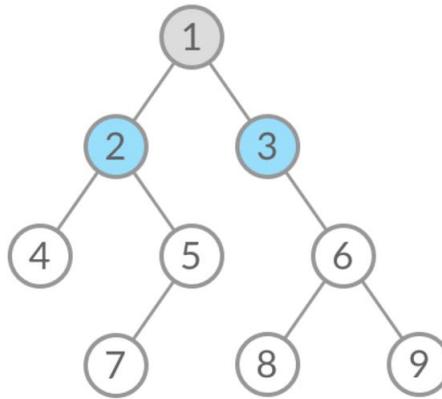
层序遍历期望的结果：二维数组

[[1]  
[2 3]  
[4 5 6]  
[7 8 9]]



# 层序遍历 (BFS)

中间状态，队列中存储了不同行的数据



# 层序遍历 (BFS)

```
vector<vector<int>> levelOrder(TreeNode* root) {  
    queue<TreeNode*> que;  
    vector<vector<int>> res;  
    if (root != nullptr) que.push(root);  
    while (!que.empty()) {  
        vector<int> tmp;  
        for(int i = que.size(); i > 0; --i) {  
            root = que.front();  
            que.pop();  
            tmp.push_back(root->val);  
            if (root->left != nullptr) que.push(root->left);  
            if (root->right != nullptr) que.push(root->right);  
        }  
        res.push_back(tmp); } }
```



# 层序遍历(DFS)

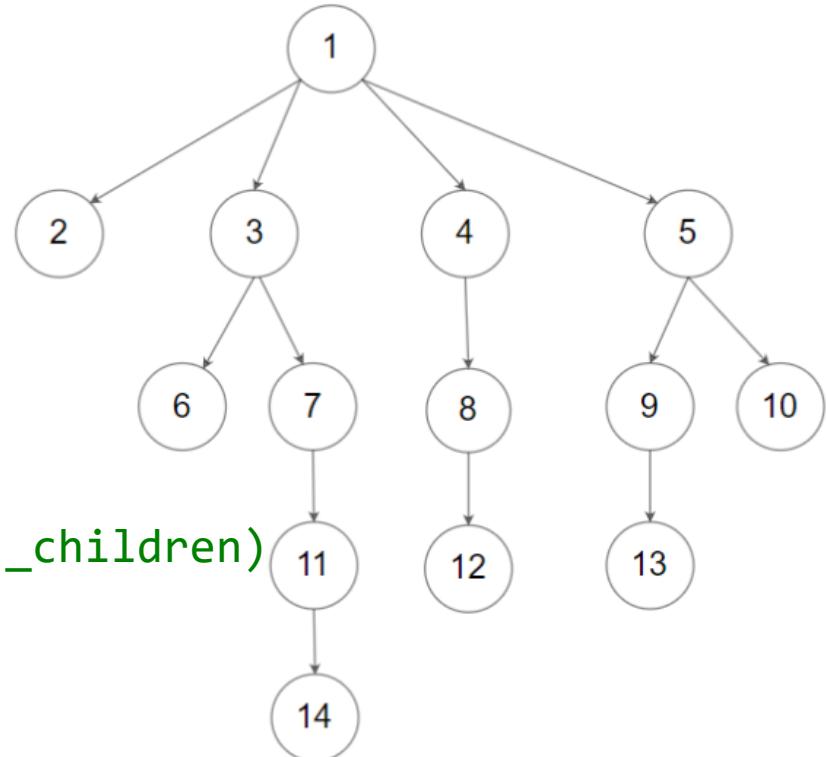
给你二叉树的根节点 root，返回其节点值的 层序遍历。（即逐层地，从左到右访问所有节点）。

```
void dfs(TreeNode* node, int level, vector<vector<int>>& result) {  
    if (node == nullptr) return;  
  
    if (result.size() == level) {  
        result.push_back(vector<int>());  
    }  
    result[level].push_back(node->val);  
    dfs(node->left, level + 1, result);  
    dfs(node->right, level + 1, result);  
}
```



# 层序遍历→N叉树

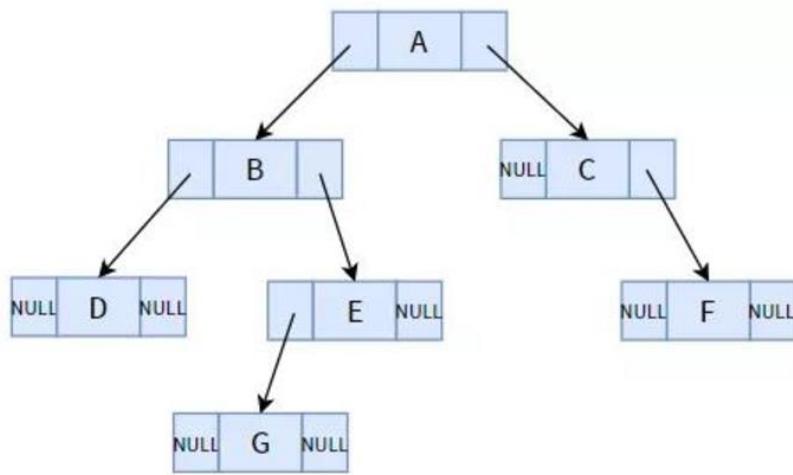
```
class Node {  
public:  
    int val;  
    vector<Node*> children;  
  
    Node() {}  
    Node(int _val) {  
        val = _val;  
    }  
    Node(int _val, vector<Node*> _children) {  
        val = _val;  
        children = _children;  
    }  
};
```



# 二叉链表的定义

若二叉链表中有 $n$ 个结点，则共有 $2n$ 个链域；其中 $n-1$ 不为空（ $n-1$ 条边），指向孩子。

$n+1$ 个空指针域难以避免



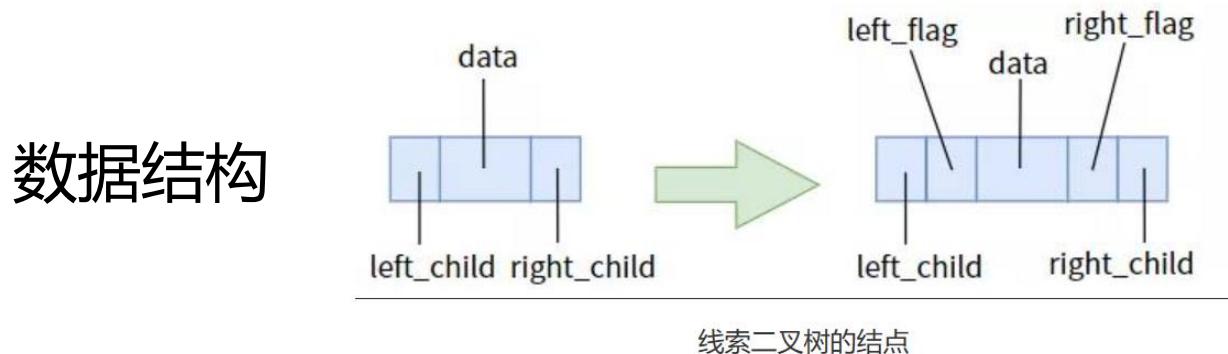
满二叉树时：

深度为 $h$ , 有 $2^h - 1 = n$  个节点  
有 $2^{h-1}$ 个叶子节点,  $2^h$  空域

如何充分利用空的链域？

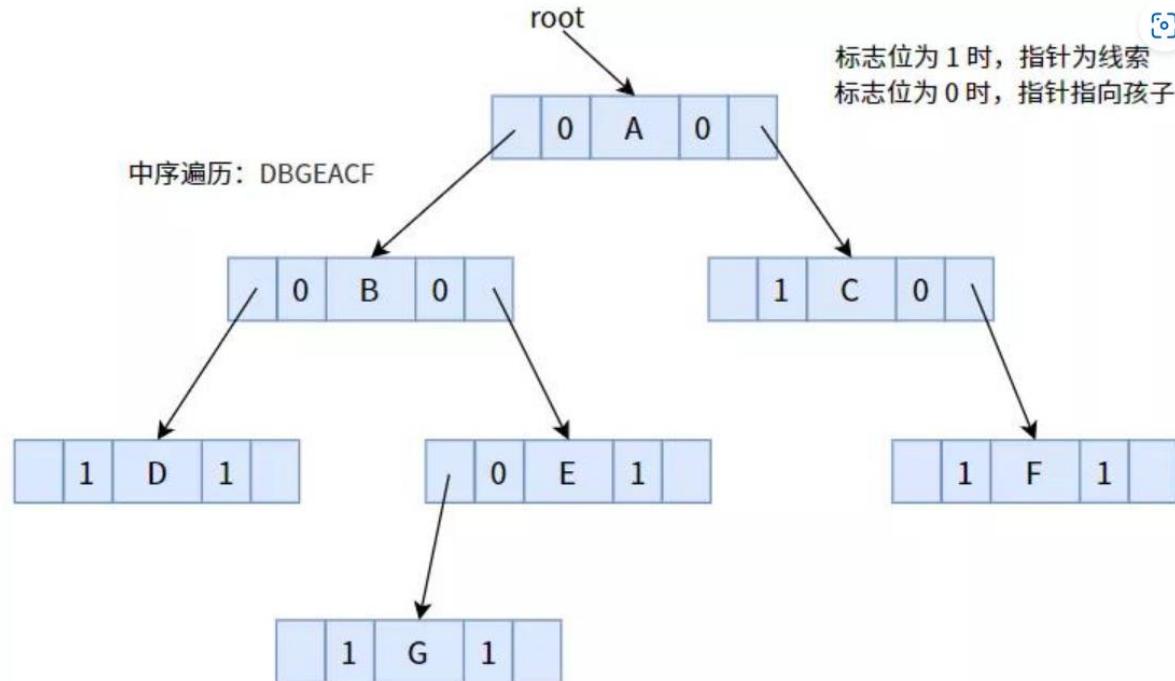
# 线索二叉树的定义

- 链域要么指向某个孩子节点，要么指向直接前驱/后继：
  - 需2个额外的标志（**Lflag**和**Rflag**），其占用的内存空间要少于指针变量。
  - C语言的布尔类型(**\_Bool**)

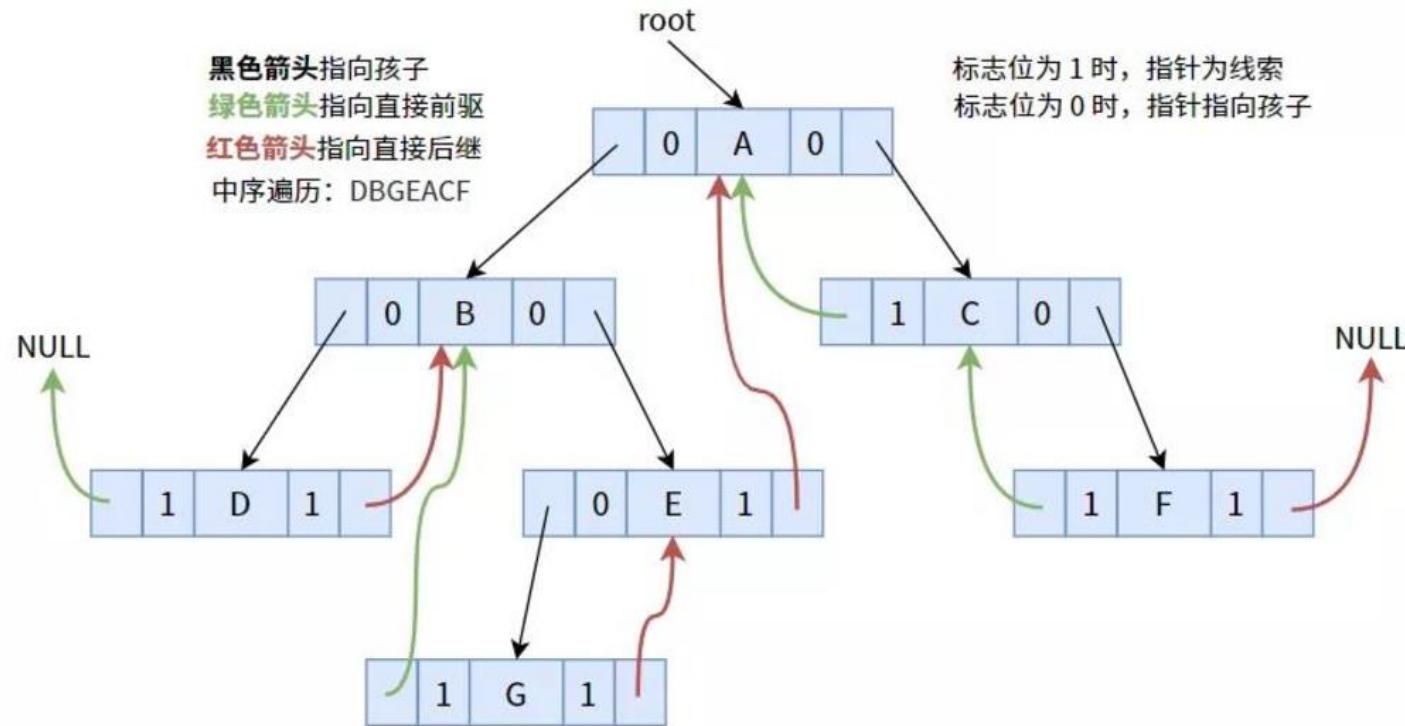


# 线索二叉树

中序遍历下，结点 E 的左孩子 G 就是结点 E 的直接前驱；  
结点 A 是其直接后继。



# 线索二叉树（示例）



不能直接通过二叉树得到的，二叉树中只有双亲和孩子结点之间的直接关系



# 线索二叉树（实现）

**思路：(按照遍历次序修改空域)**

记录 previous与 current 节点

**1. 初始化：** 定义全局变量 prev，初始值为 NULL。

**2. 递归遍历：**

- 如果根节点为空，则返回。
- 否则，先递归处理左子树。

**3. 处理当前节点：**

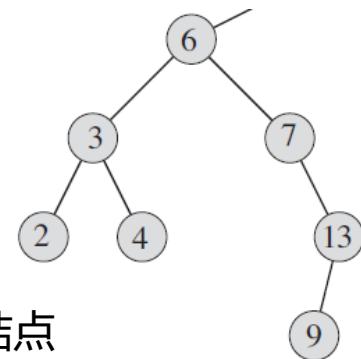
- 如果当前节点**没有左孩子**，设置左标志位，并将**左指针指向 prev**。
- 如果 **prev 不为空并且 prev 的右孩子为空**，设置右标志位，并将 prev 的右指针指向当前节点。

**4. 更新 prev：** 将 prev 更新为当前节点。

**5. 递归遍历右子树：** 继续对右子树进行同样的处理



# 线索二叉树（实现）



TreeNode \*prev = NULL; // 全局变量 prev 指针，指向刚访问过的结点

```
// 中序线索化函数
void inorder_th(TreeNode *root) {
    if(root == NULL) return;

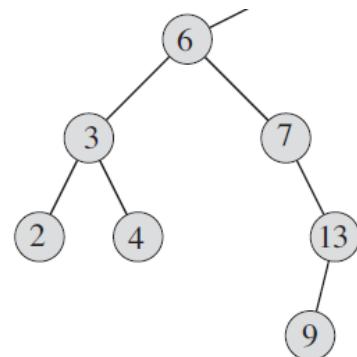
    inorder_th(root->left_child);
    if(root->left_child == NULL) {// 将左孩子指针改为指向前一个访问过的节点
        root->left_flag = 1;
        root->left_child = prev;}

    if(prev != NULL && prev->right_child == NULL) {
        prev->right_flag = 1; //将前一个节点的右孩子指针改为指向当前节点
        prev->right_child = root;}

    prev = root; // 更新 prev 指针
    // 中序遍历右子树
    inorder_th(root->right_child);
}
```



# 线索二叉树（遍历）



// 中序遍历算法

```
Function Inorder(root: TBTNode) {  
    p = First(root)  
    While p != NULL {  
        Print p.data  
        p = Next(p)  
    }  
}
```

// 中序序列下第一个结点访问算法

```
Function First(p: TBTNode) -> TBTNode {  
    While p.ltag == 0 {  
        p = p.lchild  
    }  
    Return p}
```

// 中序序列下后继结点访问算法

```
Function Next(p: TBTNode) -> TBTNode {  
    If p.rtag == 0 {  
        Return First(p.rchild)  
    }  
    Return p.rchild  
}
```

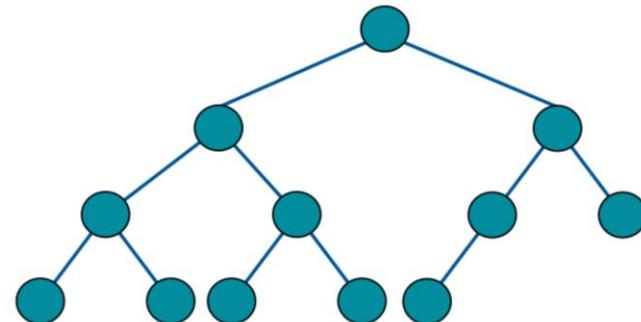


# 二叉树例题-1

- 例题：给定一颗完全二叉树，如何快速计算该二叉树有多少个节点

```
int countNodes(TreeNode* root) {  
    if (root == nullptr) return 0;  
    return 1 + countNodes(root->left)  
+countNodes(root->right);  
}
```

复杂度如何？  $O(N)$



如何优化？



2024/12/9

# 二叉树例题-1

- 例题：给定一颗完全二叉树，如何快速计算该二叉树有多少个节点

完全二叉树的左右子树中必有一颗满二叉树

满二叉树则，深度为k, 结点数为  $2^k - 1$

思路：转变为寻找满二叉树，直接公式求解

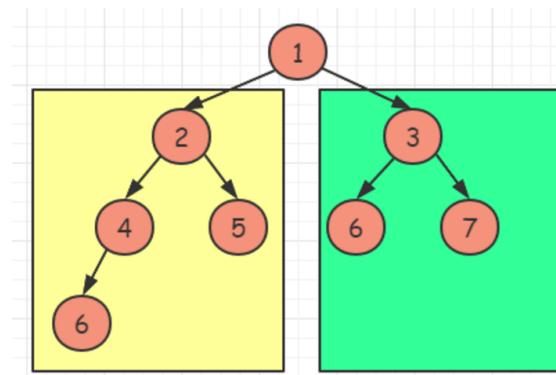
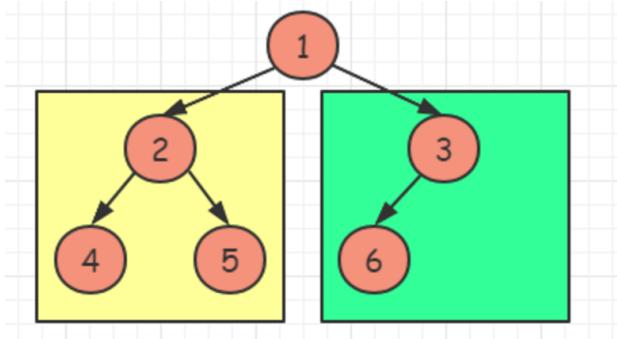
- 如果根节点的左子树深度等于右子树深度，则左子树为满二叉树
- 如果根节点的左子树深度大于右子树深度，则右子树为满二叉树



# 二叉树例题-1

- 例题：给定一颗完全二叉树，如何快速计算该二叉树有多少个节点

```
int countLevels(TreeNode* root) {  
    int levels = 0;  
    while (root) {  
        root = root->left; levels += 1;}  
return levels;}
```



```
int countNodes(TreeNode* root){  
    if(root == nullptr) return 0;  
    int left_levels = countLevels(root->left);  
    int right_levels = countLevels(root->right);  
    if(left_levels == right_levels){  
        return countNodes(root->right) + (1<<left_levels);  
    }else{  
        return countNodes(root->left) + (1<<right_levels);}
```

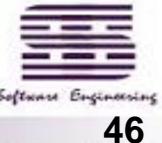


# 二叉树例题-1

- 例题：给定一颗完全二叉树，如何快速计算该二叉树有多少个节点

```
int countNodes(TreeNode* root) {  
    if (root == nullptr) return 0;  
    int lh=0,rh=0;  
    TreeNode* l = root->left;  
    TreeNode * r = root->right;  
    while(l!=nullptr) {lh++;l=l->left;}  
    while(r!=nullptr) {rh++;r=r->right;}  
    if (lh==rh){ return (1<<(lh+1))-1; }  
    return 1 + countNodes(root->left) +countNodes(root->right);  
}
```

复杂度 $\log N * \log N$



# 二叉树例题-2

- 给定满二叉树：填充每个节点的下一个右侧节点指针

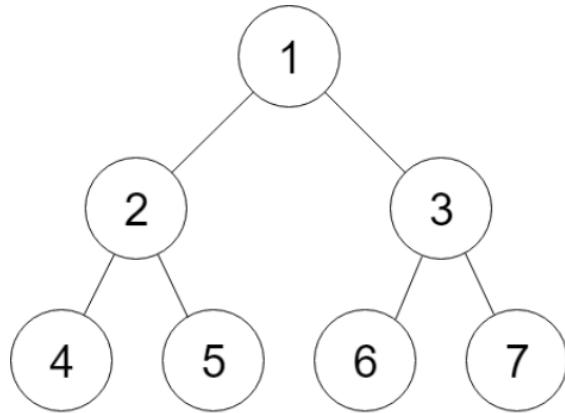


Figure A

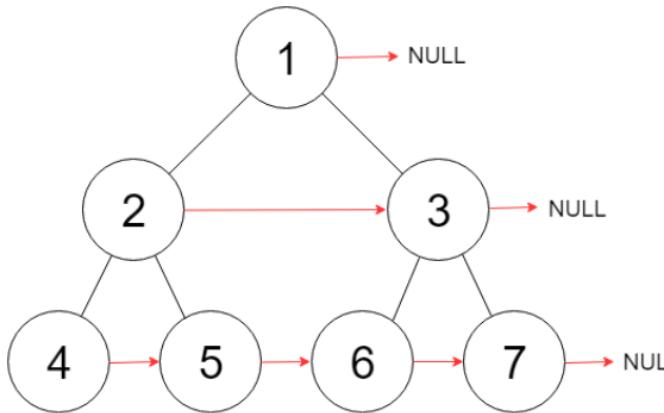


Figure B

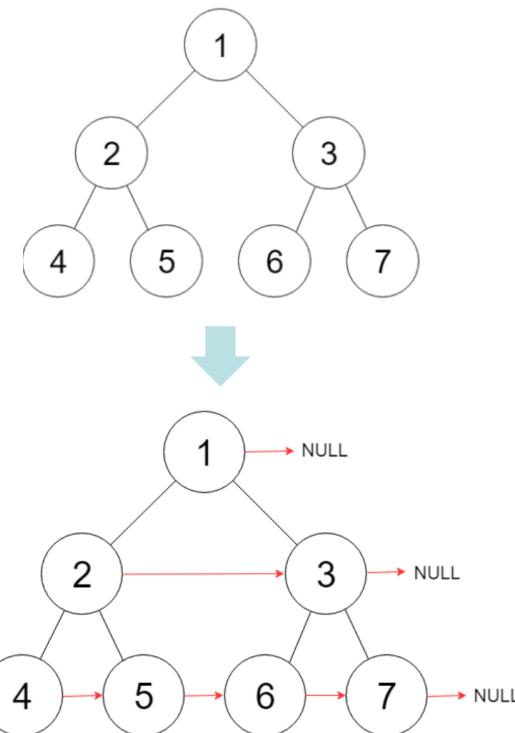
输入: root = [1,2,3,4,5,6,7]  
输出: [1,#,2,3,#,4,5,6,7,#]

- 树中节点的数量在  $[0, 2^{12} - 1]$  范围内
- $-1000 \leq \text{node.val} \leq 1000$

```
class Node {  
public:  
    int val;  
    Node* left;  
    Node* right;  
    Node* next;};
```

# 二叉树例题-2

- 例题：填充每个节点的下一个右侧节点指针



层序遍历，借助 $O(N)$   
的空间复杂度

# 二叉树例题-2

- 例题：填充每个节点的下一个右侧节点指针

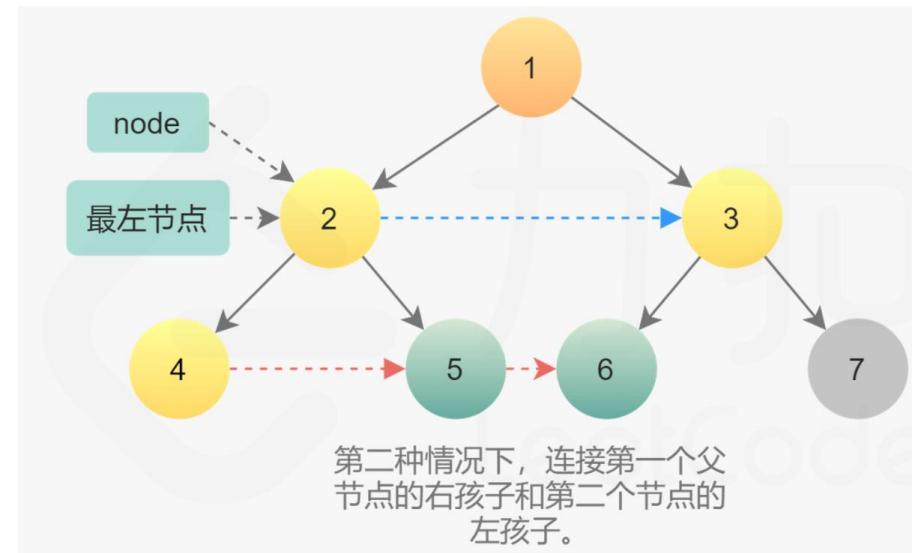
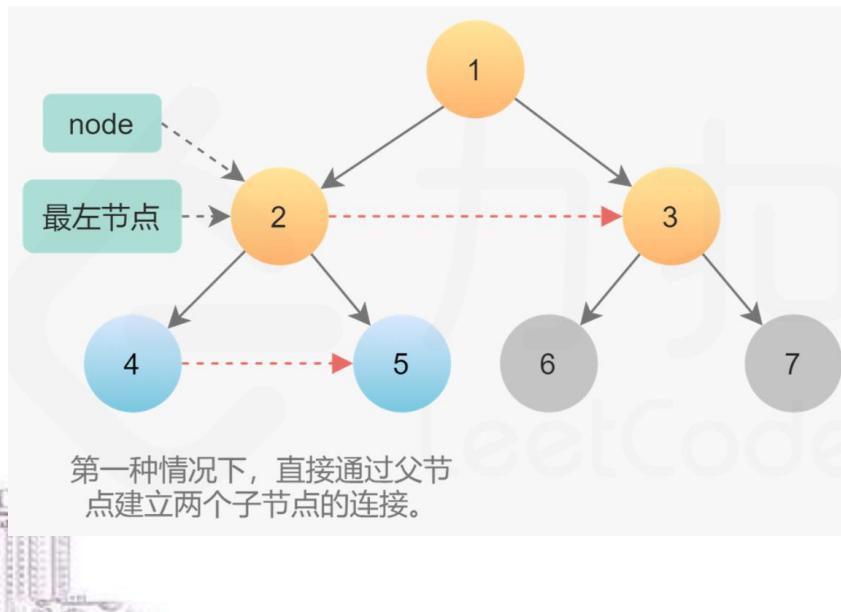
```
Node* connect(Node* root) {  
    queue<Node*> que;  
    if (root != nullptr) que.push(root);  
    while (!que.empty()) {  
        for(int i = que.size(); i > 0; --i) {  
            Node * tmp = que.front();  
            que.pop();  
            if(i>1) tmp->next = que.front();  
            if (tmp->left != nullptr) que.push(tmp->left);  
            if (tmp->right != nullptr) que.push(tmp->right);  
        }  
    }  
    return root;  
}
```



# 二叉树例题-2

- 例题：填充每个节点的下一个右侧节点指针

常量级空间复杂度？



# 二叉树例题-2

- 例题：填充每个节点的下一个右侧节点指针

思路：

## 1. 初始化

- 从根节点开始，无需处理第0层单个节点。

## 2. 逐层处理

- 对于每层节点，首先确保其**next指针已建立**。
- 处理当前层的同时，准备下一层的next指针构建。

注意：

- 我们需要在**第N-1层时**，为第N层构建**next指针**。



# 二叉树例题-2

- 例题：填充每个节点的下一个右侧节点指针

```
Node* connect(Node* root) {
    if (root == nullptr)  return root;
    // 从根节点开始
    Node* leftmost = root;
    while (leftmost->left != nullptr) {
        // 遍历层节点组织成的链表，为下一层的节点更新 next 指针
        Node* head = leftmost;
        while (head != nullptr) {
            head->left->next = head->right; // CONNECTION 1
            if (head->next != nullptr) { // CONNECTION 2
                head->right->next = head->next->left;
            }
            head = head->next; // 指针向后移动
        }
        leftmost = leftmost->left; // 去下一层的最左的节点
    }
    return root;
}
```

# 二叉排序树(二叉查找树)

- 二叉排序树(二叉查找树)

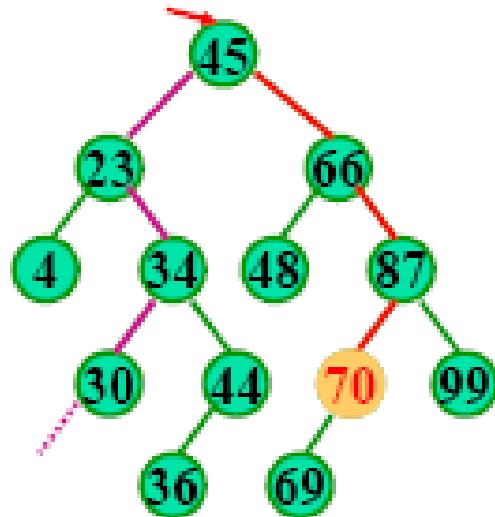
## Binary Sort/Search Tree

- 定义(递归)：或者是一棵空树或者是具有如下特性的二叉树：
  - 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；
  - 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；
  - 它的左、右子树也分别是二叉排序树。
- 中序遍历二叉排序树可得到一个关键字的有序序列。



# 二叉查找树的基本操作——查找

- 示例：



```
1 if  $x == \text{NIL}$  or  $k == x.key$ 
2   return  $x$ 
3 if  $k < x.key$ 
4   return TREE-SEARCH( $x.left, k$ )
5 else return TREE-SEARCH( $x.right, k$ )
```

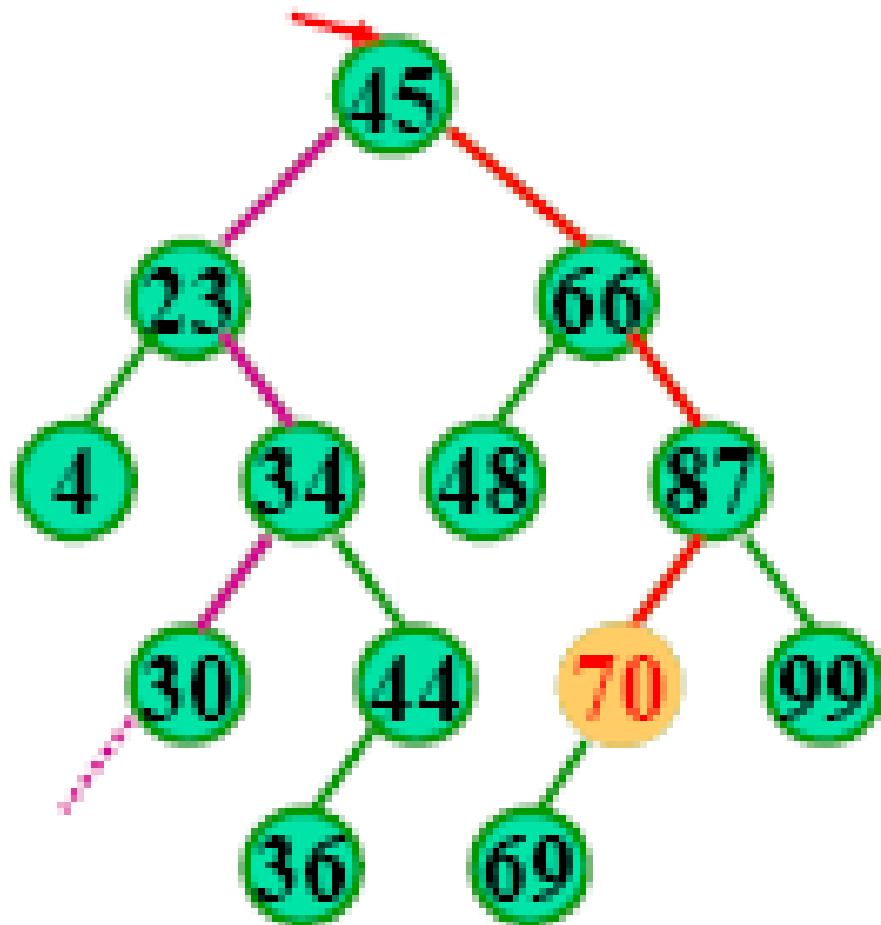
查找key=70?

查找key=28?



# 二叉查找树的基本操作——遍历

- 先序遍历:
- 中序遍历:
- 后序遍历:



# 二叉查找树的前驱和后继

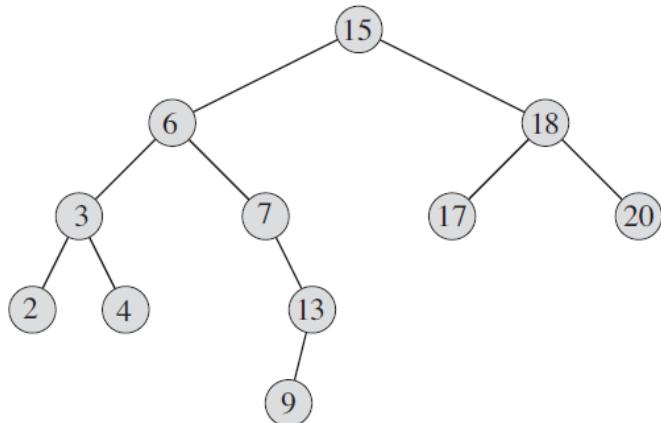
最大关键字和最小关键字

TREE-MAXIMUM( $x$ )

```
1 while  $x.right \neq \text{NIL}$ 
2      $x = x.right$ 
3 return  $x$ 
```

TREE-MINIMUM( $x$ )

```
1 while  $x.left \neq \text{NIL}$ 
2      $x = x.left$ 
3 return  $x$ 
```



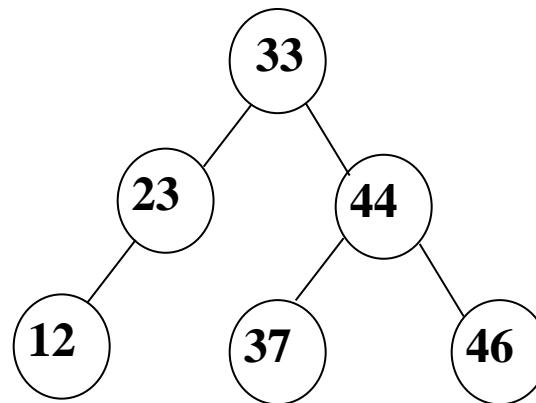
TREE-SUCCESSOR( $x$ )

```
1 if  $x.right \neq \text{NIL}$ 
2     return TREE-MINIMUM( $x.right$ )
3  $y = x.p$ 
4 while  $y \neq \text{NIL}$  and  $x == y.right$ 
5      $x = y$ 
6      $y = y.p$ 
7 return  $y$ 
```



# 二叉查找树的基本操作——插入

- 示例：从空树出发，待插的关键字序列为  
**33,44,23,46,12,37**
- 创建的树如图所示：

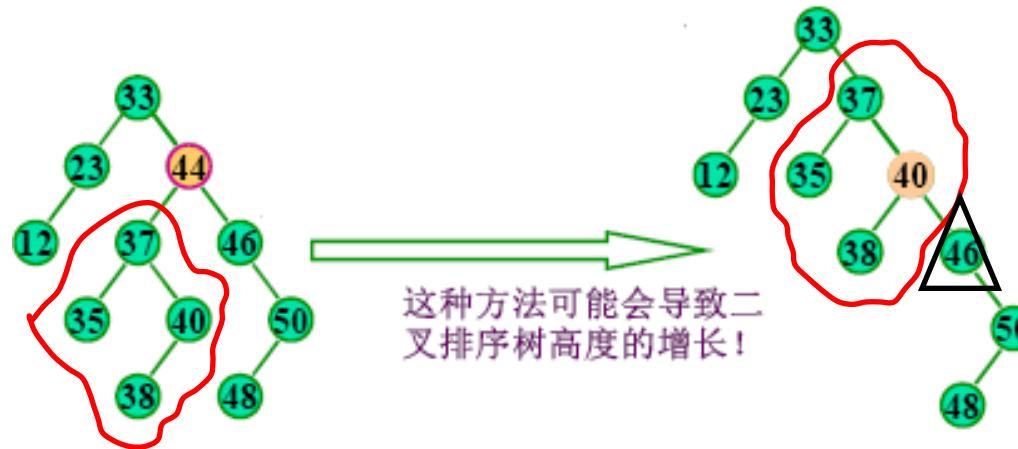


- 中序遍历二叉排序树可得到一个关键字的有序序列。
- 该例中，中序遍历结果为：**12, 23, 33, 37, 44, 46**

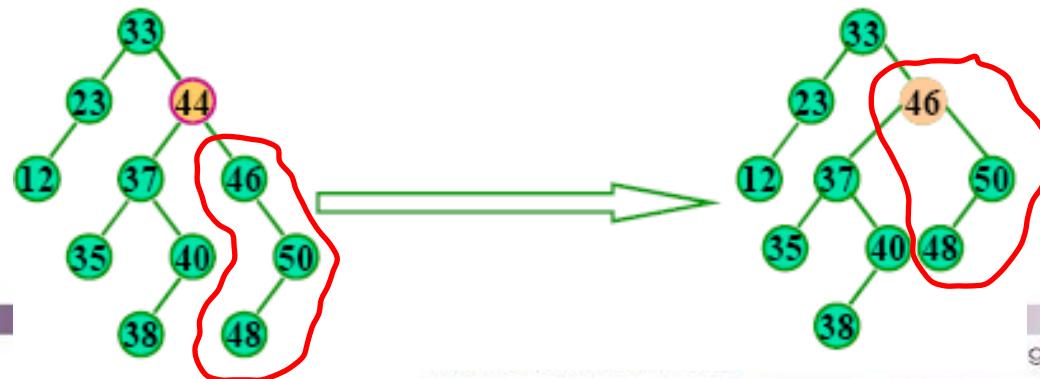


# 二叉查找树的基本操作——删除

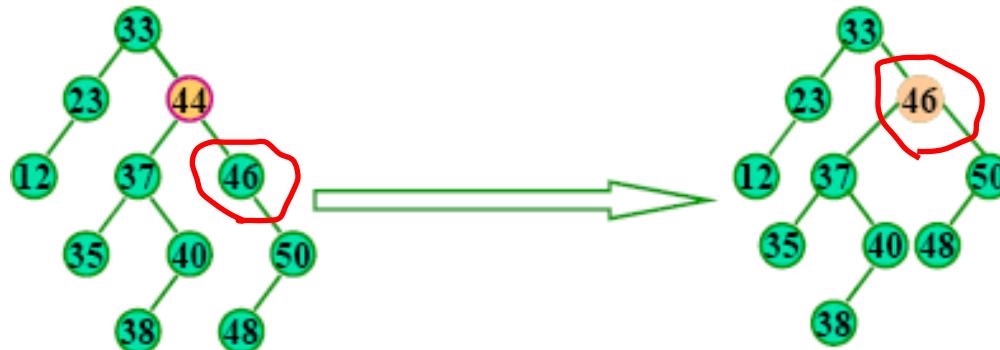
- 方法1：用其左孩子37替换



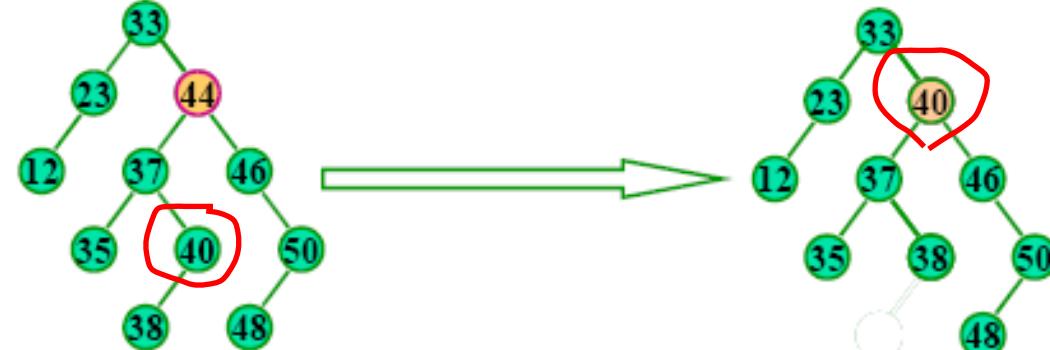
- 方法2：若其右孩子46无左孩子，则用右孩子46替换



- 方法3：（用中序遍历的直接后继替换）



- 方法4：（用中序遍历的直接前驱替换）



# 程序清单6-1

- 知识点：二叉查找树，红黑树，伸展树的综合应用。
- 程序的功能，输入和输出。
- 程序的结构：
  - 用到了哪些数据结构？
  - 包括哪些函数的声明和定义？



2024/12/9

```

#ifndef REDBLACK
#define RBONLY(x) x
#else
#define RBONLY(x)
#endif

#define BINTREE_STUFF(x) struct x *link[2] \
RBONLY(;int red)

typedef struct sBnode {
    BINTREE_STUFF(sBnode);
} Bnode;
typedef struct sBintree {
    Bnode *DummyHead;
    CompFunc Compare;
    int DuplicatesOk;
    size_t NodeSize;
} Bintree;

/* Our binary tree is made up of these */
typedef struct sMynode {
    BINTREE_STUFF(sMynode);
    char text[20];
} Mynode;

```

若当前为BST（即：未定义红黑树），则：

```

typedef struct sBnode {
    struct sBnode *link[2];
} Bnode;

```

```

typedef struct sBintree {
    Bnode *DummyHead;
    CompFunc Compare;
    int DuplicatesOk;
    size_t NodeSize;
} Bintree;

```

```

typedef struct sMynode {
    struct sMynode *link[2];
    char text[20];
} Mynode;

```

## Typedef 的常见用法

```
typedef struct sBnode {  
    struct sBnode *link[2];  
} Bnode;
```

```
typedef int IntArray[4];  
IntArray temp = {10, 20, 30, 40};
```

```
typedef int (*CompFunc)(void *node1, void *node2);  
int compareNodes(void *node1, void *node2) { // 比较逻辑}  
  
int main() {  
    CompFunc cmpFunc = compareNodes;  
}
```



# 二叉查找树——创建空树

- 创建一个带头节点的空二叉链表

```
/* Create an empty tree */
Bintree *NewBintree (Bnode *dummy,
                     CompFunc cf,
                     int dup_ok,
                     size_t node_size)
```

{

```
Bintree *t;
```

```
t = tmalloc(sizeof(Bintree));
t -> DummyHead = dummy;
t -> Compare = cf;
t -> DuplicatesOk = dup_ok;
t -> NodeSize = node_size;
```

```
return t;
```

}

```
typedef struct sMyNode {
    struct sMyNode *link[2];
    char text[20];
} MyNode;
```

```
Bnode *InitBintreeNode(size_t size)
```

{

```
    Bnode *n;
    n = tmalloc(size);
    n -> link[LEFT] = n -> link[RIGHT] = NULL;
    RBONLY(n -> red = 0);
    return n;
}
```

```
main(int argc, char **argv)
```

```
{.....
```

```
/* create a dummy node for the tree algorithms */
dummy = (MyNode *)
InitBintreeNode(sizeof(MyNode));
dummy->text[0] = 0; /* must contain valid data */

/* create a tree */
tree = NewBintree((Bnode *) dummy,
                  CompareFunc, 1, sizeof(MyNode));
```

}

Bnode类型的指针可以显式的转换为MyNode类型的指针！

# 二叉排序树——查找

- 在二叉查找树中查找指定节点

**Bnode \*FindBintree(Bintree \*t, Bnode \*n)**

- 输入：待查找的二叉树t，查找项n；
- 输出：若查找成功，则返回指向匹配节点的指针，否则，返回Null。
- 算法设计思路：
  - Step1：从根节点开始，将当前节点p指向根节点
  - Step2：若p非空，则将当前节点p与查找项n进行比较
  - Step3：根据比较结果进行相应处理。
    - 若 $p==n$ ，则返回匹配节点的指针；
    - 若为 $n < p$ ，则匹配项可能在当前节点的左子树上，因此将当前节点的左节点与查找项进行比较，重复Step2；
    - 若为 $n > p$ ，则匹配项可能在当前节点的右子树上，因此将当前节点的右节点与查找项进行比较，重复Step2；
  - Step4：如p为空，则表明二叉树t中没有与n相匹配的节点，则返回Null

- 在二叉查找树中查找指定节点

```

/* Find node n in tree t */
Bnode *FindBintree(Bintree *t, Bnode *n)
{
    Bnode *s;
    int dir;

    s = t -> DummyHead -> link[RIGHT];
    while (s != NULL) {
        dir = (t -> Compare) (n, s);
        if (dir == 0)
            return s;
        dir = dir < 0;
        s = s -> link[dir];
    }
    return NULL; /* no match */
}

```

```

#define LEFT      1
#define RIGHT     0

```

```

void FindString(Bintree *t, char *string)
{
    Mynode m, *r;
    strncpy(m.text, string, sizeof(m.text));
    m.text[sizeof(m.text) - 1] = 0;
    if ((r = (Mynode *) FindBintree(t,
        (Bnode *) &m)) == NULL)
        puts(" Not found.\n");
    else
        printf(" Found '%s'.\n", r -> text);
}

```



# 二叉排序树——插入

- 在二叉排序树中插入指定节点：

**int InsBintree (Bintree \*t, Bnode \*n)**

- 输入：待查找的二叉树t，待插入节点n；
- 输出：若t中已存在重复节点且不允许重复时，返回-1；否则，返回0。
- 算法设计思路：
  - 向下遍历树，查找插入位置（需知晓待插入位置的双亲节点的位置）。
  - 当找到一个空节点时，则用新节点替换该空节点。



## • 在二叉排序树中插入指定节点

```
/* Insert node n into tree t */
int InsBintree (Bintree *t, Bnode *n)
{
    int p_dir;
    Bnode *p, *s;

    /* Search until we find an empty arm. */
    p = t -> DummyHead;
    p_dir = RIGHT; /* direction from p to s */
    s = p -> link[RIGHT];
    while (s != NULL) {
        p = s;
        p_dir = (t -> Compare) (n, s);
        if (p_dir == 0 && t -> DuplicatesOk == 0)
            return TREE_FAIL; /* duplicate */
        p_dir = p_dir < 0;
        s = s -> link[p_dir];
    }

    /* Add the new node */
    p -> link[p_dir] = n;
    return TREE_OK;
}
```

```
int LoadString(Bintree *t, char *string)
{
    Mynode *m;

    m = (Mynode *)
    InitBintreeNode(sizeof(Mynode));
    strncpy(m->text, string, sizeof(m-
>text));
    m->text[sizeof(m->text) - 1] = 0;

    return InsBintree(t, (Bnode *) m);
}
```

# 二叉排序树——删除

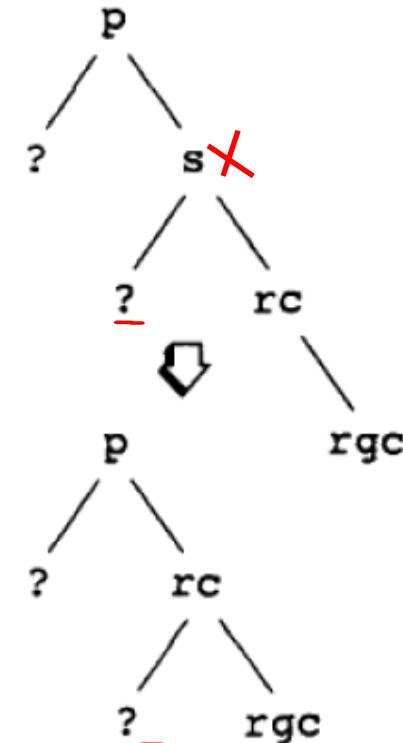
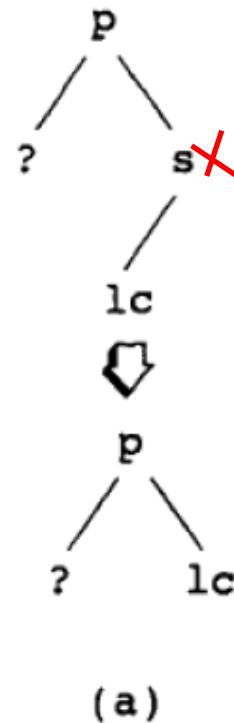
- 在二叉排序树中删除指定节点：

**Bnode \* DelBintree (Bintree \*t, Bnode \*n)**

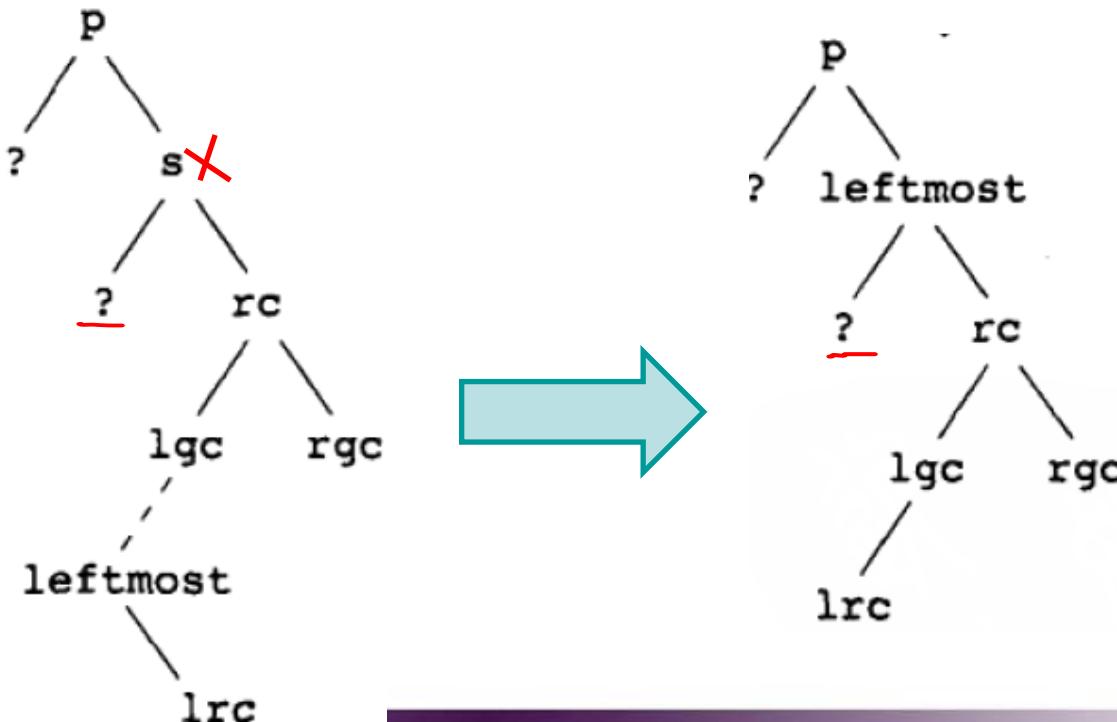
- 输入：待查找的二叉树t，待删除节点n；
- 输出：若t中存在与n相匹配的节点，则删除，并维持二叉查找树的特点，返回0；否则返回-1。
- 算法设计思路：
  - 向下遍历树，查找与待删除节点n相匹配的项。
  - 若找到匹配项，则删除它，并维持二叉查找树的特性（保证所有左子节点的key都比其双亲节点小，所有右子节点的key都比其双亲节点大）。
  - 关键：删除节点后，如何维持二叉查找树的特性？
  - 分三种情况讨论



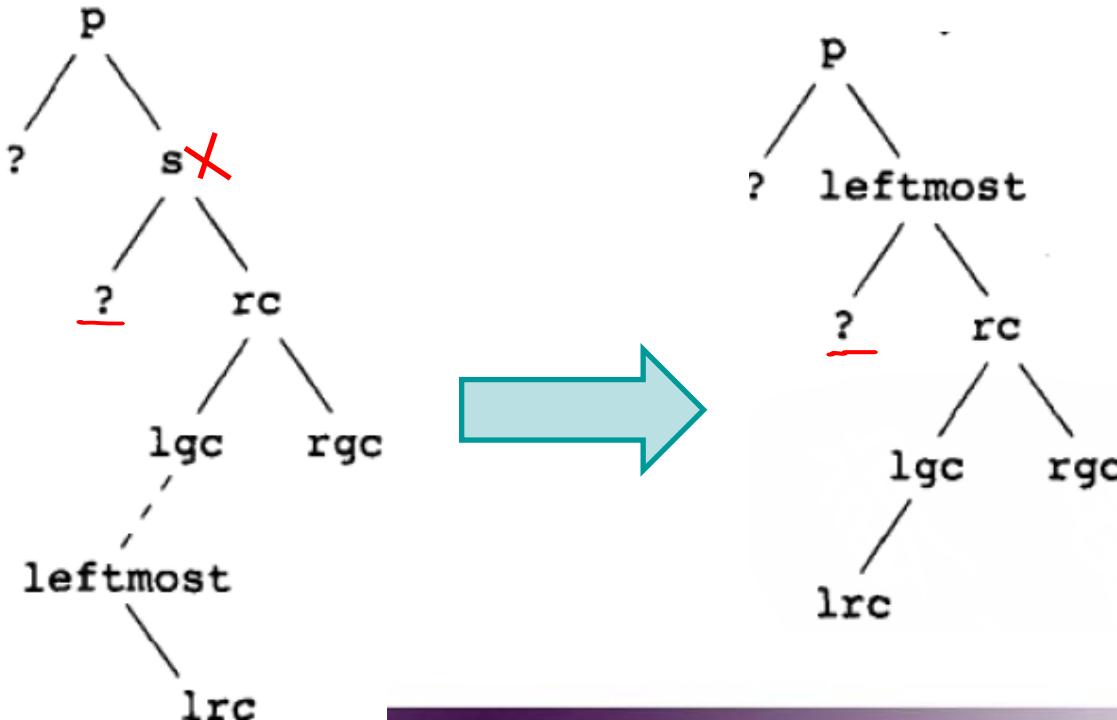
- 假设待删节点为 **s**
- 情况1：若 **s**缺少右子节点，  
，则用 **s**的左子节点替换 **s**；(用左孩子替换)
- 情况2：若 **s**的右子节点没  
有左子节点，那么就使 **s**  
的左子节点转变为 **s**右子  
节点的左子节点，并用 **s**  
的右子节点替换 **s**；(用中  
序遍历的直接后继替换)



- 情况3：若s的右子节点有它自己的两个子节点。则首先找到s之后的下一个最大的节点”**leftmost**”，并将s的两个子节点作为”**leftmost**”的子节点。（用中序遍历的直接后继替换）



- 情况3：若s的右子节点有它自己的两个子节点。则首先找到s之后的下一个最大的节点”**leftmost**”，并将s的两个子节点作为”**leftmost**”的子节点。（用中序遍历的直接后继替换）



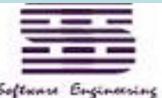
## • 在二叉树中删除指定节点

```
/* Delete node n from tree t. Returns a pointer to the deleted node */
Bnode * DelBintree (Bintree *t, Bnode *n)
{
    Bnode *p, *s, *save;
    int dir, dir_old;

    p = t -> DummyHead;
    s = p -> link[RIGHT];
    dir_old = dir = RIGHT;

    /* Look for a match */
    while (s != NULL && (dir = (t->Compare)(n, s)) != 0) {
        p = s;
        dir = dir < 0;
        dir_old = dir;
        s = p -> link[dir]; }
    if (s == NULL)
        return NULL; /* no match found */
    save = s;
    /* First case: if s has no right child, then replace s with s's left */
    if (s -> link[RIGHT] == NULL)
        s = s -> link[LEFT];
    /* Second case */
    else if (s -> link[RIGHT] -> link[LEFT] == NULL) {
        s = s -> link[RIGHT];
        s -> link[LEFT] = save -> link[LEFT]; }
    /* Final case: */
    else {
        Bnode *small;
        small = s -> link[RIGHT];
        while (small -> link[LEFT] -> link[LEFT])
            small = small -> link[LEFT];
        s = small -> link[LEFT];
        small -> link[LEFT] = s -> link[RIGHT];
        s -> link[LEFT] = save -> link[LEFT];
        s -> link[RIGHT] = save -> link[RIGHT]; }
    p -> link[dir_old] = s;
    RBONLY(s -> red = 0;)
    return save;}
```

```
void DeleteString(Bintree *t, char
*string)
{
    Mynode m, *n;
    strncpy(m.text, string, sizeof(m.text));
    m.text[sizeof(m.text) - 1] = 0;
    n = (Mynode *) DelBintree(t, (Bnode
*) &m);
    if (n)
        free (n);
    else
        fprintf(stdout, " Did not find
'%s'.\n", string);
}
```



# 二叉排序树——中序遍历

- 遍历二叉树

```
/* Recursive tree walk routines.  
The entry point is WalkBintree.  
It will do an in order traversal of  
the * tree, call df() for each  
node and leaf. */
```

```
void rWalk(Bnode *n, int level,  
DoFunc df)  
{  
    if (n != NULL) {  
        rWalk(n -> link[LEFT], level +  
1, df);  
        df(n, level);  
        rWalk(n -> link[RIGHT], level +  
1, df);  
    }  
}
```

```
int WalkBintree(Bintree *t, DoFunc  
df)  
{  
    if (t -> DummyHead -> link[RIGHT]  
== NULL) {  
        fputs("Empty tree\n", stdout);  
        return TREE_FAIL;  
    }  
  
    rWalk(t -> DummyHead ->  
link[RIGHT], 0, df);  
    return TREE_OK;  
}
```

# 二叉排序树例题-1

- 例题：给定一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。
  - 如果该二叉树的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；它的左右子树也为二叉搜索树

设计一个递归函数 `judge(root, lower, upper)` 来递归判断



# 二叉排序树例题-1

- 例题：给定一个二叉树的根节点 **root**，判断其是否是一个有效的**二叉搜索树**。

```
bool jug(TreeNode* root, long lower, long upper) {  
    if (root == nullptr) {  
        return true;  
    }  
    if (root -> val <= lower || root -> val >= upper) {  
        return false;  
    }  
    return  
        jug(root -> left, lower, root -> val)  
        && jug(root -> right, root -> val, upper);  
}
```



# 二叉排序树例题-1

- 例题：给定一个二叉树的根节点 `root`，判断其是否是一个有效的**二叉搜索树**。
  - 中序遍历的性质 -> 有序

```
bool inorder(TreeNode * cur){  
    bool l=true,r=true;  
    if (cur->left) {  
        l = inorder(cur->left);  
        if(!l) return false;  
    }  
    if (prev!=nullptr && cur->val <= prev->val) return false;  
    prev = cur;  
    if(cur->right) r = inorder(cur->right);  
    return l&&r;  
}
```



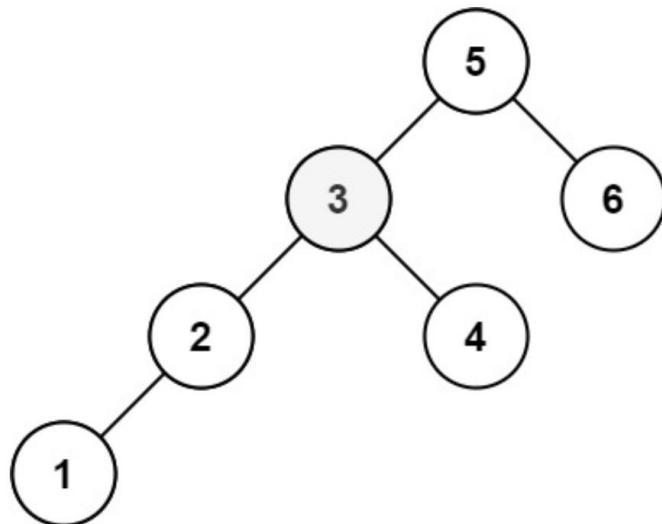
# 二叉排序树例题-1

- 例题：给定一个二叉树的根节点 `root`，判断其是否是一个有效的**二叉搜索树**。
  - 中序遍历的性质 -> 有序

```
while (!stack.empty() || root != nullptr) {  
    while (root != nullptr) {  
        stack.push(root);  
        root = root -> left;  
    } // 借助栈来保存中序遍历的节点  
    root = stack.top();  
    stack.pop();  
    if (root -> val <= inorder) return false;  
    inorder = root -> val;  
    root = root -> right; // 右子树仍要进入左子树节点压栈  
}
```

# 二叉排序树例题-2

- BST中第K小的元素：给定一个根节点 `root`，返回第K小的元素（从1计数）。



输入： `root = [5,3,6,2,4,null,null,1]`,  $k = 3$

输出： 3



# 二叉排序树例题-2

- BST中第K小的元素：给定一个根节点 `root`，返回第K小的元素（从1计数）。
  - 中序遍历的性质 ->有序

```
class Solution {  
public:  
    int kthSmallest(TreeNode* root, int k) {  
        this->k = k;  
        inorder(root);  
        return res;}  
  
private:  
    int res, k;  
    void inorder(TreeNode*cur){  
        if(cur->left != nullptr) inorder(cur->left);  
        if (--k==0) {res = cur->val; return;}  
        if(cur->right != nullptr) inorder(cur->right);  
    };
```

# 二叉排序树例题-2

- BST中第K小的元素：给定一个根节点 `root`，返回第K小的元素（从1计数）。
  - 二分查找

步骤: 开始于根节点 `node`, 计算其左子树的节点数 `left`.

**判断1:**

若  $left < k - 1$ , 则第k小元素位于右子树。

**操作:** 将 `node` 设为其右子节点, 更新  $k = k - left - 1$ , 继续搜索

**判断2:**

若  $left == k - 1$ , 找到第k小元素。结果: 返回当前节点 `node`.

**判断3:**

若  $left > k - 1$ , 则第k小元素位于左子树。

**操作:** 将 `node` 设为其左子节点, 继续搜索。



# 二叉排序树例题-2

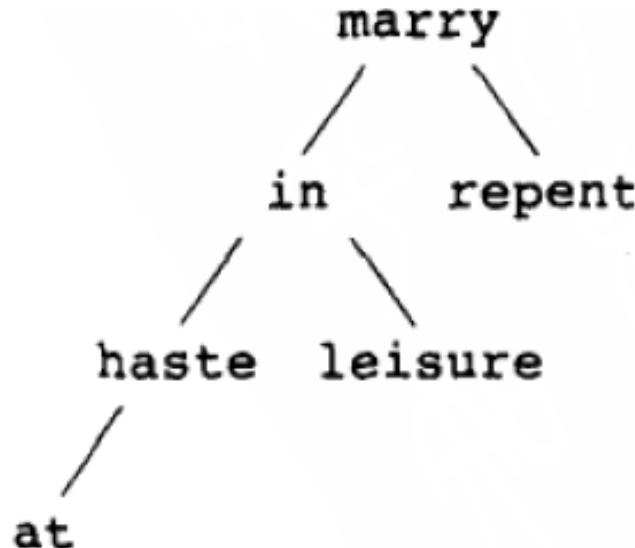
- BST中第K小的元素：给定一个根节点 `root`，返回第K小的元素（从1计数）。
  - 二分查找

```
int kthSmallest(int k) {  
    TreeNode *node = root;  
    while (node != nullptr) {  
        int left = getNodeNum(node->left);  
        if (left < k - 1) {  
            node = node->right;  
            k -= left + 1;  
        } else if (left == k - 1) {  
            break;  
        } else {  
            node = node->left; }}  
    return node->val;}
```

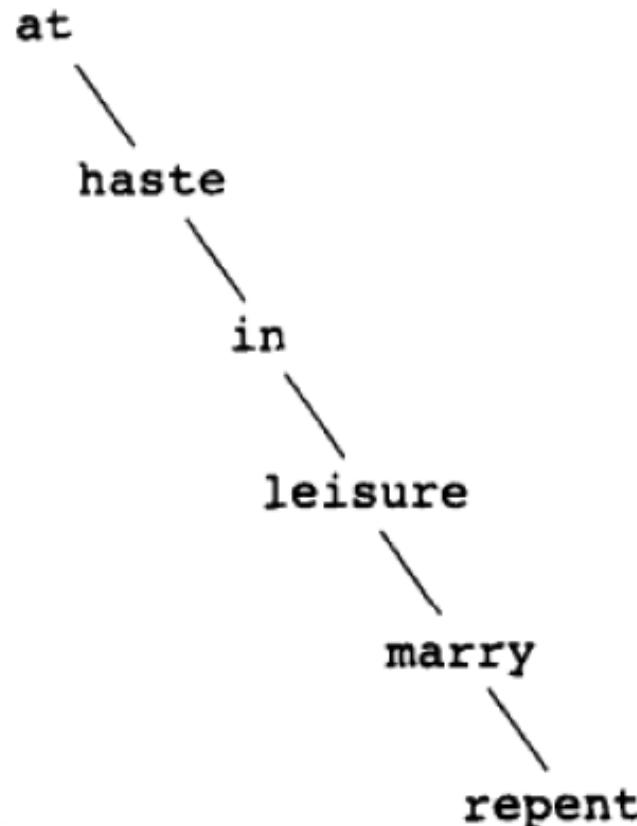


# 回顾：二叉查找树的性能

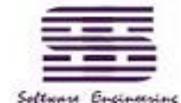
- 构造的二叉查找树的形状依赖于数据项，且依赖于它们加载的顺序。
- 示例1：加载字符串 “**marry in haste, repent at leisure**”



- 示例2：加载字符串 “at haste in leisure marry repent”



- 示例3：加载字符串 “at repent haste marry in leisure”



- 遗留问题：若构造的二叉查找树平衡性很差（即失衡），该如何解决？
  - AVL（平衡二叉树）
  - 红黑树
  - 伸展树



2024/12/9

# 两种应用场景下的平衡二叉树

- 创建平衡二叉树
  - 创建高度平衡的二叉树 (**height-balanced tree**)：假定所有数据项使用的频率是相同的。
    - AVL树
    - 红黑树
  - 创建加权的平衡二叉树 (**weight-balanced tree**)：将常用的数据项放在树中接近根部的位置；
    - 伸展树



# (1) AVL树 (*Adel'son-Vel'skii & E. M. Landis tree*)

- 定义(递归): 或者是一棵空树, 或者是具有如下特性的二叉查找树
  - 左子树和右子树的高度最多相差1;
  - 它的左、右子树也分别是平衡二叉树。
- 二叉树上结点的平衡因子BF: 该结点的左子树的高度减去它的右子树的高度。



# (1) AVL树

- AVL树的深度和 $\log_2 N$ 是同数量级的(其中的N为结点个数)。
- 如何证明

设  $f_n$  为高度为 n 的 普通树所包含的最少节点数  
对于普通的树，则  $f_n = f_{n-1} + 1$

对于高度为n的AVL树，可以包含两个树高为n-1的左右子树，或者n-1、n-2的左右子树 ( $n > 2$ )  
但左右子树高差小于等于1



# (1) AVL树

- **AVL树的深度和 $\log_2 N$ 是同数量级的(其中的N为结点个数)。**
- 如何证明

设  $f_n$  为高度为  $n$  的 AVL 树所包含的最少节点数

$$f_n = \begin{cases} 1, & n = 1 \\ 2, & n = 2 \\ f_{n-1} + f_{n-2} + 1, & n > 2 \end{cases}$$

$$f_n = \frac{5 + 2\sqrt{5}}{5} \left(\frac{1 + \sqrt{5}}{2}\right)^{n-1} + \frac{5 - 2\sqrt{5}}{5} \left(\frac{1 - \sqrt{5}}{2}\right)^{n-1} - 1$$

$$n < 1 + \log_{\frac{1+\sqrt{5}}{2}}(f_n + 1) < \alpha \log_2(f_n + 1) + c$$



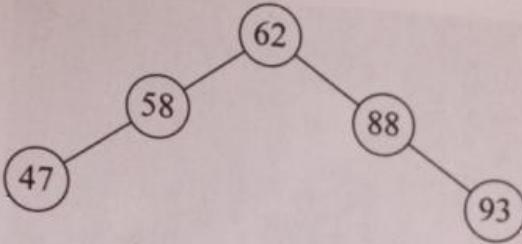


图1 平衡二叉树

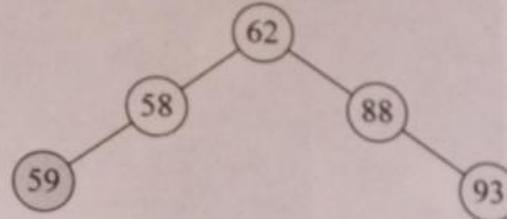


图2 不是平衡二叉树

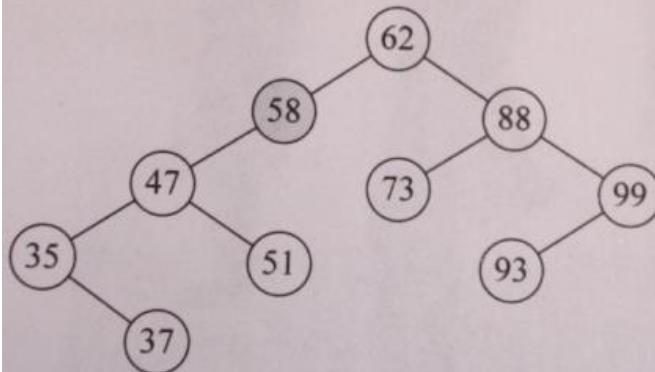


图3 不是平衡二叉树

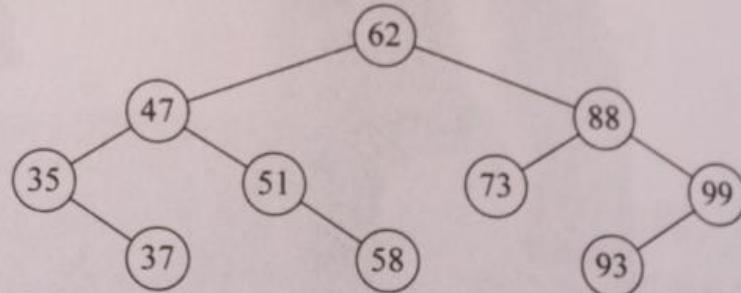
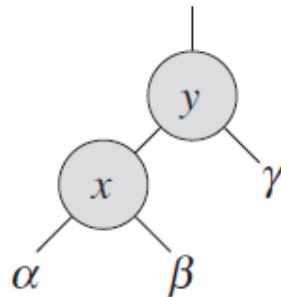
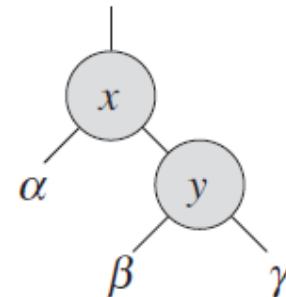


图4 平衡二叉树

# 左旋、右旋操作



LEFT-ROTATE( $T, x$ )  
-----  
RIGHT-ROTATE( $T, y$ )



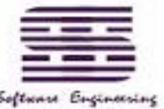
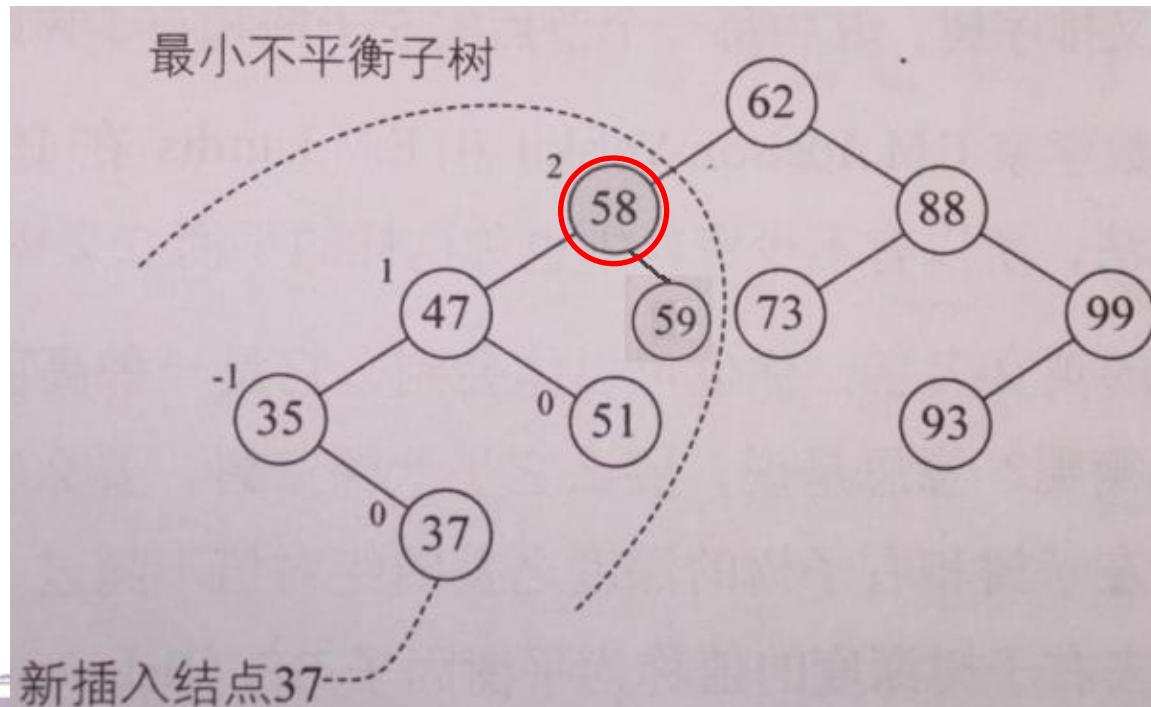
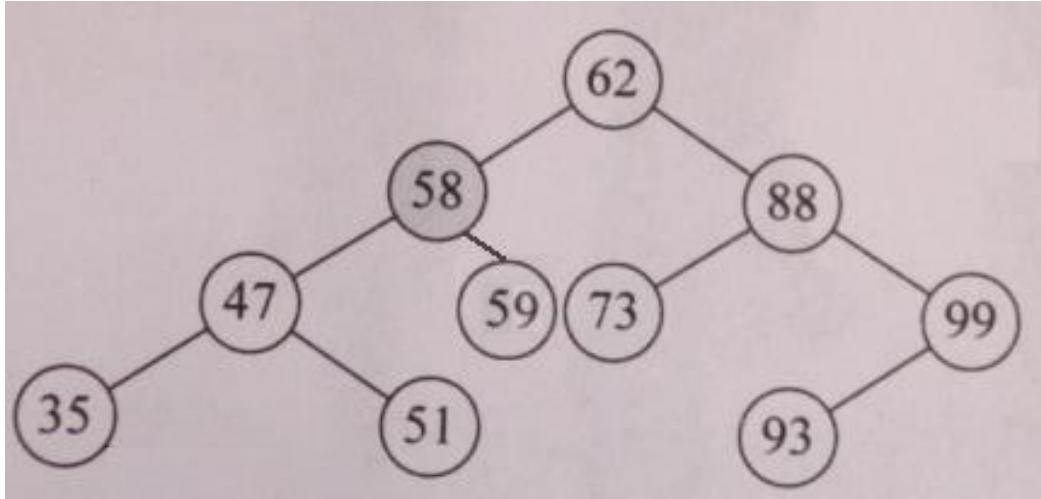
LEFT-ROTATE( $T, x$ )

```
1   $y = x.right$           // set y
2   $x.right = y.left$     // turn y's left subtree into x's right subtree
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$ 
5   $y.p = x.p$            // link x's parent to y
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$            // put x on y's left
12  $x.p = y$ 
```



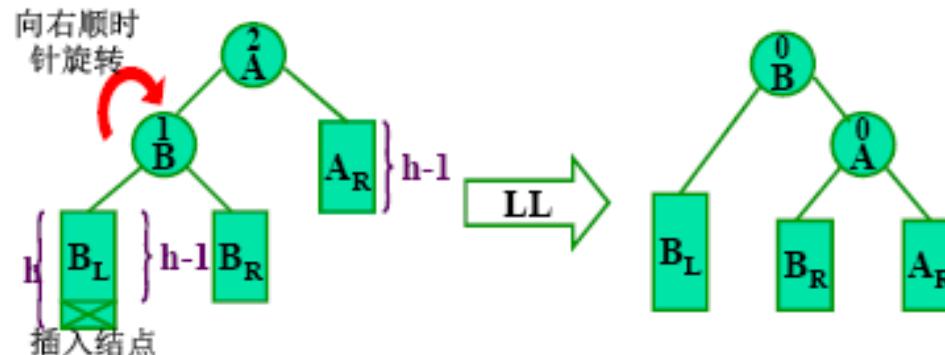
- 如何构造AVL树?
  - 与构造二叉排序树（不断插入新节点）的过程相似，不同之处在于：在出现失衡，应立即采取措施进行调整。
  - 关键问题为：如何进行调整节点以维持高度的平衡。
- 如何调整节点以维持高度的平衡?
  - Step1：找到**不平衡**（即左右子树高度差距大于1）的**最小子树的根节点**（记为A）
  - Step2：依据新插入节点与节点A的相对位置，进行相应处理。
    - **LL型**：在A结点的左孩子的左子树上插入结点，导致A结点失去平衡。
    - **RR型**：在A结点的右孩子的右子树上插入结点，导致A结点失去平衡。
    - **LR型**：在A结点的左孩子的右子树上插入结点，导致A结点失去平衡。
    - **RL型**：在A结点的右孩子的左子树上插入结点，导致A结点失去平衡。

关键点：找到失衡节点A；并定位 新插入节点与节点A的相对位置关系，以便准确判断失衡的类型



# LL型失衡的调整规则

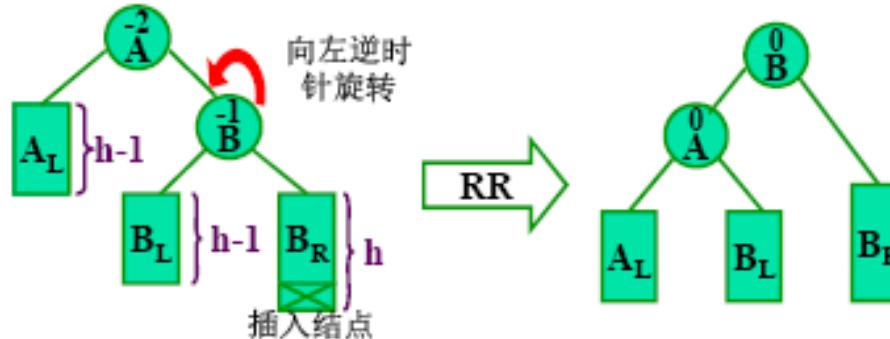
- 假设A是由于在二叉排序树上插入结点而失去平衡的最小子树根结点
- LL型失衡: 在A结点的左孩子的左子树上插入结点,导致A结点失去平衡.



单向右旋平衡处理(LL型)

# RR型失衡的调整规则

- 假设A是由于在二叉排序树上插入结点而失去平衡的最小子树根结点
- RR型失衡:** 在A结点的右孩子的右子树上插入结点,导致A结点失去平衡.



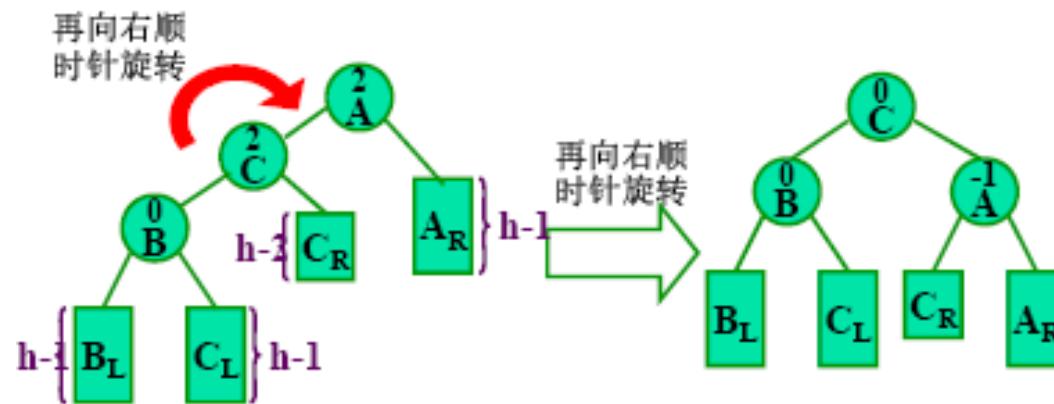
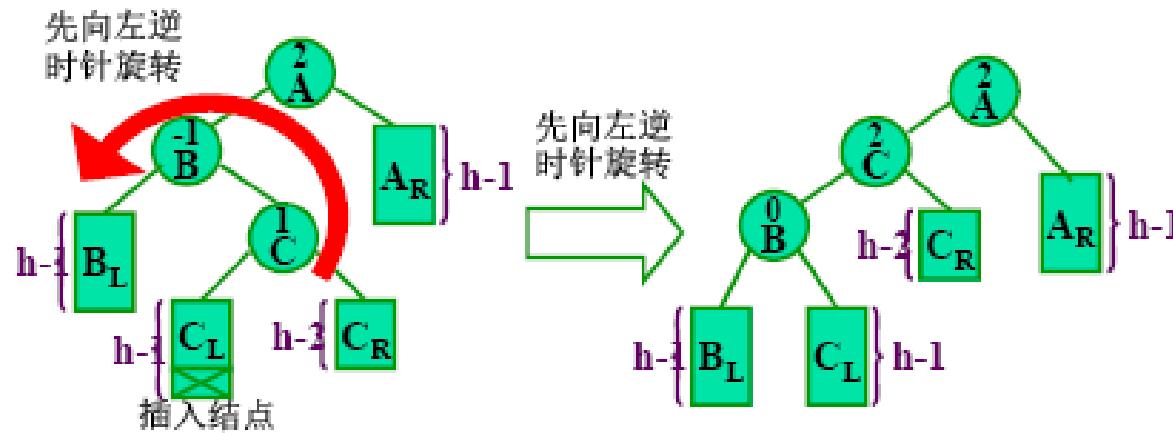
单向左旋平衡处理(RR型)

# LR型失衡的调整规则

- 假设A是由于在二叉排序树上插入结点而失去平衡的最小子树根结点
- LR型失衡: 在A结点的左孩子的右子树上插入结点,导致A结点失去平衡.
- 双向旋转: 先左后右平衡处理型
  - 先转换为LL型失衡
  - 再解决LL型失衡



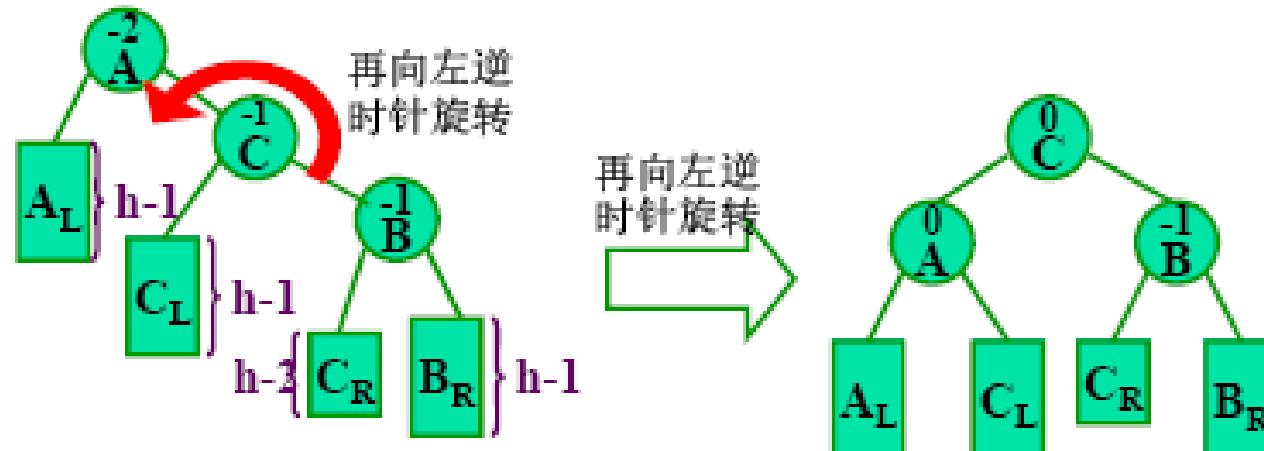
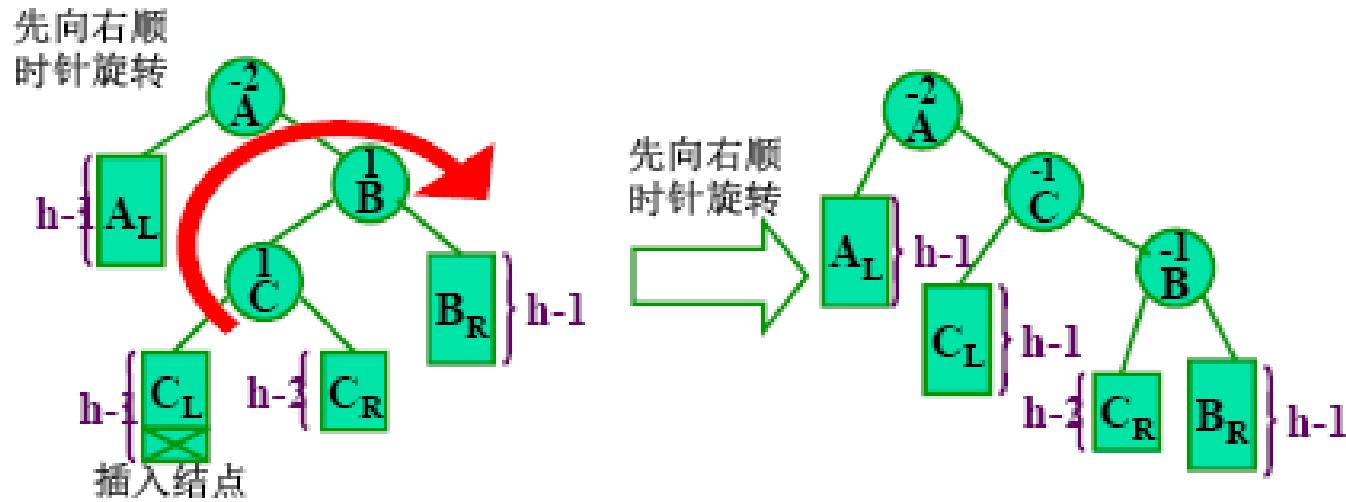
2024/12/9



# RL型失衡的调整规则

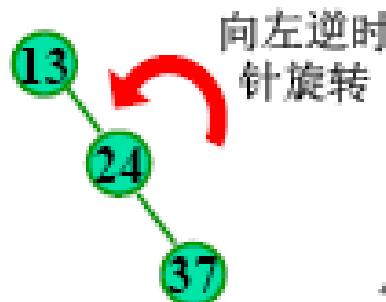
- 假设**A**是由于在二叉排序树上插入结点而失去平衡的最小子树根结点
- RL型失衡: 在**A**结点的右孩子的左子树上插入结点,导致**A**结点失去平衡.
- 双向旋转: 先右后左平衡处理型
  - 先转换为**RR**型失衡
  - 再解决**RR**型失衡



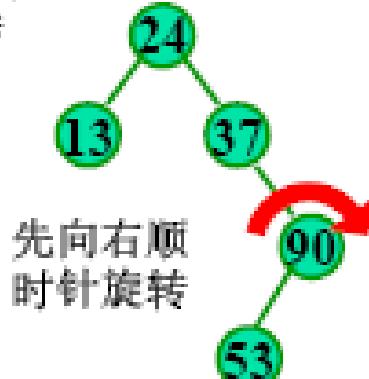


# 示例1：构造AVL树

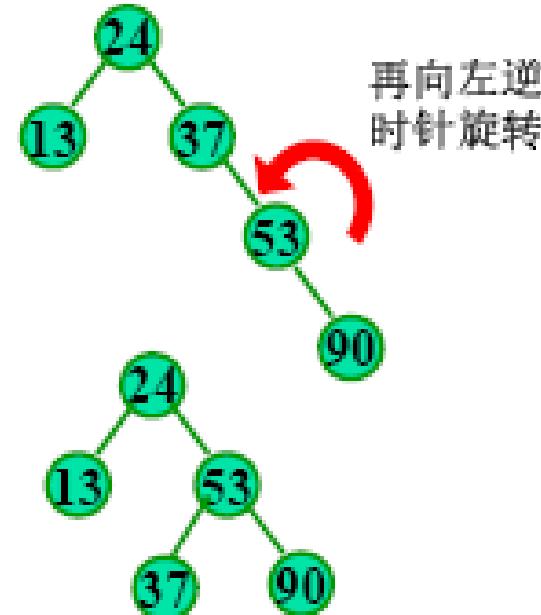
- 从空树出发，待插的关键字序列为  
**13,24,37,90,53**



结点13的BF值由-1变成-2，出现不平衡！

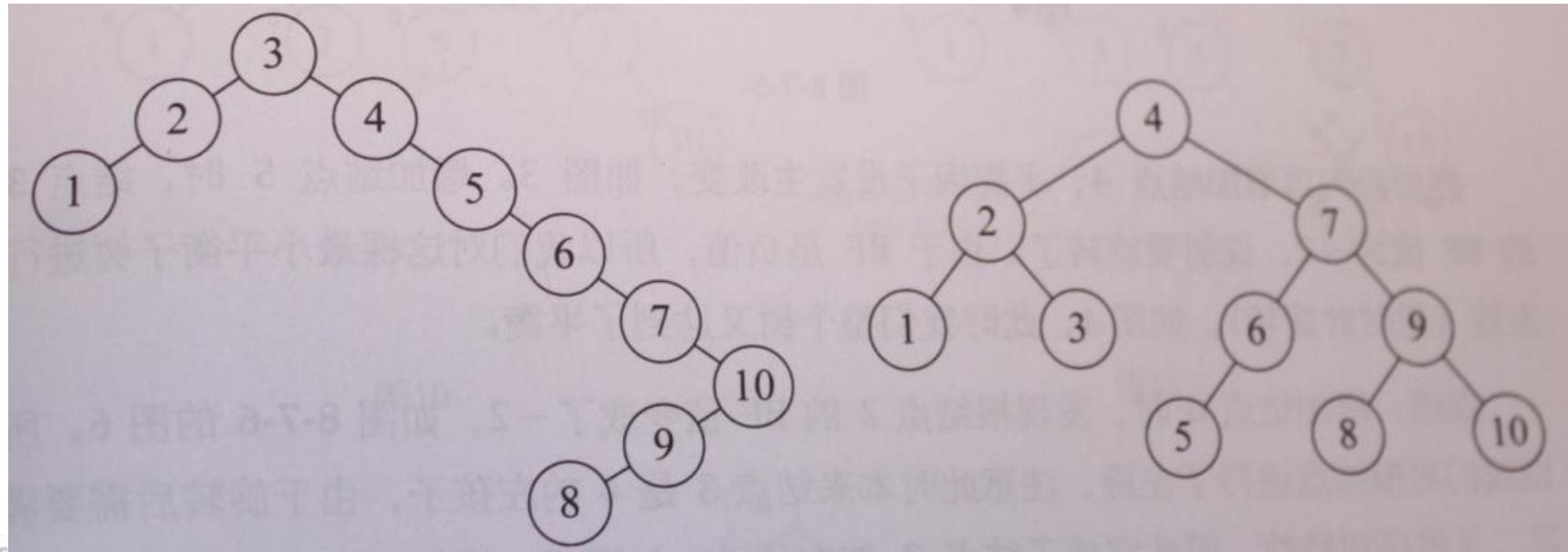


结点37的BF值由-1变成-2，出现不平衡！



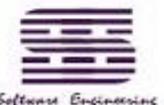
# 示例2：构造AVL树

- 从空树出发，待插的关键字序列为  
**03,02,01,04,05,06,07,10,09,08**



二叉查找树

AVL树



Software Engineering

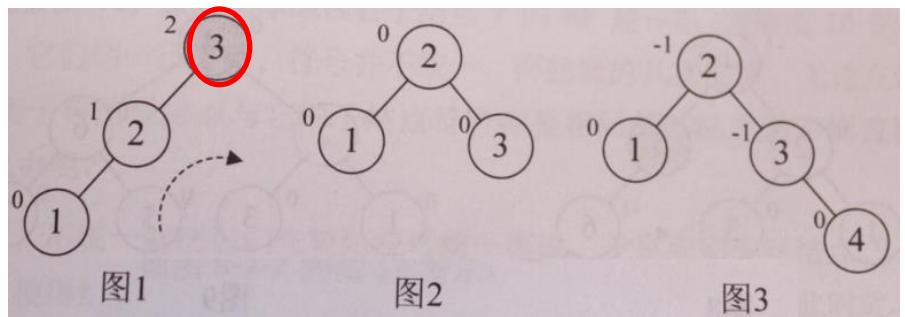


图2

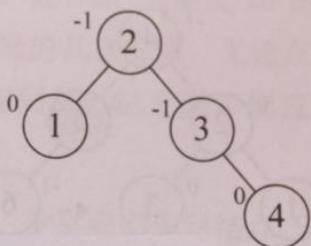


图3

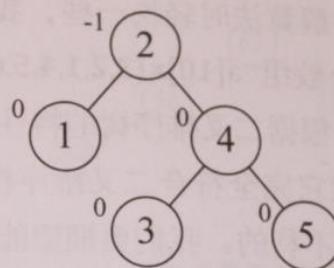
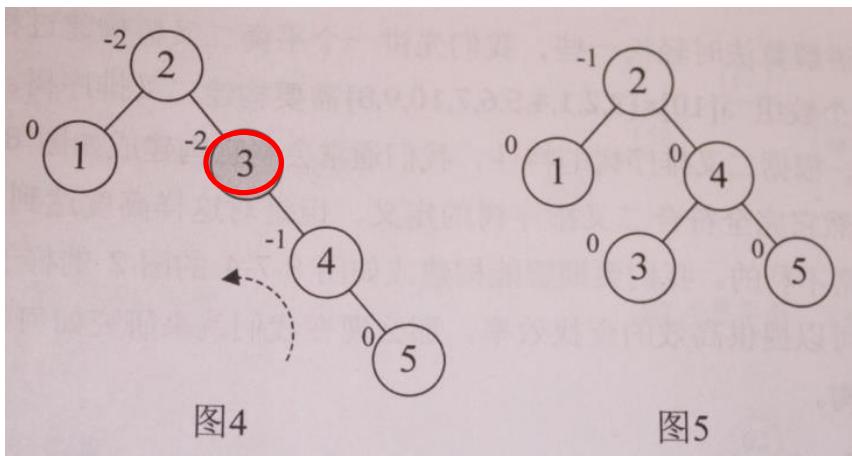
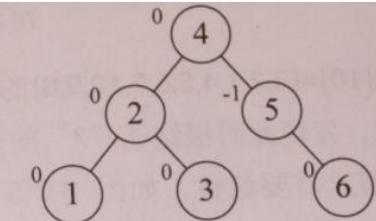
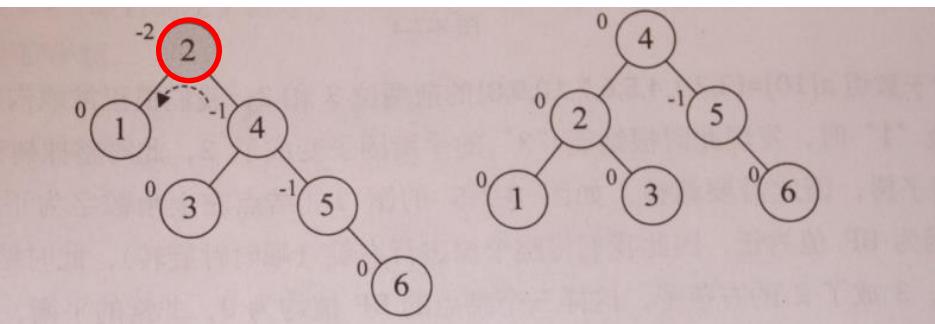


图5

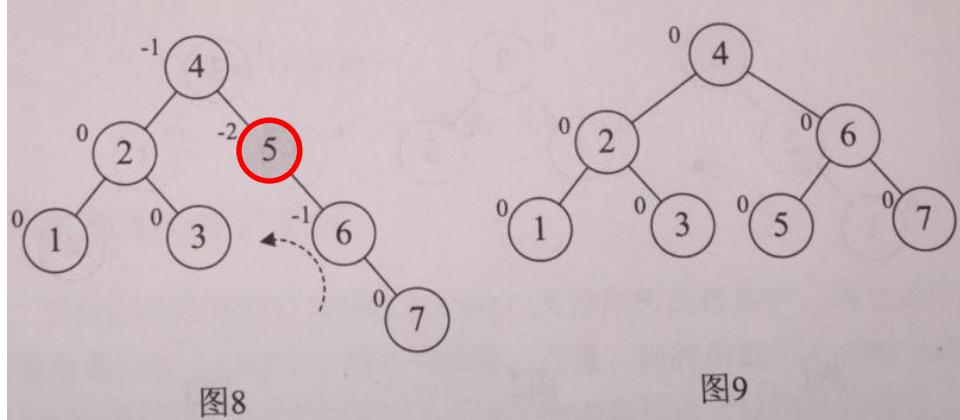


图9

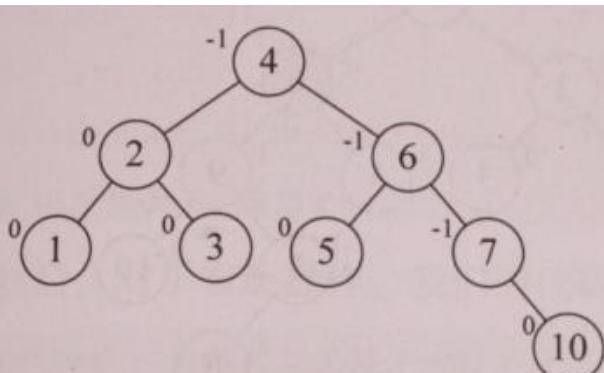


图10

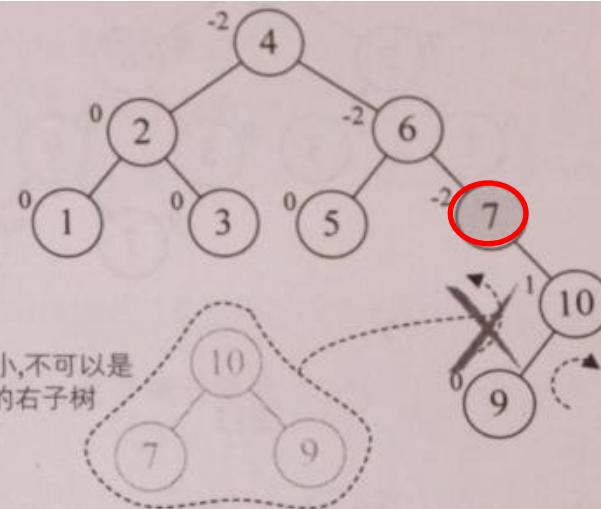


图11

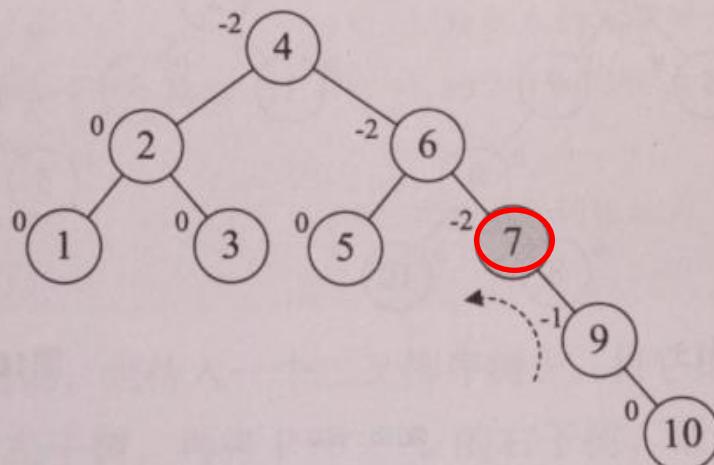
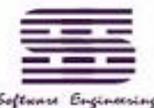


图12



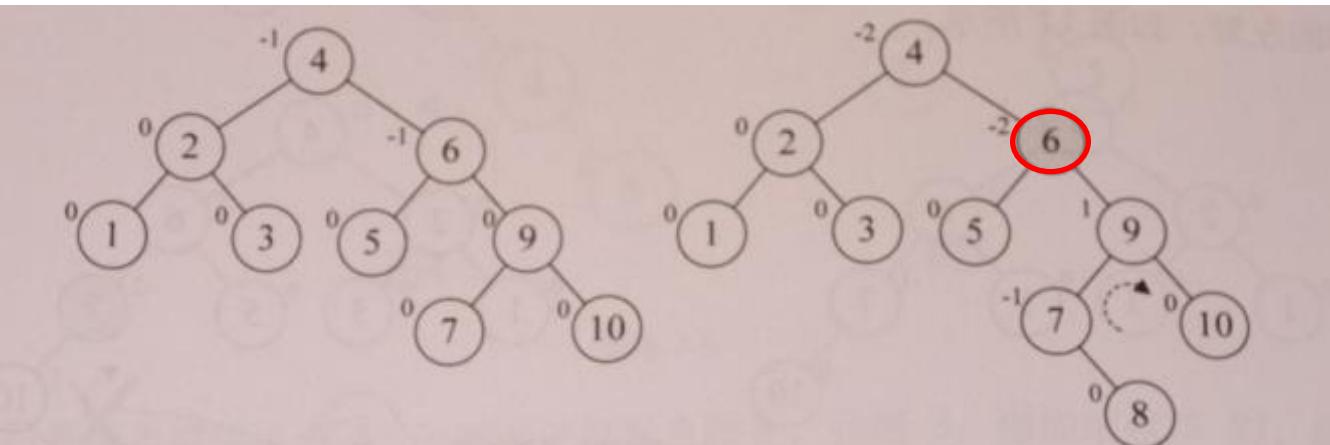


图13

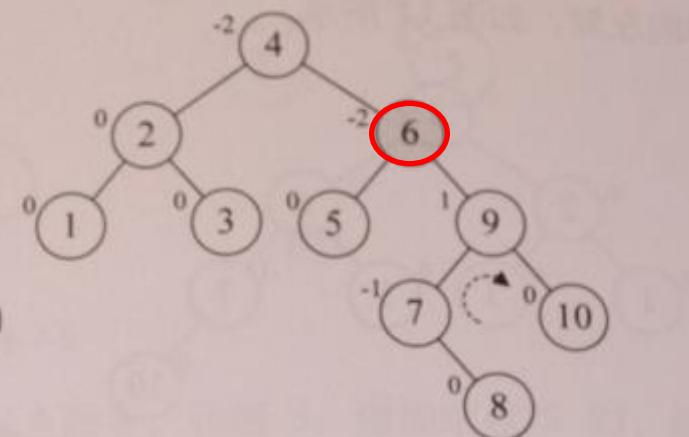


图14

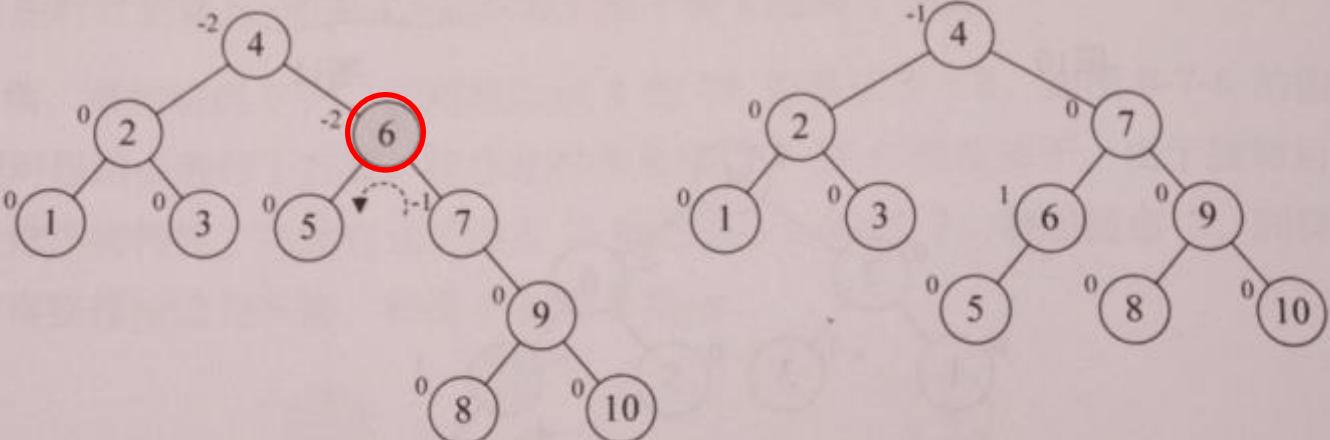


图15

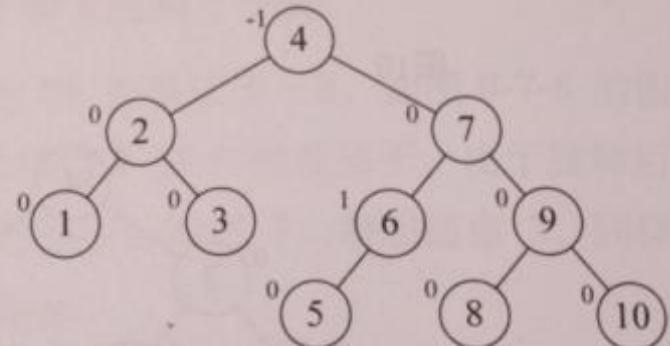
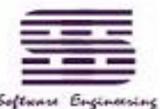


图16



# 构建AVL树：

- 1) 从空树出发，待插的关键字序列为  
**1,2,3,4,5,6**
- 2) 从空树出发，待插的关键字序列为  
**6,5,4,3,2,1**
- 3) 从空树出发，待插的关键字序列为  
**10,20,30,40,35,28,8,9**



# AVL树实现

- 数据结构
- 插入
- 删除
- 高度维护



2024/12/9

# AVL树删除

- Case 1
  - 删除叶子结点。操作：直接删除，然后依次向上调整高度
- Case 2
  - 删除非叶子结点。操作：使用前驱或后继代替，然后依次向上调整高度



2024/12/9

# AVL树实现

- 数据结构

```
template <typename T>
struct AVLTreeNode
{
    T m_key;                      // 关键字
    int m_height;                  // 高度
    AVLTreeNode *m_leftChild;     // 左孩子
    AVLTreeNode *m_rightNode;      // 右孩子
    AVLTreeNode(T value, AVLTreeNode *l, AVLTreeNode *r) :
        m_key(value), m_height(0), m_leftChild(l), m_rightNode(r)
    {}
};
```

# AVL树实现

- 数据结构 (**height**)
- 插入 (节点的插入和BF维护)
- 删除(节点的删除和BF维护)
- 高度维护

参见代码



2024/12/9

# 总结：AVL树

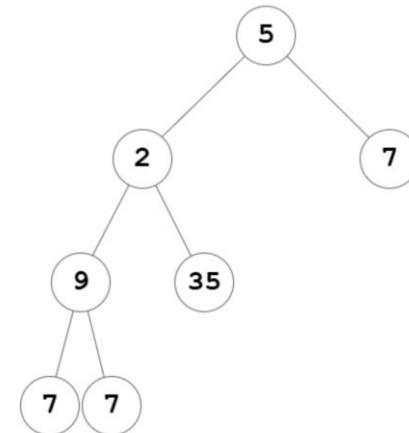
- **AVL规则的作用是：**确保二叉排序树永远不会变为失衡状态
- **N个节点的AVL树的高度将与 $\log_2 N$ 成正比。**
- **实现AVL树较麻烦：**
  - 插入和删除操作，执行时间与 $\log_2 N$ 成正比
  - 为修复失衡所进行的旋转操作可能需要对插入和删除期间所采用的访问路径执行向后遍历。  
(当需要多次旋转时)

# 平衡二叉树例题-1

- 判断是否是平衡二叉树。输入一棵二叉树的根节点，若任意节点的左右子树的深度相差不超过1，（未要求是AVL树）。

root = [5,2,7,9,35,null,null,7,7]

```
/* Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
 *     left(left), right(right) {}
 * };
```



# 平衡二叉树例题-1

- 判断是否是平衡二叉树。输入一棵二叉树的根节点，若任意节点的左右子树的深度相差不超过1，（未要求是AVL树）。

直接自顶向下递归

```
int height(TreeNode* cur){  
    if(cur == nullptr){ return 0; }  
    else return max(height(cur->right),height(cur->left)) +1;}  
  
bool isBalanced(TreeNode* root) {  
    if(root != nullptr) {  
        return abs(height(root->left) - height(root->right)) <=1 \&& isBalanced(root->left) && isBalanced(root->right);}  
    return true; }
```

# 平衡二叉树例题-1

- 判断是否是平衡二叉树。输入一棵二叉树的根节点，若任意节点的左右子树的深度相差不超过1，（未要求是AVL树）。

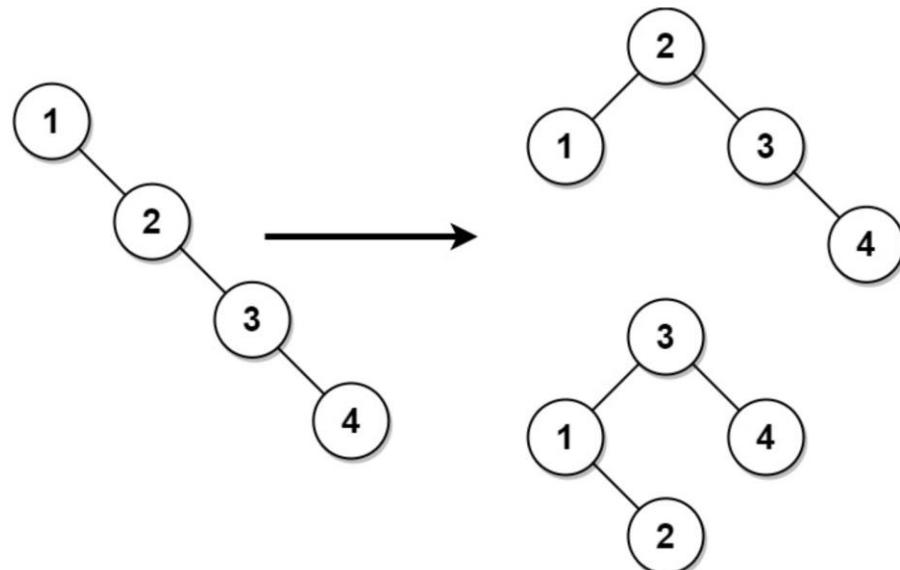
直接自底向上递归，在计算height的同时判断是否平衡

```
int height_2(TreeNode* cur){  
    if(cur == nullptr){ return 0; }  
    else{  
        int lh = height_2(cur->left);  
        int rh = height_2(cur->right);  
        if ( lh == -1 or rh == -1 or abs(lh-rh) > 1 )  
            return -1;  
        else return max(lh,rh) + 1; } }
```



# 平衡二叉树例题-2

- 将二叉搜索树变平衡：给定一棵二叉搜索树，返回一棵 **平衡后** 的二叉搜索树（任意一种即可）。



AVL树在插入/删除时  
进行平衡维护

输入： root = [1,null,2,null,3,null,4,null,null]

输出： [2,1,3,null,null,null,4]

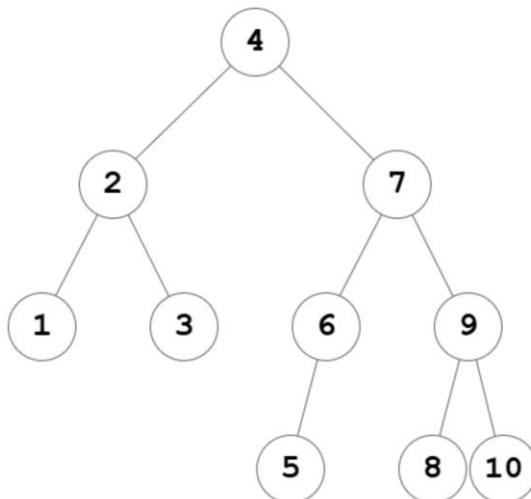
或[3,1,4,null,2,null,null]

# 平衡二叉树例题-2

- 将二叉搜索树变平衡：给定一棵二叉搜索树，返回一棵 **平衡后** 的二叉搜索树（任意一种即可）。

避免对原树调整，借助BST中序遍历的有序性，重新构造

AVL树查找过程



二分查找

```
int binarysearch(int *a, int t){  
    int l=0,r=N-1;  
    while (l<r){  
        int mid = (l+r) >>1;  
        if(a[mid] > t){ r = mid-1; }  
        else if(a[mid] < t ) { l = mid+1; }  
        else return mid;  
    }  
    return (a[l] == t)? l : -1;  
}
```

二分查找更加  
严格

# 平衡二叉树例题-2

- 将二叉搜索树变平衡：给定一棵二叉搜索树，返回一棵 **平衡后** 的二叉搜索树（任意一种即可）。

借助额外  $O(N)$  的数组空间 + 贪心（二分查找）

对长度为  $x$  的有序序列进行二分查找，查找成功时的最大比较次数为  $\lfloor \log_2 x \rfloor + 1$ ，即为  $h(x)$ 。而且二分查找过程中，**左右部分长度差最大为1**

转为证明：对长度为  $k$  和  $k+1$  按照二分方法构造的二叉树，其高度差不超过1

$$\lceil \log_2(k+1) \rceil - \lceil \log_2(k) \rceil \leq 1 \quad \text{这里是下取整}$$



# 平衡二叉树例题-2

## 1. 中序遍历转换

- 首先将原始的二叉搜索树通过中序遍历转化为一个有序序列。

## 2. 递归建树

- 定义区间：**给定一个有序序列的区间  $[L, R]$ ，表示待构建平衡二叉搜索树的节点值范围。

## 3. 选择根节点

- $mid = (R+L) \gg 1$ ，将该位置的值作为当前子树的根节点值。

## 4. 构建左子树

- $L \leq mid - 1$  成立，表示左半区间存在，构造左子树
- 递归调用建树过程，处理区间  $[L, mid-1]$  作为左子树的节点值范围。

## 5. 构建右子树(与左子树类似)

- $R \geq mid + 1$  成立，表示右半区间存在，构造右子树
- 递归调用建树过程，处理区间  $[mid+1, R]$  作为右子树的节点值范围。

仍然以  $\{10, 20, 30, 40, 35, 28, 8, 9\}$  为例子



# 平衡二叉树例题-2

```
void inorder(TreeNode * cur){  
    if(cur==nullptr) return;  
    if(cur->left!=nullptr){inorder(cur->left);}  
    lst.push_back(cur->val);  
    if(cur->right!=nullptr){inorder(cur->right);}  
}  
  
TreeNode * build(int l, int r){  
    int mid = (l+r)>>1;  
    TreeNode * node = new TreeNode(lst[mid]);  
    if(l< mid) node -> left = build(l,mid-1);  
    if(r> mid) node -> right = build(mid+1,r);  
    return node;  
}
```



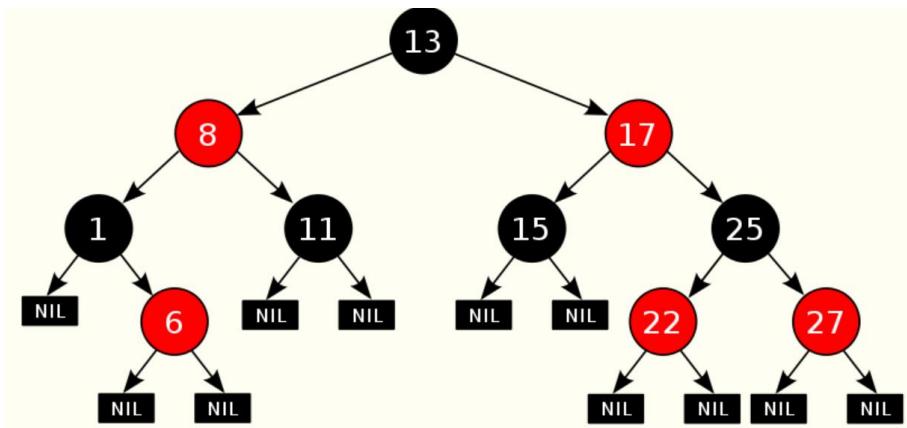
## (2) 红黑树 (red-black tree)

- 红黑树是满足下列条件的二叉查找树：
  - 每个节点都带有红色或黑色。节点的颜色由以下规则确定：
    - 根节点是黑色的。
    - 所有叶节点都是黑色的。
    - 在沿着从根出发的任何路径上都不允许出现两个连续的红色节点，即：“红色”结点的两个子结点都是“黑色”的。
    - 从任一节点到其每个子孙叶子节点的所有简单路径都包含相同数目的黑色节点（简称黑色高度）
      - 节点X的黑色高度：从节点X到其子孙叶子节点的简单路径中的黑色链的数量。
      - 红黑树的黑色高度：根节点的黑色高度（称为：根节点的阶）

## (2) 红黑树 (red-black tree)

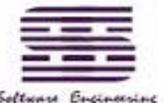
- 红黑树的两种不同定义：
  - 《算法导论》：叶子节点是指，扩充外部叶结点。即叶子节点为空的“黑色”节点
  - 《程序员实用算法》：数据只存储在叶子节点中，内部节点只用于引用；

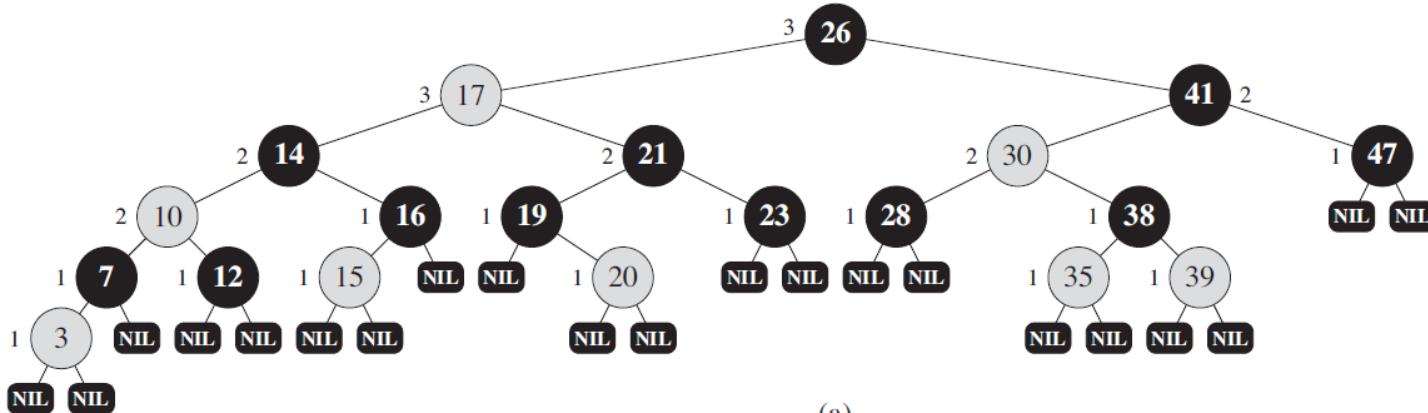
- 节点为红色或黑色
- 根节点为黑色
- NIL 节点（空叶子节点）为黑色
- 红色节点的子节点为黑色
- 从根节点到 NIL 节点的每条路径上的黑色节点数量相同



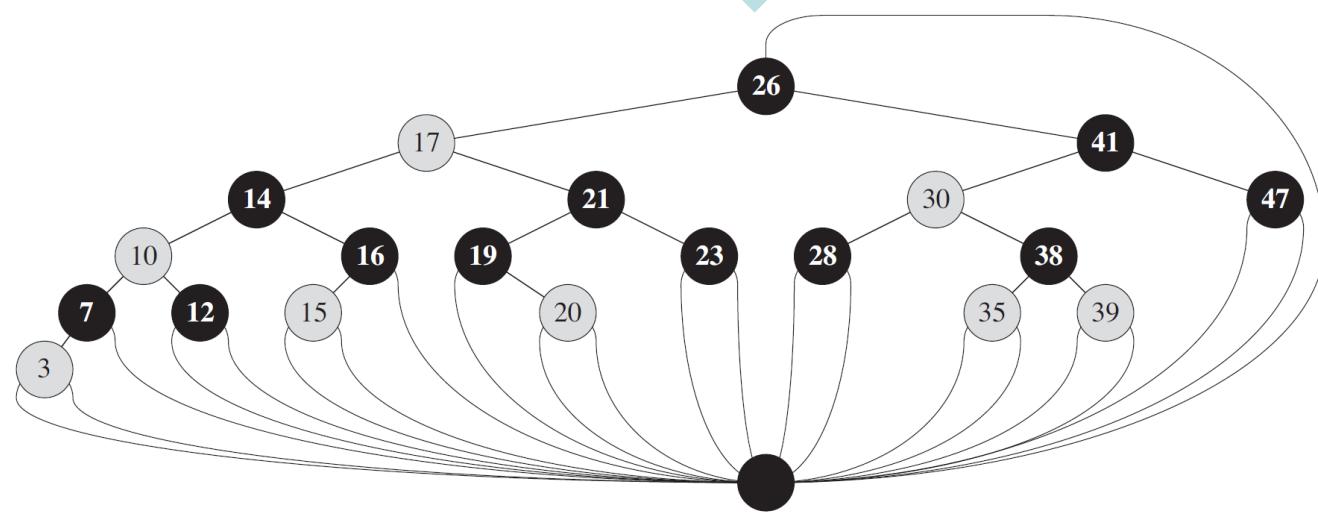
最长路径中结点个数不会超过最短路径结点个数的两倍？

《算法导论》中的RB-Tree的示例



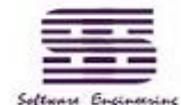


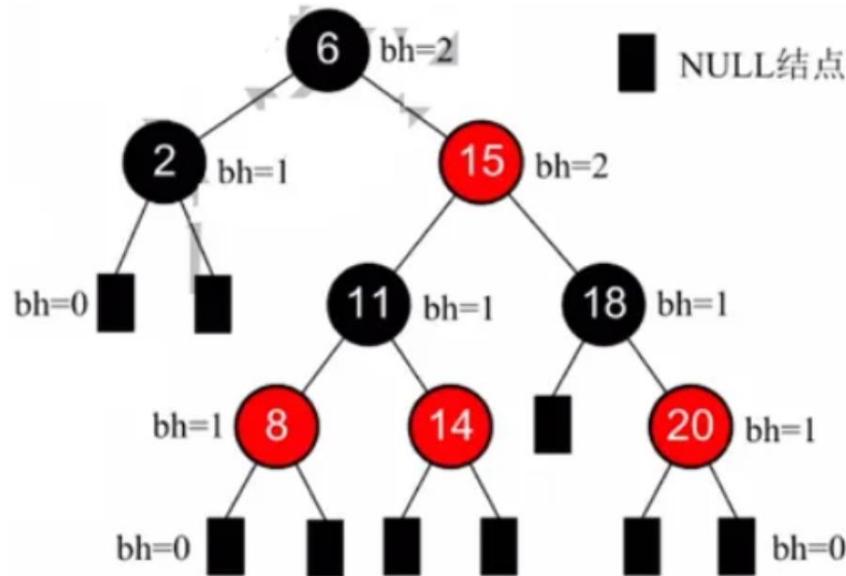
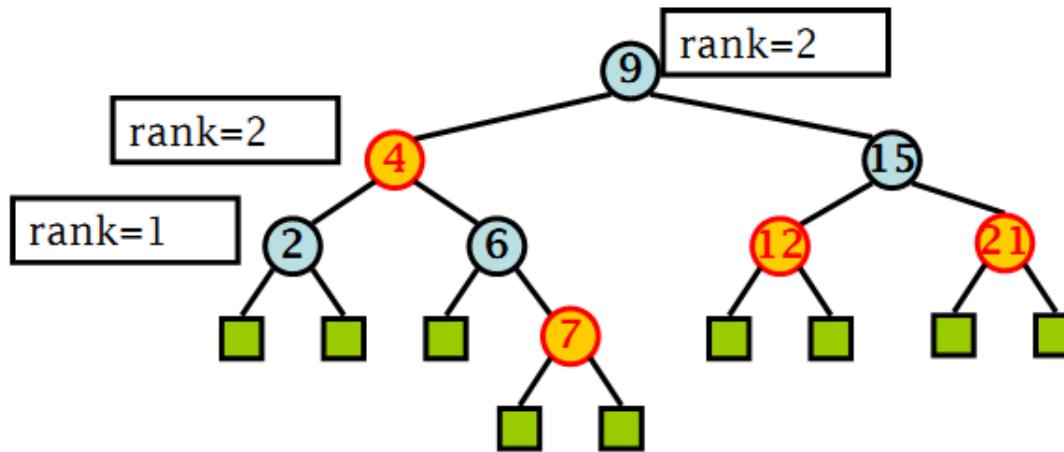
(a)



(b)

## 《算法导论》中的RB-Tree的示例

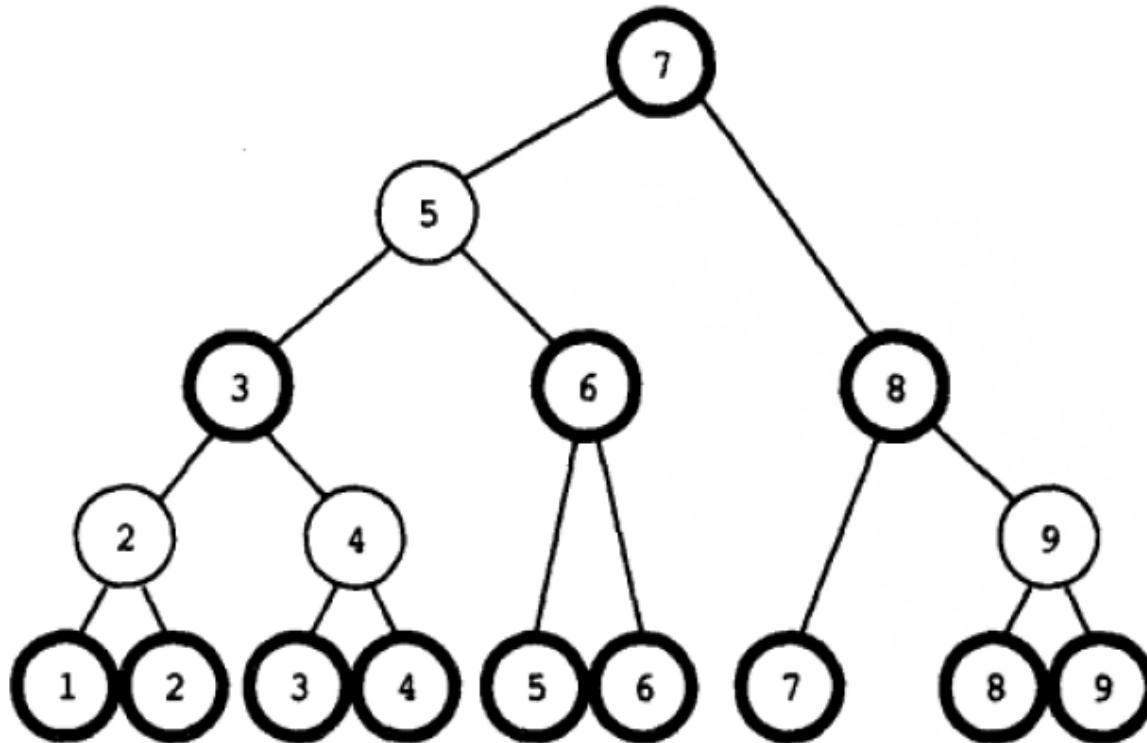




结点X的阶（rank，也称“黑色高度”）：从该节点到其子孙叶子节点的路径中的黑色链的数量。

- 从该结点到子孙叶子结点的黑色结点数量
- 不包括 X 结点本身，包括叶结点
- 根的阶称为该树的阶；
- 阶为 k 的红黑树路径长度：最短是 k，最长是 2k

- 示例：《程序员实用算法》中的RB-Tree



红黑树，其中黑色节点是用粗边框绘制的

注意，只有叶子节点中才是真实的数据！

# 红黑树的有效性

一个具有n个节点的红黑树的高度至多是 $2\lg(n+1)$

如何证明？

如何证明

先证明，任一节点x为根的子树至少包含  $2^{bh(x)} - 1$  节点

红红不相邻，则高度为h的的红黑树，黑高至少为/2

$$n \geq 2^{h/2} - 1$$

$$\log(n + 1) \geq h/2$$

归纳证明即可，子节点黑高为hb(x)或hb(x)-1



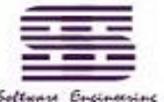
- 关键操作：
  - 红黑树的插入和构造：先确定插入位置，再插入新节点。
    - **关键点**：当插入一个新节点后，如何保持当前树仍为红黑树？（通过着色的调整和旋转来调整失衡）
  - 红黑树的删除：先确定删除位置，再删除节点。
    - **关键点**：当删除一个已有节点，如何保持当前树仍为红黑树？（通过着色的调整和旋转来调整失衡）



# 红黑树插入 —— 平衡维护

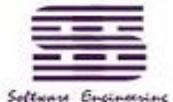
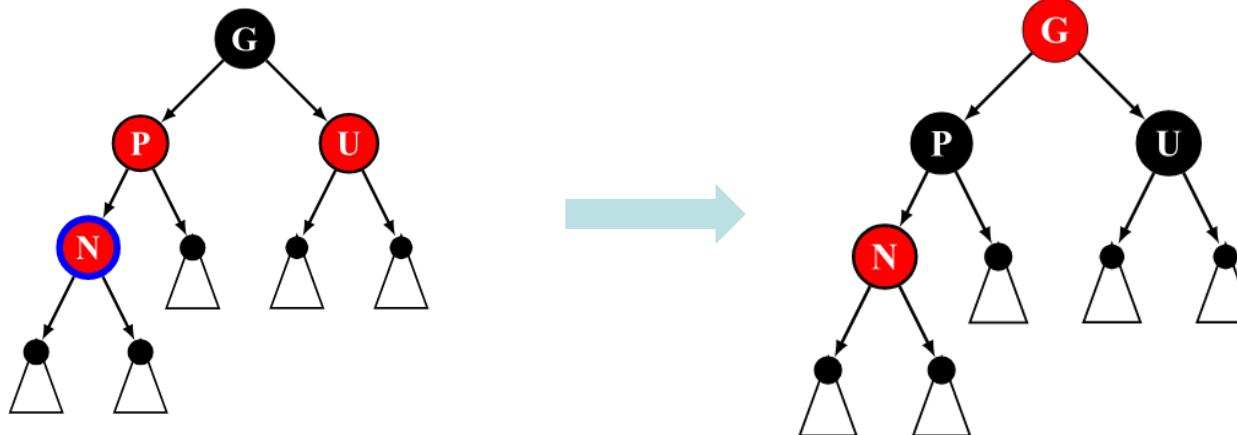
红黑树的插入操作与普通的 **BST** 类似，对于红黑树来说，新插入的节点初始为**红色**，完成插入后需根据插入节点及相关节点的状态进行修正以满足四条性质。

- **Case 1:** 该树原先为空，插入第一个节点
  - 不需要进行修正。
- **Case 2:** 当前的节点的父节点为黑色
  - 不需要进行修正。



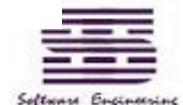
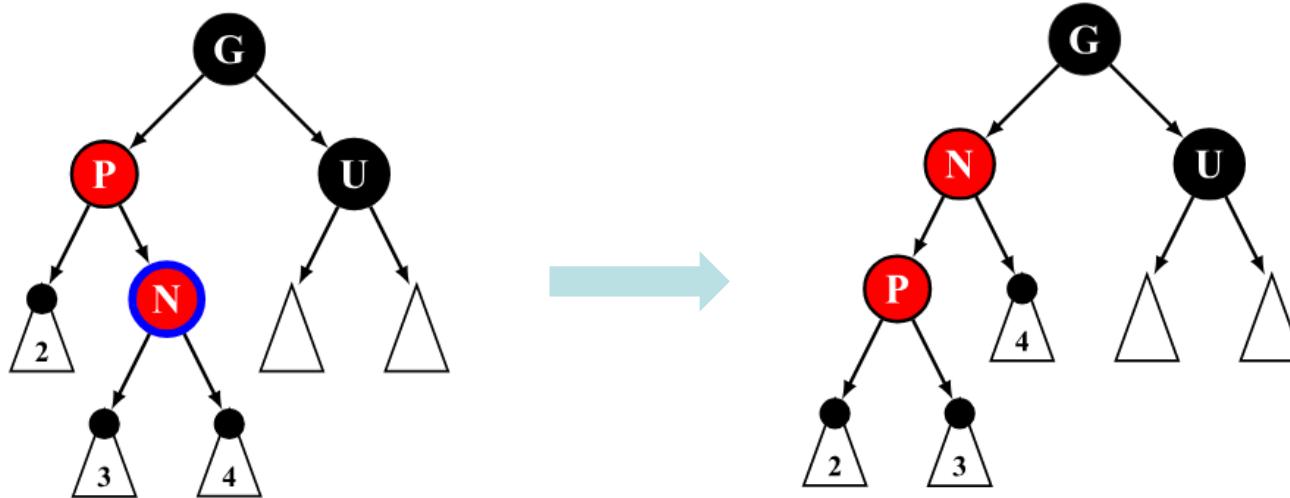
# 红黑树插入 —— 平衡维护

- **Case 3:** 当前节点 **N** 的父节点 **P** 是根节点且为红色
  - 将父节点染为黑色即可。
- **Case 4:** 当前节点 **N** 的父节点 **P** 和叔节点 **U** 均为红色
  - 将 **P**, **U** 节点染黑，将 **G** 节点染红
  - 递归维护 **G** 节点



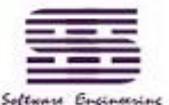
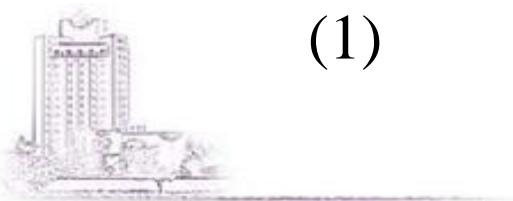
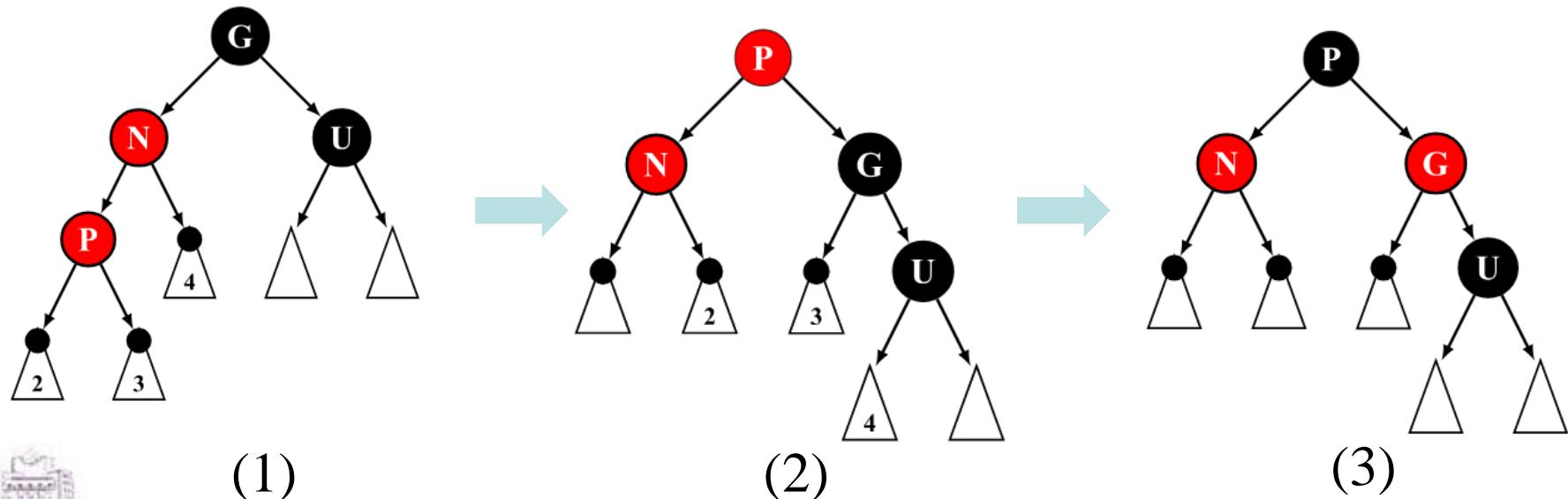
# 红黑树插入 —— 平衡维护

- **Case 5:** 当前节点 **N** 与父节点 **P** 的方向相反（即 **N** 节点为右子节点且父节点为左子节点，或 **N** 节点为左子节点且父节点为右子节点。）
  - 无法直接维护，先旋转 **p** 节点调整为 **case 6** 状态，后续维护
  - 注意： **u** 节点可能为 **NIL** 节点



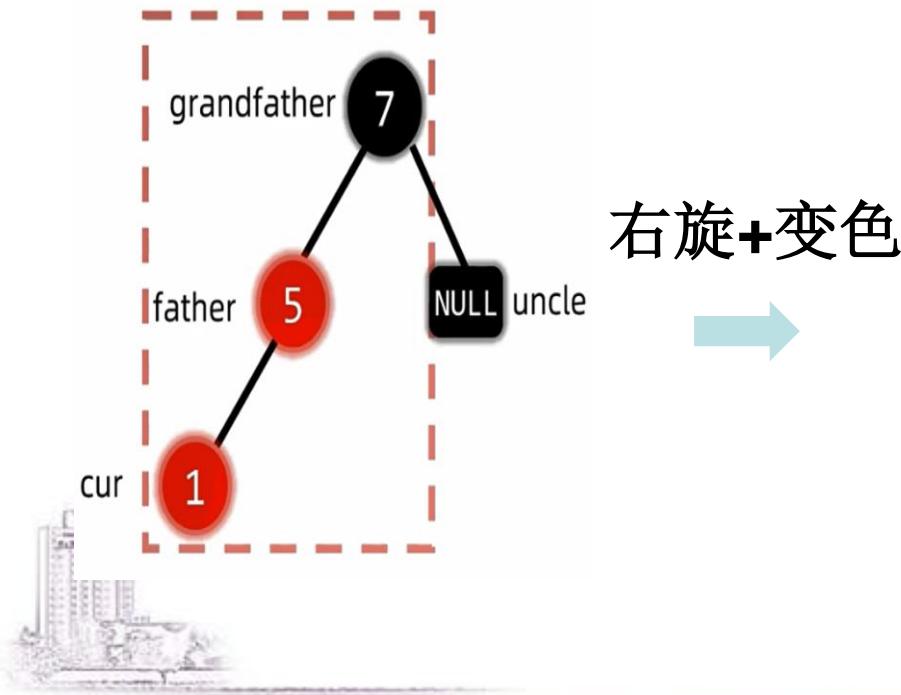
# 红黑树插入 —— 平衡维护

- **Case 6:** 当前节点 **N** 与父节点 **P** 的方向相同
  - 若 **N** 为左子节点则右旋祖父节点 **G**, 否则左旋祖父节点 **G**.
  - 重新染色, 将 **P** 染黑, 将 **G** 染红

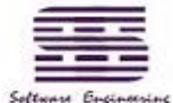
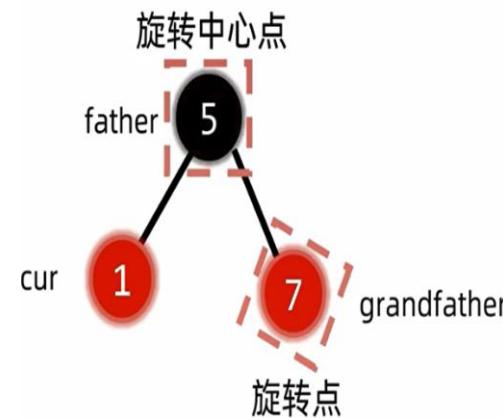
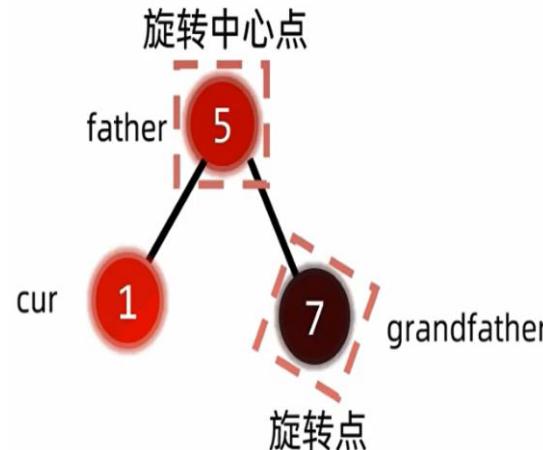


# 红黑树插入 —— 归类

仅当插入节点的叔节点为黑时，才能直接维护  
LL型（以G视角）

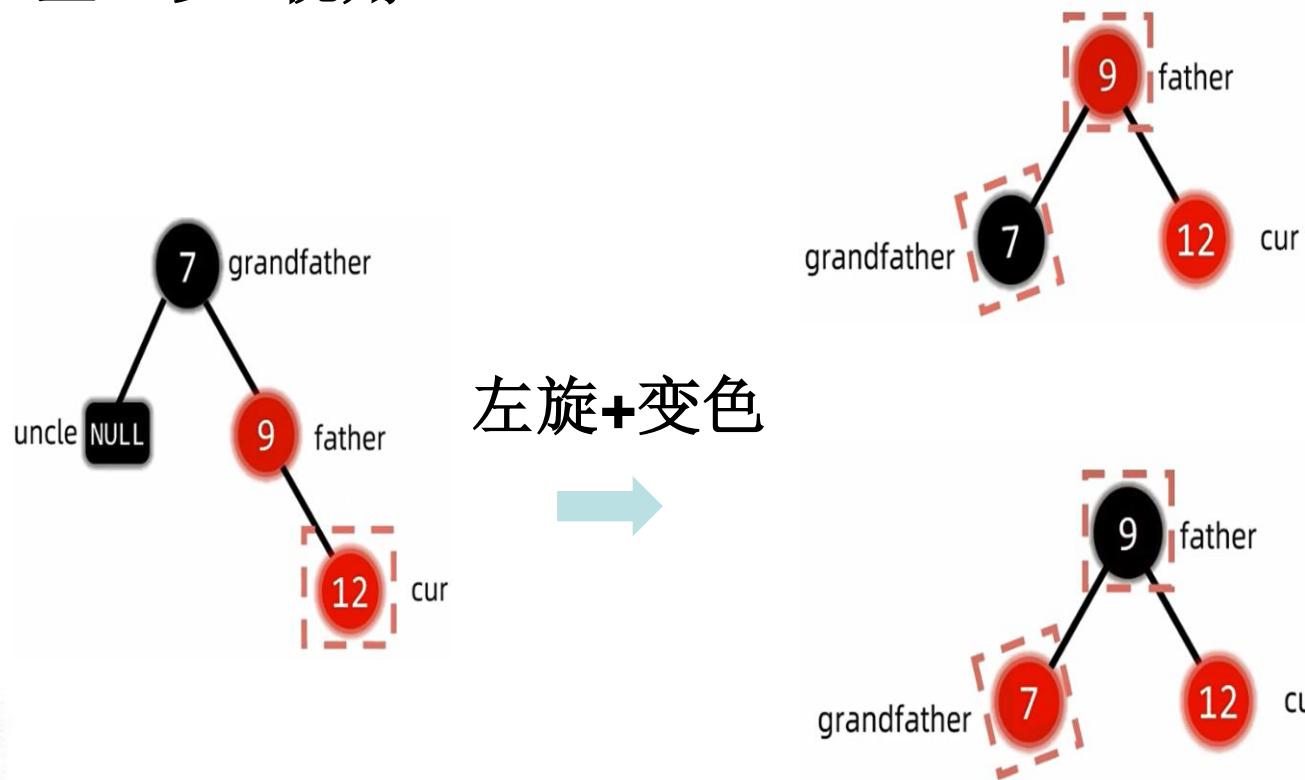


右旋+变色



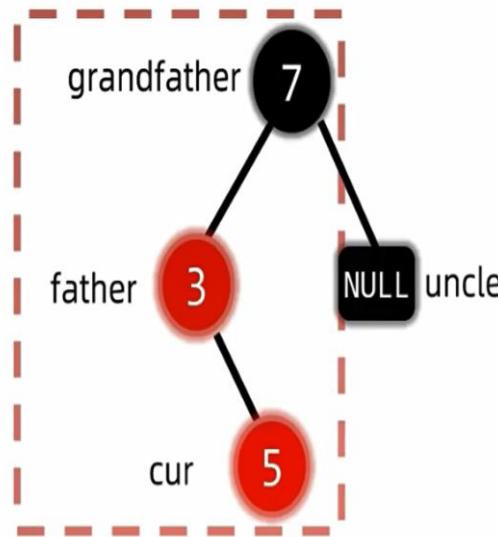
# 红黑树插入 —— 归类

仅当插入节点的叔节点为黑时，才能直接维护  
RR型（以G视角）

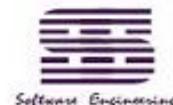
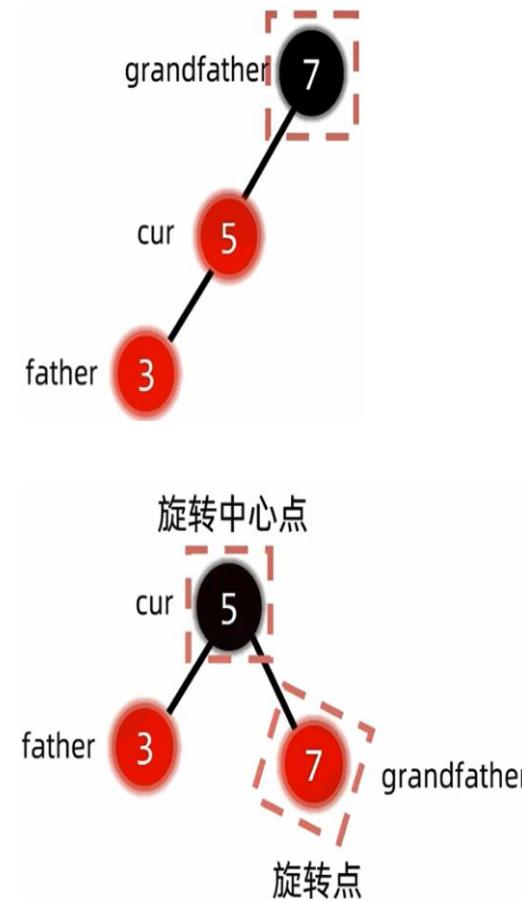


# 红黑树插入 —— 归类

仅当插入节点的叔节点为黑时，才能直接维护  
**LR型（以G视角）**

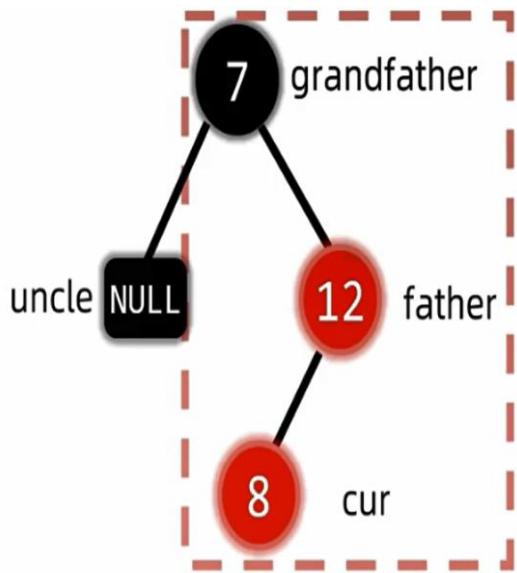


左旋左孩  
右旋本身  
后变色

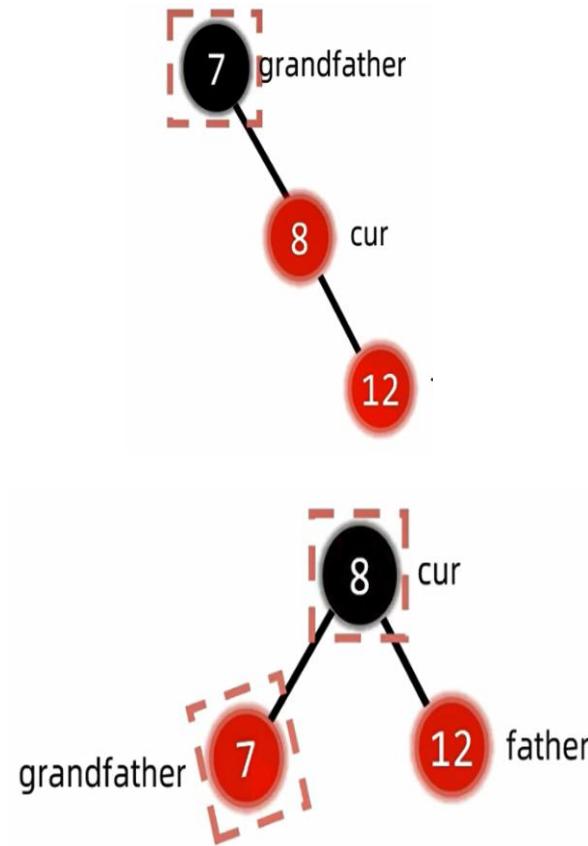


# 红黑树插入 —— 归类

仅当插入节点的叔节点为黑时，才能直接维护  
**RL型（以G视角）**



右旋右孩  
左旋本身  
后变色



# 红黑树插入 —— 总结

红  
黑  
树  
  
插入

左根右  
根叶黑  
不红红  
黑路同

插入结点默认是红色结点

如果插入后性质被破坏, 则根据下面三种情况做调整:

插入结点是根结点 → 直接变黑

破坏了根叶黑

插入结点的叔叔是红色 → 叔父变色, 爷爷变插入结点

插入结点的叔叔是黑色 → (LL,RR,LR,RL)旋转, 然后变色

破坏了不红红

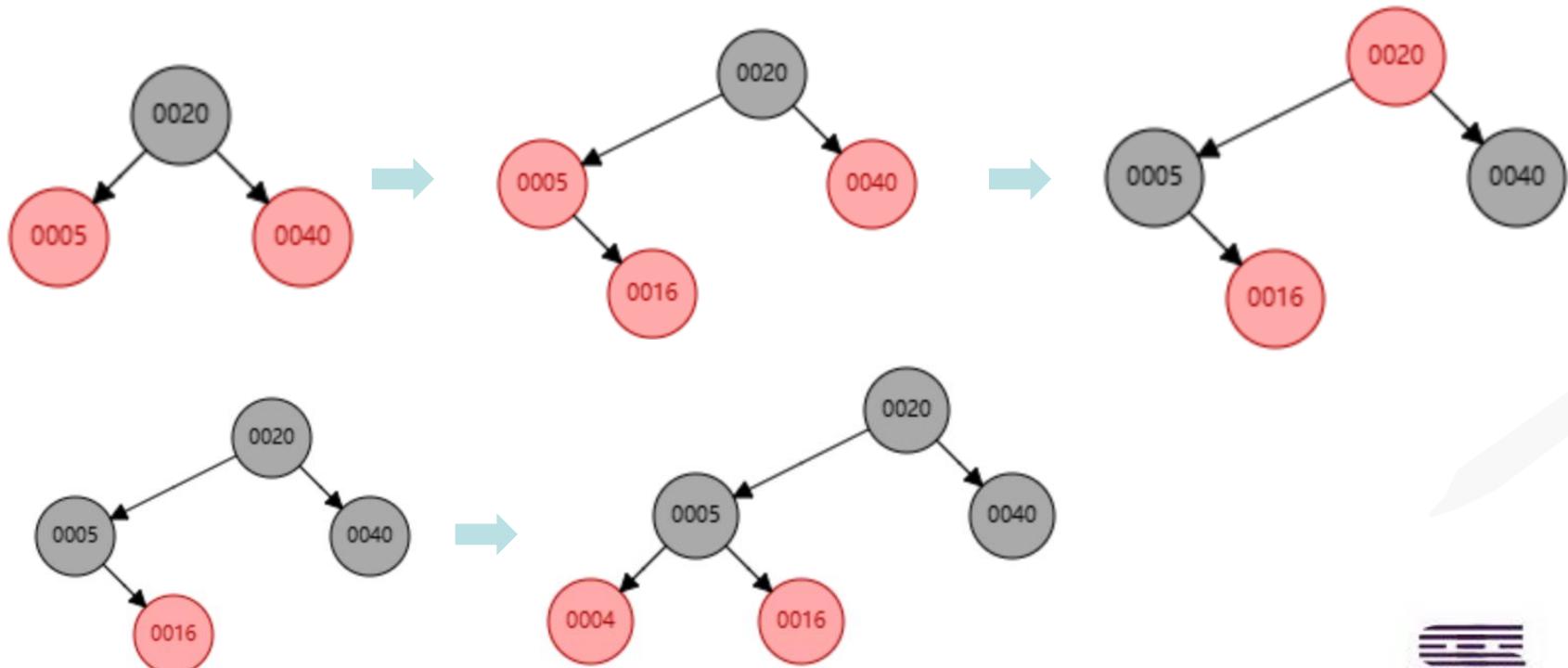
参考

[https://www.bilibili.com/video/BV1Xm421x7Lg/?spm\\_id\\_from=333.337.search-card.all.click&vd\\_source=459f25d1b71e72fac82c5bee26e60bb7](https://www.bilibili.com/video/BV1Xm421x7Lg/?spm_id_from=333.337.search-card.all.click&vd_source=459f25d1b71e72fac82c5bee26e60bb7)



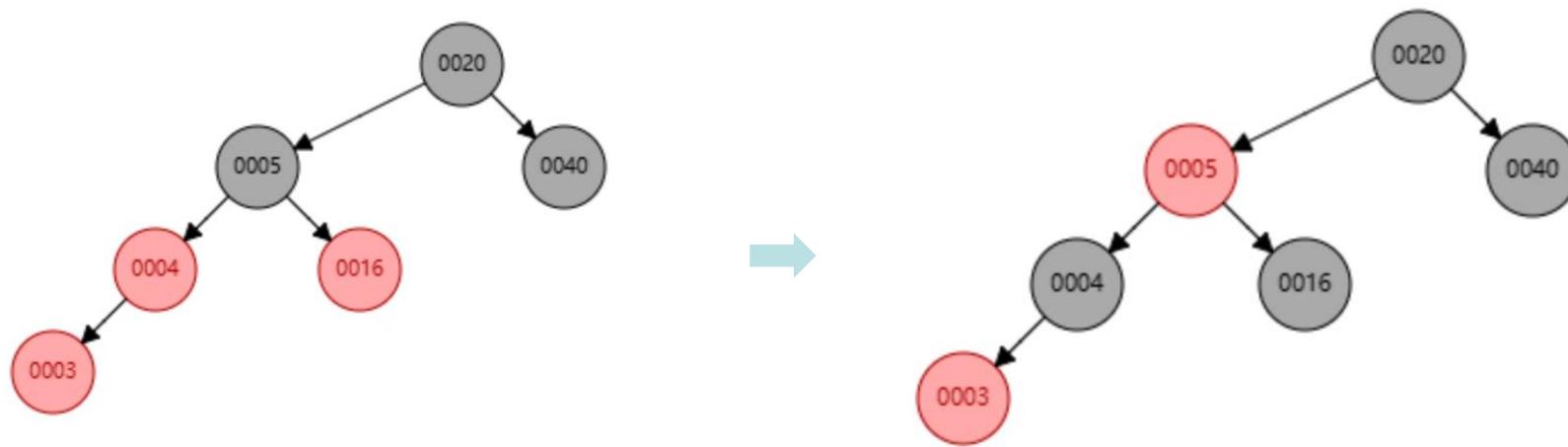
# 红黑树插入 —— 示例

按序插入[20, 5, 40, 16, 4, 3, 2, 1]

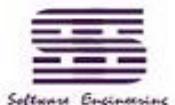


# 红黑树插入 —— 示例

按序插入[20, 5, 40, 16, 4, 3, 2, 1]



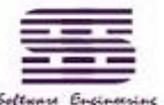
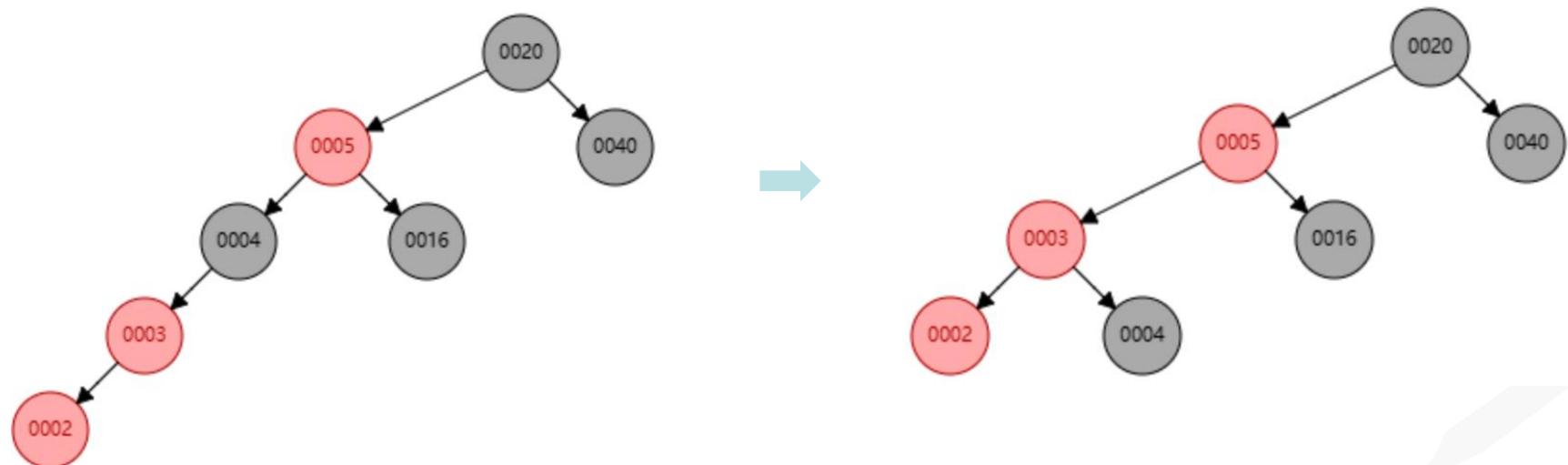
简单的改变颜色，未发生旋转  
BF=2



# 红黑树插入 —— 示例

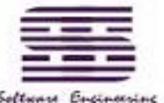
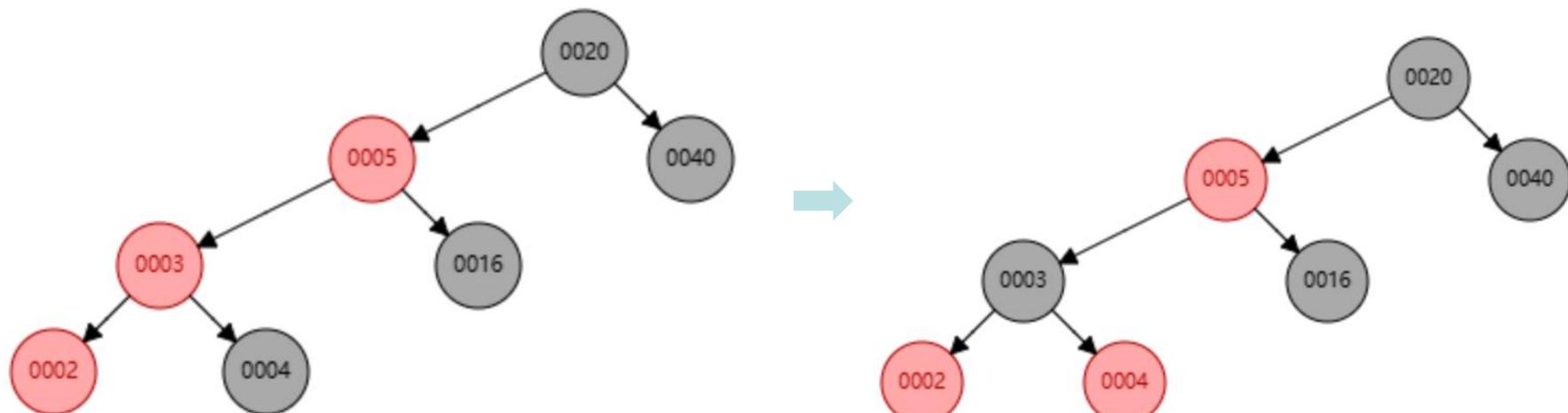
按序插入[20, 5, 40, 16, 4, 3, 2, 1]

ement



# 红黑树插入 —— 示例

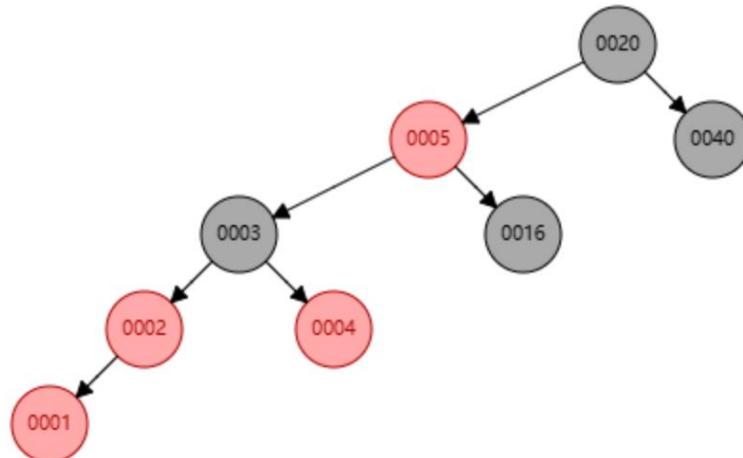
按序插入[20, 5, 40, 16, 4, 3, 2, 1]



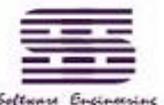
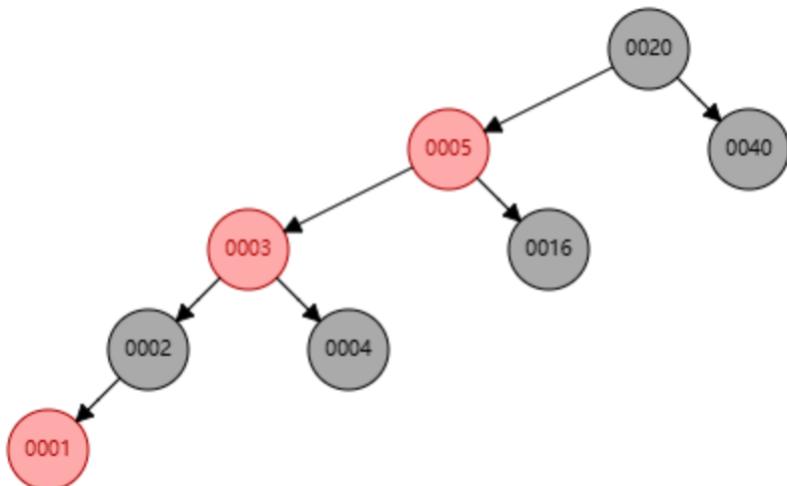
# 红黑树插入 —— 示例

按序插入[20, 5, 40, 16, 4, 3, 2, 1]

\ inserting element

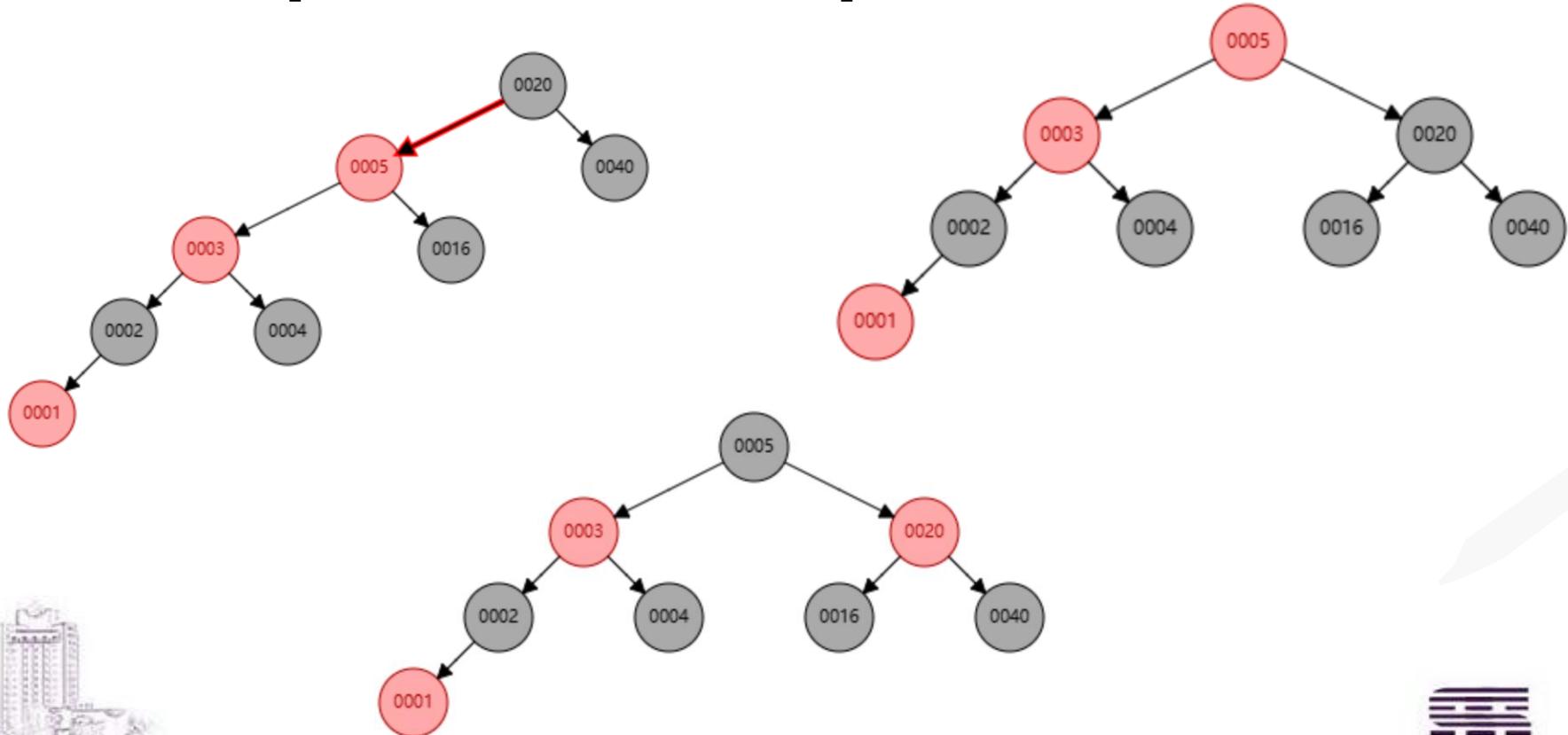


Uncle of node is red -- push blackness down from grandparent



# 红黑树插入 —— 示例

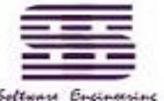
按序插入[20, 5, 40, 16, 4, 3, 2, 1]



# 红黑树删除 —— 先删除后维护

不改变节点颜色和引用

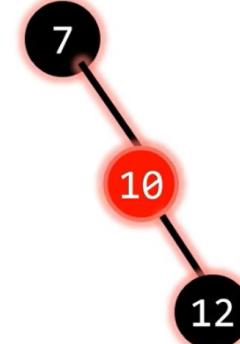
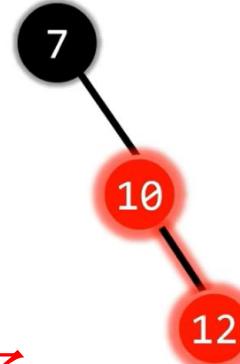
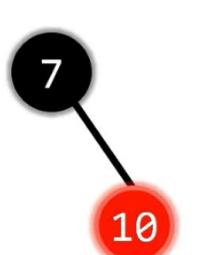
- **Case 0**
  - 若待删除节点为根节点的话，直接删除即可，
- **Case 2-1**
  - 待删除节点为叶子节点，若该节点为红色，直接删除即可
- **Case 2-2**
  - 待删除节点为叶子节点，若该节点为黑色，直接删除后维护。



# 红黑树删除 —— 先删除后维护

- Case 3

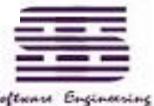
- 若待删除节点 **N** 有且仅有一个非 **NIL** 子节点，则子节点 **S** 一定为红色。子节点 **S** 替代 **N** 并将其染黑后即可
- 如果子节点 **S** 为黑色，则 **S** 的黑深度和待删除结点的黑深度不同 不改变节点颜色和引用



问题变为黑节点只有左或右孩子

违反不红红

违反黑路同



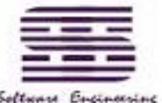
# 红黑树删除 —— 先删除后维护

- Case 1

- 若待删除节点 **N** 既有左子节点又有右子节点，则需找到它的前驱或后继节点进行替换，后面删除该前驱或后继并维护。
- 前驱或后继节点保证不会是一个既有非 **NIL** 左子节点又有非 **NIL** 右子节点的节点（不会套娃）

- 删除
- | 没有孩子 - 直接删除
  - | 只有左子树/只有右子树 - 直接代替
  - | 左右子树都有 - 直接后继(或前驱)代替值, 然后删除(转换成前两种情况)

删除后最容易破坏黑路同的性质

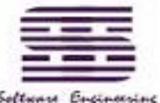
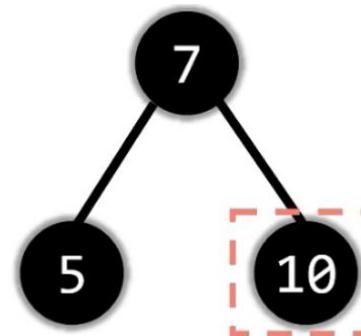


# 红黑树删除 —— 需要维护

删除

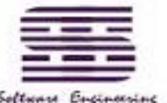
- | 没有孩子 - 直接删除
- | 只有左子树/只有右子树 - 直接代替
- | 左右子树都有 - 直接后继(或前驱)代替值, 然后删除(转换成前两种情况)

- { 只有左孩子/只有右孩子: 代替后变黑
- { 红结点: 删除后无需任何调整
- 没有孩子 { 黑结点



# 红黑树删除 —— 情况

- **Case1:**兄弟节点 **S**红，侄节点 **C, D** 黑色，父节点 **P** 黑
- **Case2:**兄弟节点 **S**黑，侄节点 **C, D** 黑色，父节点 **P** 红
- **Case3:**兄弟节点 **S**黑，侄节点 **C, D** 黑色，父节点 **P** 黑
- **Case4:**兄弟节点 **S**黑，近侄**C**红，远侄黑，父节点红/黑
- **Case5:**兄弟节点 **S**黑，近侄**C**黑，远侄红，父节点红/黑
- **Case6:**兄弟节点 **S**黑，近侄**C**红，远侄红，父节点红/黑

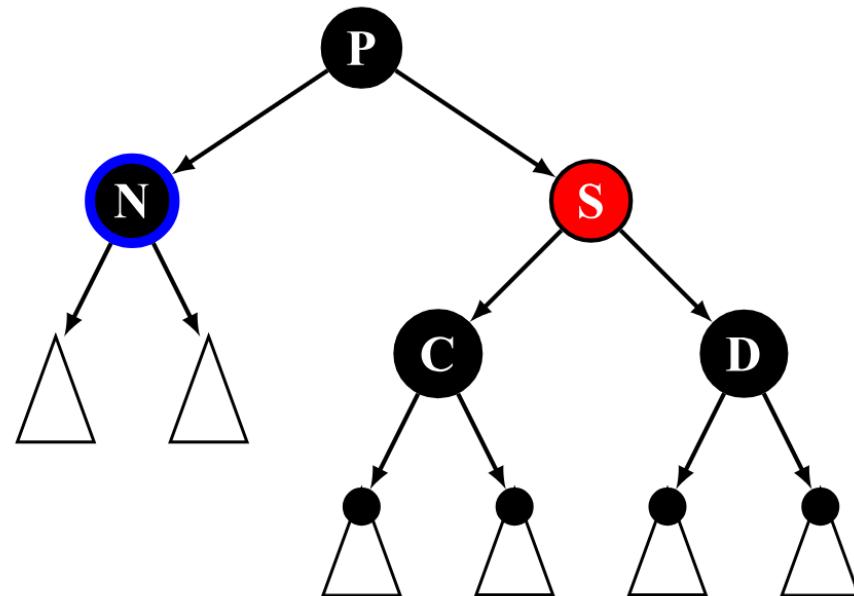


# 红黑树删除 —— 删除后维护

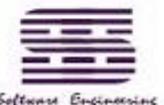
空三角代表子树， 带黑点表示黑高加1的子树

- **Case 1**

- 兄弟节点 (**sibling node**) **S** 为红色
- 则父节点 **P** 和侄节点 (**nephew node**) **C** 和 **D** 必为黑色



N这里可能是非空叶节点  
或者是向上维护到该节点  
了。因此，N的子树是比  
C/D黑高小1的。因为删去  
了一个黑色节点



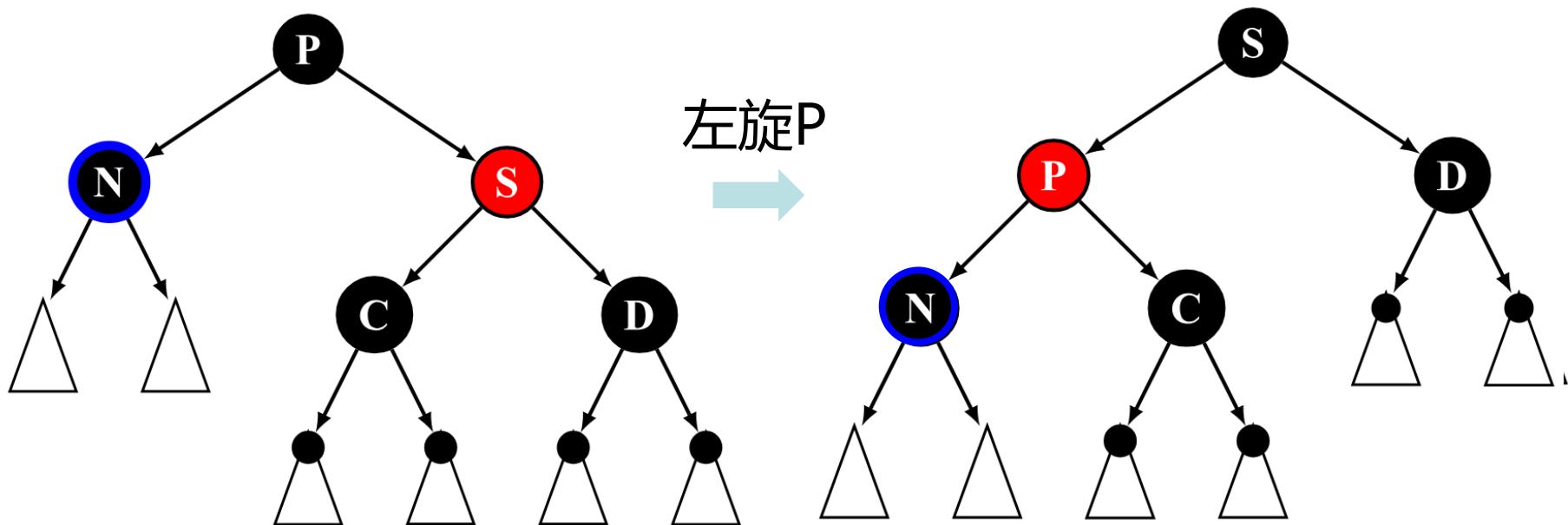
# 红黑树删除 —— 删除后维护

空三角代表子树， 带黑点表示黑高加1的子树

- **Case 1**

将P染红，这里理解起来稍微有点绕。可以逻辑上把有问题的节点（蓝色框）当作双黑节点

- 若待删除节点 N 为左子节点，左旋 P; 若为右子节点，右旋 P。
- 将 S 染黑，P 染红，继续对节点 N 进行维护（Case2）

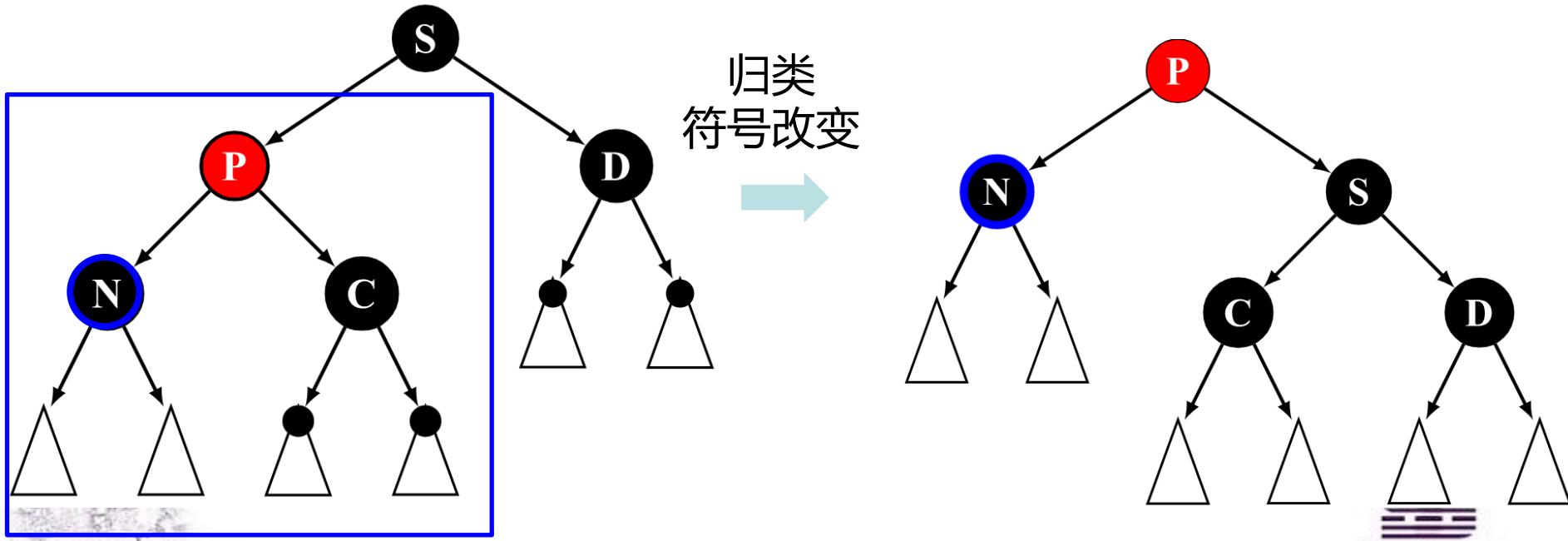


# 红黑树删除 —— 删除后维护

空三角代表子树， 带黑点表示黑高加1的子树

- **Case 2**

- 兄弟节点 **S** 为黑， 且侄节点**C,D**为黑， 父**P**为红

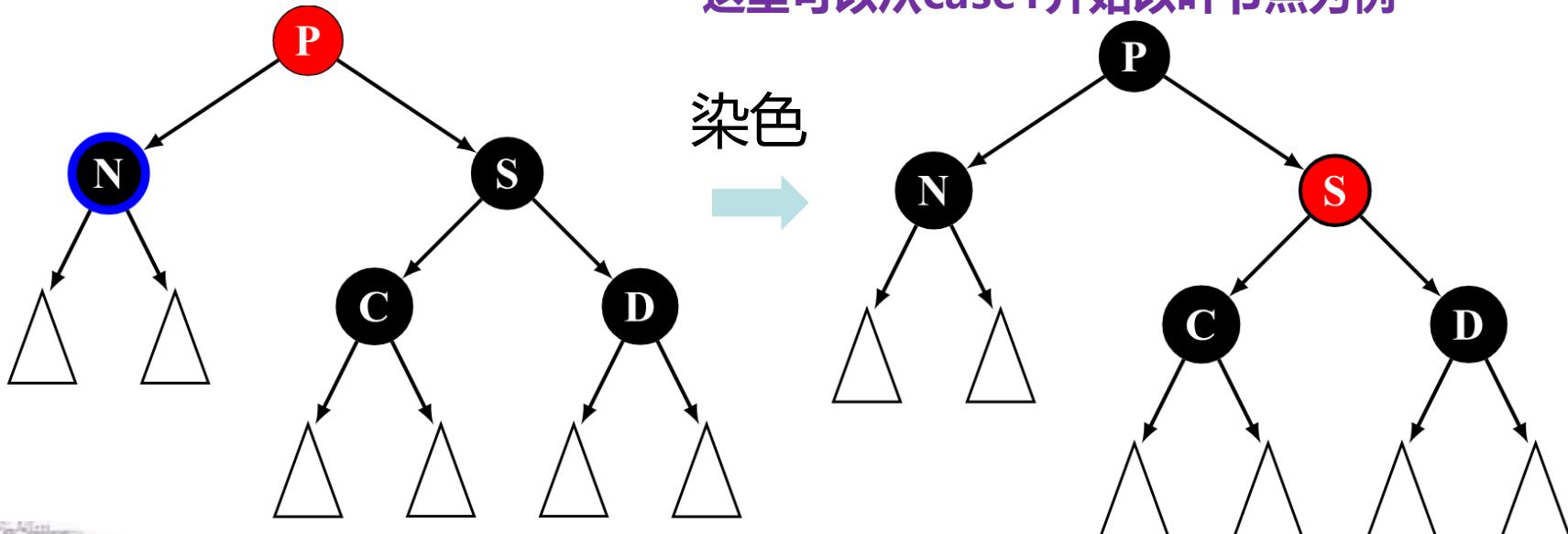


# 红黑树删除 —— 删除后维护

空三角代表子树或NIL， 带黑点表示黑高加1的子树或NIL

- **Case 2**

- 兄弟节点 **S** 为黑， 且侄节点**C,D**为黑， 父**P**为红
- **S**染红， **P**染黑即可 **P右子树黑高-1**， 因此**N**所在的**P**的左子树维护完成  
这里可以从case1开始以叶节点为例

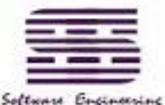
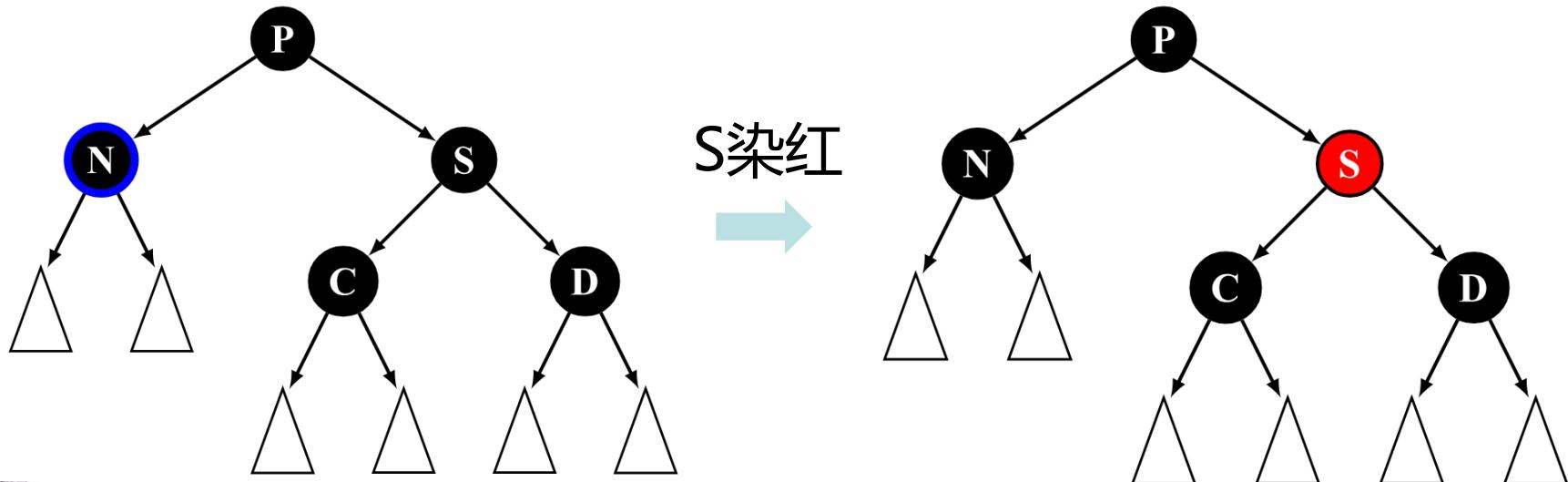


# 红黑树删除 —— 删除后维护

空三角代表子树或NIL， 带黑点表示黑高加1的子树或NIL

- **Case 3**

- 兄弟节点 **S** 黑， 僵节点**C,D**黑， 父节点**P**黑

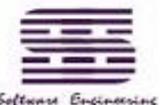
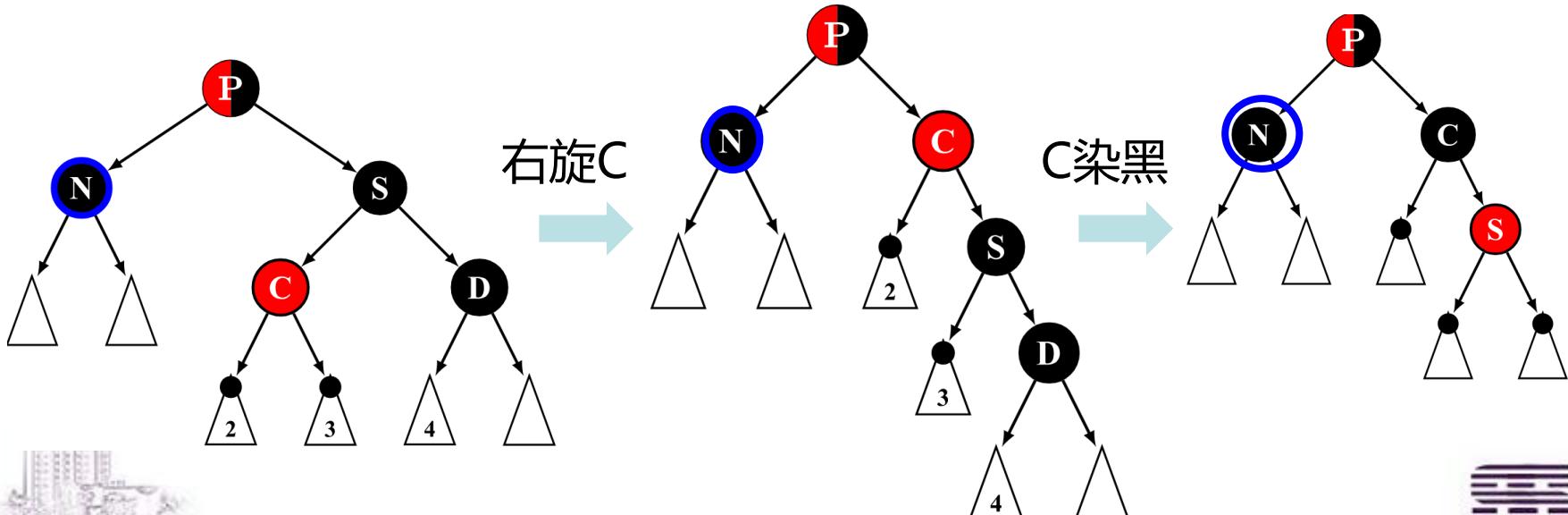


# 红黑树删除 —— 删除后维护

空三角代表子树或NIL， 带黑点表示黑高加1的子树或NIL

- **Case 4**

- 兄弟节点 **S** 黑， 近侄**C**红， 远侄**D**黑， 父节点**P**红/黑
- 若 **N** 为左子节点， 右旋 **C**， 否则左旋 **C**
- **S**染红， **C**染黑， 进入**Case5**

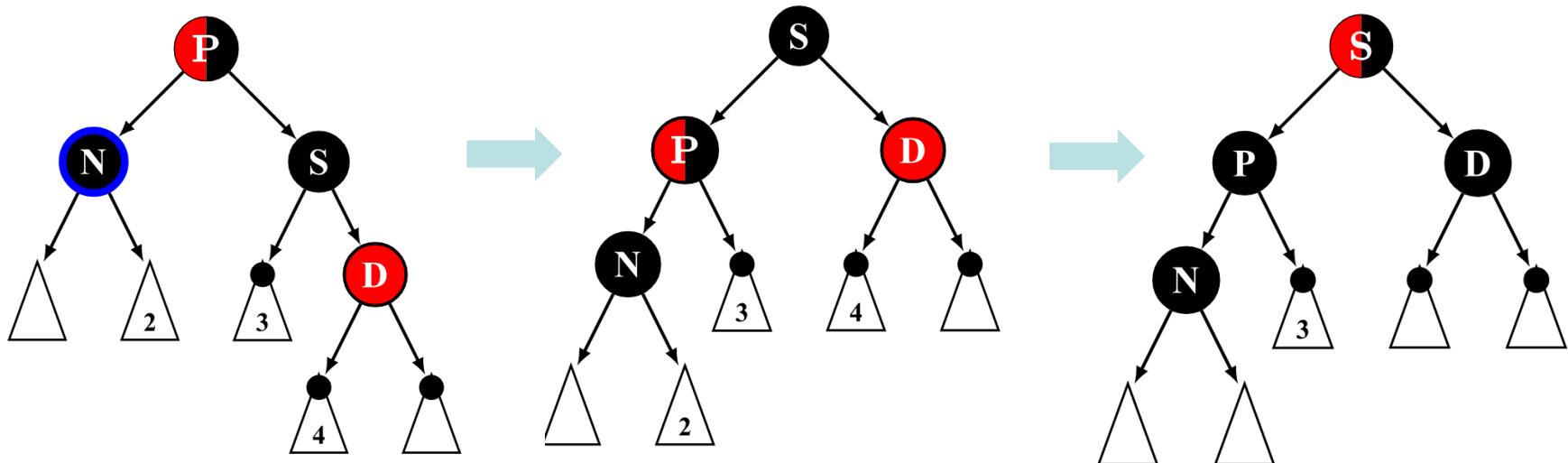


# 红黑树删除 —— 删除后维护

空三角代表子树或NIL， 带黑点表示黑高加1的子树或NIL

- **Case 5**

- 兄弟节点 **S** 黑， 近侄**C**黑， 远侄**D**红， 父节点**P**红/黑
- 若 **N** 为左子节点， 左旋 **P**， 反之右旋 **P**
- 交换父节点 **P** 和兄弟节点 **S** 的颜色
- 将远侄染黑

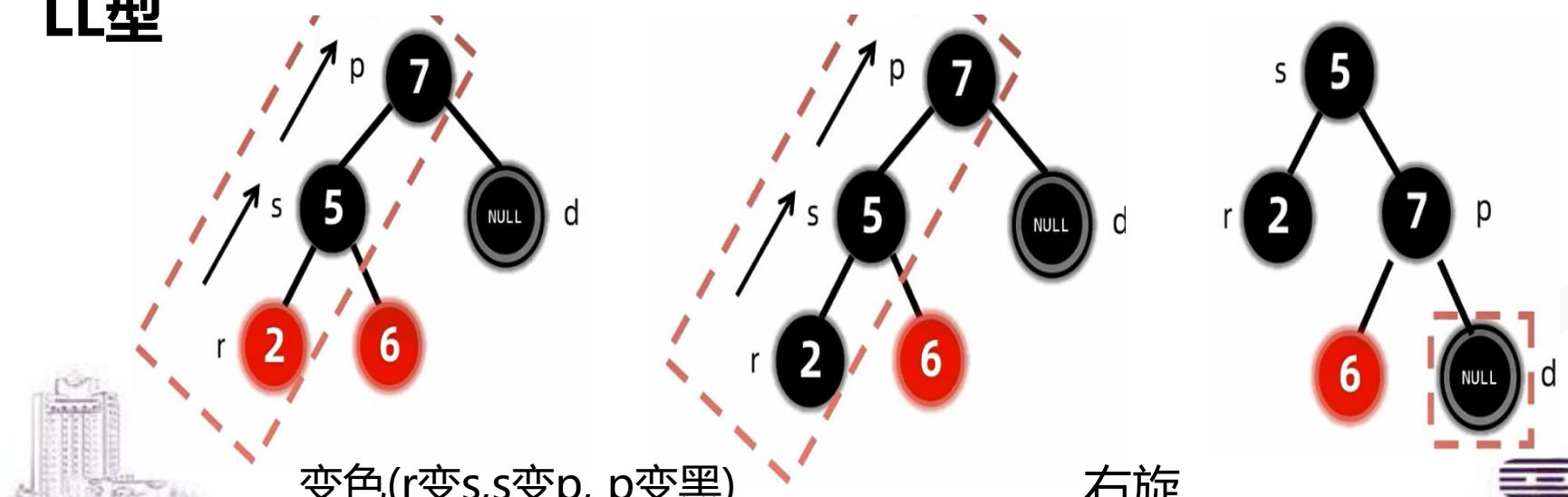


# 红黑树删除调整 归类

- Case1: 兄弟 S红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 黑
- Case4: 兄弟 S黑, 近侄C红, 远侄黑, 父节点红/黑
- Case5: 兄弟 S黑, 近侄C黑, 远侄红, 父节点红/黑

兄黑,  
至少一  
红侄

LL型

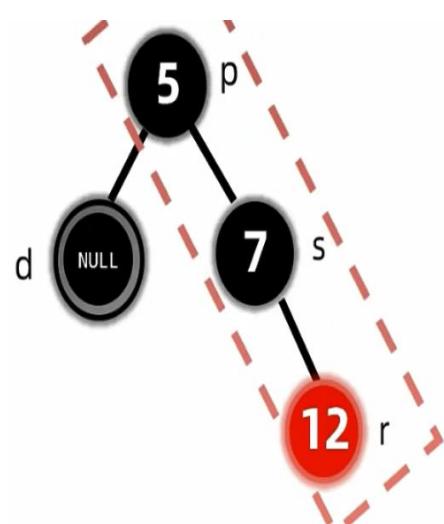


# 红黑树删除调整 归类

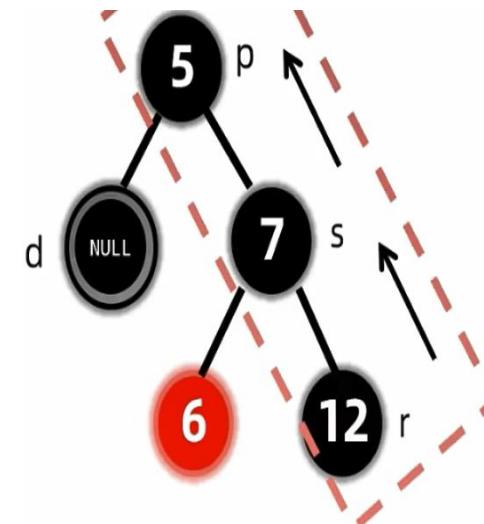
- Case1: 兄弟 S红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 黑
- **Case4:** 兄弟 S黑, 近侄C红, 远侄黑, 父节点红/黑
- **Case5:** 兄弟 S黑, 近侄C黑, 远侄红, 父节点红/黑

兄黑,  
至少一  
红侄

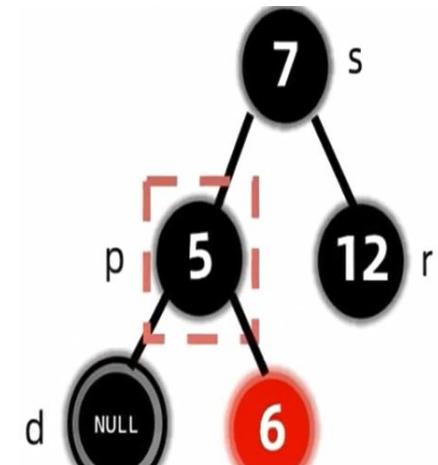
RR型



变色(r变s,s变p, p变黑)



左旋

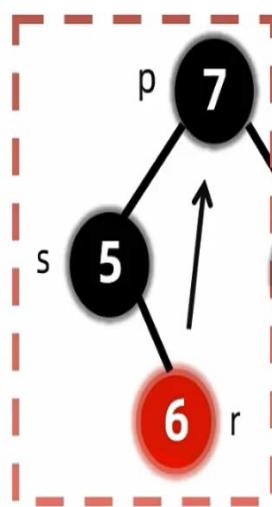


# 红黑树删除调整 归类

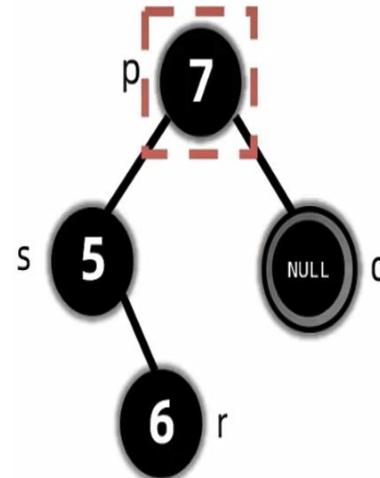
- Case1: 兄弟 S红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 黑
- Case4: 兄弟 S黑, 近侄C红, 远侄黑, 父节点红/黑
- Case5: 兄弟 S黑, 近侄C黑, 远侄红, 父节点红/黑

兄黑,  
至少一  
红侄

LR型



变色(r变p, p变黑)



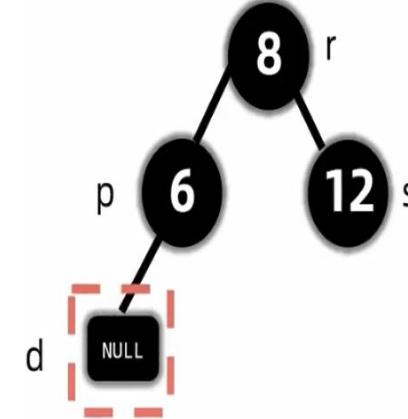
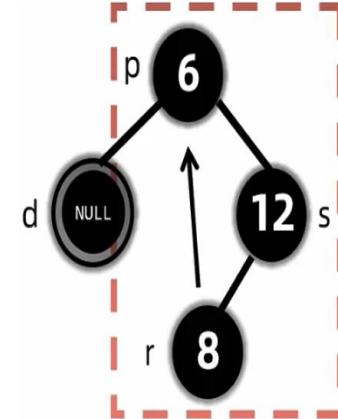
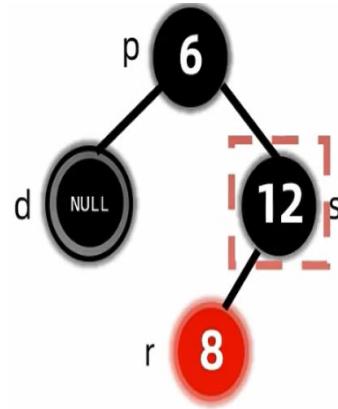
左旋左孩, 然后右旋

# 红黑树删除调整 归类

- Case1: 兄弟 S红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S黑, 倾节点 C, D 黑色, 父节点 P 黑
- **Case4:** 兄弟 S黑, 近侄C红, 远侄黑, 父节点红/黑
- **Case5:** 兄弟 S黑, 近侄C黑, 远侄红, 父节点红/黑

兄黑,  
至少一  
红侄

RL型



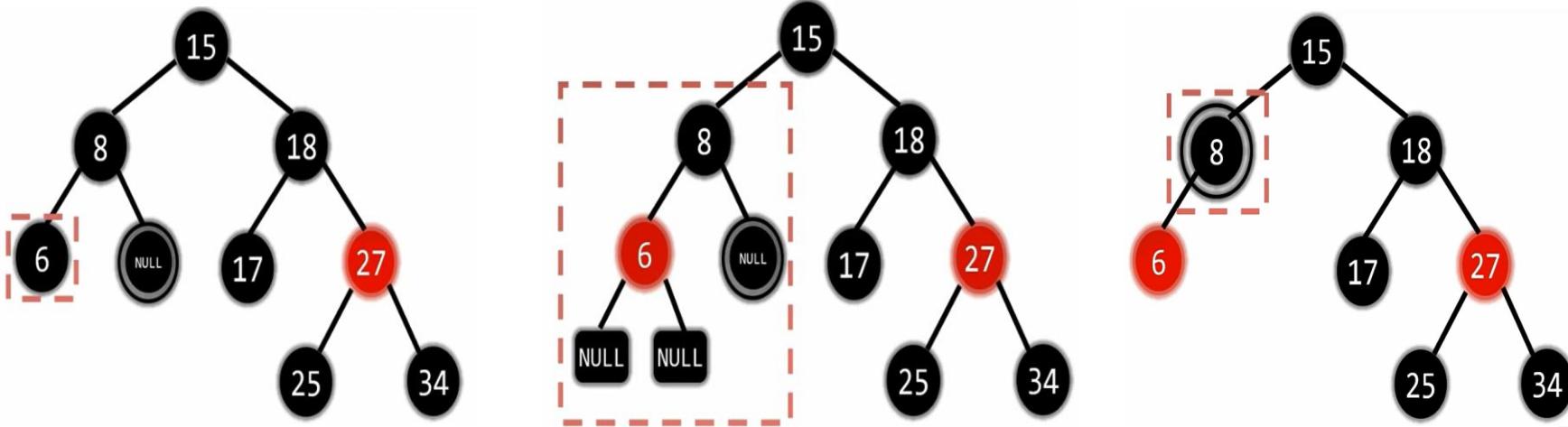
变色(r变p, p变黑)

右旋右孩, 然后左旋

# 红黑树删除调整 归类

- Case1: 兄弟 S 红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S 黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S 黑, 倾节点 C, D 黑色, 父节点 P 黑

兄黑,  
侄黑



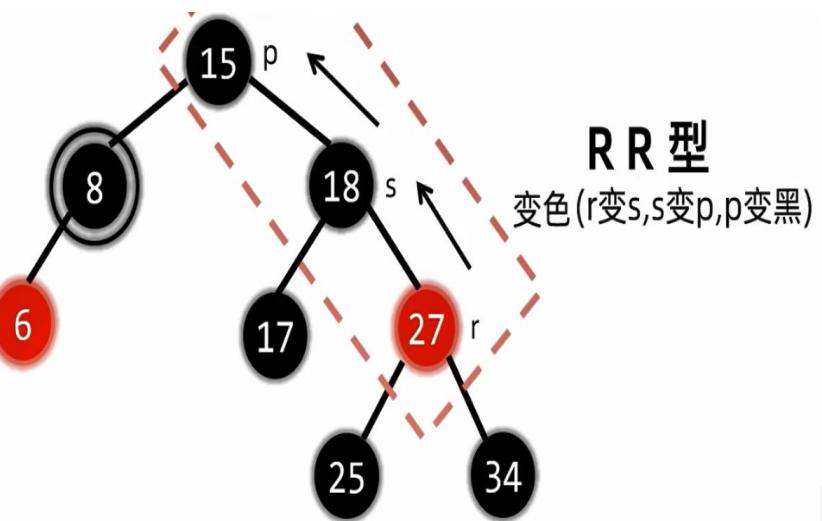
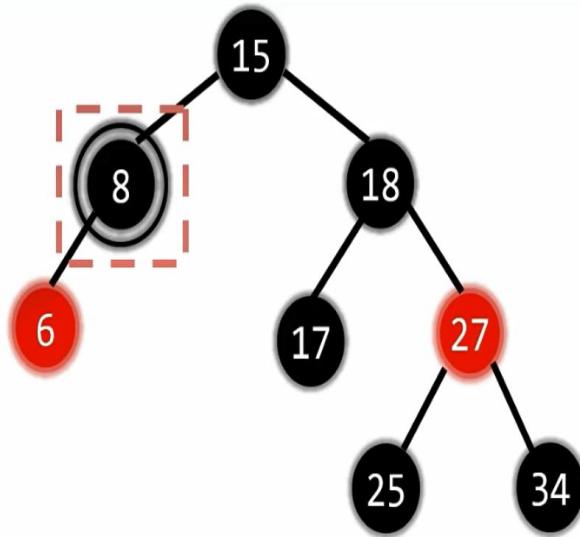
兄弟变红，双黑上移



# 红黑树删除调整 归类

- Case1: 兄弟 S 红, 僵节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S 黑, 僵节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S 黑, 僵节点 C, D 黑色, 父节点 P 黑

兄黑,  
侄黑



RR型

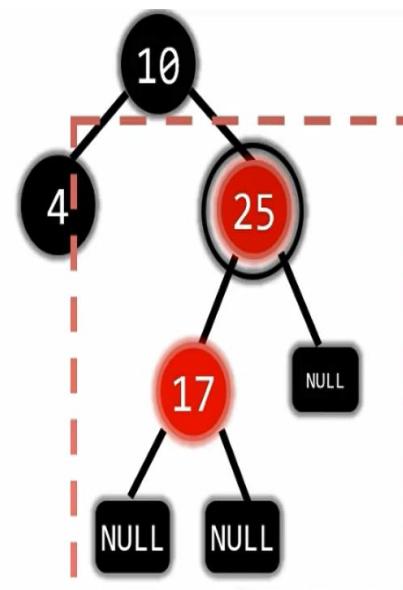
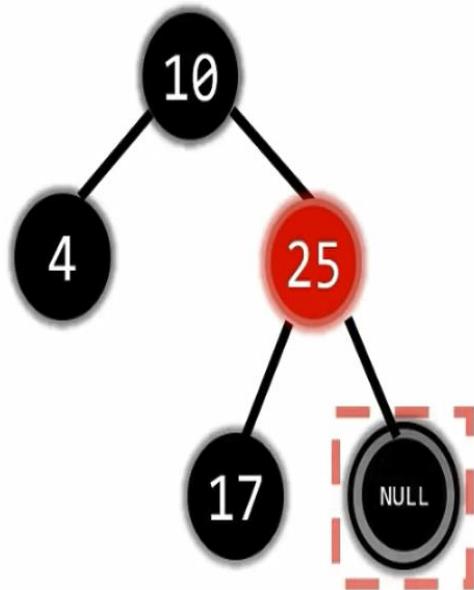
变色(r变s,s变p,p变黑)



# 红黑树删除调整 归类

- Case1: 兄弟 S 红, 倾节点 C, D 黑色, 父节点 P 黑
- Case2: 兄弟 S 黑, 倾节点 C, D 黑色, 父节点 P 红
- Case3: 兄弟 S 黑, 倾节点 C, D 黑色, 父节点 P 黑

兄黑,  
侄黑



兄弟变红  
双黑上移  
遇红变单黑

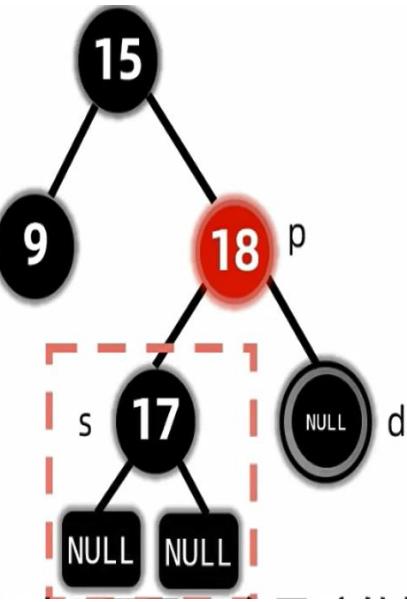
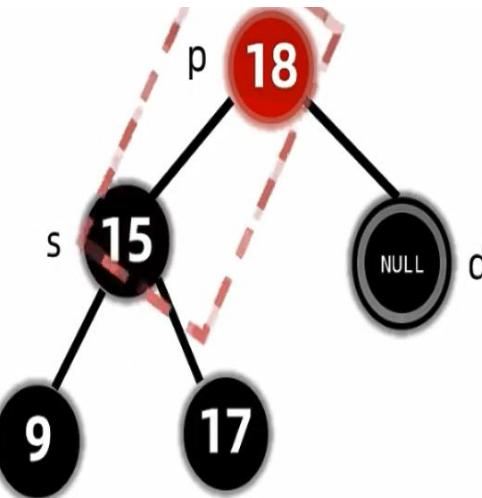
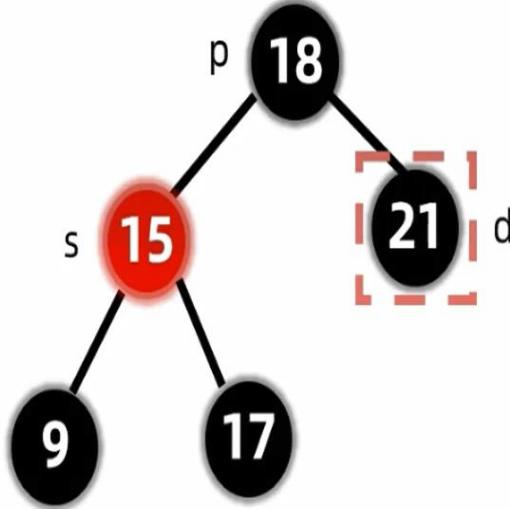


2024/12/9

# 红黑树删除调整 归类

- Case1: 兄弟 S 红, 倾节点 C, D 黑色, 父节点 P 黑

兄红

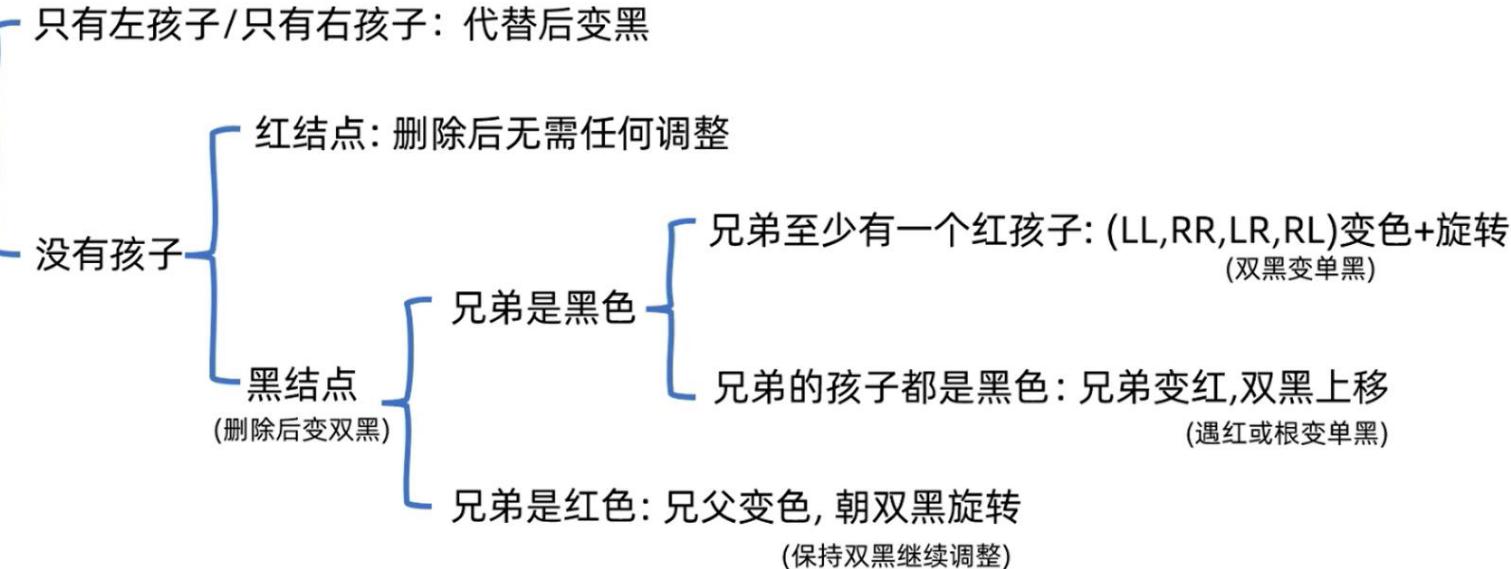


兄父变色，双黑旋转



# 总结：红黑树删除

## 红黑树 删除



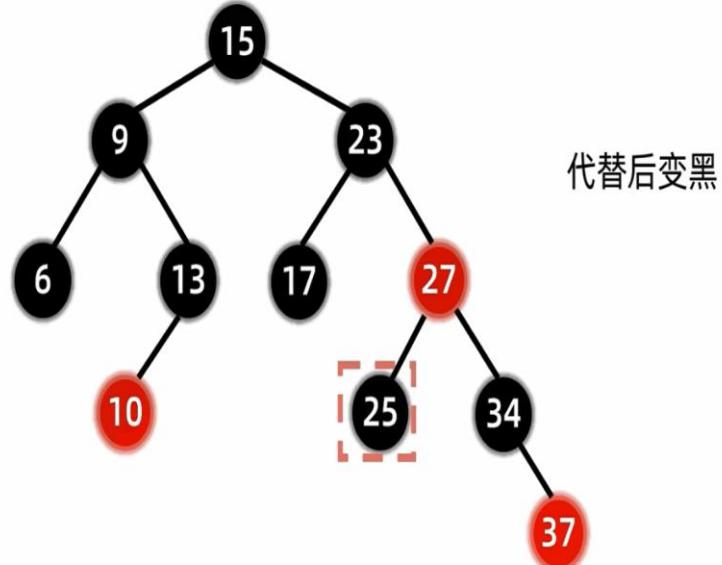
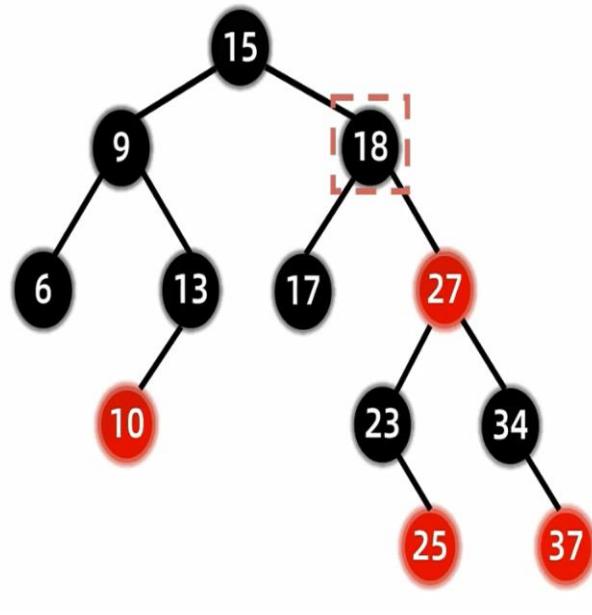
## • 参考资料

- ◆ [https://www.bilibili.com/video/BV16m421u7Tb/?spm\\_id\\_from=333.788.recommend\\_more\\_video.0&vd\\_source=459f25d1b71e72fac82c5bee26e60bb7](https://www.bilibili.com/video/BV16m421u7Tb/?spm_id_from=333.788.recommend_more_video.0&vd_source=459f25d1b71e72fac82c5bee26e60bb7)
- ◆ 操作可视化网址：<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>



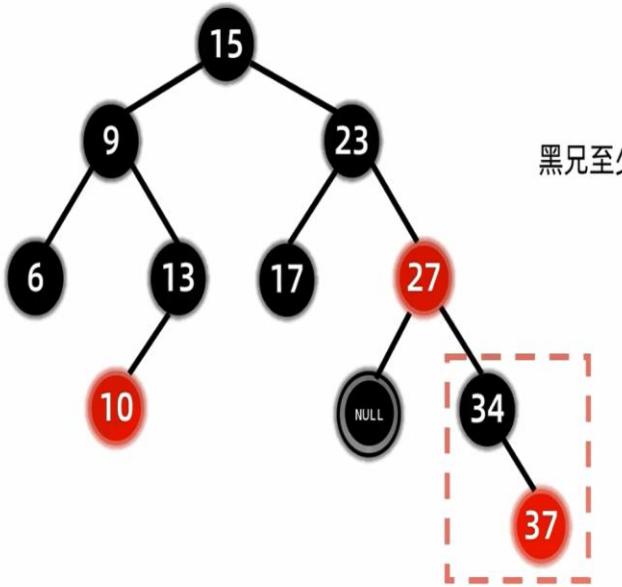
# 红黑树删除示例

18 25 15 6 13 37 27 17 34 9 10 23

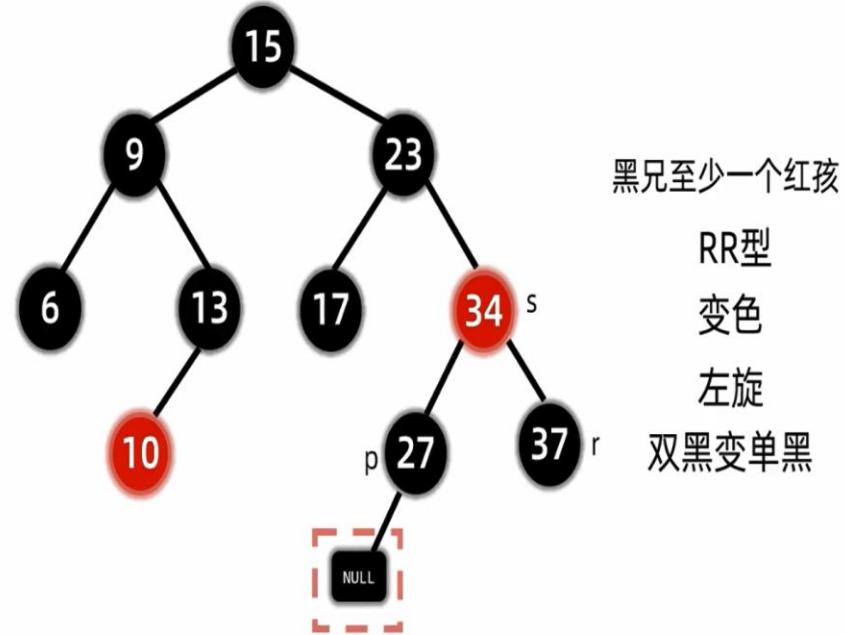


# 红黑树删除示例

25 15 6 13 37 27 17 34 9 10 23



黑兄至少一个红孩



黑兄至少一个红孩

RR型

变色

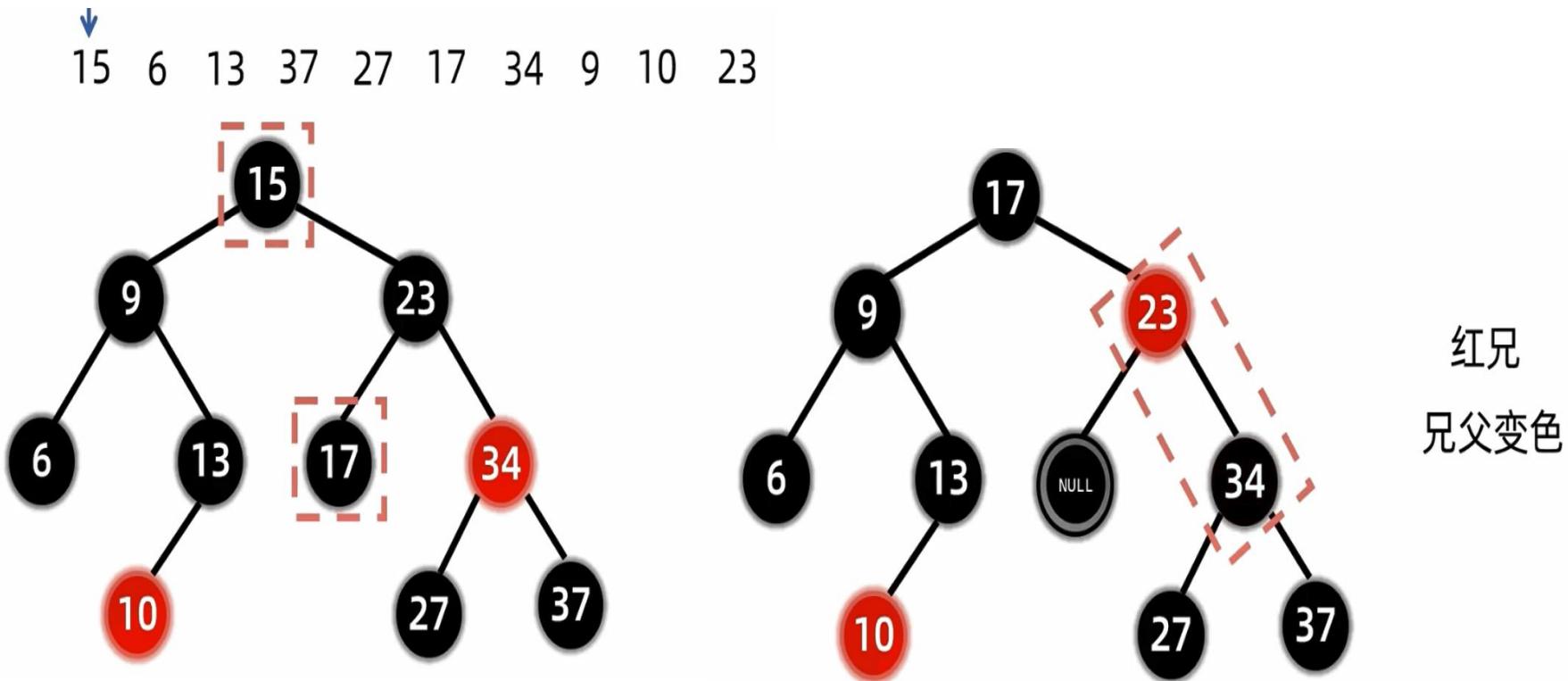
左旋

双黑变单黑



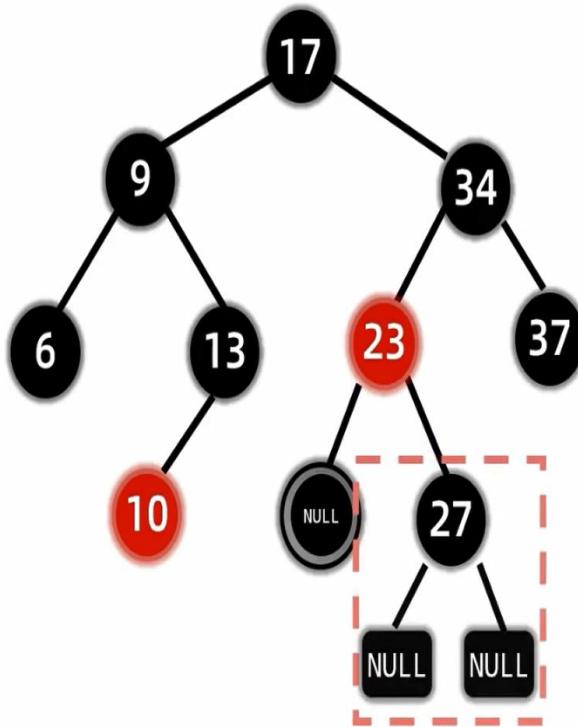
2024/12/9

# 红黑树删除示例

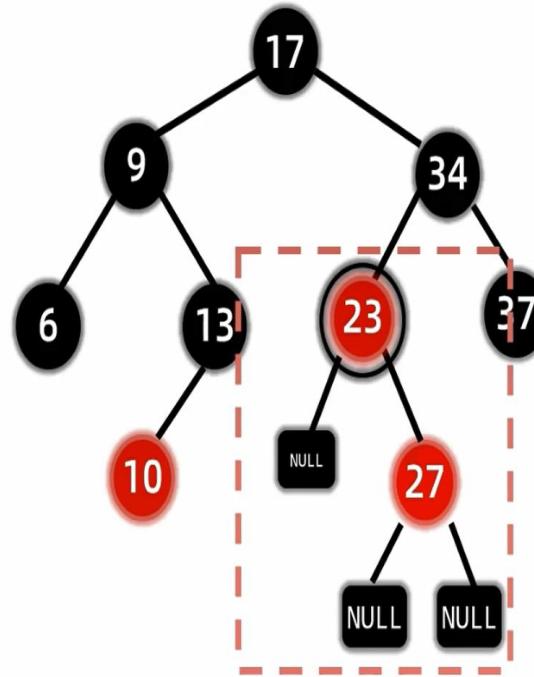


2024/12/9

# 红黑树删除示例



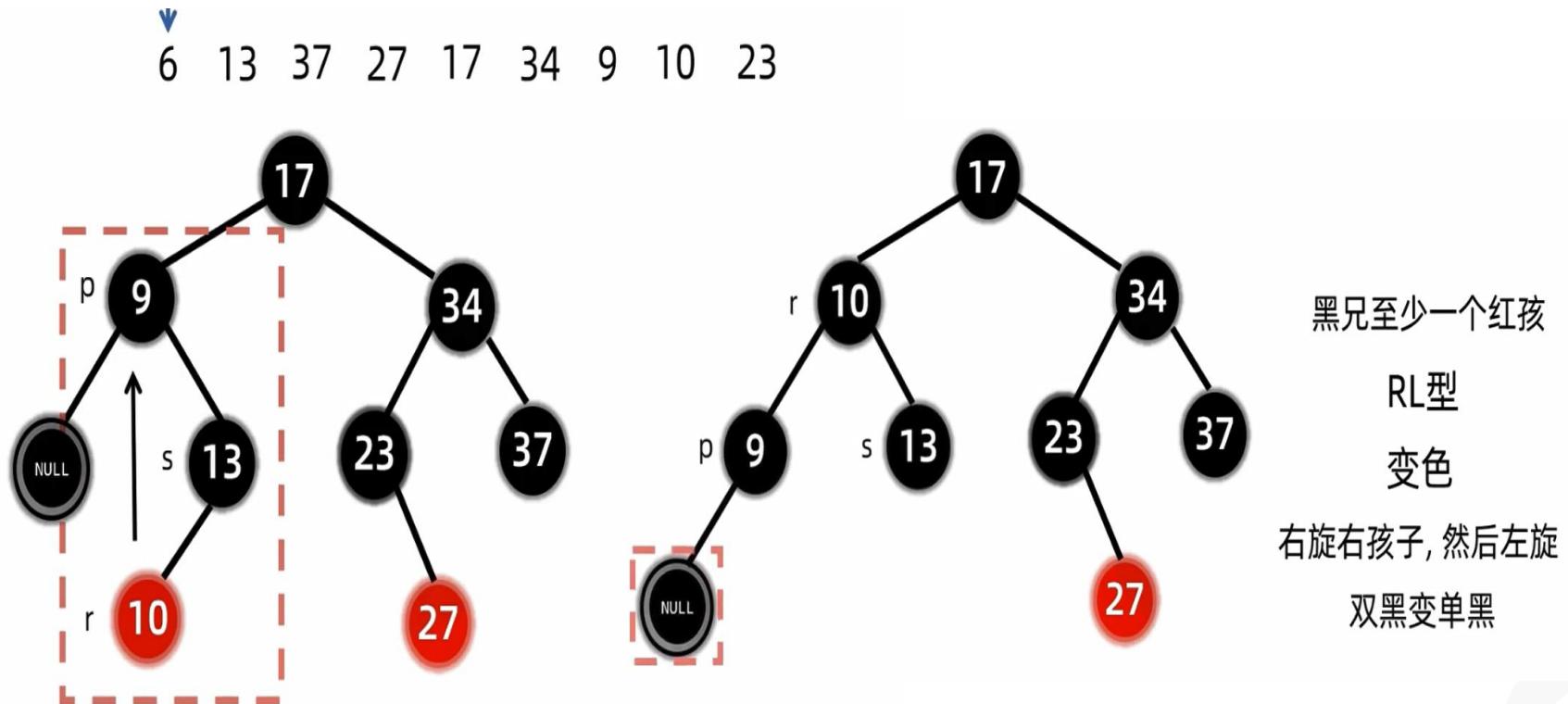
黑兄全黑孩



黑兄全黑孩  
兄弟变红  
双黑上移



# 红黑树删除示例



2024/12/9

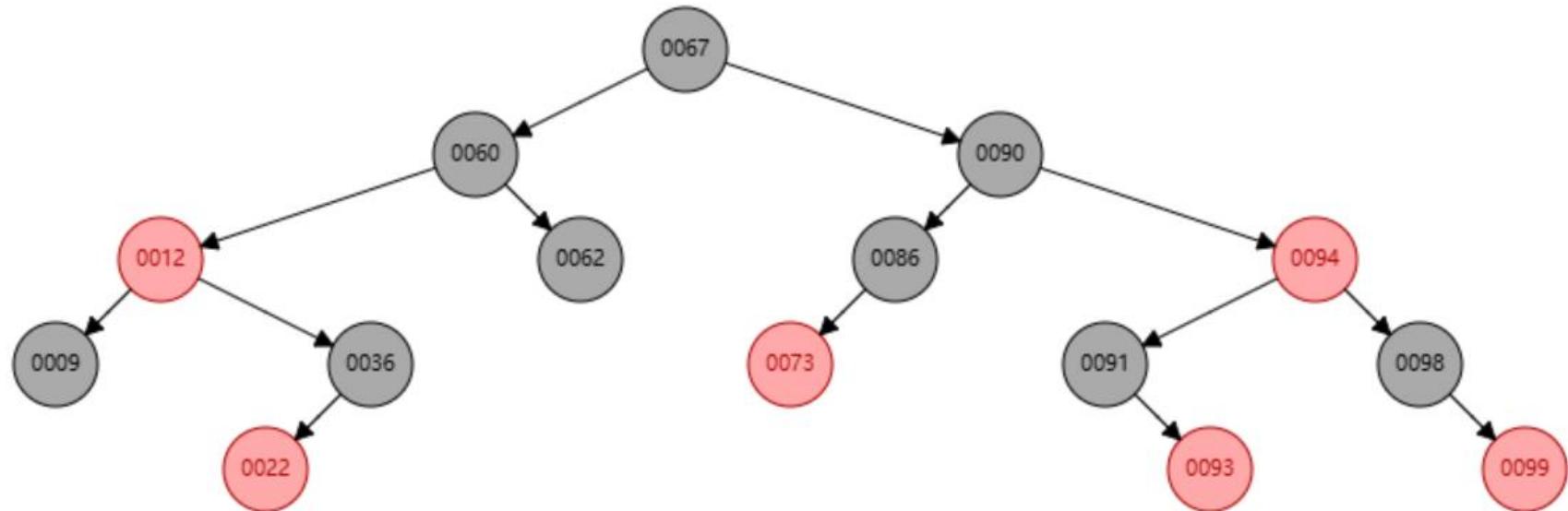


Software Engineering

167

# 红黑树删除练习

依次删除 67, 86 62 (直接前驱替代)



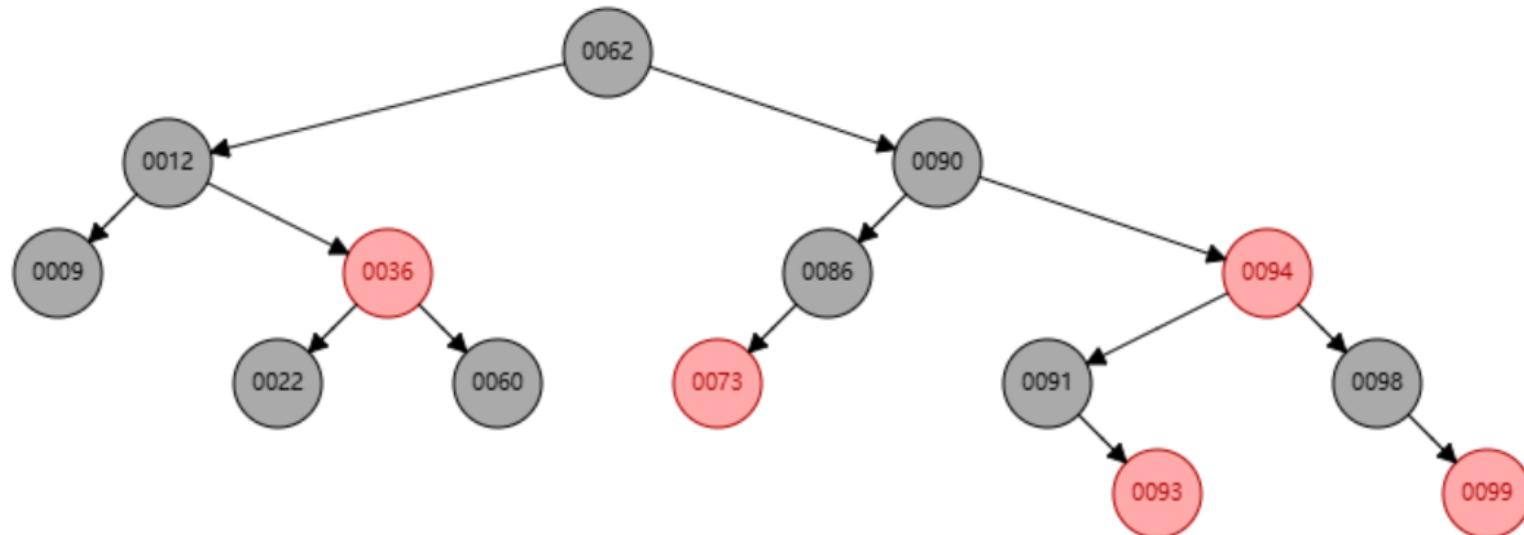
属于兄红，  
父兄变色，朝向双黑旋转。递归维护



2024/12/9

# 红黑树删除练习

依次删除 67, 86 62 (需要时直接前驱替代)



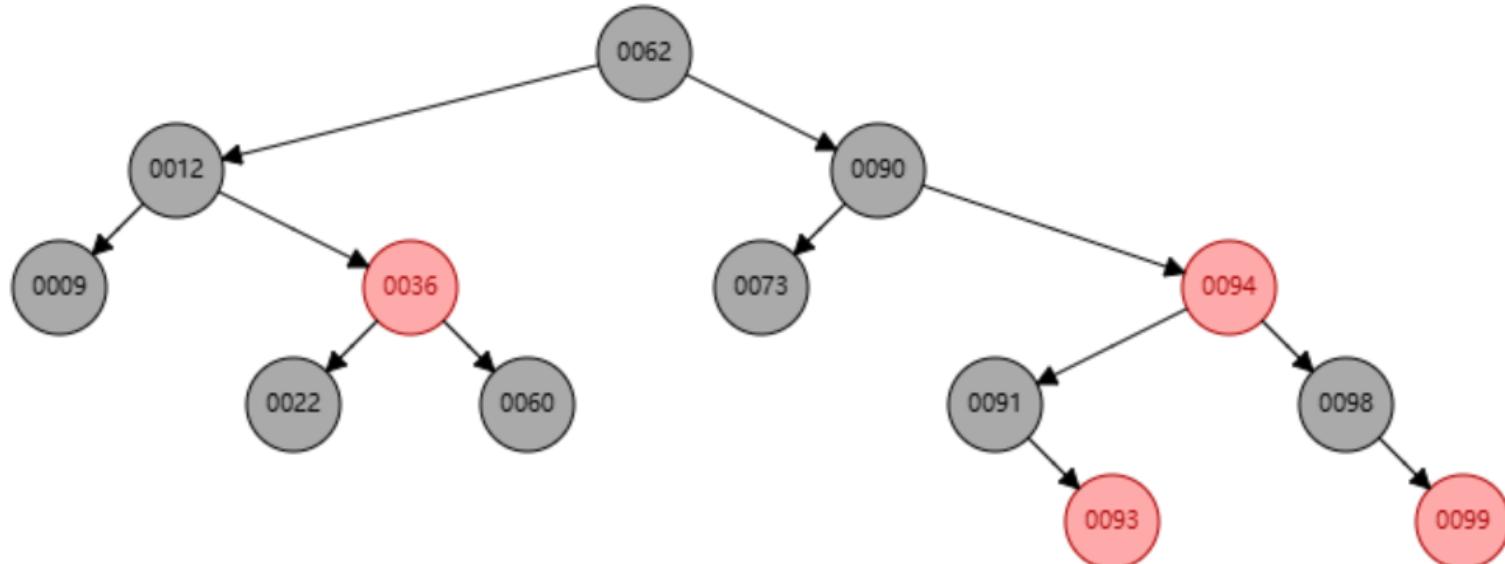
兄黑，远侄红->LL型  
颜色传递+旋转



2024/12/9

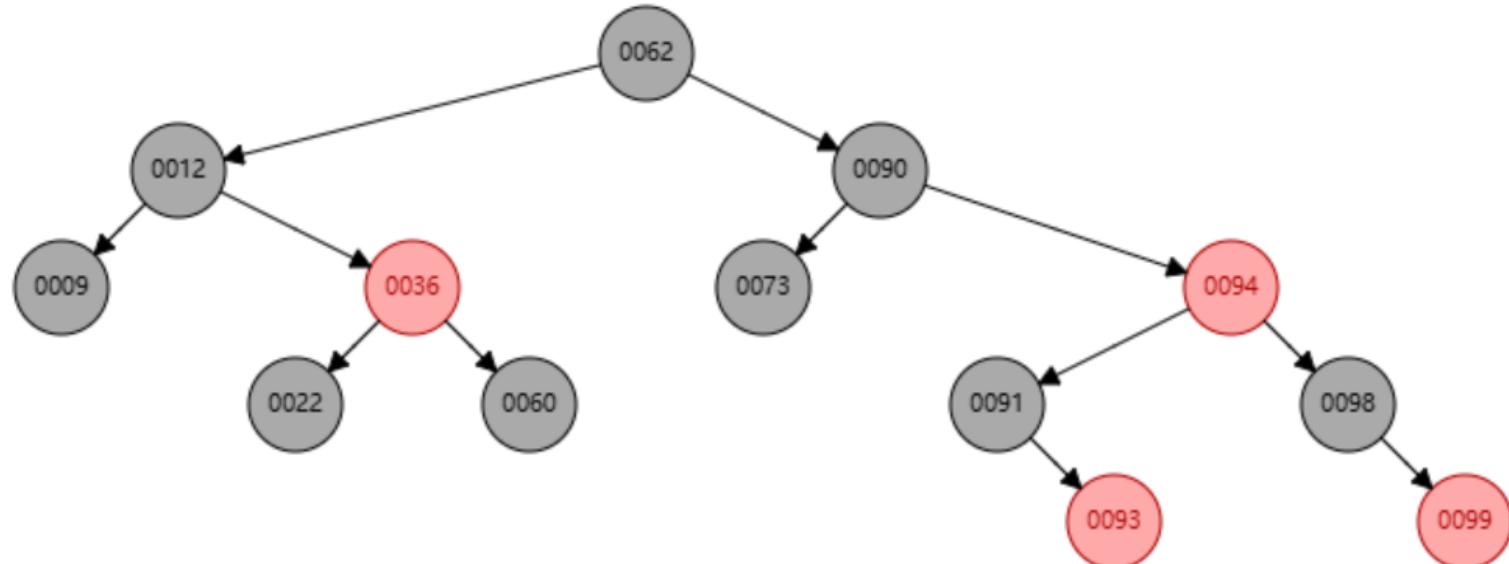
# 红黑树删除练习

依次删除 67, 86 62 (需要时直接前驱替代)



# 红黑树删除练习

依次删除 67, 86, 62 (需要时直接前驱替代)

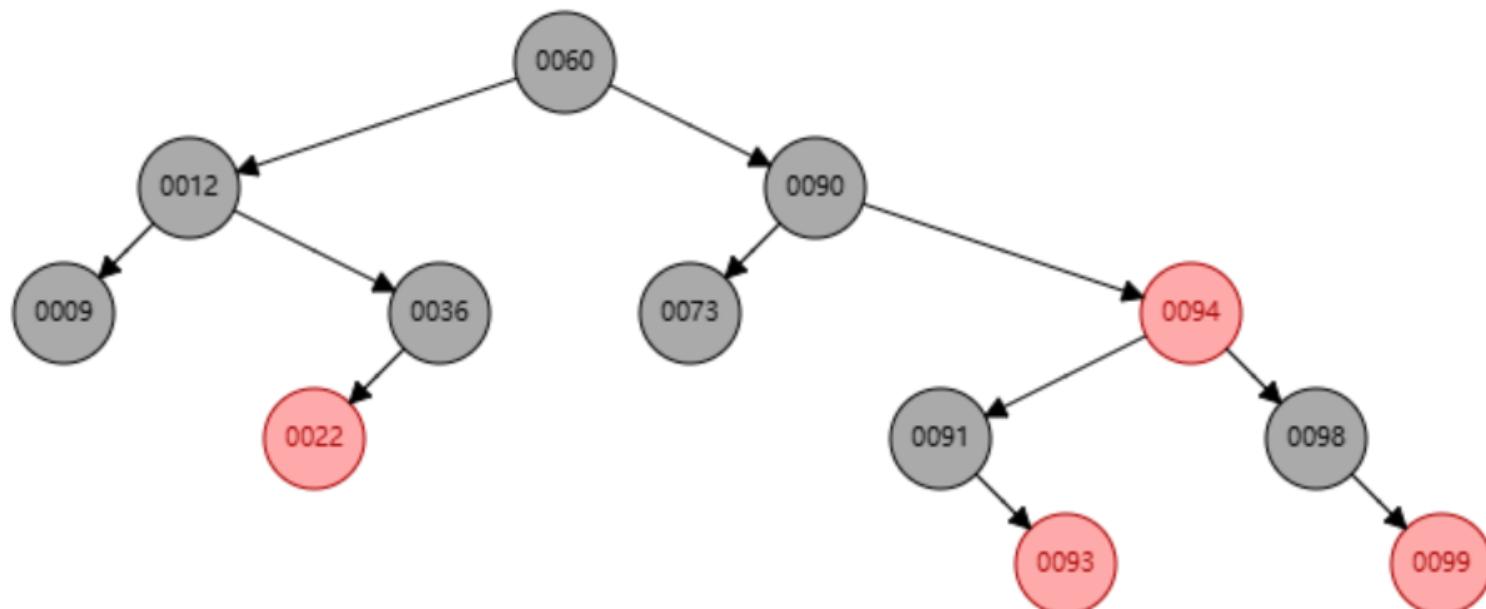


兄黑，侄子全黑  
兄变红，双黑上移，递归维护



# 红黑树删除练习

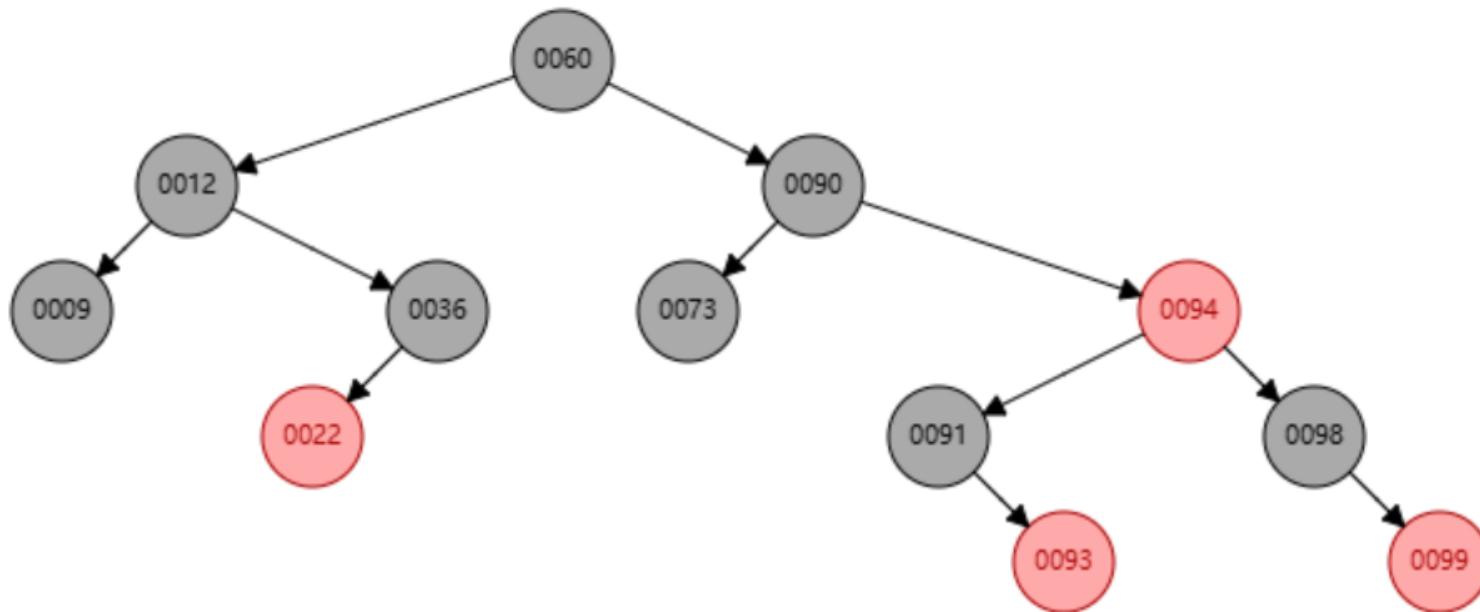
依次删除 67, 86, 62 (需要时直接前驱替代)

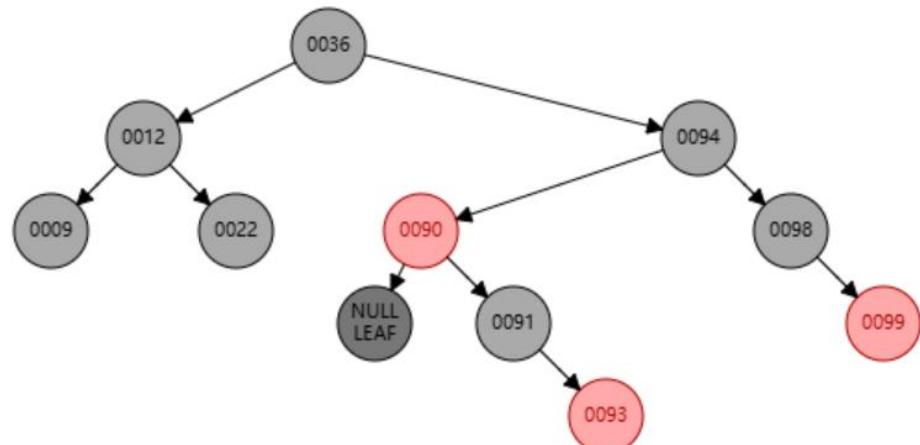


# 红黑树删除练习

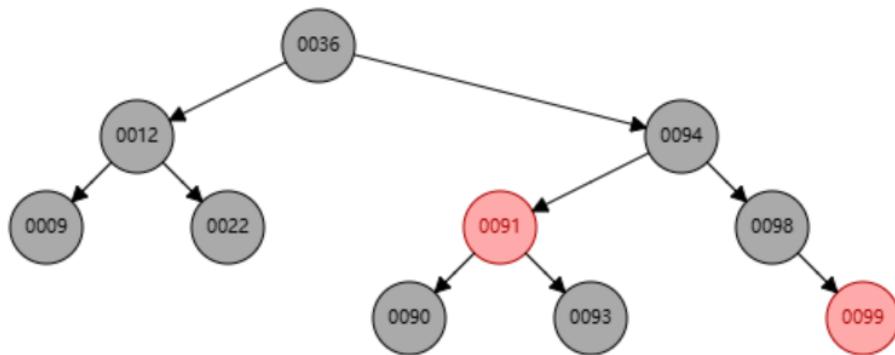
依次删除 67, 86, 62 (需要时直接前驱替代)

接着删除 60, 73, 90

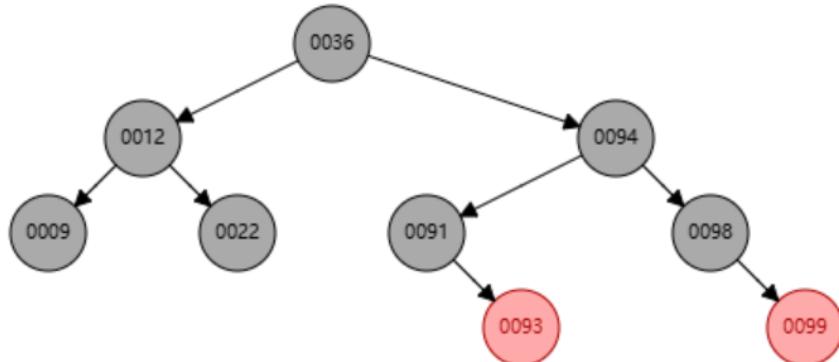




删除73



删除90



# 课堂练习⑧ (红黑树)

1) 从空树出发, 待插的关键字序列为  
**1,2,3,4,5,6,7,8**

2) 从空树出发, 待插的关键字序列为  
**8,7,6,5,4,3,2,1**

3) 从空树出发, 待插的关键字序列为  
**10,20,30,40,35,28,8,9**

