



计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：张羽丰

湖南大学 信息科学与工程学院

第 12 章

高级查找

提纲

12.1 问题引入：网上购物

12.2 线段树

12.3 跳表

12.4 红黑树

提纲

12.2.1 线段树的定义

12.2.2 线段树的存储

12.2.3 线段树的动态维护操作

12.2.4 小结



12.2.1 问题引入

- 给定数列 $[a_0, a_1, \dots, a_{n-1}]$, 求下标区间 $[l, r]$ 上的最小值

- 直接做法

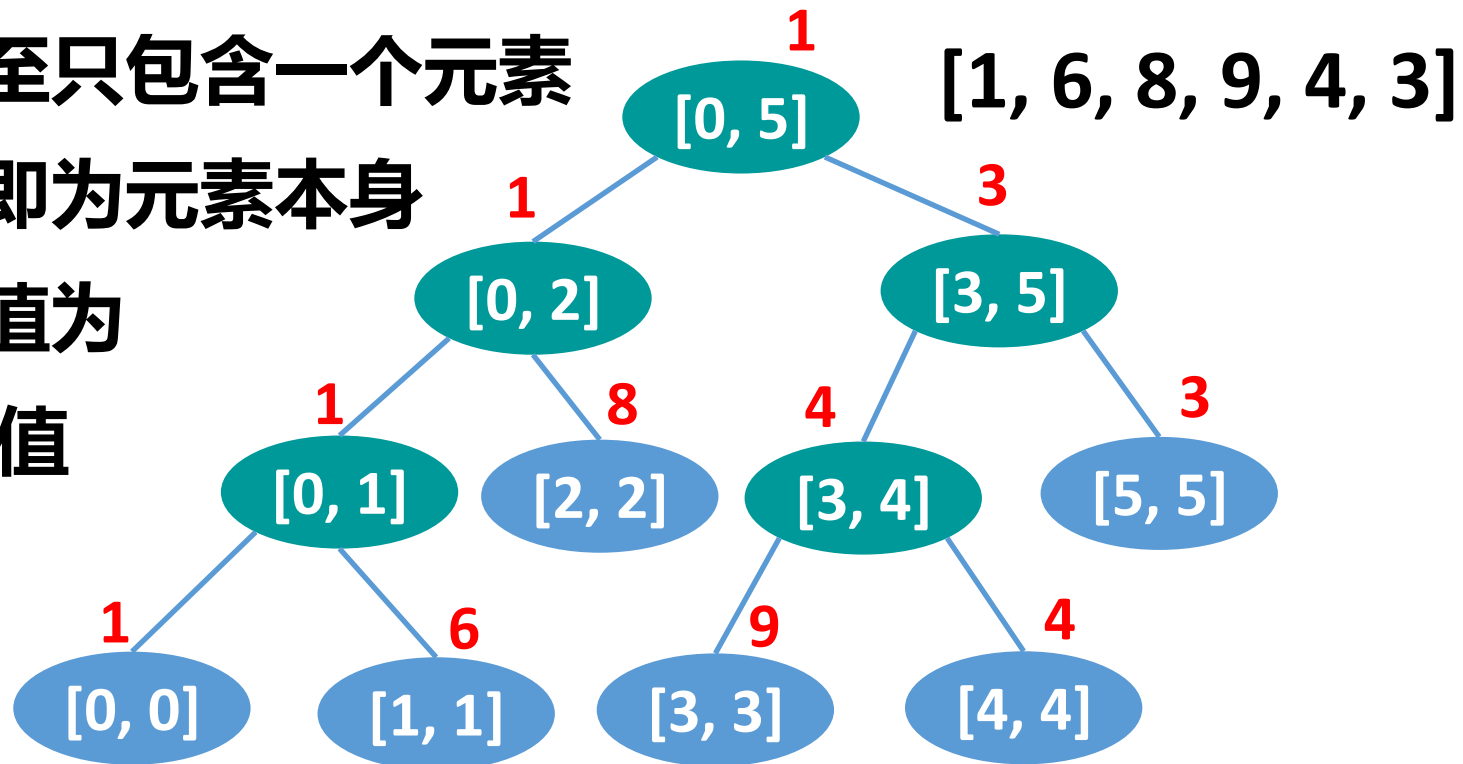
```
int minValue(int a[], int l, int r) {  
    int min = a[r];  
    for (int i = l; i < r; ++i) {  
        min = std::min(a[i], min);  
    }  
    return min;  
}
```

- 如果在不同区间执行 m 次求最小值操作
 - 平均区间长度 $n/2$, 时间复杂度 $O(nm)$
 - 有没有更高效的算法?



问题引入

- 将整个区间组织成二叉树的结构，预先计算每个子区间的最小值，基于子区间的最小值，计算给定区间的最小值
 - 递归的将整个区间二分至只包含一个元素
 - 一个元素区间的最小值即为元素本身
 - 父节点的区间元素最小值为左右孩子节点值的最小值





问题引入

- 如何求任意区间 $[l, r]$ 的最小值?

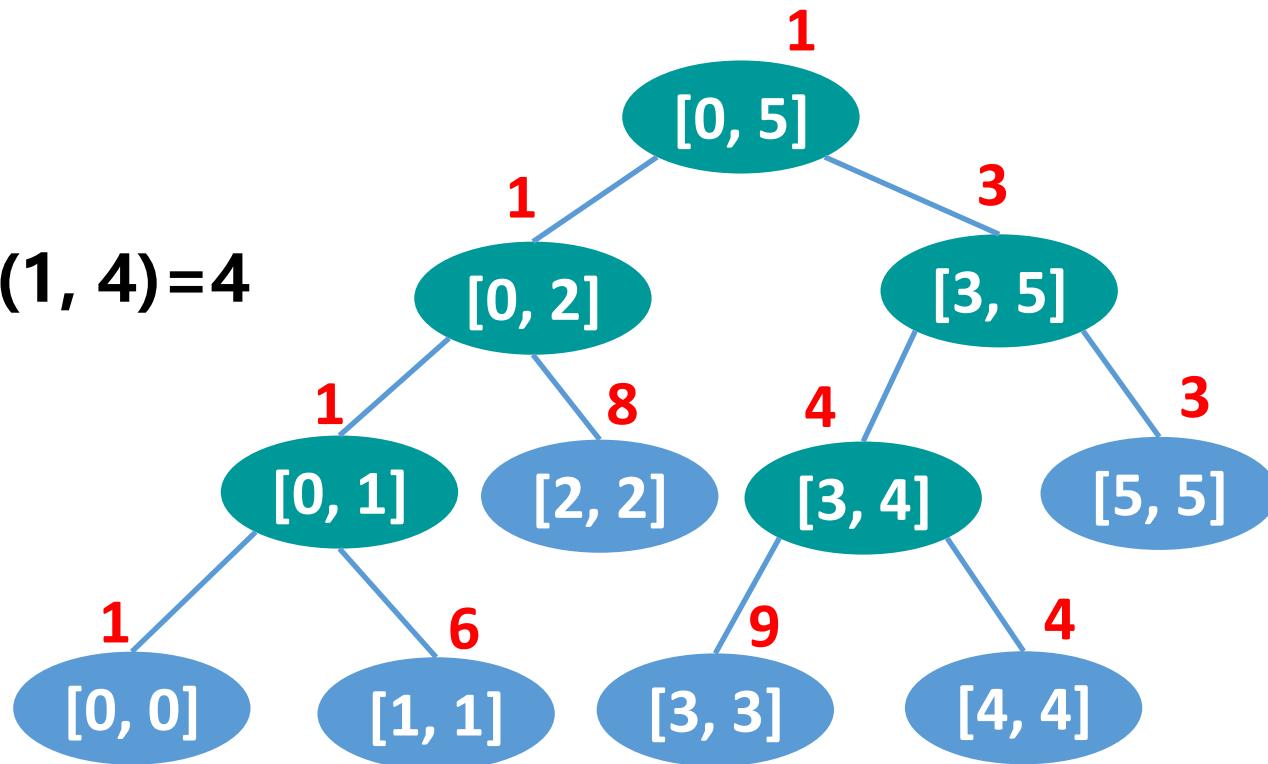
$[1, 6, 8, 9, 4, 3]$

- 查找 $[0, 4]$ 范围内的元素的最小值

- 区间 $[0, 2]$ 上的最小值为1
- 区间 $[3, 4]$ 上的最小值为4
- 区间 $[0, 4]$ 上的最小值为 $\min(1, 4) = 1$

- 时间复杂度 $O(\log_2 n)$

- 也可应用于求区间的和、最大值等





线段树的定义

- 线段树是满足如下条件的二叉查找树:

[1, 6, 8, 9, 4, 3]

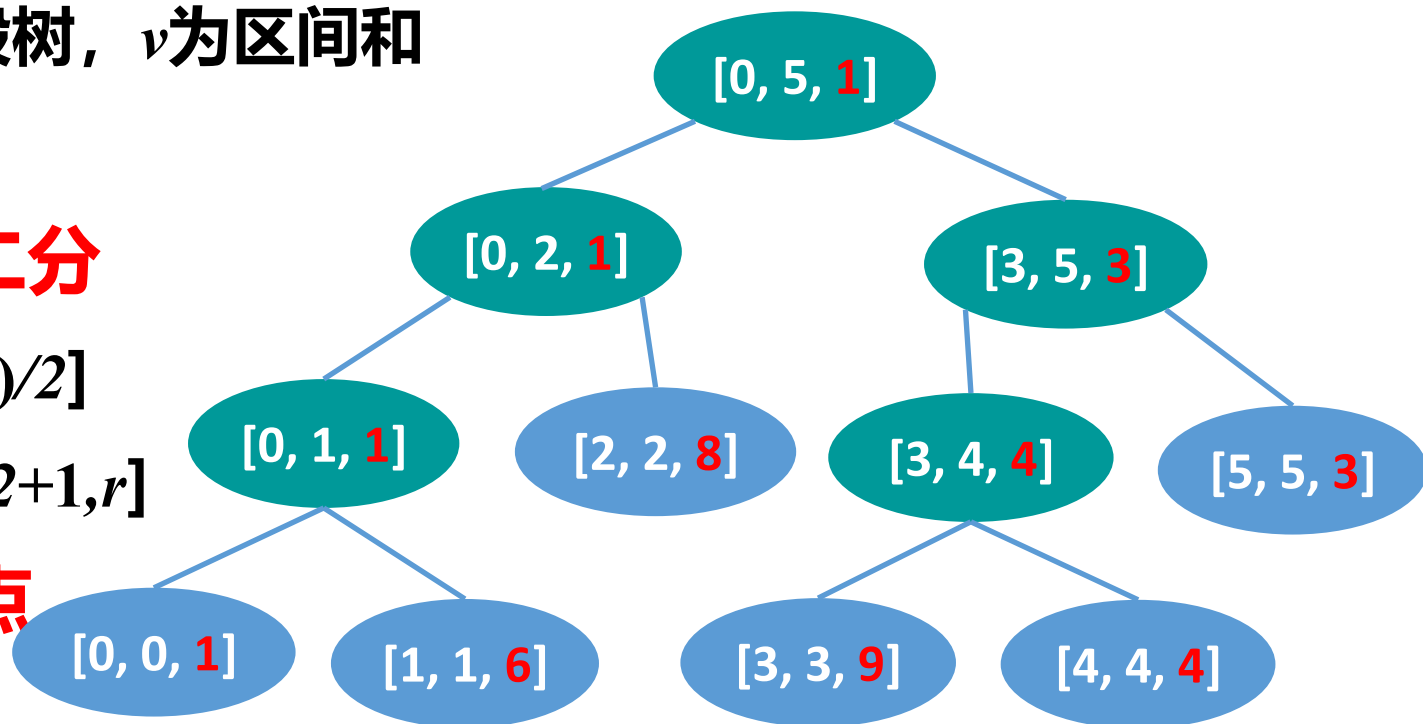
- 每个结点保存**区间范围** $[l, r]$ 及该**区间上的值** v

- 为查找区间最小值而设计的线段树, v 为最小值
- 为求区间和而设计的线段树, v 为区间和
- 区间长度 $n = r - l + 1$

- 若**区间长度** $n > 1$, 进行二分

- 左孩子结点区间为 $[l, (l+r)/2]$
- 右孩子结点区间为 $[(l+r)/2+1, r]$

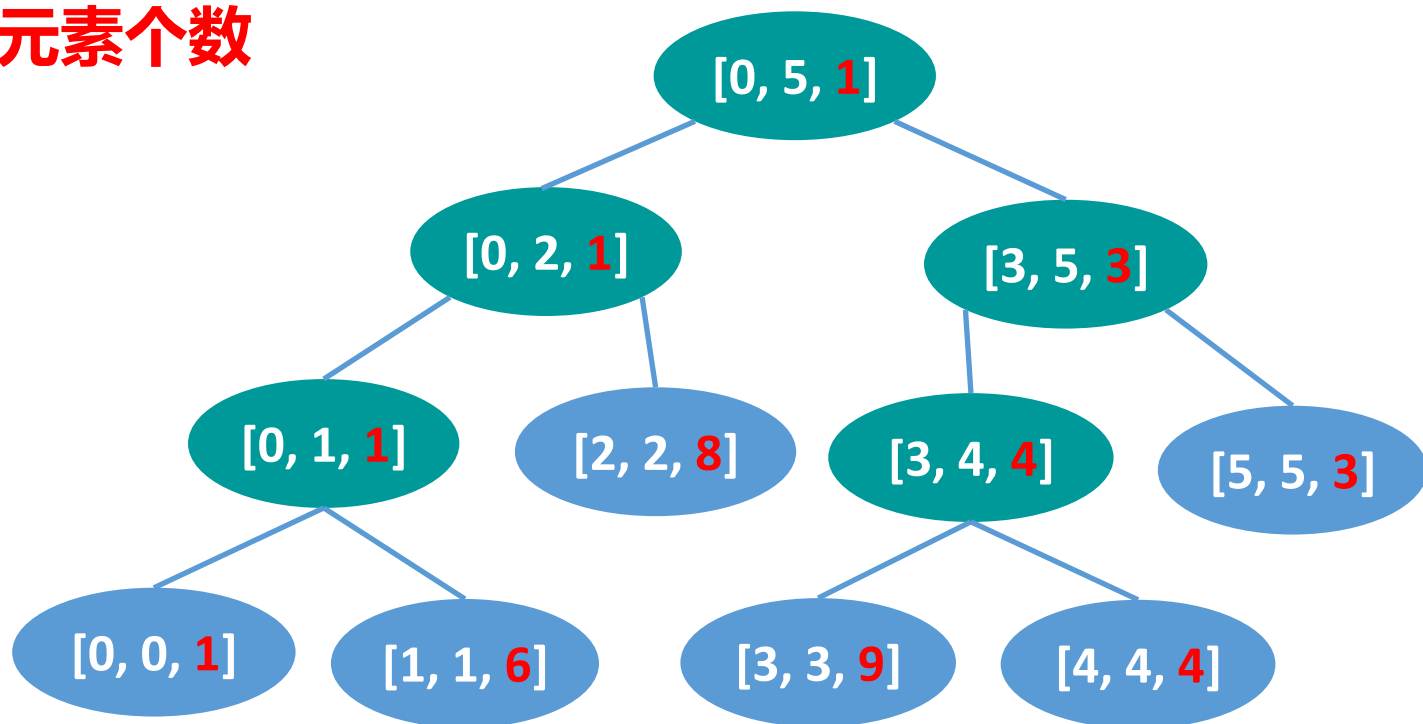
- 若**区间长度** $n = 1$ 为叶结点





线段树的定义

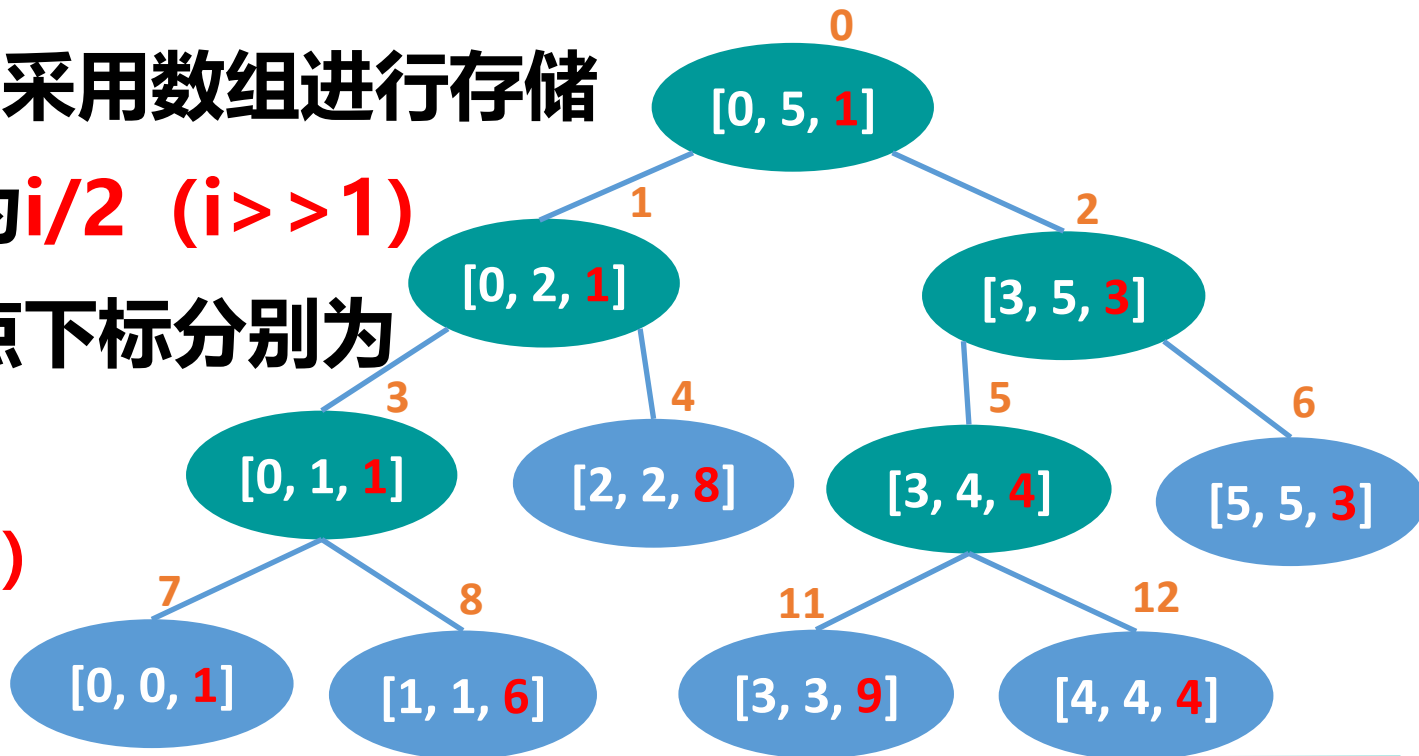
- 线段树采用二分法进行构造，因此是平衡二叉树
 - 对于整个区间长度为 n 的线段树，结点总数为 $2n-1$
 - 叶子结点个数 = 内部结点个数 + 1
 - 叶子结点个数 = 数列中的元素个数
 - 是否为完全二叉树？





12.2.2 线段树的存储

- 线段树是二叉查找树
 - 可以采用二叉链表存储
- 线段树接近完全二叉树
 - 可以**扩充**为完全二叉树，采用数组进行存储
 - 第 i 个结点的父结点下标为 $i/2$ ($i > 1$)
 - 第 i 个结点的左右孩子结点下标分别为
 - 左孩子: $2i$ ($i < 2^k$)
 - 右孩子: $2i+1$ ($i < 2^k - 1$)





线段树的构建

- 构建最小值线段树

- 以数组存储线段树：只存储区间值

- 当数列元素 n 确定，每个结点的区间范围 $[l, r]$ 是确定的，无需存储

```
// seg_tree: 构建的线段树, array: 待构建的数列
// l, r: 当前结点的区间范围, p: 当前结点在数组存储中的下标
BuildSegTree(seg_tree, array, l, r, p) {
    if (l == r) {
        seg_tree[p] = array[l];
    } else {
        m = (l + r) / 2;
        BuildSegTree(seg_tree, array, l, m, 2p);
        BuildSegTree(seg_tree, array, m+1, r, 2p+1);
        seg_tree[p] = min(seg_tree[2p], seg_tree[2p+1]);
    }
}
```

初始调用: BuildSegTree
(seg_tree, array, 0, n-1, 0)

// 叶子结点

// 二分的中点值

// 递归构建左子树

// 递归构建右子树

// 左右子树的最小值



12.2.3 线段树的动态维护操作

- **查询**

- 给定区间 $[ql, qr]$ 上的最小值、最大值、求和

- **更新**

- **单点更新**

- 更新某个元素的值

- **区间更新**

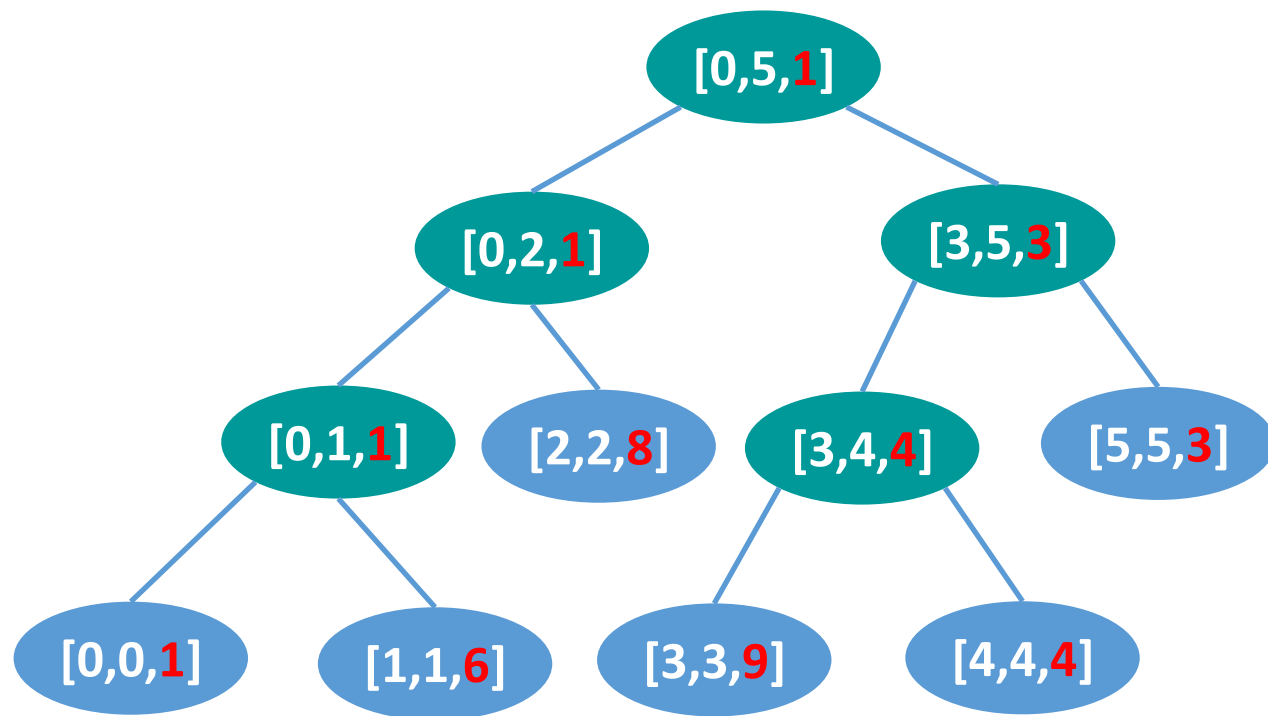
- 更新某个区间范围内的值



查询

- 查询给定区间 $[ql, qr]$ 上的最小值, $[l, r]$ 为当前结点的区间
 - 如果 $[ql, qr] \supseteq [l, r]$, 即 $[ql, qr]$ 完全包含 $[l, r]$
 - 该节点的值, 即为区间 $[ql, qr]$ 上的最小值
 - 如果 $[ql, qr]$ 与 $[l, r]$ 有交集
 - 递归查询左右子树的最小值
 - 左右子树返回值的最小值, 即为区间 $[ql, qr]$ 上的最小值
 - 如果 $[ql, qr]$ 在区间 $[l, r]$ 外
 - 返回一个大的常数 $kMaxNum$
 - 时间复杂度: $O(\log n)$
 - 最多2倍树高

查找区间 $[0, 4]$ 上的最小值





查询

- 查询给定区间 $[ql, qr]$ 上的最小值, $[l, r]$ 为当前结点的区间

```
// seg_tree: 线段树,  
// l, r: 当前结点的区间范围, p: 当前结点在数组存储中的下标,  
Query(seg_tree, l, r, p, ql, qr) {  
    if (qr < l || ql > r) {  
        return kMaxNum;  
    } else if (ql <= l && qr >= r) {  
        return seg_tree[p];  
    } else {  
        m = (l + r) / 2;  
        left = Query(seg_tree, l, m, 2p, ql, qr);  
        right = Query(seg_tree, m+1, r, 2p+1, ql, qr);  
        return min(left, right);  
    }  
}
```

初始调用: `Query(seg_tree, 0, n-1, 0, ql, qr)`

//[ql, qr]与[l, r]无交集

//[ql, qr]完全包含[l, r]

//[ql, qr]与[l, r]有交集

// 二分的中点值

//递归查询左子树

//递归查询右子树

//左右子树的最小值



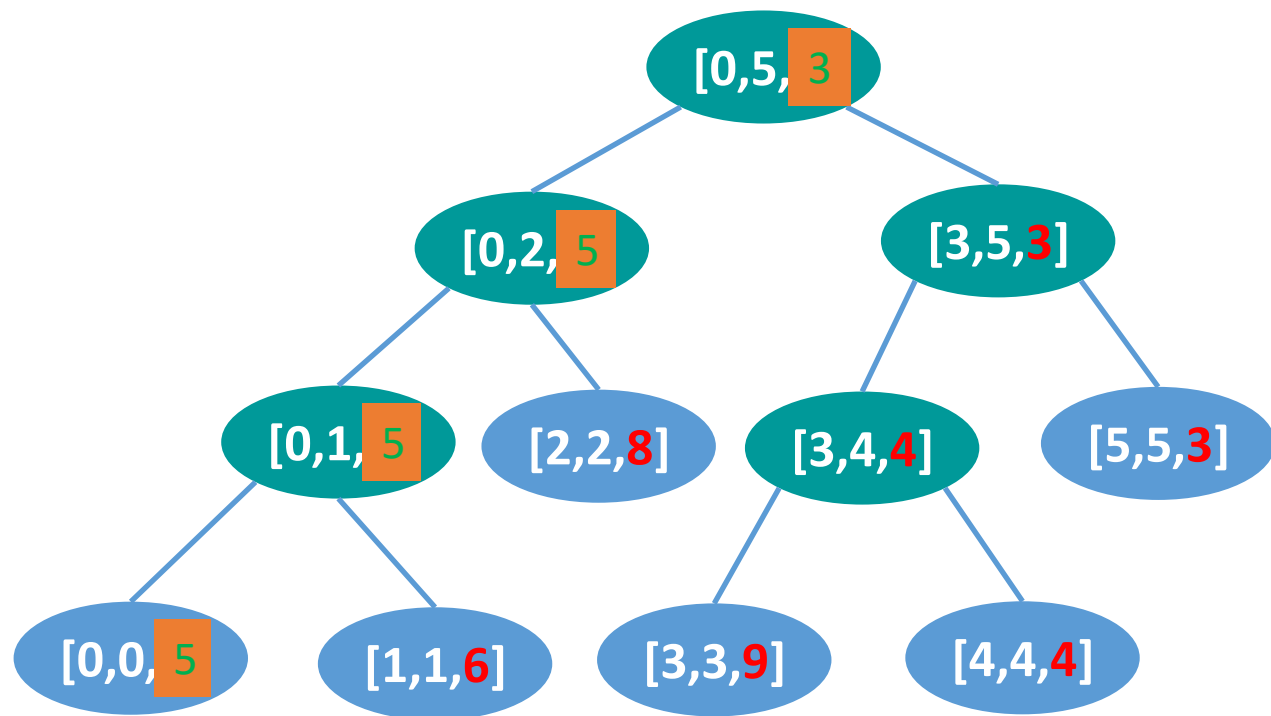
单点更新

- 修改数列array中下标为idx的值
 - 按照二分法，查找array[idx]所在的叶结点
 - 回溯时更新区间的值
 - 时间复杂度： $O(\log n)$
 - 2倍树高

[1, 6, 8, 9, 4, 3]



[5, 6, 8, 9, 4, 3]





单点更新

- 修改数列array中下标为idx的值为value

```
// seg_tree: 线段树,  
// l, r: 当前结点的区间范围, p: 当前结点在数组存储中的下标  
Update(seg_tree, l, r, p, idx, value) {  
    if (l==r) {  
        seg_tree[p] = value;  
    } else {  
        m = (l + r) / 2;  
        if (idx <= m)  
            Update(seg_tree, l, m, 2p, idx, value);  
        else  
            Update(seg_tree, m+1, r, 2p+1, idx, value);  
        seg_tree[p] = min(seg_tree[2p], seg_tree[2p+1]);  
    }  
}
```

//叶子结点

//非叶子结点

// 二分的中点值
//递归更新左子树

//递归更新右子树
//回溯更新

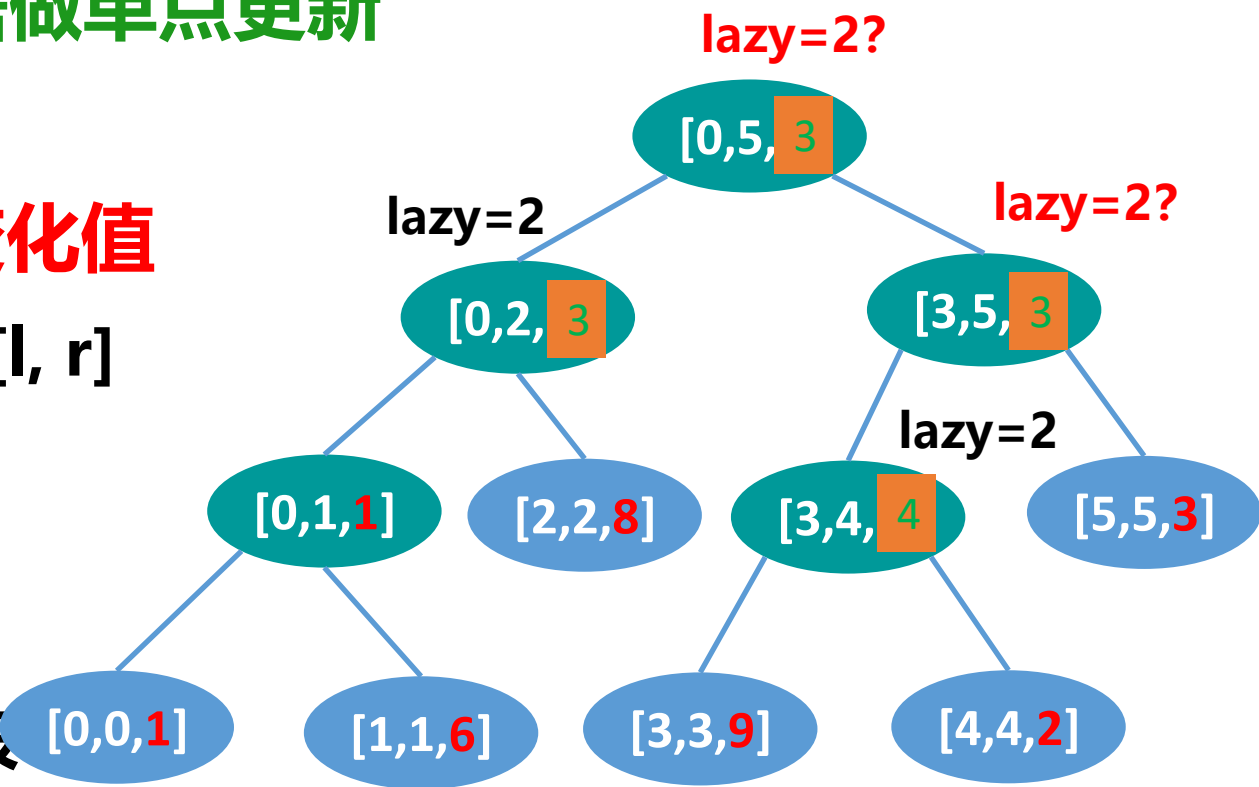
初始调用:
Update(seg_tree, 0, n-1, 0,
idx, value)



区间更新

- 对指定区间 $[ql, qr]$ 内的数据做统一修改
 - 如：将区间 $[ql, qr]$ 内的所有数据减去C
 - 直接的方法：对区间内每个数据做单点更新
 - 时间复杂度为 $O(n\log n)$ ，效率低
 - 延迟更新：增加lazy字段记录变化值
 - 若区间 $[ql, qr]$ 完全包含结点区间 $[l, r]$
 - 设置lazy变化量，更新结点值
 - 对孩子结点不进行递归更新
 - 孩子结点的更新等到查询时进行
 - 部分包含的区间能否设置lazy字段

对区间 $[0, 4]$ 加2





区间更新

- 对指定区间 $[ql, qr]$ 内的数据做统一修改

- 延迟更新：增加 $lazy$ 字段记录变化值

- 若区间 $[ql, qr]$ 完全包含结点区间 $[l, r]$

- 设置 $lazy$ 变化量： $lazy = lazy + c$

- 更新结点值： $v = v + c$

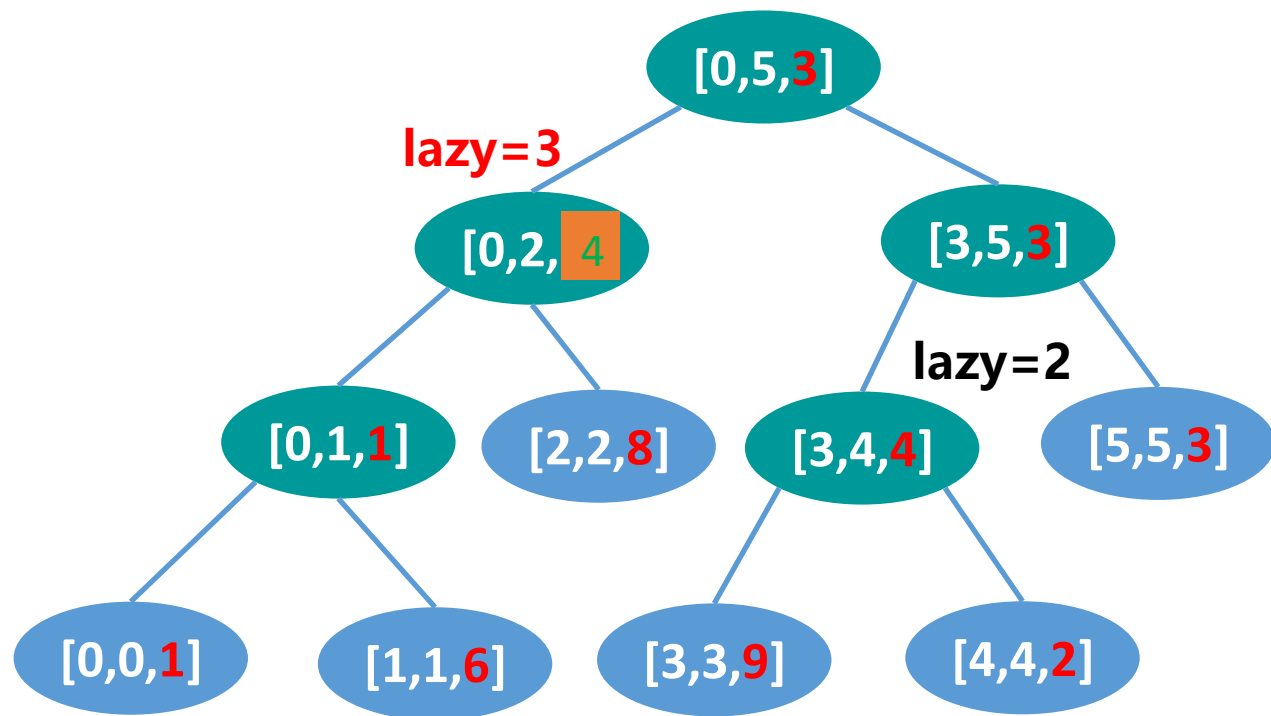
- 对孩子结点不进行递归更新

- 否则：区间有交集

- 对左右子树递归的做区间更新

- 回溯更新节点值

继续对区间 $[0, 2]$ 加1





区间更新

- 对指定区间 $[ql, qr]$ 内的数据做统一修改

- 延迟更新：增加 $lazy$ 字段记录变化值

- 若区间 $[ql, qr]$ 完全包含结点区间 $[l, r]$

- 设置 $lazy$ 变化量： $lazy = lazy + c$

- 更新结点值： $v = v + c$

- 对孩子结点不进行递归更新

- 否则：区间有交集

- 对左右子树递归的做区间更新

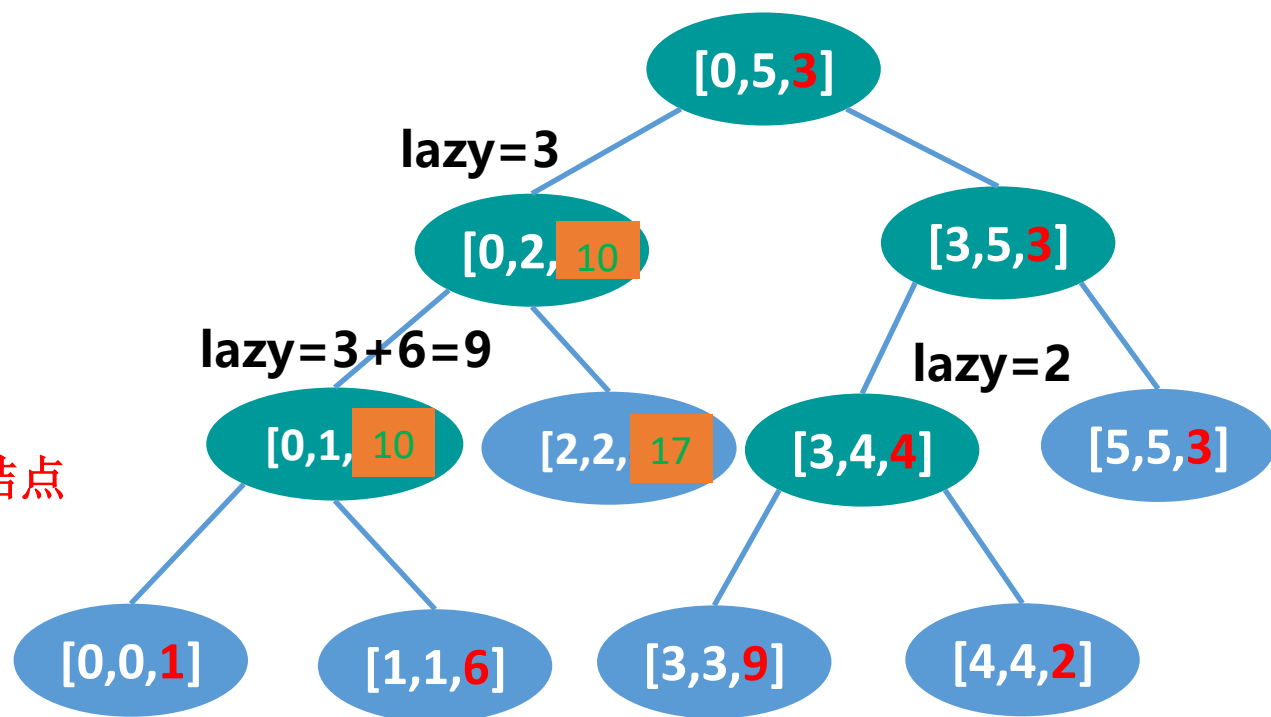
- 下推该结点的 $lazy$ 字段至左右子结点

- 回溯更新节点值

- 清除 $lazy$ 字段

- 时间复杂度： $O(\log n)$

继续对区间 $[0, 1]$ 加6





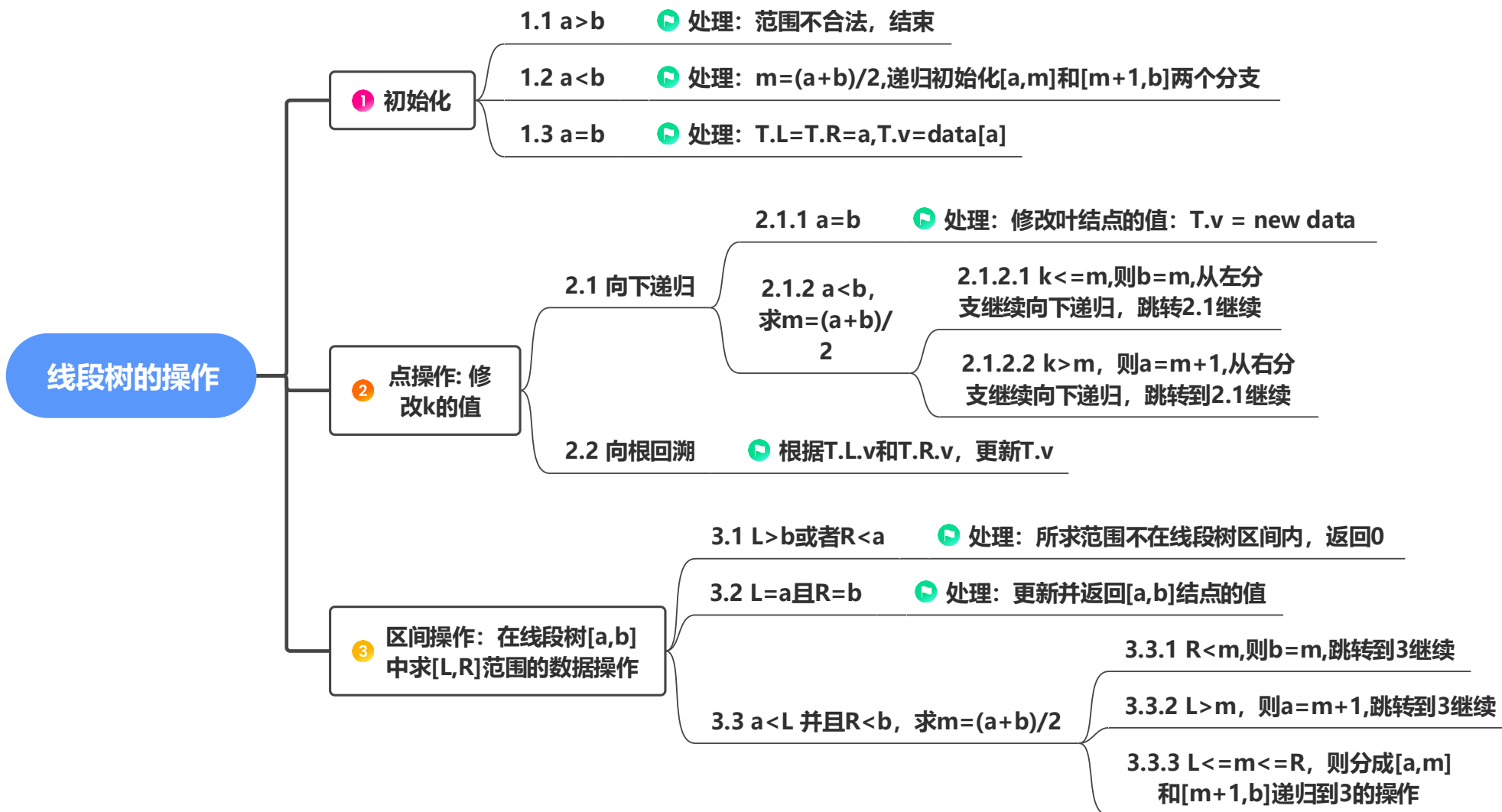
区间更新

- 对指定区间 $[ql, qr]$ 内的数据做统一修改

```
// seg_tree: 线段树, lazy: 延迟标志, ql, qr: 待修改区间, c: 区间值的增量
// l, r: 当前结点的区间范围, p: 当前结点在数组存储中的下标,
RangeUpdate(seg_tree, lazy, l, r, p, ql, qr, c) {
    if (ql<=l && r<=qr) {                                //[ql, qr]完全包含[l, r]
        lazy[p] = lazy[p] + c;                             //加延迟标志
        seg_tree[p] = seg_tree[p] + lazy[p];              //单点更新结点
    } else if (qr>=l && ql<=r) {                            //[ql, qr]与[l, r]有交集
        if (lazy[p] != 0) {                                //下推延迟标志
            lazy[2p] += lazy[p];                          //更新左孩子
            lazy[2p+1] += lazy[p];                        //更新右孩子
            lazy[p] = 0;
        }
        m = (l + r) / 2;                                    // 二分的中点值
        RangeUpdate(seg_tree, lazy, l, m, 2p, ql, qr, c); //递归更新左子树
        RangeUpdate(seg_tree, lazy, m+1, r, 2p+1, ql, qr, c); //递归更新右子树
        seg_tree[p] = min(seg_tree[2p], seg_tree[2p+1]); //回溯更新
    }
}
```



线段树的操作





习题

例题：已知10000个正整数，用A数组存储这10000个正整数。其中A[i]表示存储的第i个数($0 \leq i < 10000$)。

- (1) 编号从L到R的所有数之和为多少？其中 $0 \leq L \leq R \leq 10000$
- (2) 如果将第k个数增加C ($0 \leq k < 10000$)，则编号[L,R]的所有数之和是多少？
- (3) 如果将[m, n]区间的所有数都增加C，则编号[L,R]的所有数之和是多少？



12.2.4 树状数组——问题导入

- 给定顺序表 $A = [a_1, a_2, \dots, a_n]$ ，需要有两类操作
 - 区间求和操作：计算第 i 个元素到第 j 个元素的和
 - 单点修改操作：更改指定元素 a_i 的值
- 解决方案
 - 1: 循环遍历求解区间的和
 - 平均时间复杂度为 $O(n)$
 - 区间的平均长度为 $n/2$,
 - m 次查询为 $O(nm)$
 - 单点修改时间复杂度为 $O(1)$

```
int sum(int a[], int l, int r) {  
    int s = 0;  
    for (int i = l; i < r; ++i) {  
        s += a[i];  
    }  
    return s;  
}
```



问题导入

• 解决方案

• 1: 循环遍历计算区间和

• 2: 前缀和

• 预计算前缀和数组S, 其中 $S_i = a_1 + \dots + a_i$

- 可由递归计算得到: $S_1 = a_1$, $S_{i+1} = S_i + a_{i+1}$, 时间复杂度为 $O(n)$

• $[i, j]$ 之间的区间和: $S_j - S_{i-1}$

- 时间复杂度为 $O(1)$

• 单点修改: 若修改 a_i , 需更新 S_i 之后的所有前缀和

- 时间复杂度为 $O(n)$

```
int prefixSum(int a[], int s[], int n) {  
    s[0] = a[0];  
    for (int i = 1; i < n; ++i) {  
        s[i] = s[i - 1] + a[i];  
    }  
}
```




问题导入

• 解决方案

- 1: 循环遍历计算区间和
- 2: 前缀和
- 3: 线段树
 - 区间求和: 时间复杂度 $O(\log n)$
 - 单点更新: 时间复杂度 $O(\log n)$
 - 适用于更新和求和操作都比较频繁的场景
- 4: 树状数组
 - 区间求和及单点更新的时间复杂度均为 $O(\log n)$, 但比线段树的常数更小



树状数组

- 结合前缀和及线段树的思想

- 前缀和中的 $s_i = a_1 + \dots + a_i$ ，因此，更新 a_k 时需要更新 s_k 以后的所有前缀和
 - 时间复杂度为 $O(n)$
- 能否只计算部分和，而不是都从 a_1 开始计算？
 - 这样更新 a_k 时就不需要更新 s_k 后面的所有前缀和
 - 部分前缀和 c_k 计算起点：将二进制表示的 k 中的最低位的1变为0后+1

$$i=12=(1100)_2 \quad \longrightarrow \quad (1000)_2=8 \quad \longrightarrow \quad c_{12}=a_9+a_{10}+a_{11}+a_{12}$$

$$i=4=(0100)_2 \quad \longrightarrow \quad (0000)_2=0 \quad \longrightarrow \quad c_4=a_1+a_2+a_3+a_4$$



树状数组

• **部分前缀和的计算** $i=12=(1100)_2 \longrightarrow (1\mathbf{0}00)_2=8 \longrightarrow c_{12}=a_9+a_{10}+a_{11}+a_{12}$

• 部分前缀和 c_k 可表示为 $c_k = \sum_{f(k)+1}^k a_k$

• 其中, $f(k)$: 将**二进制表示的k中的最低位的1变为0**

• **Lowbit(k)**: $\text{Lowbit}(12)=(0\mathbf{1}00)_2=4$

• 只保留二进制表示中最低位1, 其余位为0

• $f(k)$ 可表示为: $f(k) = k - \text{Lowbit}(k)$

```
int Lowbit(int x) {
    return x & (-x);
}
```

• **Lowbit的计算**: 可以通过 **k 与 -k 的按位与运算**得到

• -k在计算机中是**补码**表示: 按位取反+1

• k与-k只在最低位1的位置相同, 其它都不同

5	101	6	110
&-5	011	&-6	010
<hr/>		<hr/>	
	001		010



树状数组

部分前缀和可以通过递推进行计算

$$c_{12} = a_9 + a_{10} + a_{11} + a_{12}$$

$$c_{10} = a_9 + a_{10}$$

$$c_{11} = a_{11}$$



$$c_{12} = c_{10} + c_{11} + a_{12}$$



$$i=12=(1100)_2$$

$$i=10=(1010)_2$$

$$i=11=(1011)_2$$



$$\text{Lowbit}(12) = (0100)_2 = 4$$

$$k=12 - \text{Lowbit} = (1000)_2$$

$$k=10=(1010)_2 = k + \text{Lowbit}/2$$

$$k=11=(1011)_2 = k + \text{Lowbit}/4$$

// a: 原始数组, c: 前缀和, n: 数组元素个数

```
PrefixSum(int a[], int n, int c[]) {
```

```
    for (int i = 1; i <= n; ++i) { // 假设下标从1开始,
```

```
        c[i] = a[i]; // 最后一个元素
```

```
        int low = Lowbit(i);
```

```
        int k = i - low; // ci的起始位置
```

```
        while (low > 1) {
```

```
            low >>= 1;
```

```
            k += low;
```

```
            c[i] += c[k];
```

```
        } //end while
```

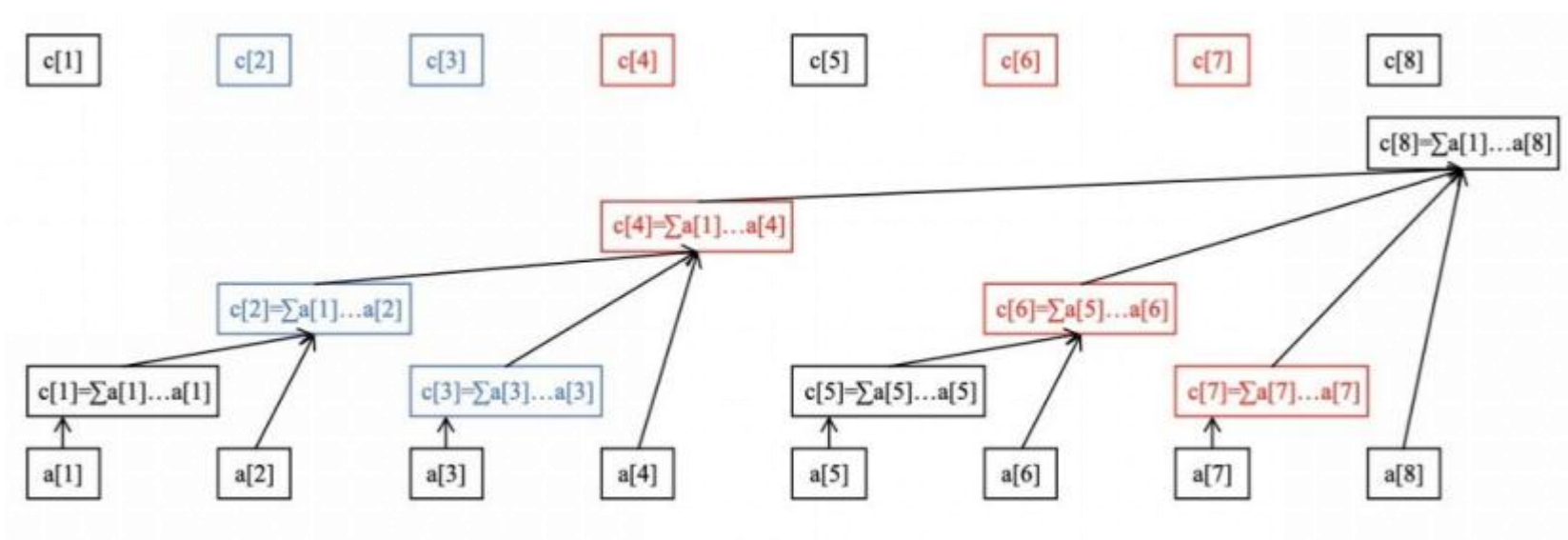
```
}}
```

只计算最近的LowBit个数字之和



树状数组

- 部分前缀和计算
 - 计算过程可以表示成一棵树





区间和计算

• 区间[l, r]上的和的计算

- $\text{sum} = s_r - s_{l-1}$

• 基于部分前缀和 c_k 计算前缀和 s_k

$$c_k = \sum_{f(k)+1}^k a_k$$

- $s_k = c_k + s_{f(k)} = c_k + c_{f(k)} + c_{f(f(k))} + \dots =$

$$s_3 = c_3 + s_{f(3)} = c_3 + c_2 + s_{f(2)} = c_3 + c_2$$

$$s_7 = c_7 + s_{f(7)} = c_7 + c_6 + s_{f(6)}$$

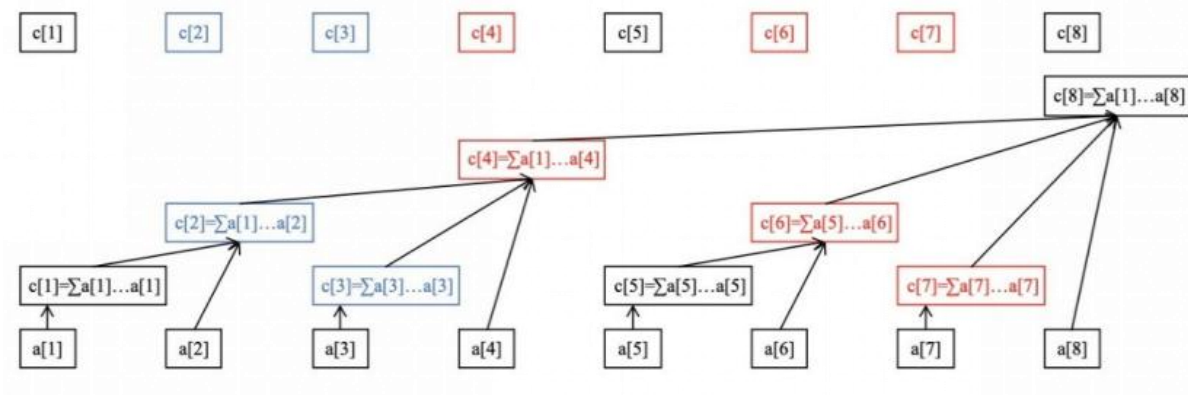
$$= c_7 + c_6 + c_4 + s_{f(4)} = c_7 + c_6 + c_4$$

```
// c: 部分前缀和, k: 第k项前缀和
int PrefixSum(int c[], int k) {
    int sum = 0;
    while (k > 0) {    // 假设下标从1开始,
        sum += c[k];
        k = k - Lowbit(k);    // k = f(k)
    } //end while
    return sum;
}
```



单点更新

- 单点更新 a_k
 - 更新所有包含 a_k 的部分前缀和 c_i
 - $g(k) = k + \text{Lowbit}(k)$
 - $k, g(k), g(g(k)), \dots$
 - 如：更新 a_3



$k=3=(0011)_2$
 $k=4=(0\mathbf{1}00)_2 = 3 + \text{Lowbit}(3)$
 $k=8=(\mathbf{1}000)_2 = 4 + \text{Lowbit}(4)$

```

// c: 部分前缀和, n: 数组长度, k: 第k项前缀和
Update(int c[], int n, int k, int d) {
    while (k <= n) { // 假设下标从1开始,
        c[k] += d;
        k = k + Lowbit(k); // k = g(k)
    } //end while
}

```



时间复杂度分析

- **区间求和**

- 对于任何正整数 $k \in [1, n]$, 最多经过 $\log_2 n$ 次 f 函数后变为 0
- 时间复杂度为 $O(\log n)$

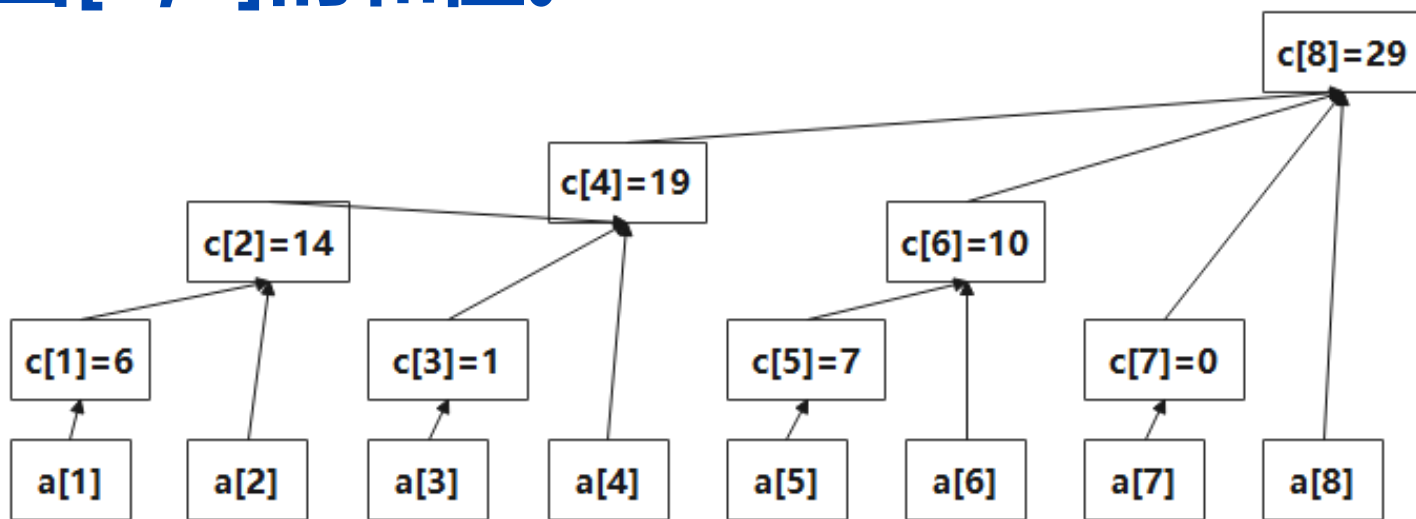
- **单点更新**

- 对于任何正整数 $k \in [1, n]$, 最多经过 $\log_2 n$ 次 g 函数后大于 n
- 时间复杂度为 $O(\log n)$



习题

- 已知数组 $A=[6, 8, 1, 4, 7, 3]$ ，请建立树状数组并根据树状数组求出 $[2, 5]$ 的和值。



$$S[2,5]=S[5]-S[1]$$

$$S[5]=C[5]+Sf[5]=C[5]+S[4]=C[5]+C[4]+Sf[4]=C[5]+C[4]=19+7=26$$

$$S[1]=C[1]+Sf[1]=C[1]=6$$

$$S[2,5]=26-6=20$$



小结

- **线段树的应用范围广，可支持动态修改和查询**
 - 区间求和，区间最大值/最小值
 - 区间修改、单点修改
- **线段树类似于分治的思想**
 - 必须满足原问题的解可由不相交的子问题的解合并得到
- **时间复杂度**
 - 查询和更新的时间复杂度均为 $O(\log n)$
 - 区间更新时，可以只做lazy标记不更新，而在做查询操作的时候，才在递归向下的时候进行真正到叶结点的值的更新

提 纲

12.3.1 跳表的定义

12.3.2 跳表的基本操作



问题导入

- **AVL树查询效率高： $O(\log n)$**
 - 树的维护需要左旋、右旋等复杂操作
- **有序顺序表查询效率高： $O(\log n)$**
 - 有序顺序表的维护代价高，需要将插入点后的数据后移
- **有序链表查询效率低： $O(n)$**
 - 插入、删除等操作代价低

有序链表达达到类似有序顺序表及AVL树的查询效率，
而保持较低的维护代价？

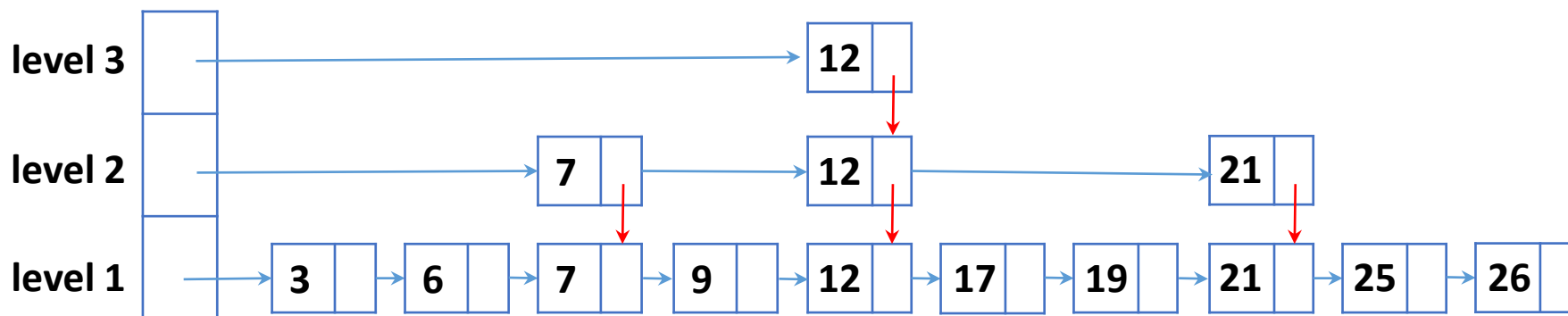


问题导入

• 多层链表：在原有链表的基础上增加多级索引

- 最底层为原始链表
- 上一层只保留下一层一半的数据：隔一个保留一个
- 最顶层的为链表的中间值
- 类似于一棵二叉查找树，查找的时间复杂度为 $O(\log n)$
- 空间复杂度： $n + n/2 + n/4 + \dots = 2n$

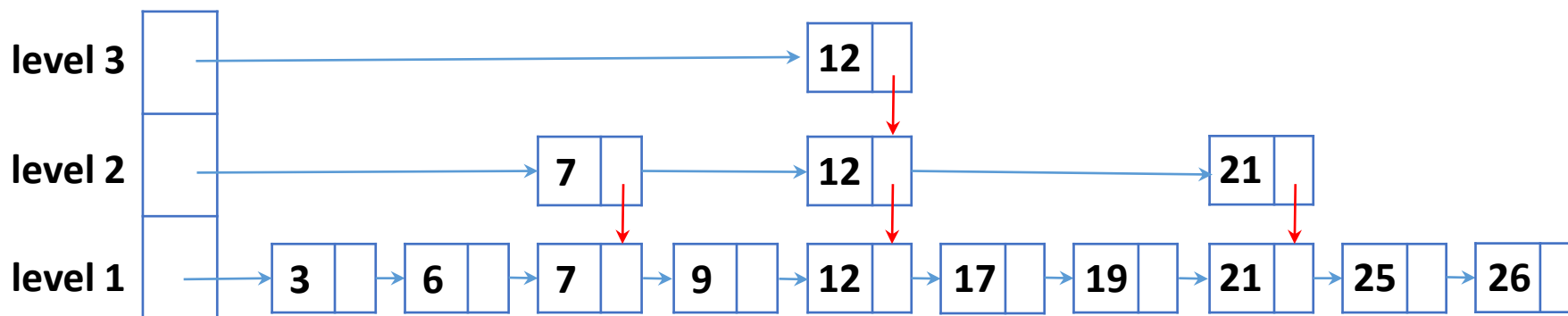
动态插入和删除难以维持平衡性：上一层链表的结点是下一层链表的中间值





12.3.1 跳表的定义

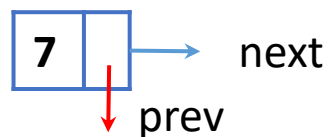
- **跳表由若干层有序链表组成**
 - **第一层为原始有序链表**
 - **之后每一层的链表只保留前一层链表的部分结点**
 - 第 i 层的结点有概率 P 被保留在第 $i+1$ 层，概率 p 一般为0.5
 - **若当前层只有一个结点，则不再分层，**
 - **以当前层作为跳表的最高层**



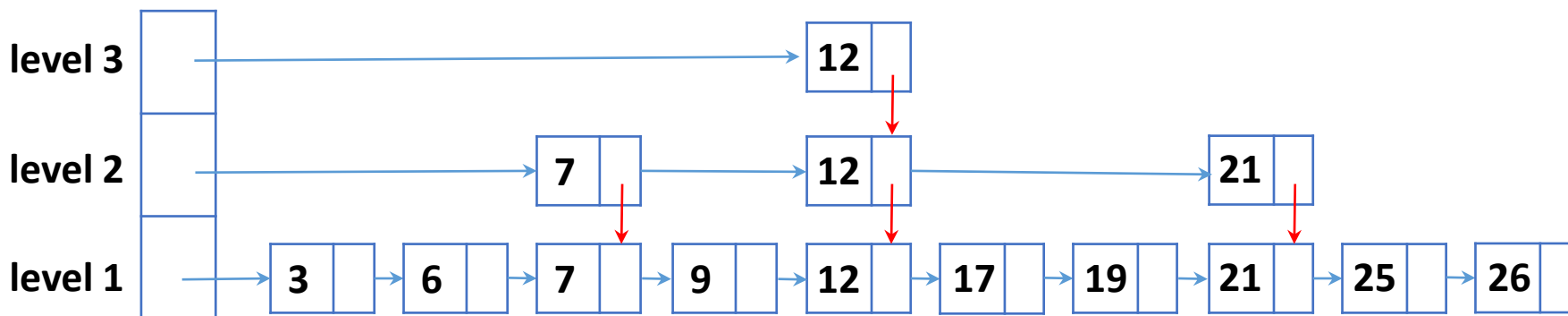


跳表的定义

- 跳表在有序链表的基础上引入“分层”的思想
 - 每个结点既有指向后一个结点的next指针，还有指向该结点对应前一层链表的结点指针



- 查找、插入和删除操作的时间复杂度能降至 $O(\log n)$





12.3.2 跳表的基本操作

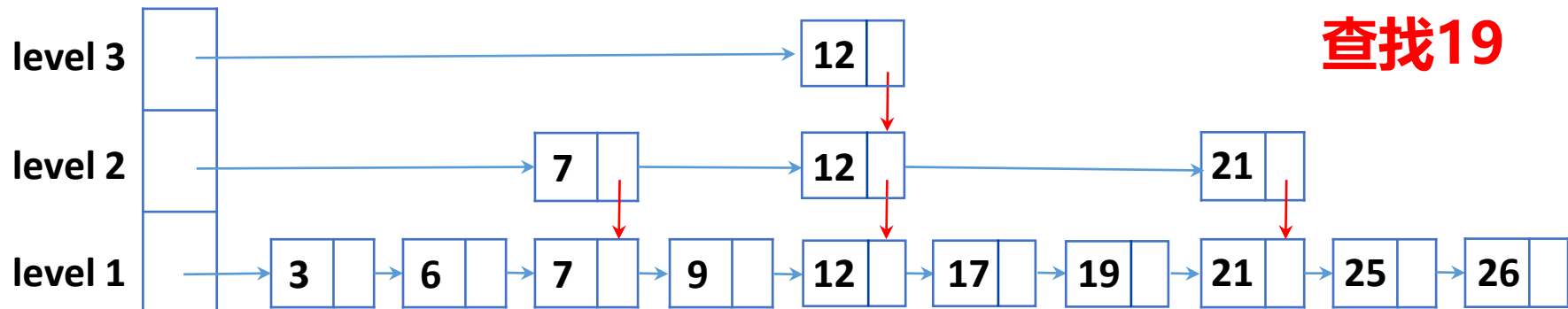
- **基本操作**
 - 查找
 - 插入
 - 删除



查找

• 查找关键字k

- 从最高层L开始从左往右找到当前层**最后一个值小于等于k的结点v**。
 - 如果结点v存在，则跳转至第L-1层与结点v对应的结点
 - 否则跳转至第L-1层的头结点。
- 在第L-1层，L-2层……继续重复上述操作，**直至到达第一层**。
 - 如果在第一层找到的结点v的值等于k，则返回查找结果，
 - 否则，值为k的元素不存在，查找失败。





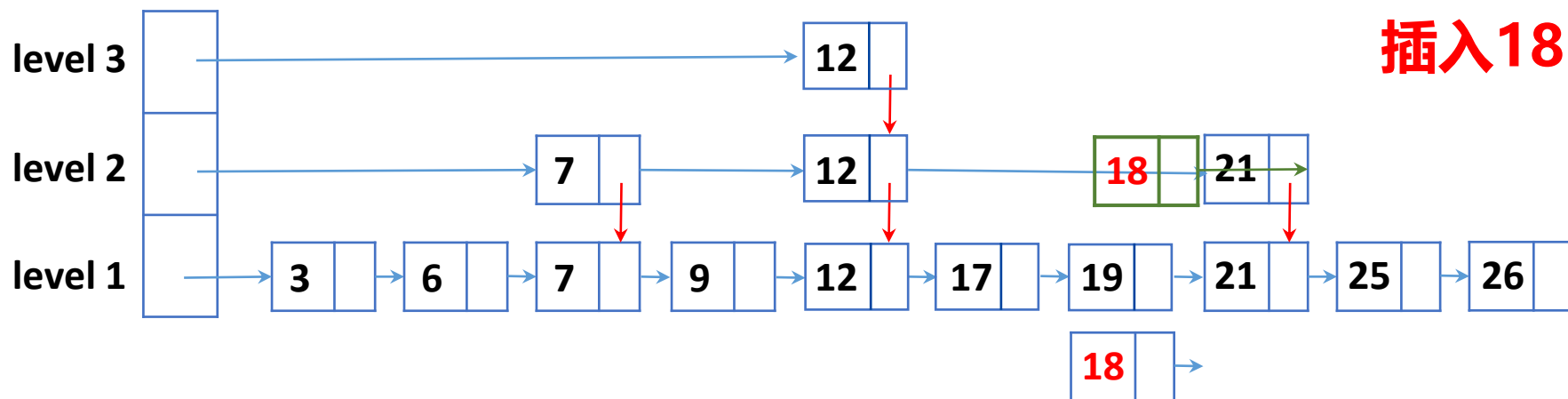
插入

- **插入过程实际上也是构建跳表的过程**
 - **先执行一遍查找操作，确定每一层有序链表中插入k的位置**
 - **在第一层，将值为k的结点插入当前层有序链表**
 - **以p的概率进行一次随机，决定是否要插入上一层**
 - **若不要插入上一层，则停止**
 - **否则，重复这一过程**
 - **在插入元素时，有可能使跳表的总层数增加**



插入

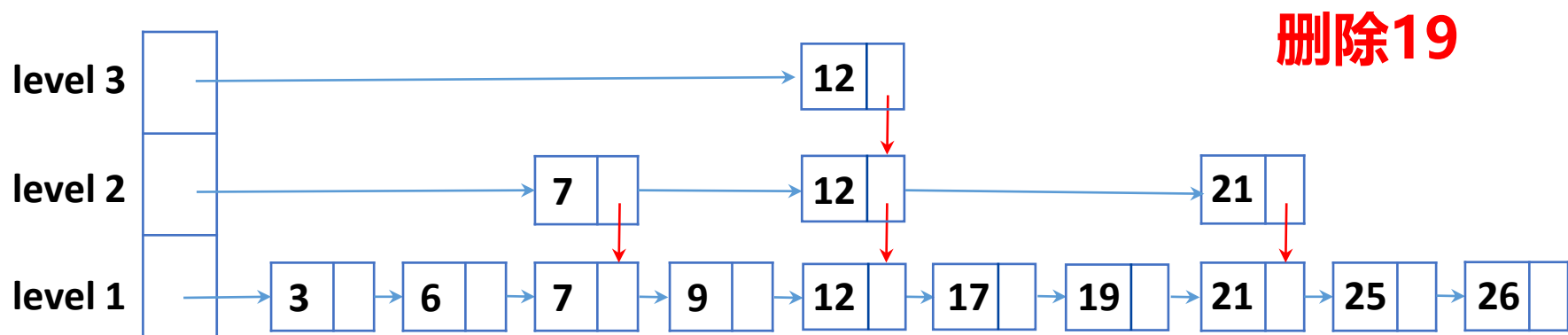
- 是否插入上一层的链表中的概率是 p
 - 插入 k 层的概率是 p^{k-1} ，越往上，插入的概率越低
 - 与AVL树不同，只是一定概率上保持平衡，不能保证一定平衡
 - 有可能在最高一层还需插入，使跳表的总层数增加
 - 可以限制最多只增加一层





删除

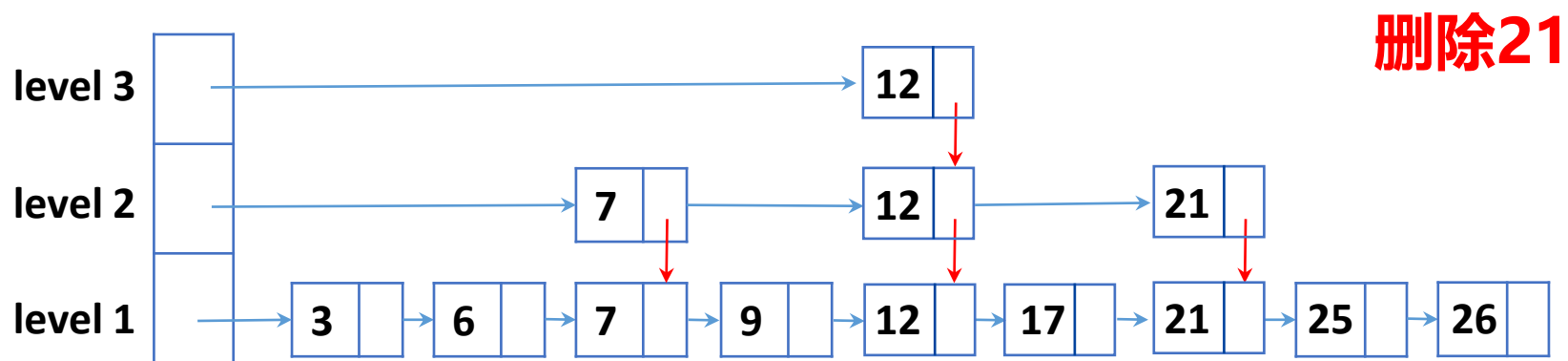
- **删除元素k**
 - 先执行查找操作
 - 若查找成功，将查找路径上等于k的结点删除





删除

- **删除元素k**
 - 先执行查找操作
 - 若查找成功，将查找路径上等于k的结点删除





跳表的复杂度分析

• 空间复杂度

- 原始链表上的每一个元素，插入时往上一层保留的概率为 p ，不保留概率为 $1-p$ 。其最高在第 i 层依然被保留的概率为 $(1-p)p^{i-1}$ ，则每个元素的最高保留层数的期望为：

$$\sum i(1-p)p^{i-1} = \frac{1}{1-p}$$

- 跳表上结点数的期望为 $\frac{n}{1-p}$
 - 保留到多少层，就有多少个结点备份，期望的最高层数为 $\frac{1}{1-p}$
 - 若 $p=0.5$ ，则跳表的结点数为 $2n$
- 跳表的空间复杂度为 $O(n)$



跳表的复杂度分析

• 时间复杂度

- 在跳表上查找元素时，如果在某一层内的查找路径包含 i 个结点，意味着其中 $i-1$ 个结点都没有在高一层被保留下来
 - 这种情况发生的概率不超过 $(1-p)^{i-1}$
 - 每一层内查找路径的长度的期望不超过： $\sum i(1-p)^{i-1} = \frac{1}{p^2}$
- 跳表层数的期望为： $\log \frac{n}{p}$
- 查找的路径长度的期望值为： $\log \frac{n}{p} / p^2$
- 查找的时间复杂度为 **$O(\log n)$**
- 插入和删除均依赖查找，时间复杂度也为 **$O(\log n)$**

提纲

12.4.1 红黑树的定义

12.4.2 红黑树的存储实现

12.4.3 红黑树的基本操作



问题引入

- **二叉查找树**

- 查找、插入、删除的平均时间复杂度均为 $O(\log n)$ ，效率较高
- 但存在退化为线性链表的不平衡情况
 - 时间复杂度为 $O(n)$

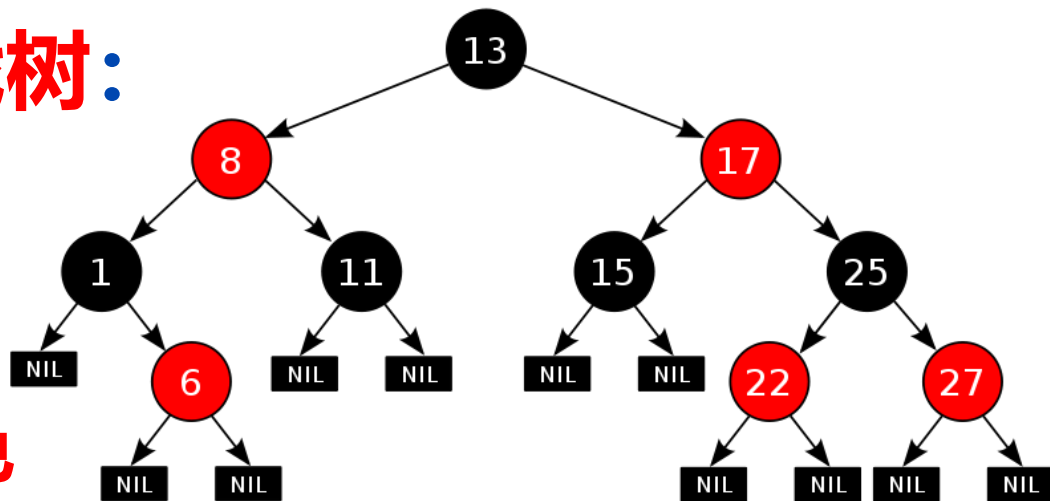
- **AVL树高度平衡，左右子树的高度差最多为1**

- 插入、删除时发生较多左旋、右旋等复杂操作，维护代价高
- **能否放宽左右子树的高度差约束，减少AVL树的维护代价？**
 - 红黑树：弱平衡二叉树



12.4.1 红黑树的定义

- **红黑树是满足如下性质的二叉查找树：**
 - 每个结点或者为黑色，或者为**红色**
 - 根结点为黑色
 - **每个空结点（NULL结点）都是黑色**
 - 如果有一个结点是红色，那么他的2个孩子结点都是黑色（**不能有2个相邻的红色结点**）
 - 对于每一个结点，从该结点到其子孙的叶子结点的路径中所包含的黑色结点数量必须相等





红黑树的性质

- **黑高度**

- 从红黑树任意结点 x 出发，到达叶子结点的任意路径上（不包含 x ）的黑色结点个数，记为 $bh(x)$
 - 空结点的黑高度为0，叶子结点的黑高度是1
 - 根结点的黑高度为红黑树的黑高度。

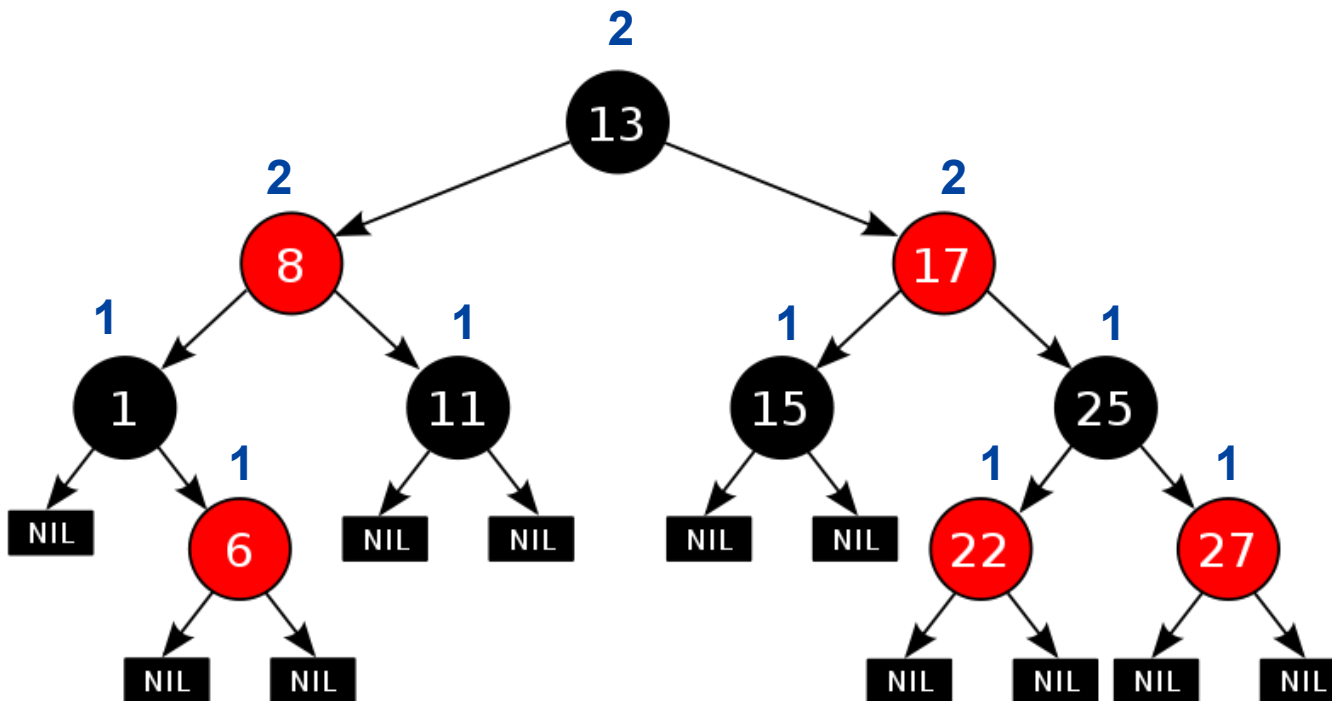
- **黑高度相等性质**

- 每个结点 x 到其所有子孙叶结点的路径中所包含的黑结点个数（**不包含结点 x** ）必须相等

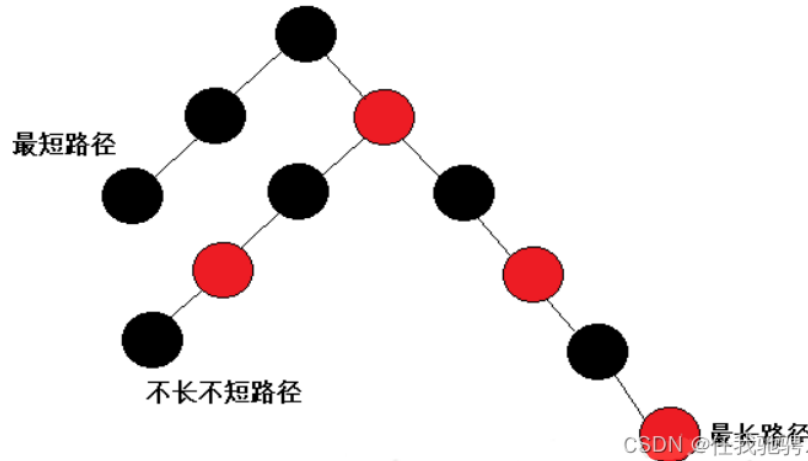


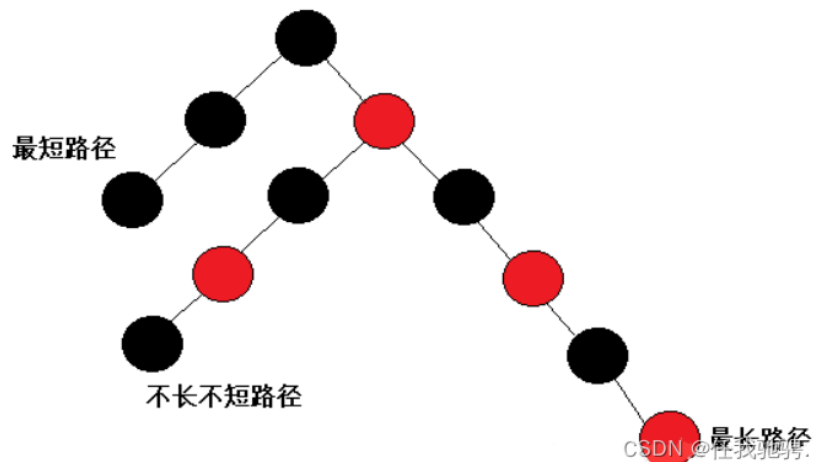
练习

- 写出红黑树的每个结点的黑高度



红黑树的性质

- **着色性质**
 - 根结点为黑色，每个结点着色为红黑两色中的一种颜色
 - 任一红结点的孩子只能为黑色（红色结点不能相邻）
 - **最长路径中结点个数不会超过最短路径结点个数的2倍**
 - 最短路径为全黑
 - 最长路径为红黑结点交替
 - 红色结点不能连续
 - 最长路径是最短路径的2倍
 - 每条路径的黑色结点相同
- 
- 最短路径
- 最长路径
- 不长不短路径
- CSDN @ 在我驰骋
- 红黑树对平衡性的要求比AVL弱



红黑树对平衡性的要求比AVL弱



红黑树的性质

- 任意一棵有 n 个结点的红黑树，其高度至多为： $2\log(n+1)$

- 查询的时间复杂度与AVL树相当

- 证明

- 若能证明：结点 x 的黑高度为 $bh(x)$ ，则以 x 为根的红黑树的结点数：

$$n \geq 2^{bh} - 1$$

- 设红黑树的高度为 h ，其黑高度至少为 $h/2$

- 红色结点不能连续，最少情况为红黑结点交替排列情况

- 因此，高度 h 的红黑树，其结点数 $n \geq 2^{h/2} - 1$ ，整理可得：

$$h \leq 2\log(n + 1)$$



红黑树的性质

- 结点x的黑高度为 $bh(x)$ ，则以x为根的红黑树的结点数 $n \geq 2^{bh(x)} - 1$
 - 采用数学归纳法进行证明：
 - 若黑高度 $bh(x)=1$ ，则只包含x结点，且x为黑色的红黑树结点数最少，n为1，成立
 - 假设黑高度 $bh(x)=j$ 时，结论成立，即以x为根的子树中，结点数 $n \geq 2^j - 1$
 - 当黑高度 $bh(x) = j+1$ 时，以x为根的子树中至少包含的结点数n：
 $n = \text{左子树至少包含的结点数} + \text{右子树至少包含的结点数} + 1$
 - 若左右孩子结点均为黑色，则结点数最少，此时左右子结点的黑高度均为 $bh(x)-1=j$
 - 由归纳假设知：左右子树的结点数 $n \geq 2^j - 1$
 - 若左右孩子结点均为红色，则结点数最多；
 - 所以，当黑高度为 $j+1$ 时，结点数 $n \geq (2^j - 1) + (2^j - 1) + 1 = 2^{j+1} - 1 = 2^{bh(x)} - 1$ ，结论成立



红黑树的存储实现

```
enum Color{RED, BLACK};           // 节点的颜色
template<typename Elem>
class RBNode {                     // 红黑树节点的定义
    RBNode(const Elem& data = Elem(), Color color = RED)
        : _left(nullptr), _right(nullptr), _parent(nullptr)
        , _data(data), _color(color) {}
    RBNode<Elem>* _left, * _right;  // 节点的左孩子、右孩子
    RBNode<Elem>* _parent;          // 节点的双亲(红黑树需要旋转, 为了实现简单给出该字段)
    Elem _data;                     // 节点的值域
    Color _color;                   // 节点的颜色
    void setLeft(RBNode<Elem>* lc) {
        _left = lc; lc->setParent(this);
    }
    void setRight(RBNode<Elem>* rc) {
        _right = rc; rc->setParent(this);
    }
    void setParent(RBNode<Elem>* parent) {
        _parent = parent;
    }
};
```




红黑树的存储实现

```
template<typename Elem>
class RBTree {                                // 红黑树的定义
    RBNODE<Elem> *_root;
public:
    RBTree() { _root = nullptr; }
    ~RBTree();
    RBNODE<Elem> *root() {
        return _root;
    }
    void setRoot(RBNODE<Elem> *root) {
        _root = root;
        if (root)
            root->setParent(nullptr);
    }
    bool find(const Elem &key);
    bool insert(const Elem &value);
    void delete(const Elem &value);
};
```



红黑树的基本操作

- **查找**

- **红黑树的查找与二叉查找树的查找完全一致**
 - 若查找的值 $<$ 根结点的值，则递归的查找左子树
 - 若左子树为空，则查找失败
 - 若查找的值 $>$ 根结点的值，则递归的查找右子树
 - 若右子树为空，则查找失败
 - 若查找的值 $==$ 根结点的值，则查找成功

- **插入**

- **删除**



回顾：左旋、右旋

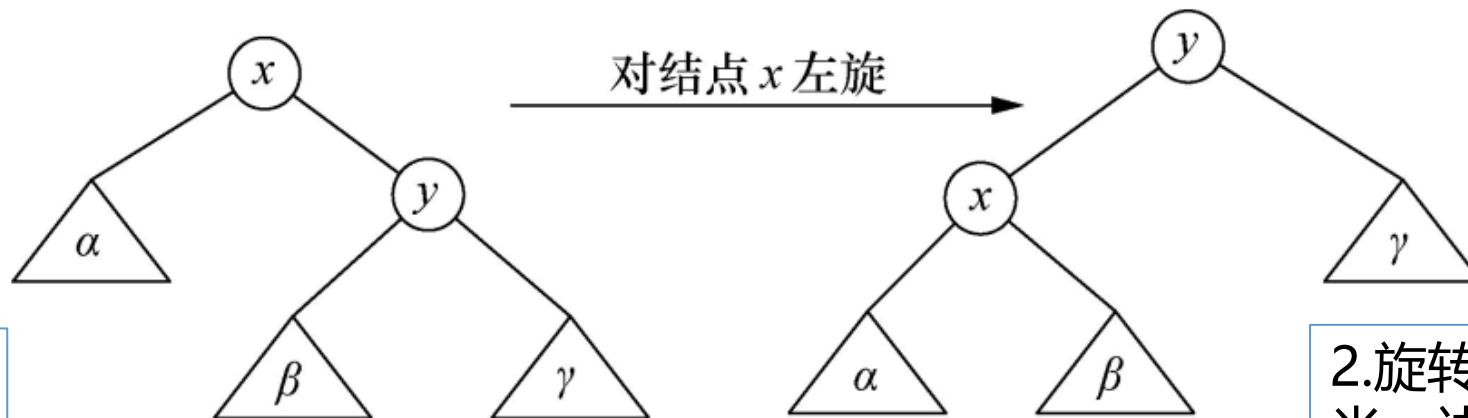


图1 红黑树左旋

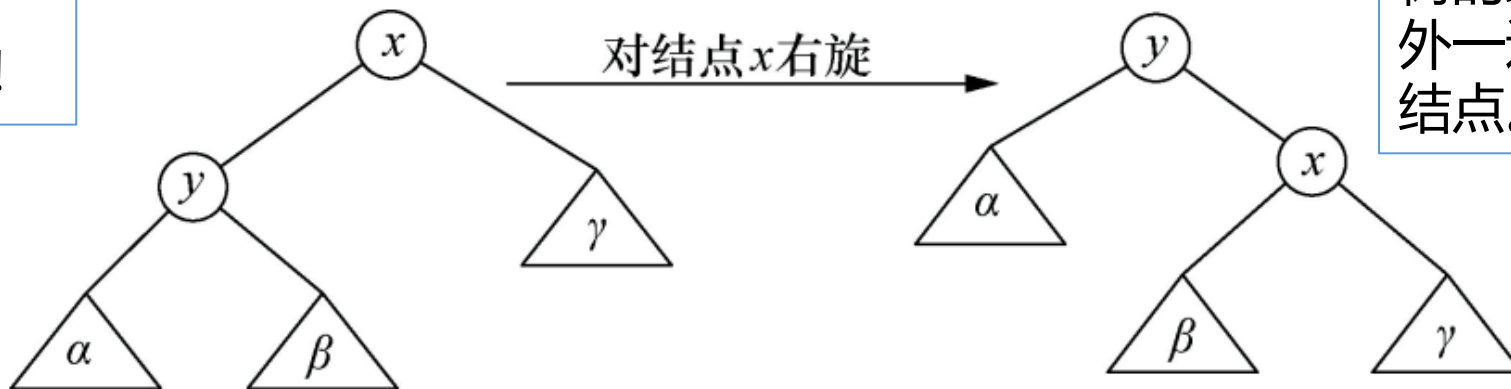


图2 红黑树右旋

1. 旋转不会影响**旋转结点的父结点**，父结点以上结点结构保持不变
2. 旋转不考虑颜色！

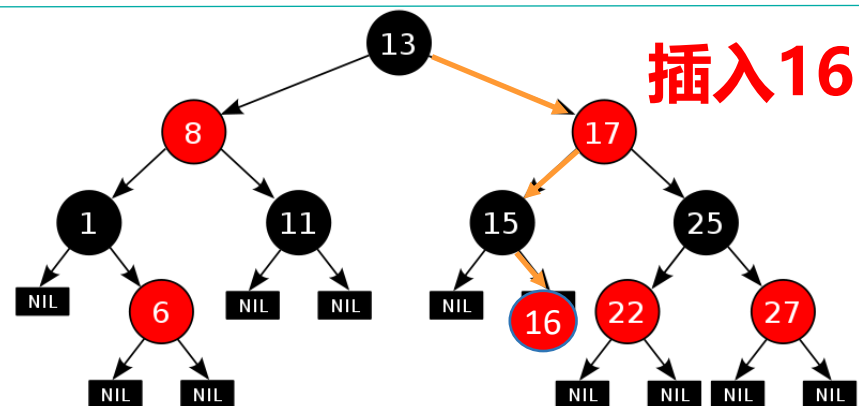
2. 旋转是局部的，目的是当一边子树的结点少了，那么向另外一边子树“借入”一些结点；当一边子树的结点多了，那么向另外一边子树“出租”一些结点。



红黑树的插入

思想

- 按照二叉检索树的规则，找到插入的位置
- 在插入位置插入一个**红色的结点**
 - 插入黑色结点**肯定会**影响到黑高相等性质，每次都需要调整
 - 插入红色结点**可能会**影响着色性质，不一定每次都需调整
 - 红色结点不能相邻
- 检查是否符合红黑树的规则，若违反规则，则进行调整
 - 只会影响着色性质，因此只需检查**父结点是否为红色结点**
 - 若父结点为黑色结点，不需调整
 - 若父结点为红色结点，**需要调整**



红黑树的插入

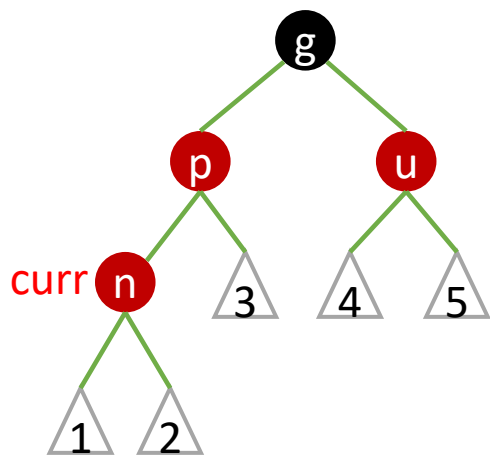
- 若父亲结点为红色，则出现连续的红色结点，违反了着色性质

- 约定：n为当前结点，p为父结点，g为祖父结点，u为叔叔结点
- 情况1：叔叔结点存在且为红色（n为红，p为红，g为黑，u为红）

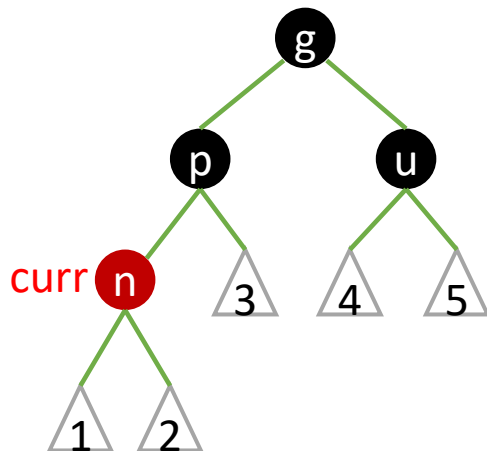
- 将父亲结点和叔叔结点的颜色改为黑色，祖父结点改为红色

- 红色改为黑色，不会违反不能出现连续红色的规则
- 父亲结点和叔叔结点同时修改，不会影响左右子树的黑高

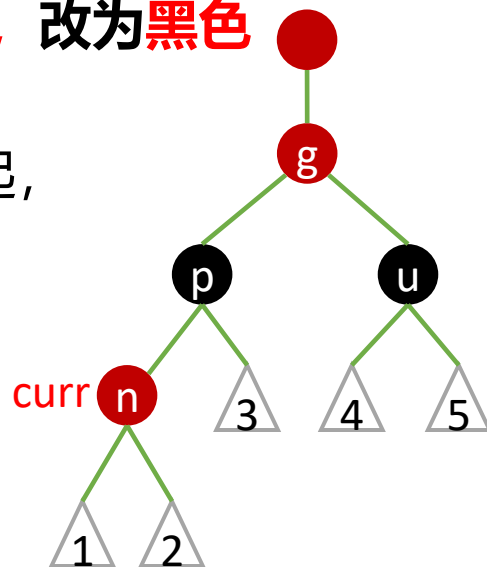
- 如果g结点的父亲结点为红色，则继续向上调整；如果g结点为根结点，改为黑色



变色
→

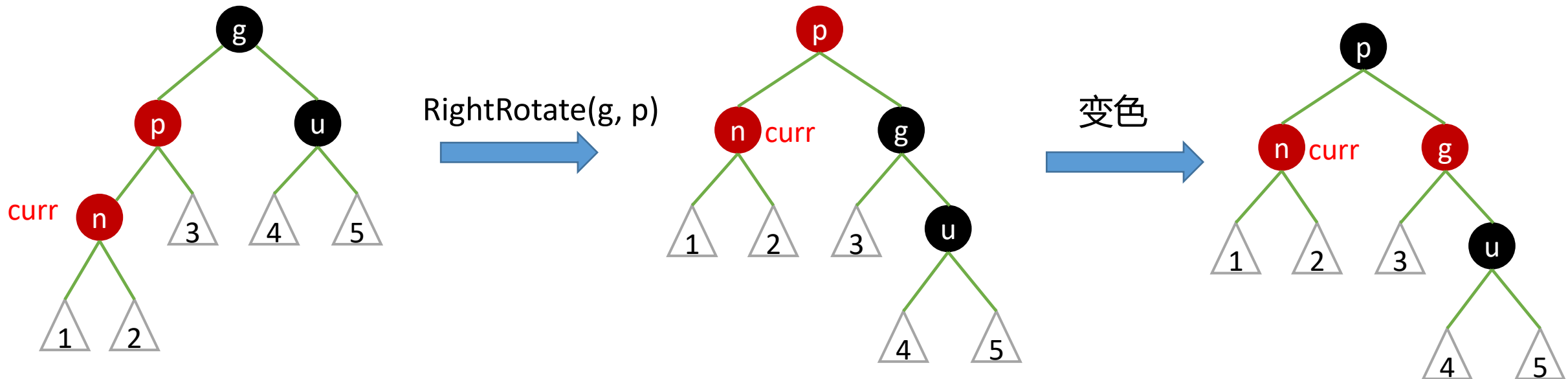


两个红色结点连在一起，
需要继续向上调整
→



红黑树的插入

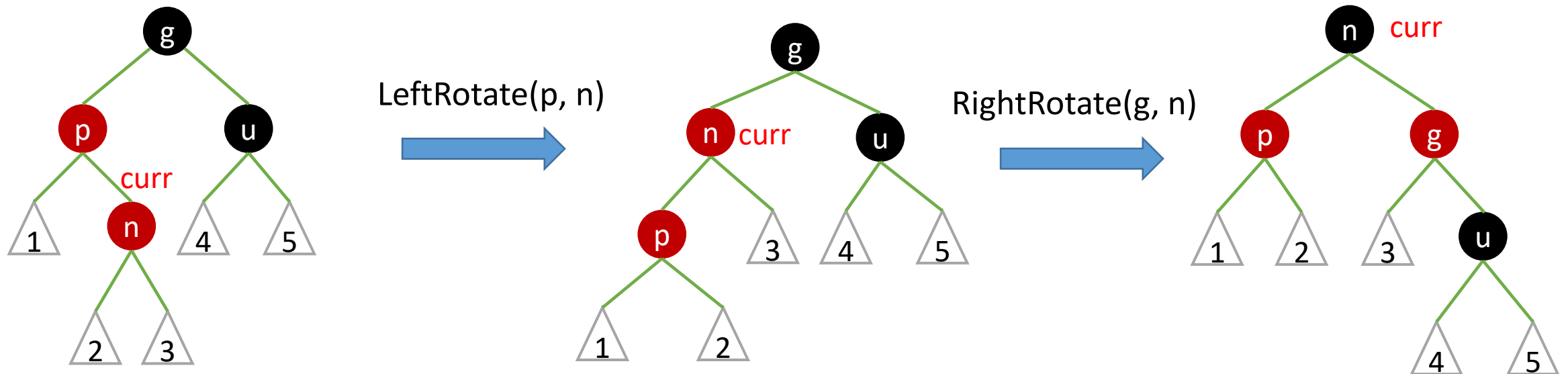
- 情况2-1：叔叔结点不存在或为黑色，p为g的左孩子
 - n为红，p为红，g为黑，u不存在或为黑色
 - 情况2-1.1：n为p的左孩子（LL型）
 - 对结点g绕结点p进行右旋，并将p的颜色改为黑色，g的颜色改为红色



连续红色消失了，黑高不变

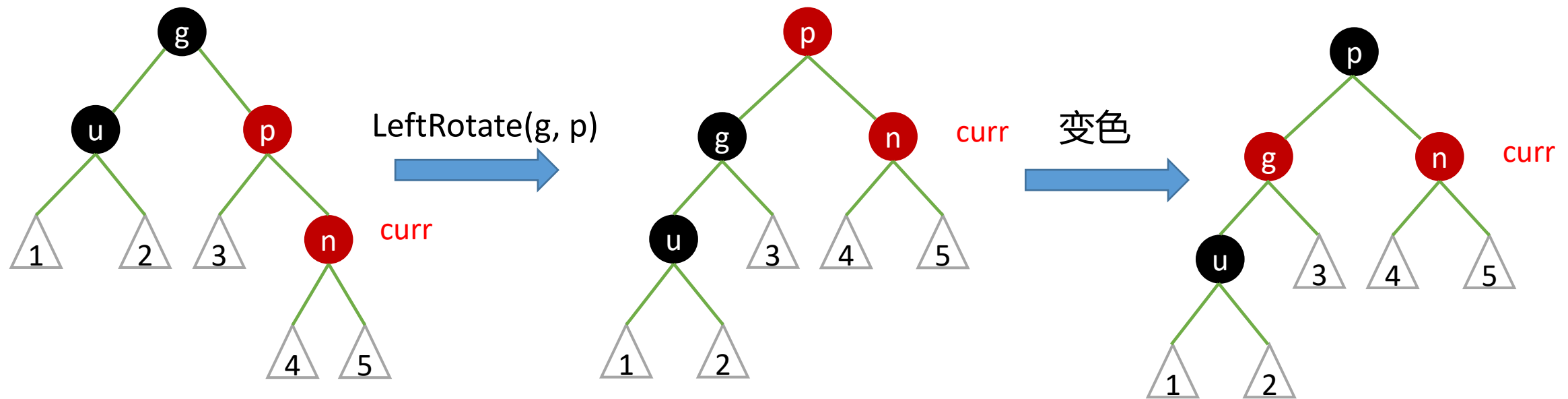
红黑树的插入

- 情况2-1：叔叔结点不存在或为黑色，**p**为**g**的左孩子
 - **n**为红，**p**为红，**g**为黑，**u**不存在或为黑色
 - 情况2-1.2：**n**为**p**的右孩子（LR型）
 - 先对结点**p**绕结点**n**进行左旋，则转换为情况2-1.1
 - 利用情况2-1.2的规则继续调整：对结点**g**绕结点**n**右旋，并交换**n**与**g**的颜色



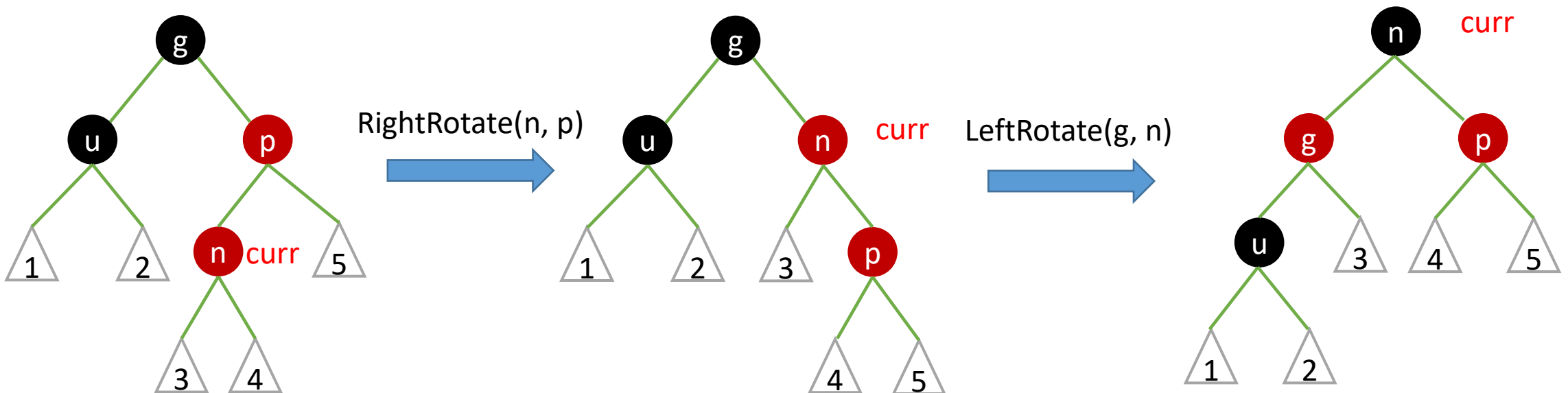
红黑树的插入

- 情况2-2：叔叔结点不存在或为黑色，**p**为**g**的右孩子
 - **n**为红，**p**为红，**g**为黑，**u**不存在或为黑色
 - 情况2-2.1：**n**为**p**的右孩子（RR型）
 - 对结点**g**绕结点**p**左旋，并将**p**的颜色改为黑色，**g**的颜色改为红色

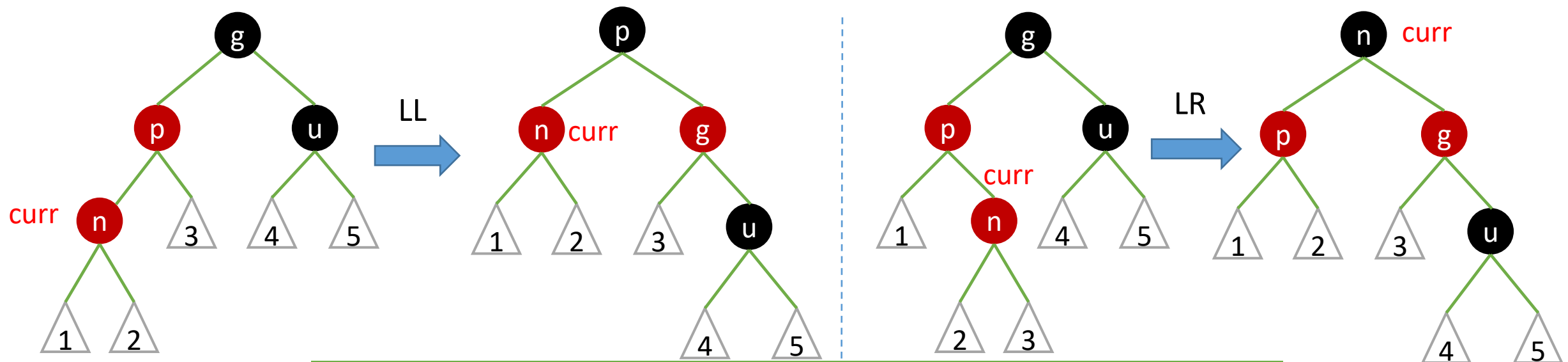


红黑树的插入

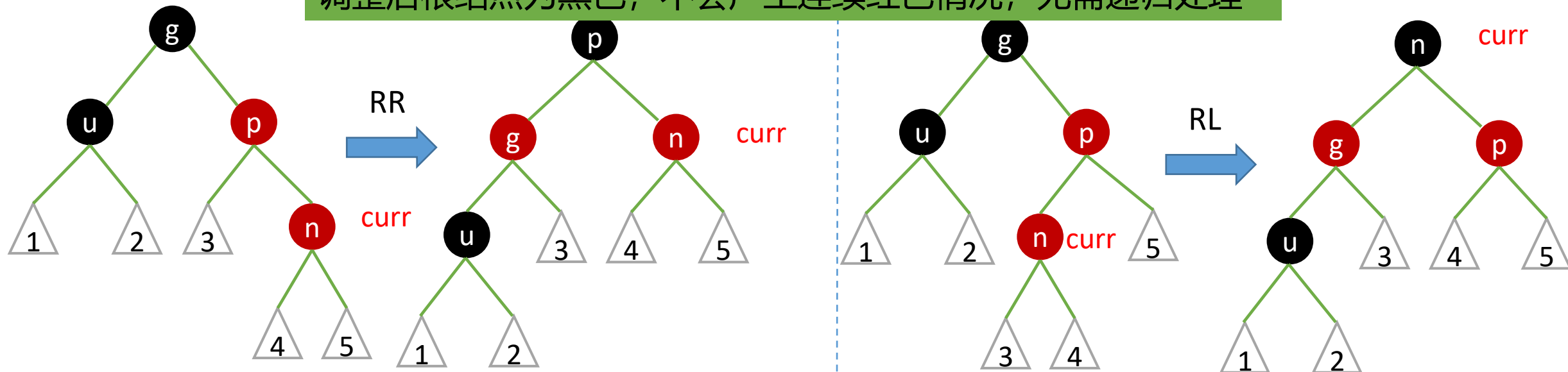
- 情况2-2：叔叔结点不存在或为黑色，p为g的右孩子
 - n为红，p为红，g为黑，u不存在或为黑色
 - 情况2-2.2：n为p的左孩子（RL型）
 - 对结点n绕结点P右旋，则转换为情况3-1
 - 利用情况3-1的规则继续调整：对结点g绕结点n左旋，并交换curr与g的颜色



红黑树的插入



调整后根结点为黑色，不会产生连续红色情况，无需递归处理



红黑树的插入

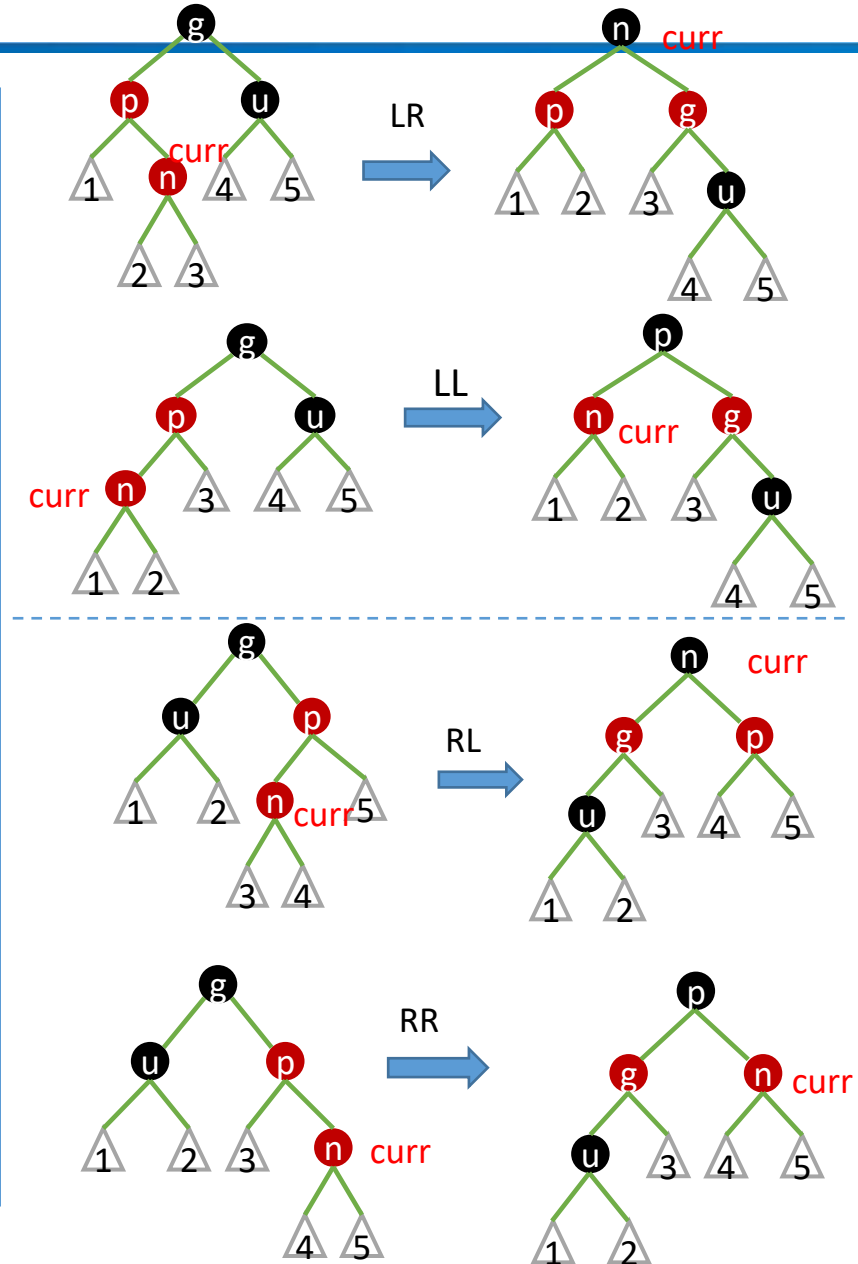
```
bool RBTREE::Insert(const Elem& value) {  
    // 1. 按照二叉搜索树的规则插入新节点  
    if (nullptr == _root) {           // 空树  
        _root = new RBTREENode(value, BLACK);  
    } else { // 非空, 按照二叉搜索树的特性查找待插入节点在树中的位置  
        RBTREENode* cur = _root, *parent = nullptr;  
        while (cur) {  
            parent = cur;  
            if (value < cur->_value) cur = cur->left();  
            else if (value > cur->_value) cur = cur->right();  
            else return false;           // 存在相同的key  
        }  
        cur = new RBTREENode(value, RED); // 插入新节点: 默认为红色  
        if (value < parent->_value) parent->setLeft(cur);  
        else parent->setRight(cur);  
        cur->setParent(parent);  
        adjustColor(cur);                // 调整, 从而满足着色性质  
    }  
}
```

红黑树的插入

```
void RBTree::adjustColor(RBTreeNode* cur) {  
    // 2. 检测新节点插入之后是否违反性质三: 即是否存在红色节点连在一起的情况  
    //     因为新插入节点cur的颜色是红色的, 如果cur双亲parent节点的颜色也是红色的  
    //     则违反了性质三  
    RBTreeNode* p = cur->parent();  
    if (p == nullptr) { // root should be black  
        cur->_color = BLACK;    return;    }  
    // no 2 continuous red: already is valid  
    if (p->_color != RED)    return;  
    RBTreeNode* g = p->parent(); // 此处grandFather一定不为空  
    // 因为: parent是红色的, 则parent一定不是根节点, parent的双亲一定是存在的  
    RBTreeNode* uncle = p == g->left() ? g->right():g->left();  
    if (uncle && RED == uncle->_color) { // 情况1: 叔叔节点存在且为红  
        p->_color = BLACK;  
        uncle->_color = BLACK;  
        g->_color = RED;  
        adjustColor(g); // adjust color in recursive  
    }  
}
```

红黑树的插入

```
else if (p == g->left()) { // 情况2-1
    if (cur == p->right()) { // 情况2-1.2: LR
        LeftRotate(cur, p);
        swap(p, n); // swap n and p
    }
    // 情况2-1.1: LR
    p->_color = BLACK;    g->_color = RED;
    RightRotate(g, cur);
} else if (p == g->right()) { // 情况2-1
    if (cur == p->left()) { // 情况2-2.2: RL
        RightRotate(cur, p);
        swap(p, n); // swap n and p
    }
    // 情况2-2.1: RL
    p->_color = BLACK;    g->_color = RED;
    LeftRotate(g, cur);
}}
```

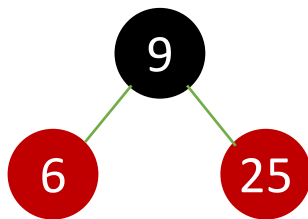


例子

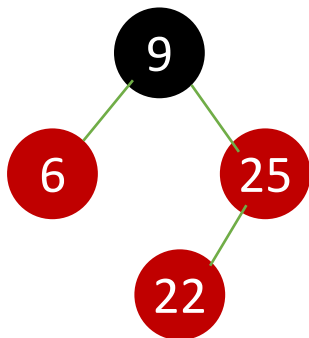
- 输入9, 6, 25, 22, 18, 20, 19, 15, 构建红黑树



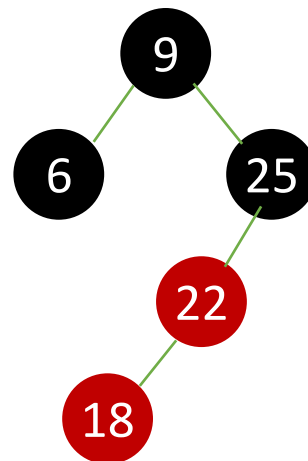
空树



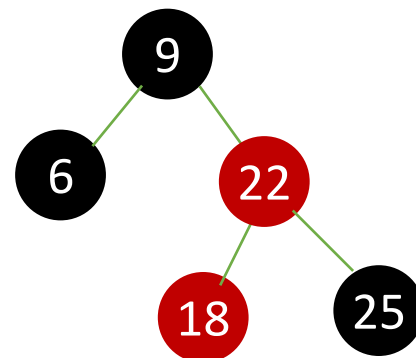
无需调整



情况1, 直接变色,
根结点变黑

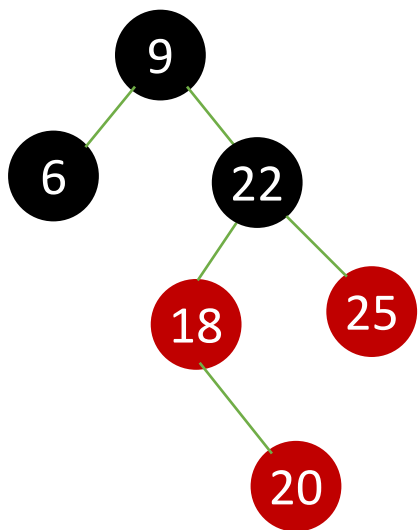


情况2-1, 先右旋, 再变色

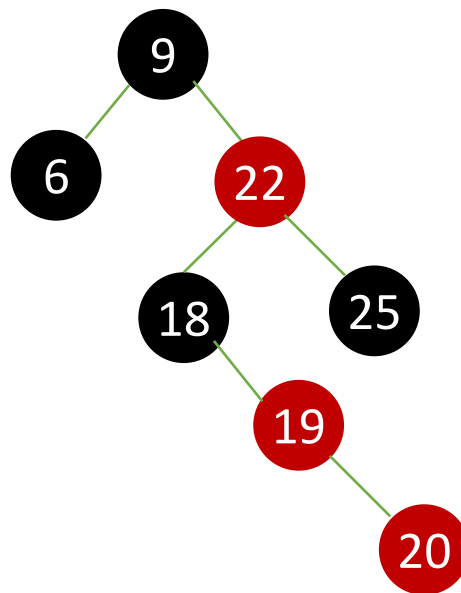
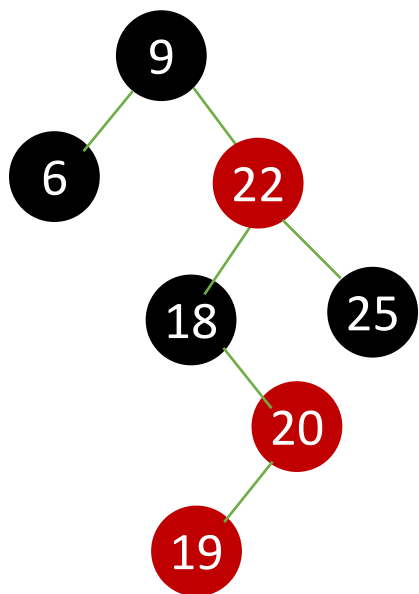


例子

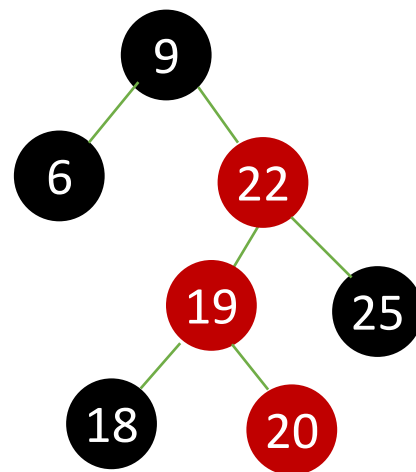
- 输入9, 6, 25, 22, 18, 20, 19, 15, 构建红黑树



情况1, 直接变色

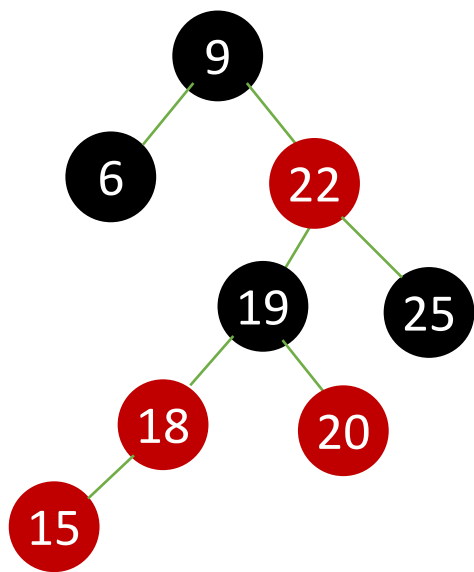


情况3-2, 先右旋变成情况3-1,
再左旋, 最后变色

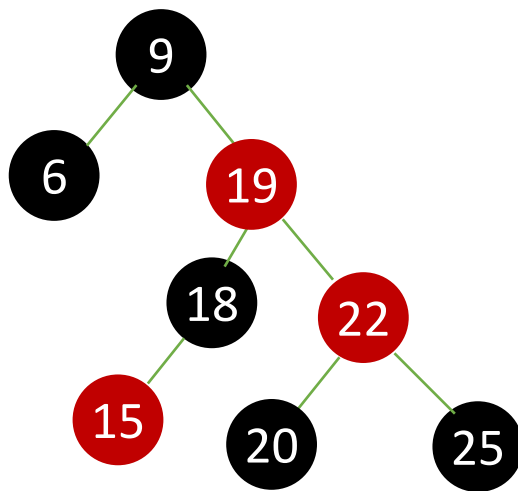


例子

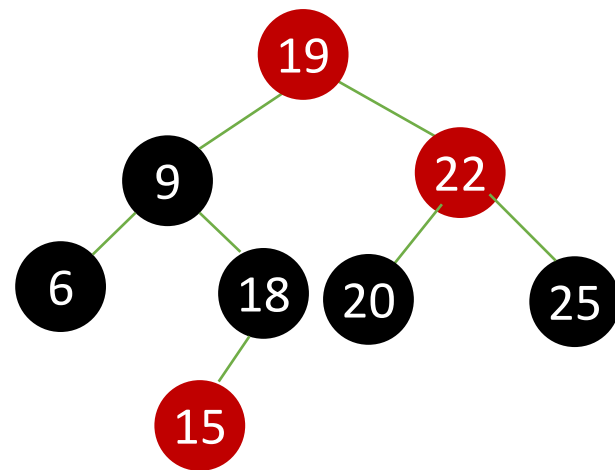
- 输入9, 6, 25, 22, 18, 20, 19, 15, 构建红黑树



情况1, 直接变色,
继续调整

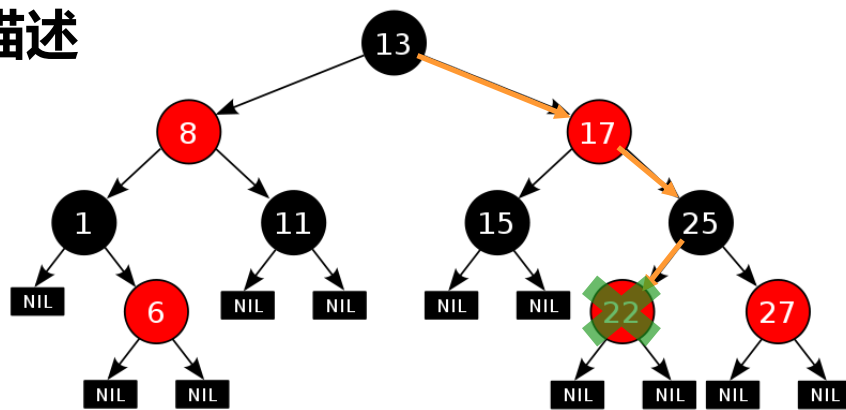


情况3-2, 先右旋变成情况3-1,
再左旋, 最后变色



红黑树的删除

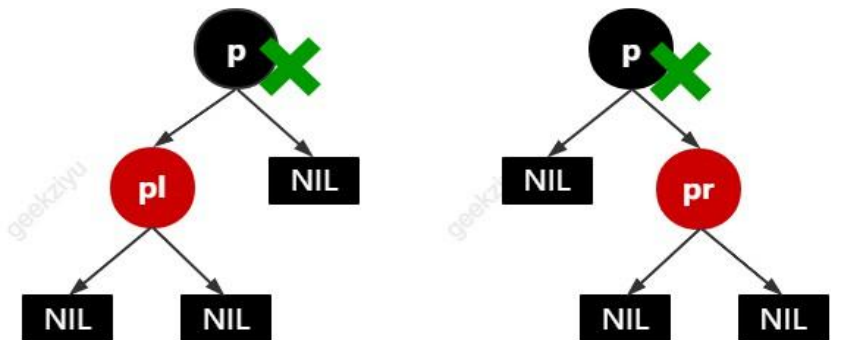
- 利用二叉查找树的规则找到待删除的结点
 - 查找失败，直接返回
- 1. 删除叶子结点
 - 1.1 叶子结点为红色，直接删除
 - 删除红色结点不会影响红黑树的平衡性
 - 1.2 叶子结点为黑色，删除该结点后，树黑高发生变化，需要进行调整
 - 较复杂，后面详细描述



删除22

红黑树的删除

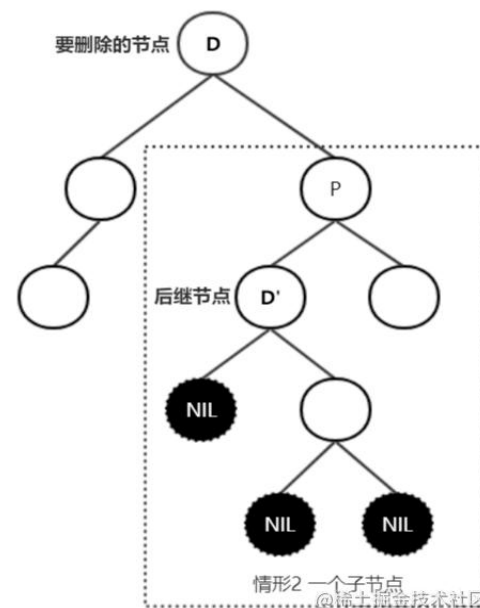
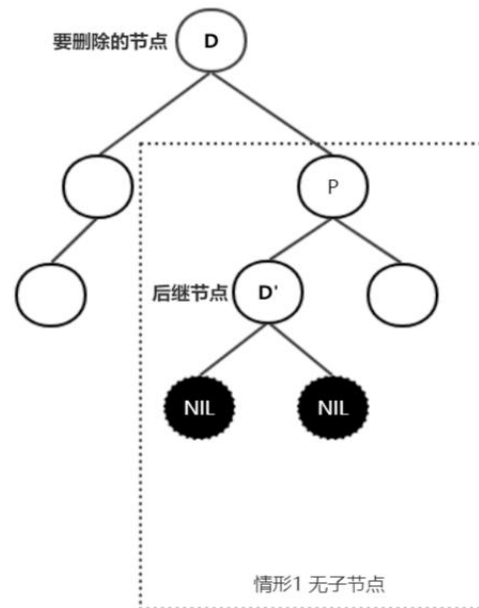
- 2. 删除的结点只有一个非空孩子结点
 - 根据黑高相等性质：每个结点到叶子结点的路径上包含相同的黑色结点
 - 孩子结点为红色结点
 - 根据着色性质：不存在连续的红色结点
 - 待删除的结点为黑色结点
 - 将孩子结点替换待删除的结点，并将颜色改为黑色
 - 删除黑色结点后，黑高-1，要保持黑高平衡，需要将颜色改为黑色



情景2.1 删除有一个红色左子结点的黑色结点 情景2.2 删除有一个红色右子结点的黑色结点

红黑树的删除

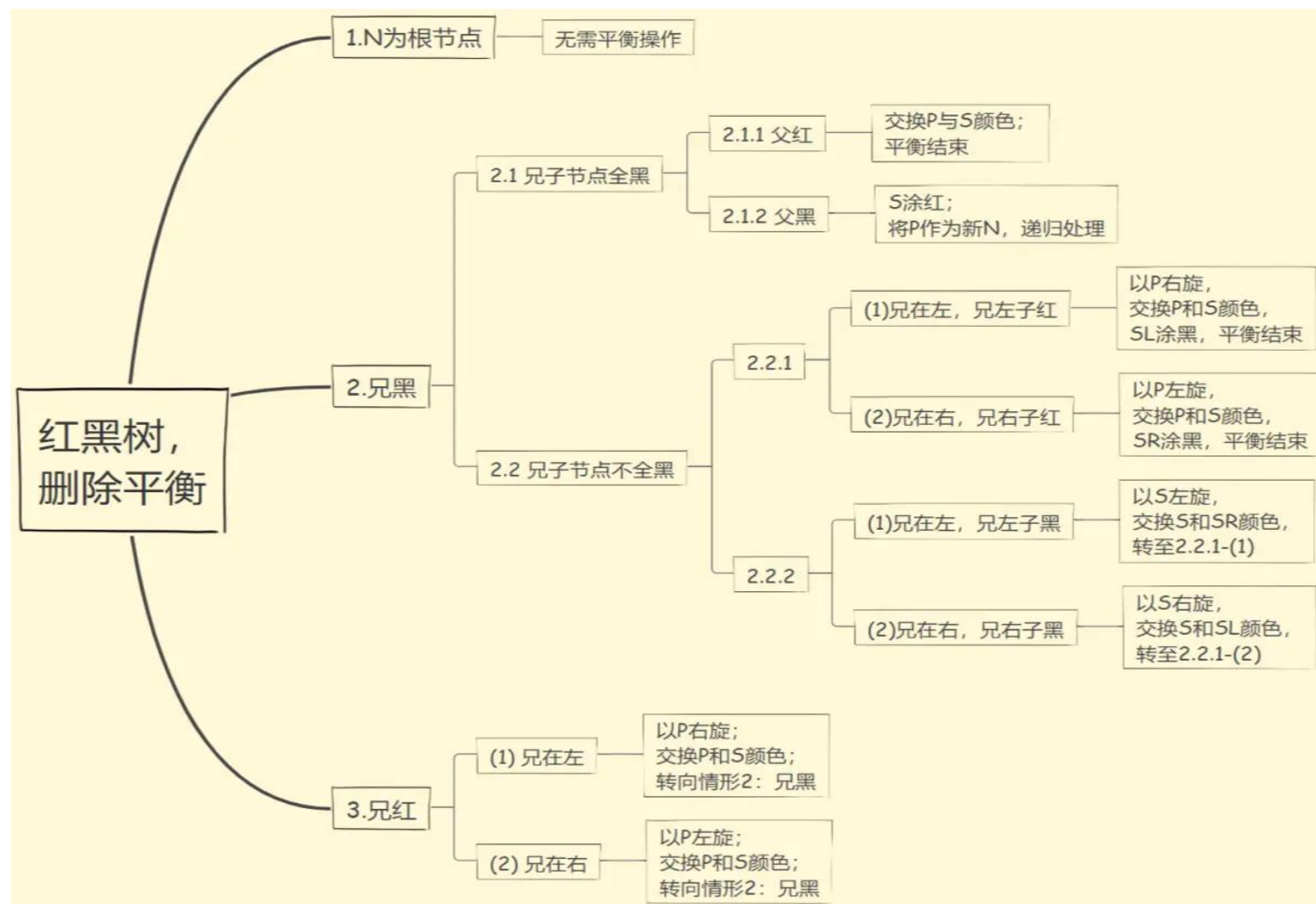
- 3. 删除的结点有两个非空孩子结点
 - 选择右子树的最小值结点，替换待删除结点，颜色设置为待删除结点的颜色
 - 删除最小值结点，此时转换为情况1和情况2的两种情况
 - 为什么会是这两种情况？



因为最小值结点不可能有两个子结点

情况1.2中黑色叶子结点的删除

- 黑色结点删除后，所在子树的黑高-1，左右子树的黑高不平衡
- 根据兄弟结点及兄弟结点的子结点进行区分



黑色叶子结点的删除

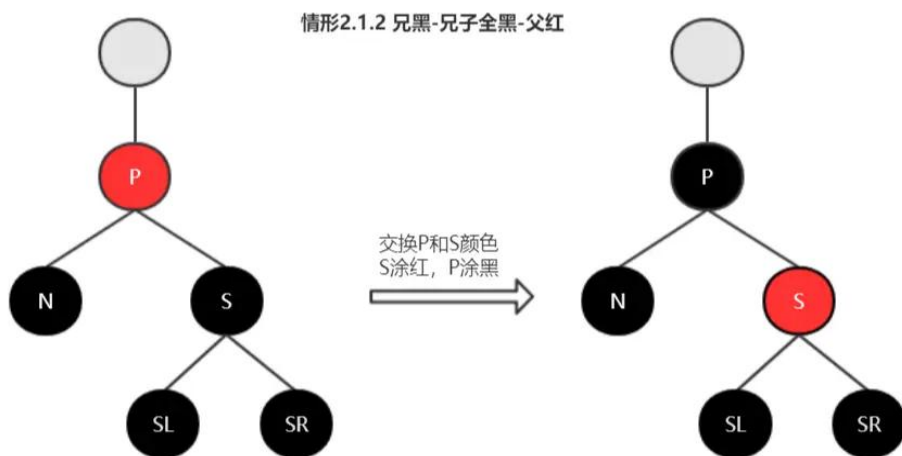
- 1. 待删除结点为根结点，删除该结点，将根结点置空，无需额外操作
- 2. 兄弟结点为黑色

- 2.1 兄弟结点的子结点全黑

N: 删除后的空结点, S: 兄弟结点, p: 父节点
SL: 兄弟结点的左孩子, SR: 兄弟结点的右孩子

- 2.1.1 父亲结点为红色: **交换P和S的颜色**

- 结点S的颜色改为红色，所以结点P的左右子树黑高相等
- 结点P的颜色改为黑色，以结点P为根结点的子树黑高保持不变



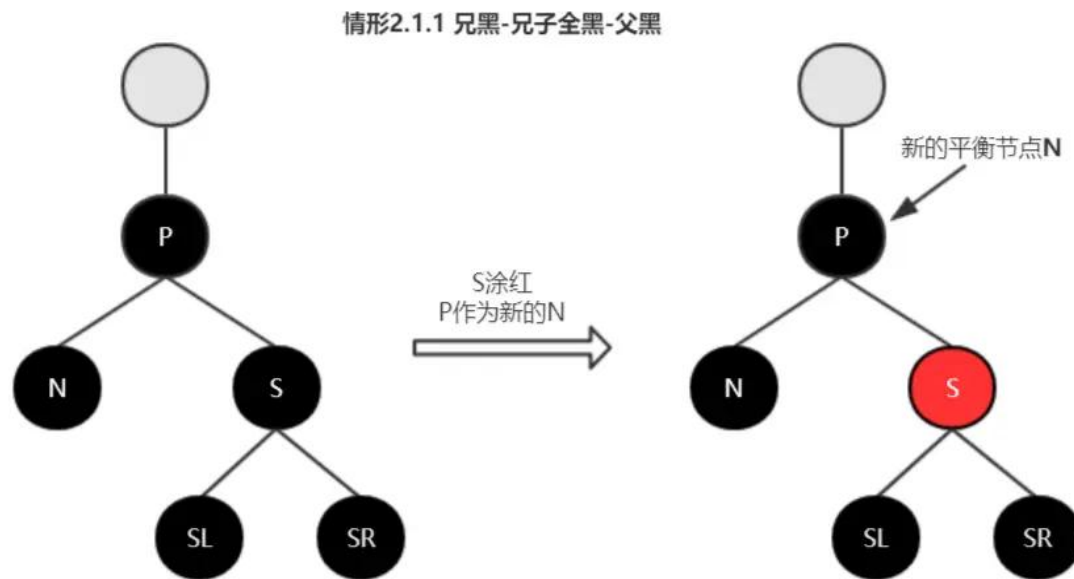
2. 兄弟结点为黑色

- 2.1 兄弟结点的子结点全黑

N: 删除后的空结点, S: 兄弟结点, p: 父节点
SL: 兄弟结点的左孩子, SR: 兄弟结点的右孩子

- 2.1.2 父亲结点为黑色

- 将S的颜色改为红色: 父节点P的左右子树的黑高平衡
- 递归处理父节点P: 以结点P为根节点的黑高-1, 不平衡, 所以递归处理父节点p



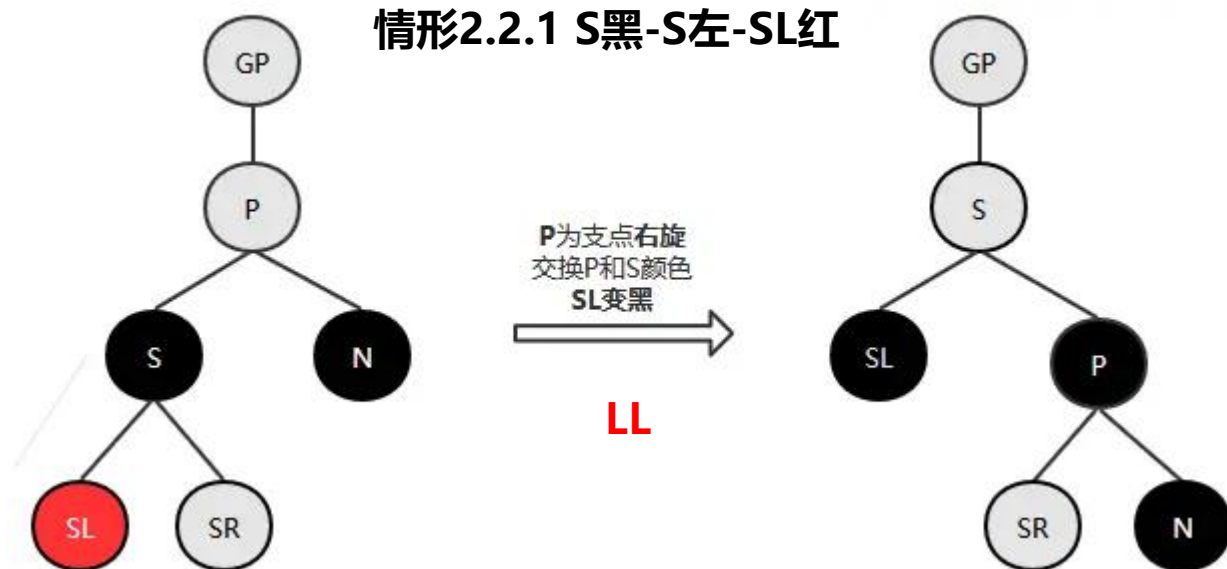
2. 兄弟结点为黑色

- 2.2 兄弟结点的子结点不全黑，兄弟结点为P的左孩子

- 2.2.1 兄弟结点的左孩子为红 (S左-SL红)

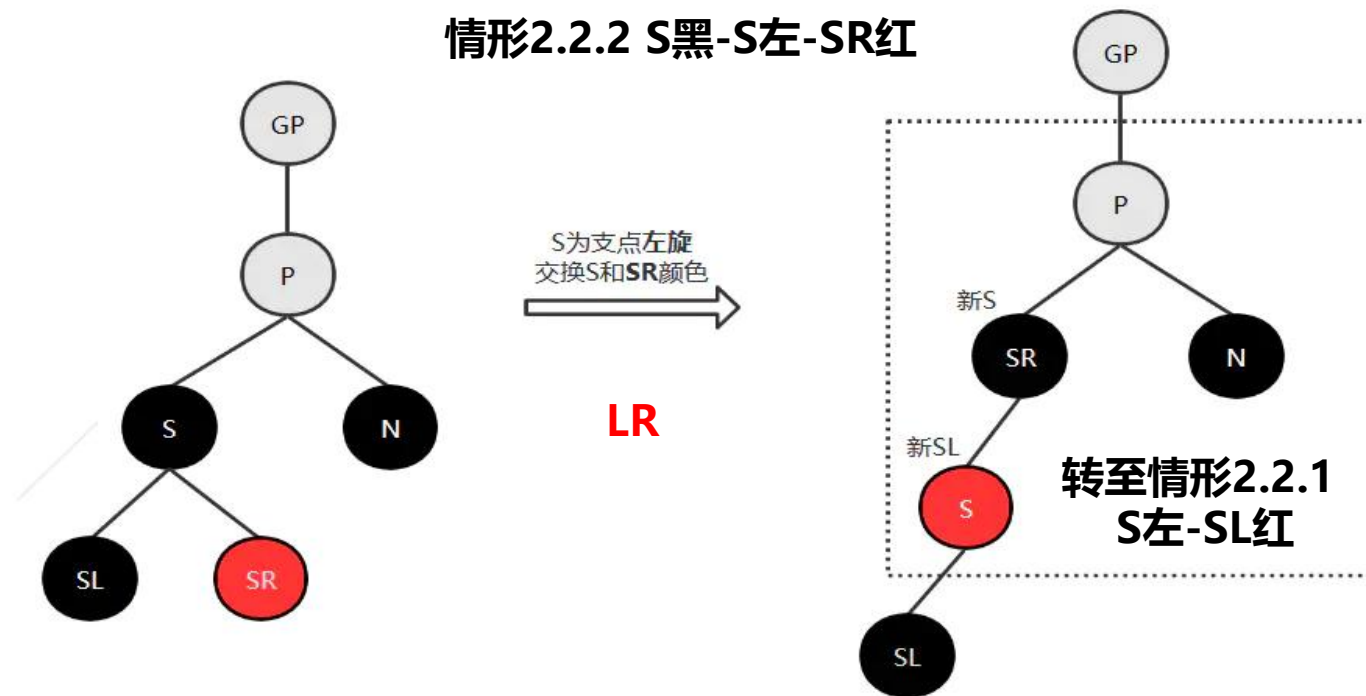
- 对结点P绕结点S右旋，交换P和S的颜色

- SL变黑



2. 兄弟结点为黑色

- 2.2 兄弟结点的子结点不全黑，兄弟结点为P的左孩子
 - 2.2.2 兄弟结点的右孩子为红（S左-SR红）
 - 结点SR绕结点S左旋，并交换S和SR的颜色，
 - 转换为情况2.2.1

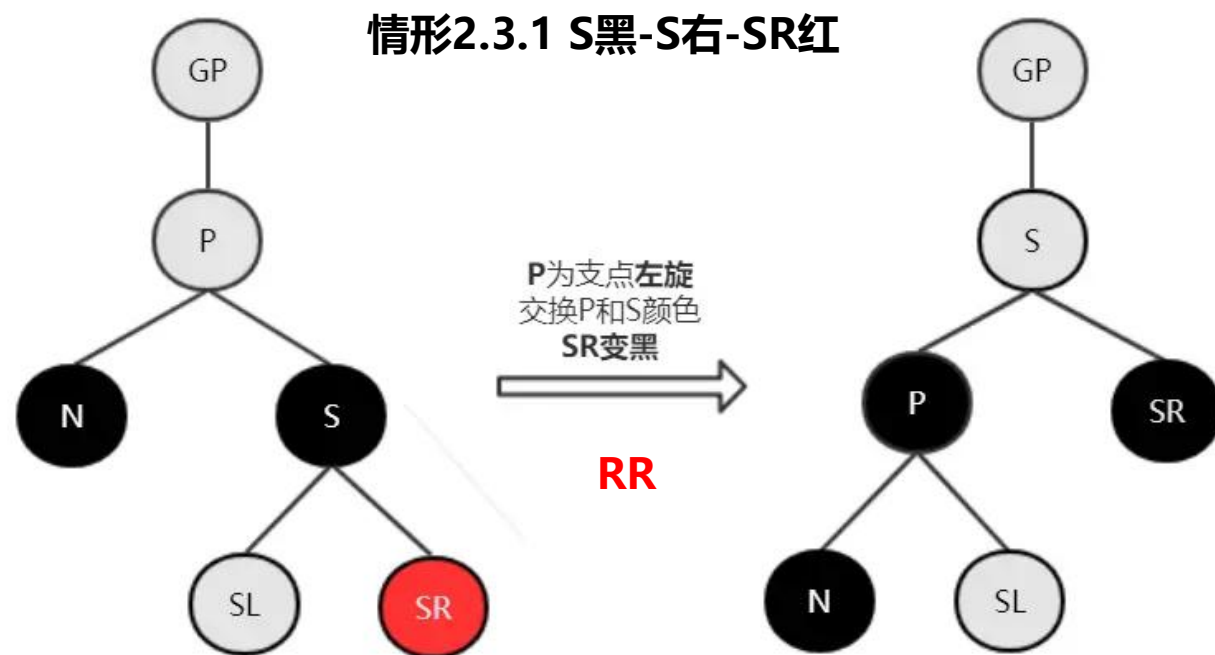


2. 兄弟结点为黑色

- 2.3 兄弟结点的子结点不全黑，兄弟结点为P的右孩子

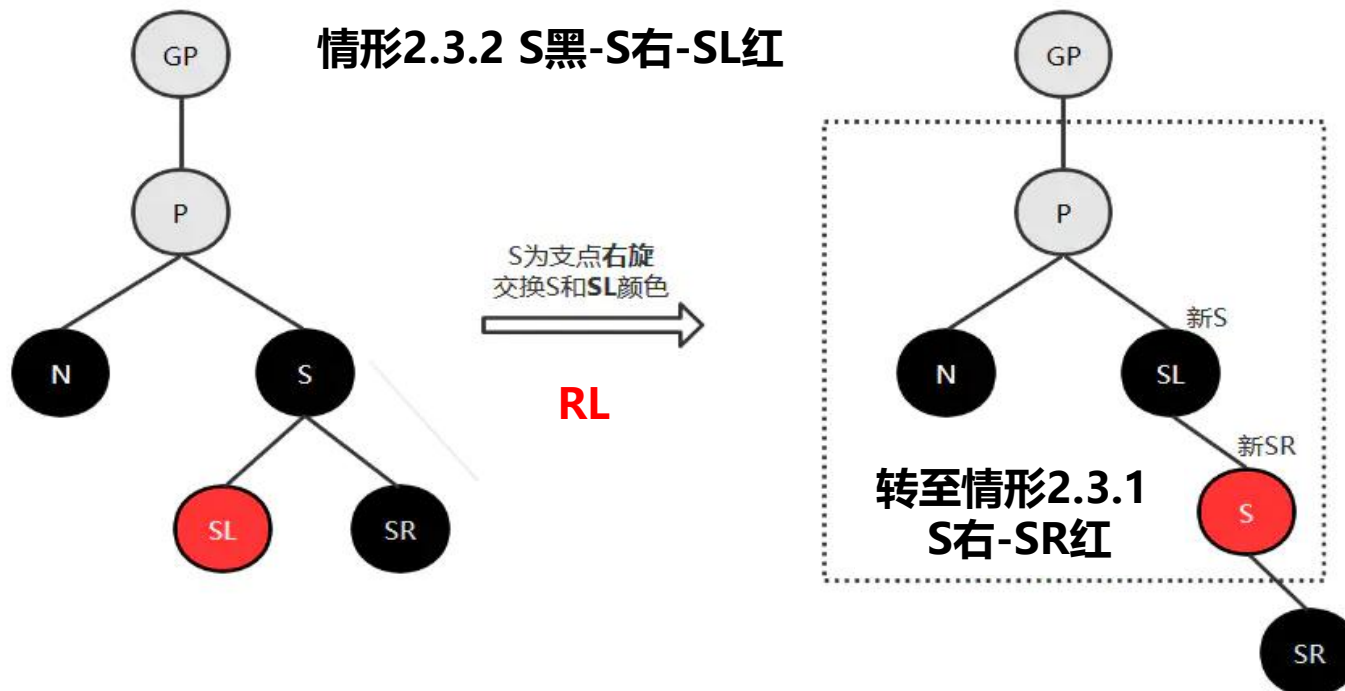
- 2.3.1 兄弟结点的右孩子为红（S右-SR红）

- 结点p绕结点S左旋，并交换P和S的颜色
- SR变黑



2. 兄弟结点为黑色

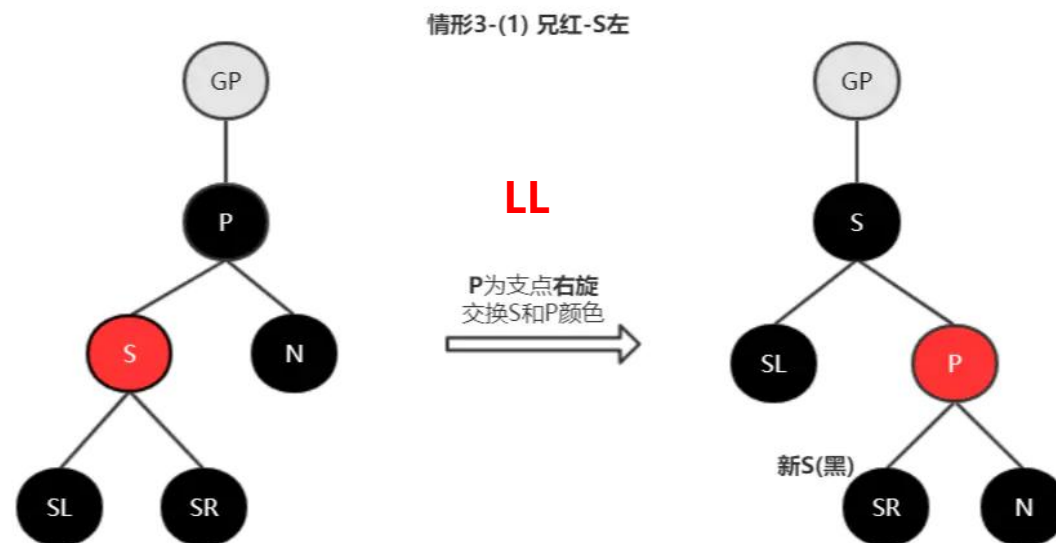
- 2.3 兄弟结点的子结点不全黑，兄弟结点为P的右孩子
 - 2.3.2 兄弟结点的右孩子为黑（S右-SL红）
 - 结点SL绕结点S右旋，并交换S和SL的颜色
 - 转换为情况2.3.1



3. 兄弟结点为红色

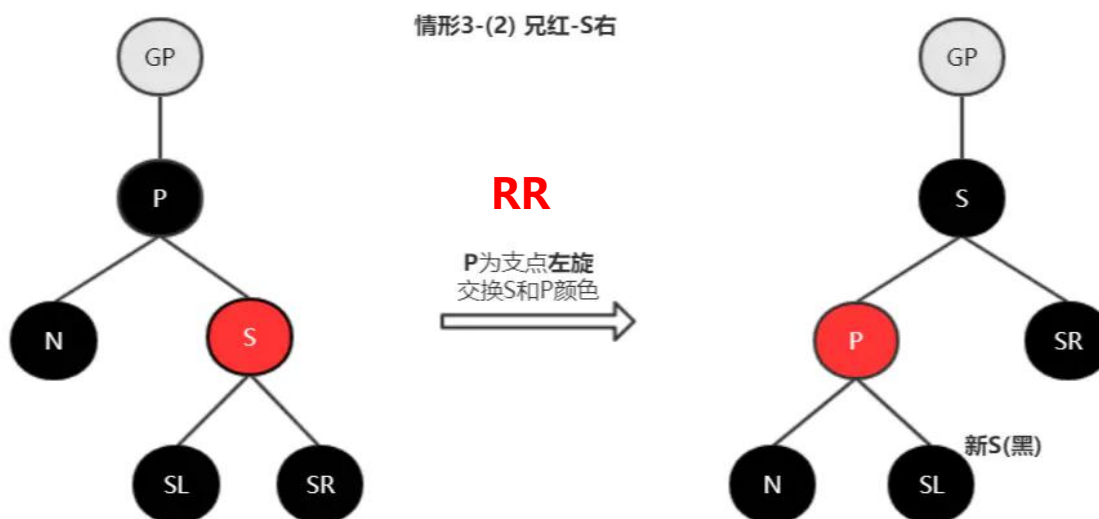
• 3.1 兄弟结点为P的左孩子

- 对结点P绕结点S右旋
- 交换S和P的颜色
- 以SR为S递归调整
 - 转化为情况2：兄弟结点为黑色



• 3.2 兄弟结点为P的右孩子

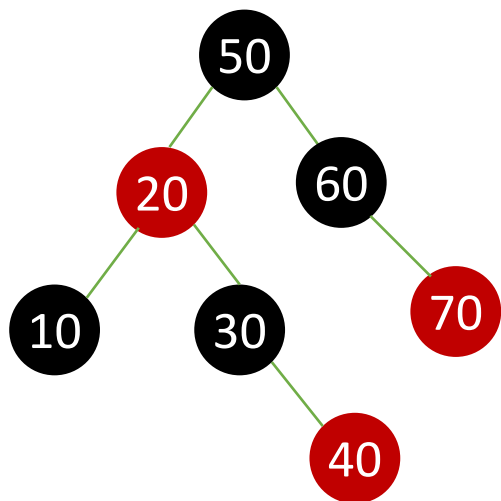
- 对结点P绕结点S左旋
- 交换S和P的颜色
- 以SL为S递归调整
 - 转化为情况2：兄弟结点为黑色



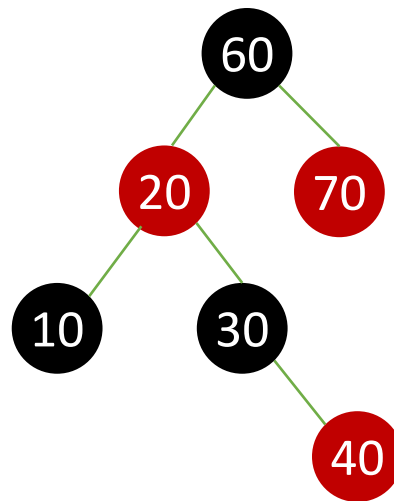
红黑树的删除

- 例子

用右子树的最小值60替代50
转化为删除一个叶子结点的情况



删除50



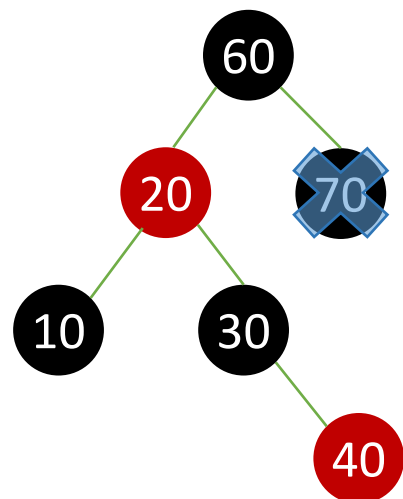
删除70



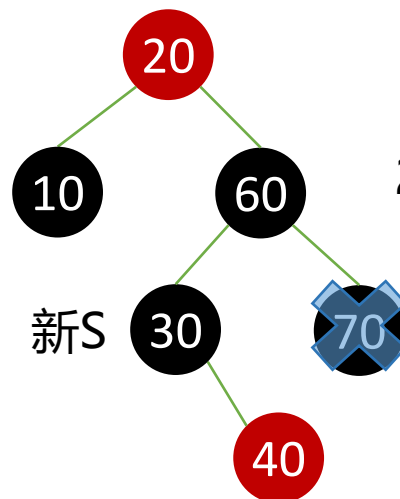
删除70

结点60右旋，转换成case 2.2.2

结点40左旋，转换成case 2.2.1



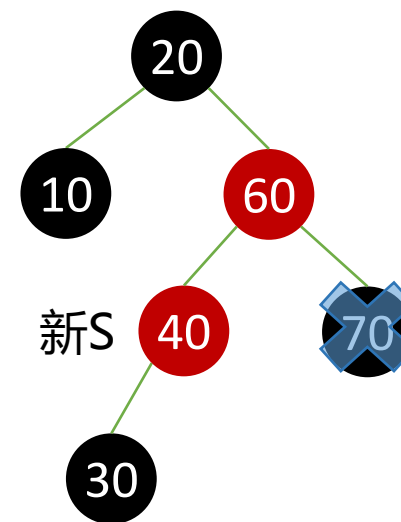
3.1 左兄红



2.2.2 S左SR红



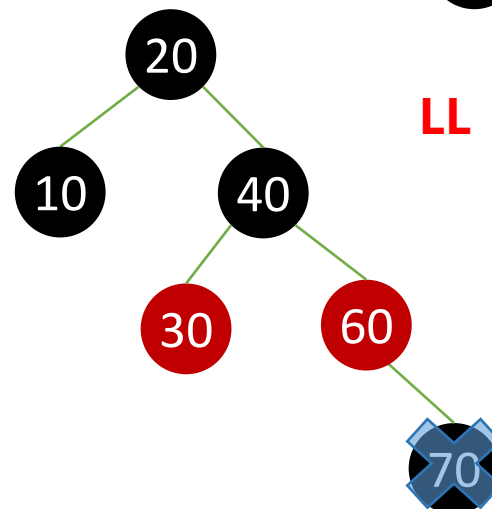
LR



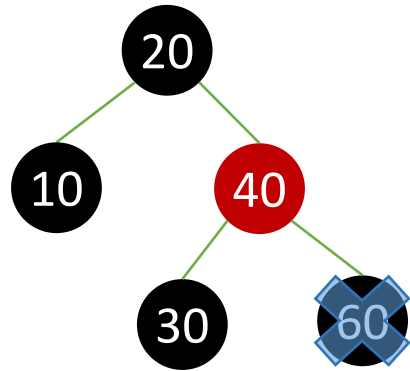
LL



2.2.1 S左SL红



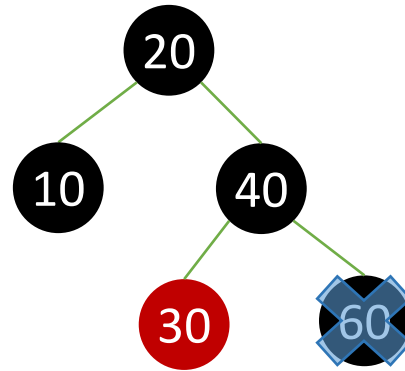
删除60



2.1.1 父红,
兄子全黑



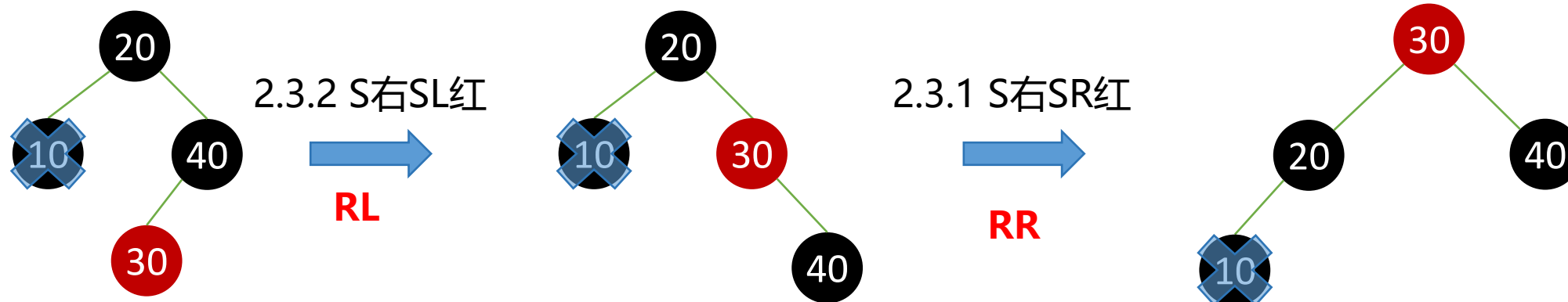
交换30和40的颜色



删除10

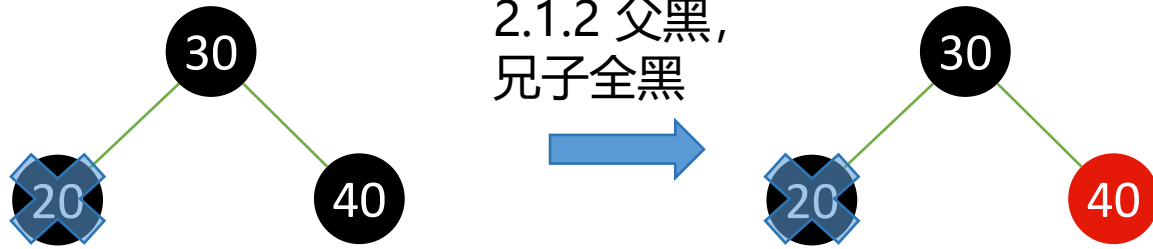
结点30左旋，转换成case 2.3.1

结点20左旋



删除20

将S的颜色改为红色，递归处理父节点



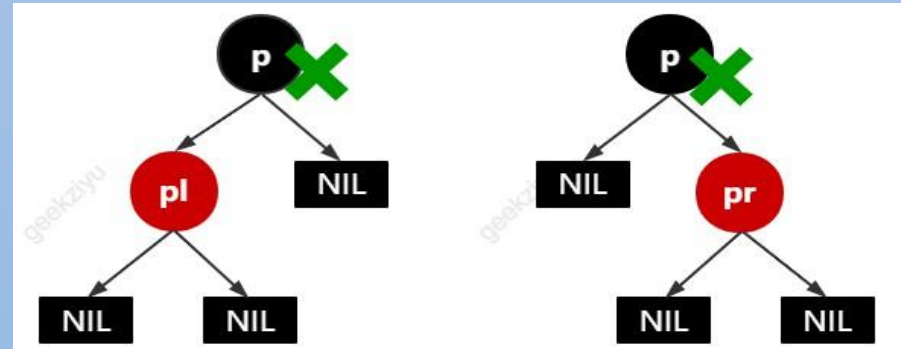
红黑树的删除

```
Elem RBTree::Remove(const Key& key) {  
    RBNode* node = BSTree::find(key);           // 1. 按照二叉搜索树的规则查找  
    if (node) {                                  // 查找成功  
        Elem it = node->element();  
        if (node->left() && node->right()) {      // case 3: 两个非空孩子结点  
            auto temp = getMin(node->right());   // minimum node in right subtree  
            node->setElement(temp->element());  
            node->setKey(temp->key());  
            // convert to case 1 and 2: at least one child is empty  
            node = temp;  
        }  
        //case1 and 2: at least one child is empty  
        removeHelp(node);  
        delete node;  
        return it;  
    }  
    return Elem();                               // 查找失败  
}
```

红黑树的删除

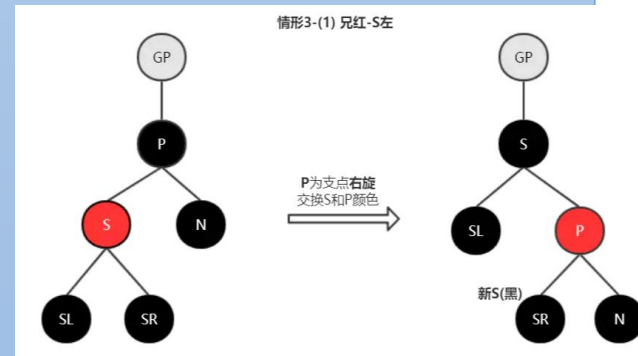
```
Elem RBTREE::removeHelp(RBNode *rt) {  
    RBNode* temp = rt;          auto p = rt->parent();  
    if (rt->left()) {             // case 2: 一个非空结点: 将颜色修改为黑色  
        rt = rt->left();  
        rt->setColor(BLACK);  
    } else if (rt->right()) {     // case 2: 一个非空结点: 将颜色修改为黑色  
        rt = rt->right();  
        rt->setColor(BLACK);  
    } else rt = nullptr;  
    if (!p) setRoot(rt);  
    bool isLeftSibling = true;  
    if (p->left() == temp) {  
        p->setLeft(rt);  
        isLeftSibling = false;  
    } else  
        p->setRight(rt);  
    if (!rt && temp->getColor() == BLACK) // case 1: 删除黑色叶子结点,  
        removeBlackLeafNode(p, isLeftSibling);  
}
```

// case 1: 叶子结点
// delete root



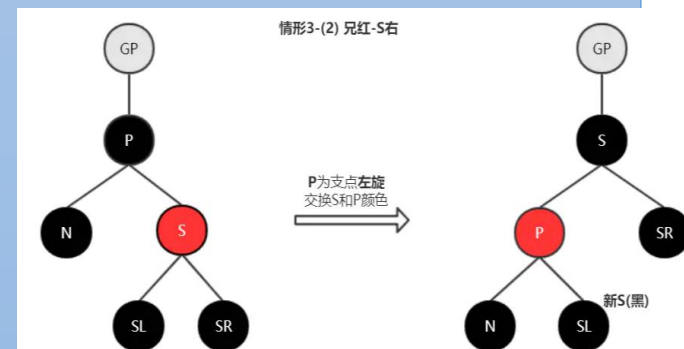
红黑树的删除

```
void RBTree::removeBlackLeafNode(RBNode *p, bool isLeftSibling) {
    if (!p) return;
    if (!isLeftSibling) {        // right sibling, left child
        auto s = p->left();
        if (s->isRed()) {        // case 3.1: Left sibling is red
            auto sr = s->right();
            RightRotation(s);
            s->setColor(BLACK);    // swap color of p and s
            p->setColor(RED);
            s = sr;                // continue to adjust
            if (s) {                // convert to case 2
                p = sr->parent();
                removeBlackCase2(s, p, true);
            }
        } else                    // case 2: sibling is black
            removeBlackCase2(s, p, true);
    } else {                    // left sibling, right child
    }
```



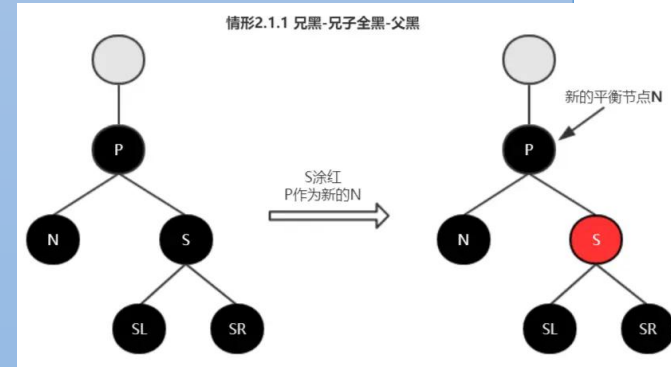
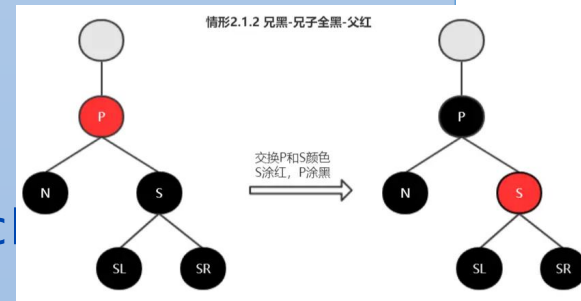
红黑树的删除

```
void RBTree::removeBlackLeafNode(RBNode *p, bool isLeftSibling) {  
    if (!isLeftSibling) {    // left sibling, right child  
        .....  
    } else {                // right sibling, left child  
        auto s = p->right();  
        if (s->isRed()) {    // case 3.2: right sibling is red  
            auto sl = s->left();  
            LeftRotation(p, s);  
            s->setColor(BLACK);    // swap color of p and s  
            p->setColor(RED);  
            s = sl;  
            if (s) {          // continue to adjust  
                // convert to case 2  
                p = sl->parent();  
                removeBlackCase2(s, p, true);  
            }  
        } else              // case 2: sibling is black  
            removeBlackCase2(s, p, false);  
    }  
}
```



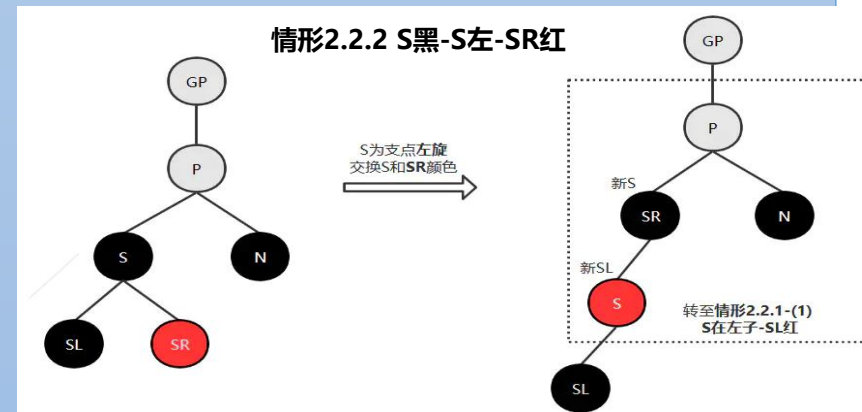
红黑树的删除

```
void RBTree::removeBlackCase2(RBNode *p, bool isLeftSibling) {  
    // case2.1: both children of s are black  
    auto sl = s->left(), sr = s->right();  
    if ((!sl || sl && sl->isBlack()) && (!sr || sr && sr->isBlack())) {  
        if (p->isRed()) {                // case 2.1.1: parent is red  
            s->setColor(RED);              // swap color of p and s  
            p->setColor(BLACK);  
        } else {                          // case 2.1.2  parent is black  
            s->setColor(RED);  
            auto g = p->parent();          // process in recursive: to remove p  
            if (g)  
                removeBlackLeafNode(g, p == g->right());  
        }  
    } else {                              // case 2.2 & 2.3:  
        .....  
    }  
}
```

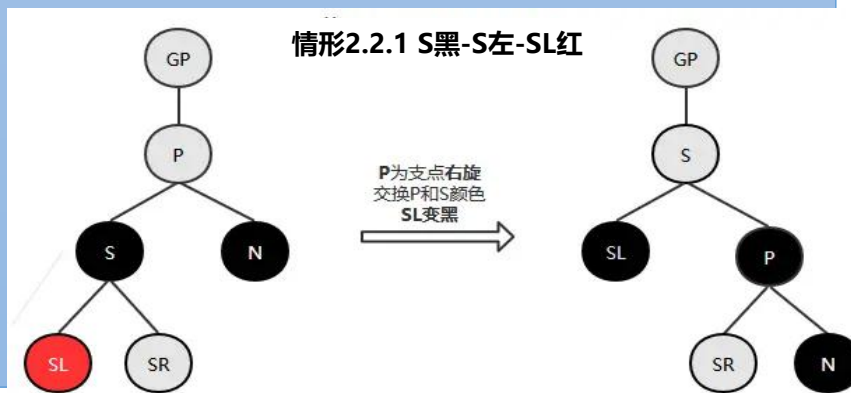


红黑树的删除

```
void RBTree::removeBlackCase2(RBNode *p, bool isLeftSibling) {  
    // case 2.2: one of children of s are red, sibling node at left  
    if (isLeftSibling) { //S左  
        if (!sl || sr->isRed()) { // case 2.2.2: S左-SR红  
            LeftRotation(sr, s);  
            s->setColor(sr->color());  
            sr->setColor(BLACK);  
            // convert to case 2.2.1: S左 - SL红  
            sl = s;    s = sr;  
        }  
        // case 2.2.1: S左 - SL红  
        s->setColor(p->color());  
        p->setColor(BLACK);  
        sl->setColor(BLACK);  
        RightRotation(p, s);  
    }  
    else {  
        // S右  
    }  
}
```

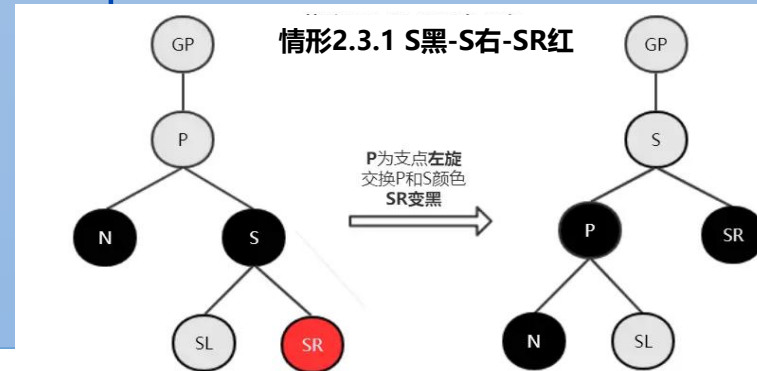
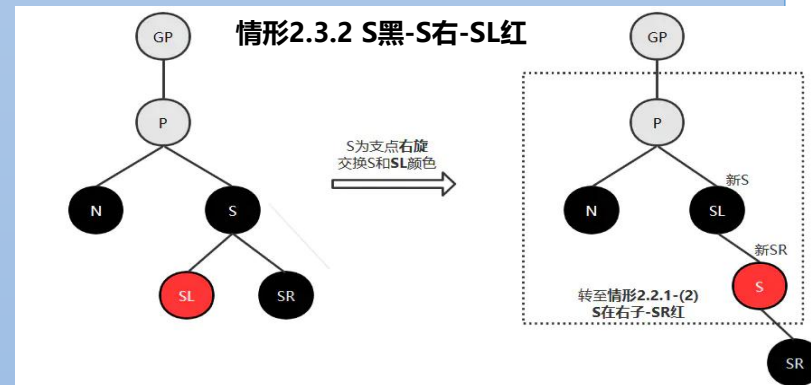


// swap color of p and s



红黑树的删除

```
void RBTree::removeBlackCase2(RBNode *p, bool isLeftSibling) {  
    // case 2.3: one of children of s are red, sibling node at right  
    if (isLeftSibling) { //S左  
    } else { // S右  
        if (!sr || sl->isRed()) { // case 2.3.2: S右 - SL红  
            RightRotate(sl, s);  
            s->setColor(sl->color());  
            sl->setColor(BLACK);  
            sr = s; // convert to case 2.3.1: S右 -  
            s = sl;  
        }  
        // case 2.3.1: S右 - SR红  
        s->setColor(p->color());  
        p->setColor(BLACK);  
        sr->setColor(BLACK);  
        LeftRotate(p, s);  
    }  
}
```



红黑树 VS AVL树

- 红黑树和AVL树都是高效的平衡二叉树，增删改查的时间复杂度都是 $O(\log_2 N)$
 - 红黑树不追求绝对平衡，只需保证最长路径不超过最短路径的2倍，降低了旋转的次数
 - AVL树的高度差不超过1，平衡的要求更高
- 在经常进行增删的应用中，红黑树的性能比AVL树更优
 - 如STL中的map就是基于红黑树实现的

红黑树 VS AVL树

	红黑树	平衡二叉树
相同点	都是二叉排序树	都是二叉排序树
查找效率	一般时间复杂度为 $O(\log N)$ ，最坏情况差于AVL	时间复杂度稳定在 $O(\log N)$
插入效率	需要旋转操作和变色操作，插入结点最多只需要2次旋转；变色需要 $O(\log N)$ ；	插入结点最多只需要1次旋转， $O(\log N)$ 级别
删除效率	删除一个结点最多需要3次旋转操作	每一次删除操作最多需要 $O(\log N)$ 次旋转
优劣势	数据读取效率低于AVL，维护性强于AVL	数据读取效率高，维护性较差
应用场景	搜索，插入，删除操作差不多	搜索的次数远远大于插入和删除  业余码农