数据结构与算法

DATA STRUCTURE

第十八讲 平衡二叉查找树

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

课堂内容

- 平衡二叉查找树

二叉查找树回顾

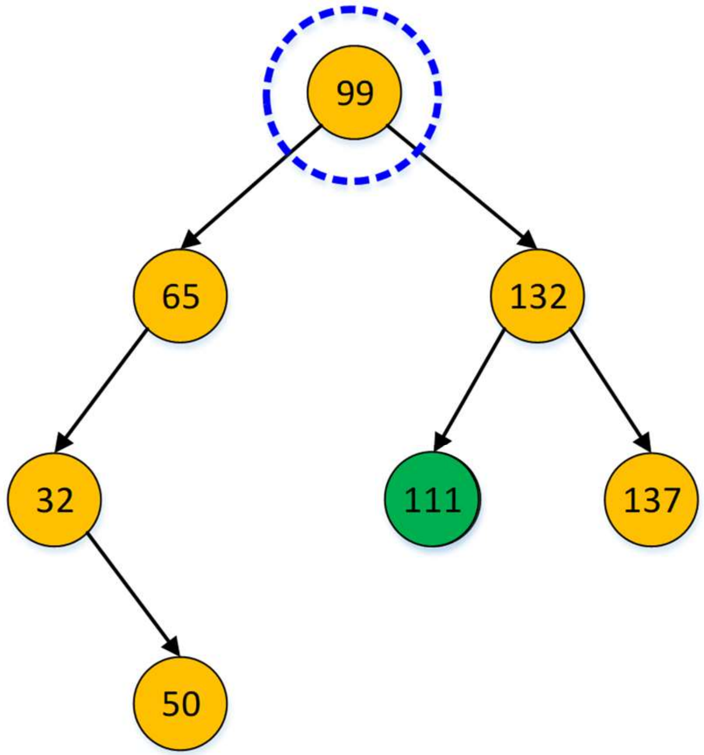
Binary Search Tree

内容

- 为什么使用二叉查找树
- 什么是二叉查找树Binary Search Tree
- 怎么动态维护二叉查找树：

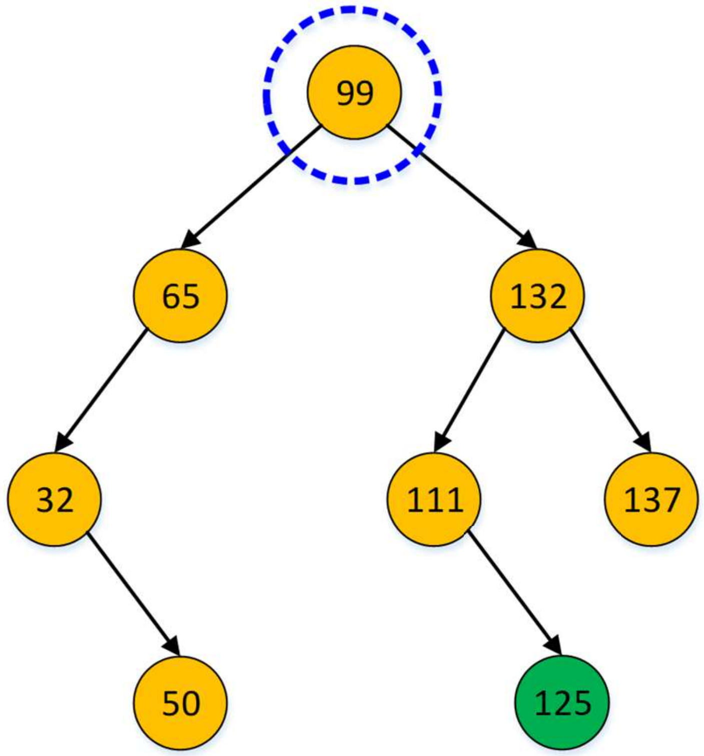
  查找/插入/删除

查找算法(111)



因为对查找特别友好，所以称为二叉查找树

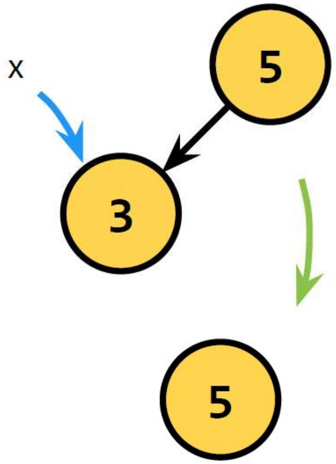
插入算法(125)



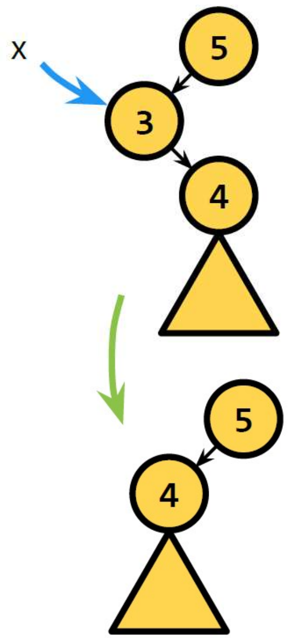
先查找到叶节点，然后根据大小插入为左/右子节点

删除算法总结

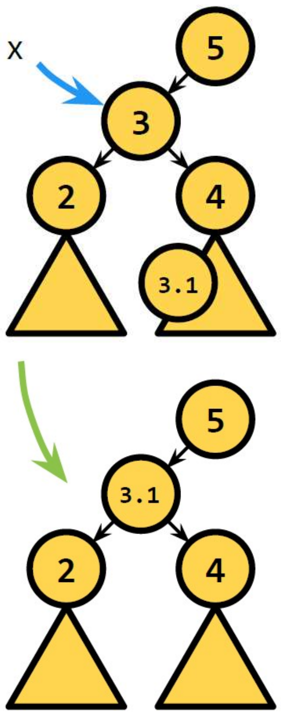
Case 1： x是叶节点，直接删除



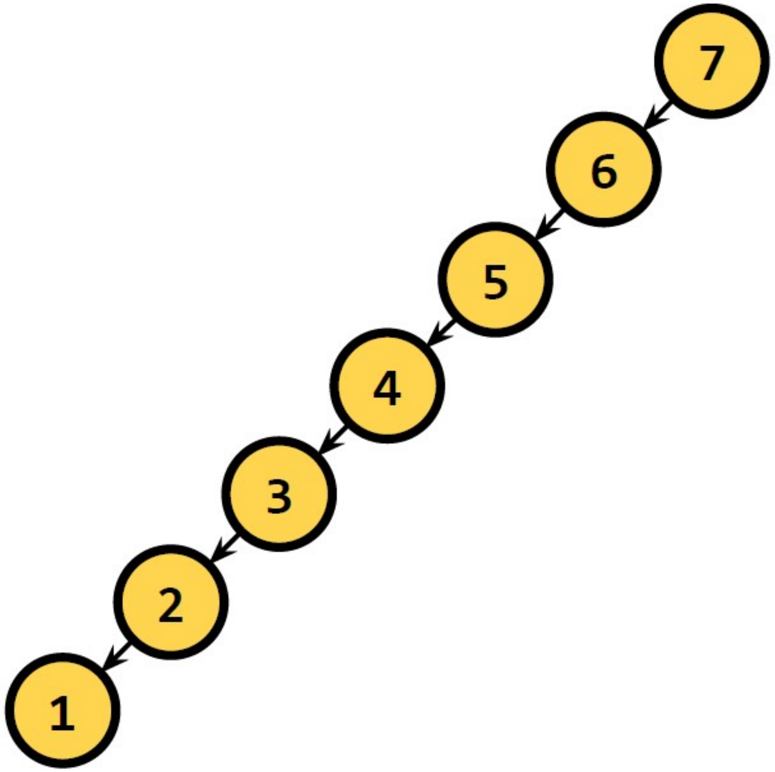
Case 2： x有一个子节点，直接把子节点的子节点接到父节点



Case 3： x有两个子节点，用x的直接后继代替， 然后删除后继原来的节点（最多有一个子节点）



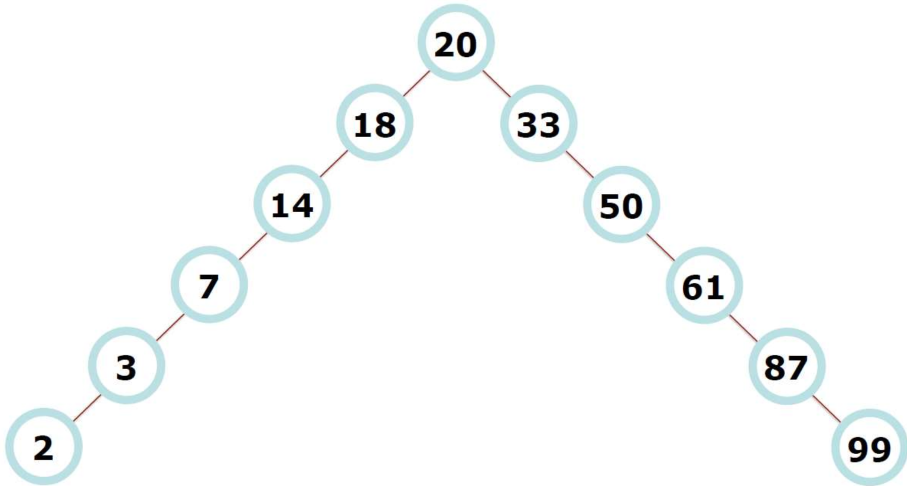
问题是……



- 这还是一个有效的BST
- 但是树的高度是n, 不再是 $O(\log(n))$
- 其实退化成了链表

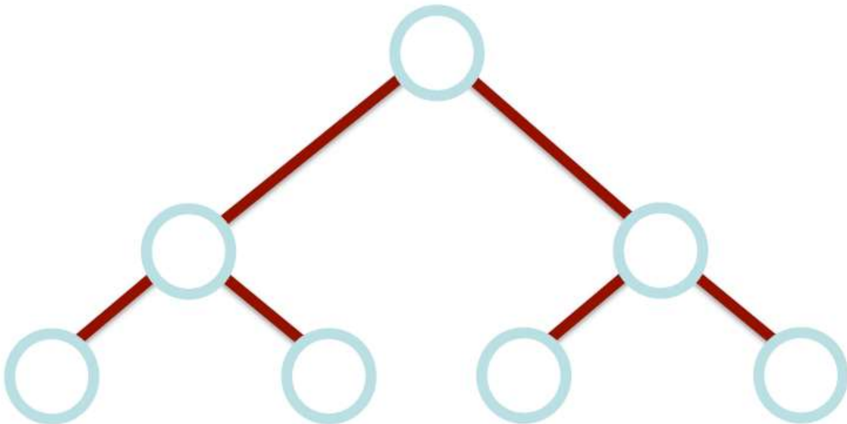
维持O(log n) 复杂度的思路

- 保持二叉查找树的左右节点个数平衡
- 不过不是下面这种平衡，即只有根节点是平衡的



维持O(log n) 复杂度的思路

- 另一方面，如果要求所有节点的左右子树完全平衡
- 却是太难达到要求，只有满的完全的二叉树才是
- 思路是我们只需要维持大体平衡，
- 希望所有节点的左右子树的节点个数只差常数倍

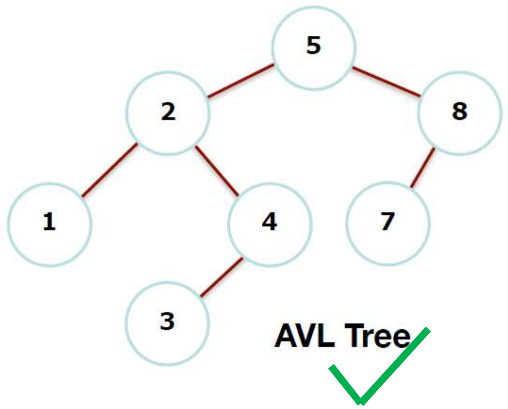


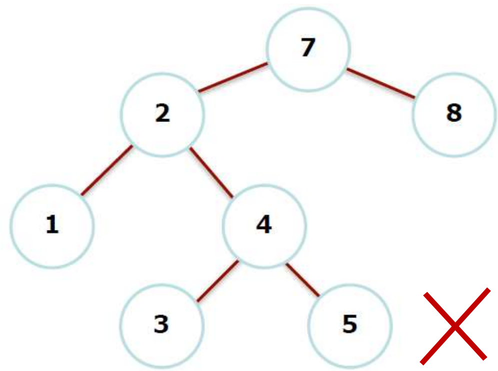
平衡二叉查找树

- AVL树
- 红黑树

AVL树

- AVL tree (Adelson-Velskii and Landis)
- 同样是一个二叉查找树
- 满足任意节点的左子树和右子树的高度最多相差1
- 空的树高度为-1，只有一个节点的树高度为0



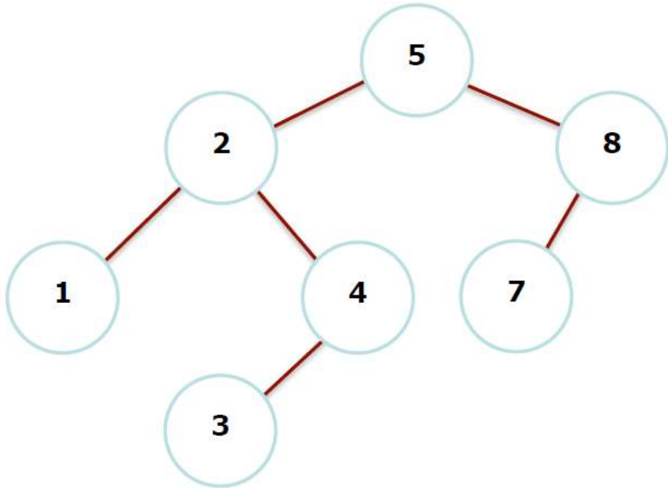


判断树平衡

- 证明有n个节点的AVL树高是$logn$

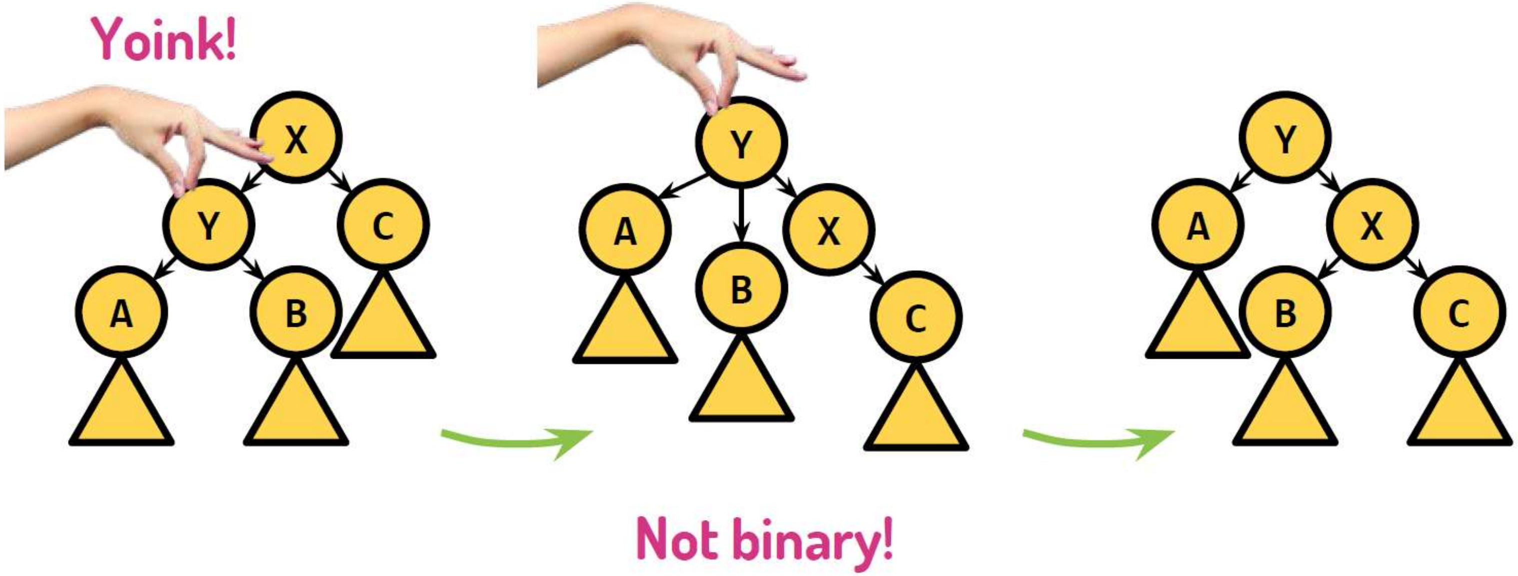
- 如何保持AVL的特性
  - 每个节点需要保存height的信息
  - 记得需要更新相关节点的高度，在每次插入/删除后

- 例如：如果再要求插入6，看节点8是不是破坏平衡了？



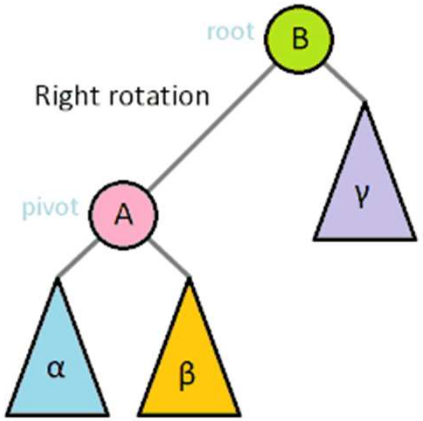
如何保持平衡

旋转

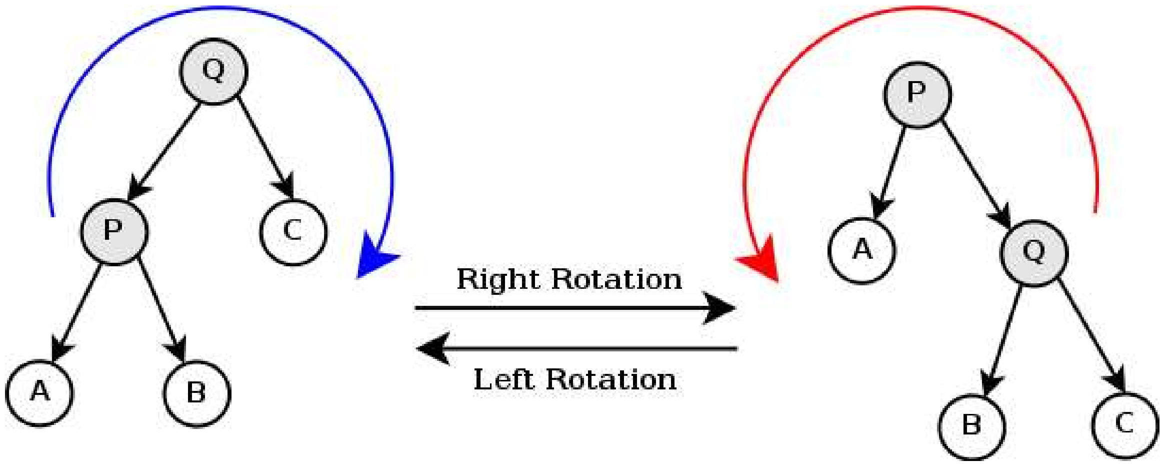


二叉树里保持平衡一直会使用的方法
反过来也一样

旋转



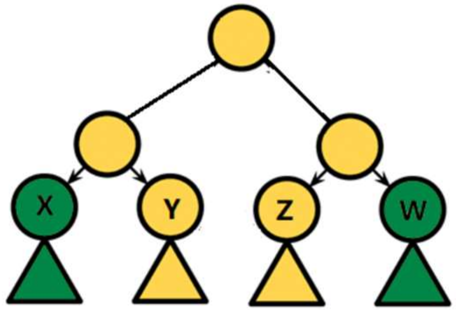
root
B
Right rotation
pivot
A
γ
α
β



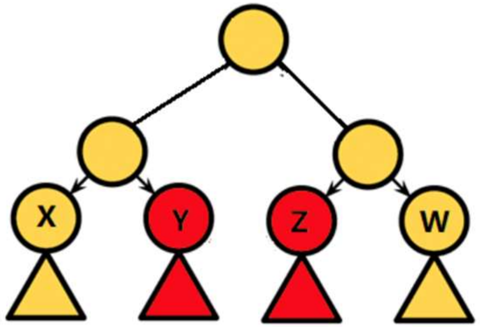
Q
P
C
A
B
Right Rotation
Left Rotation
P
A
Q
B
C

分情况旋转

1. 如果插入的节点在X、W子树中

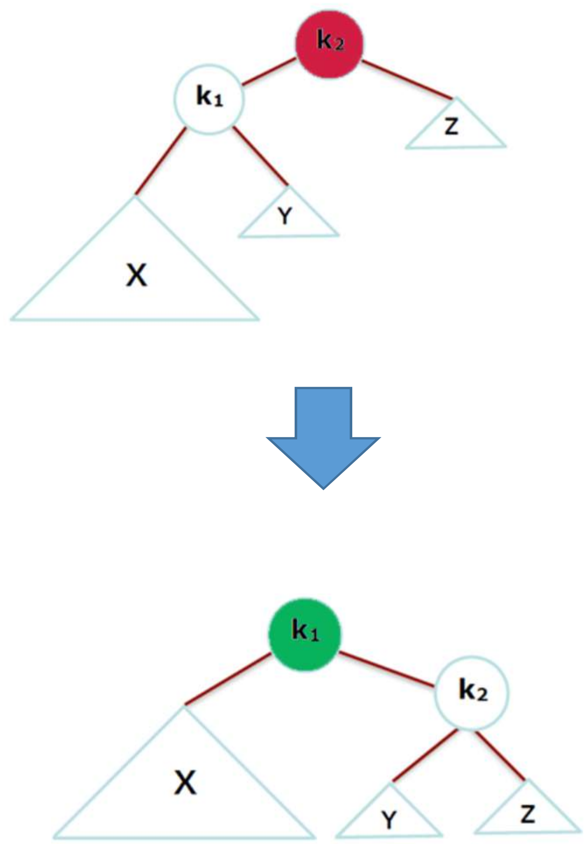


2. 如果插入的节点在Y、Z子树中



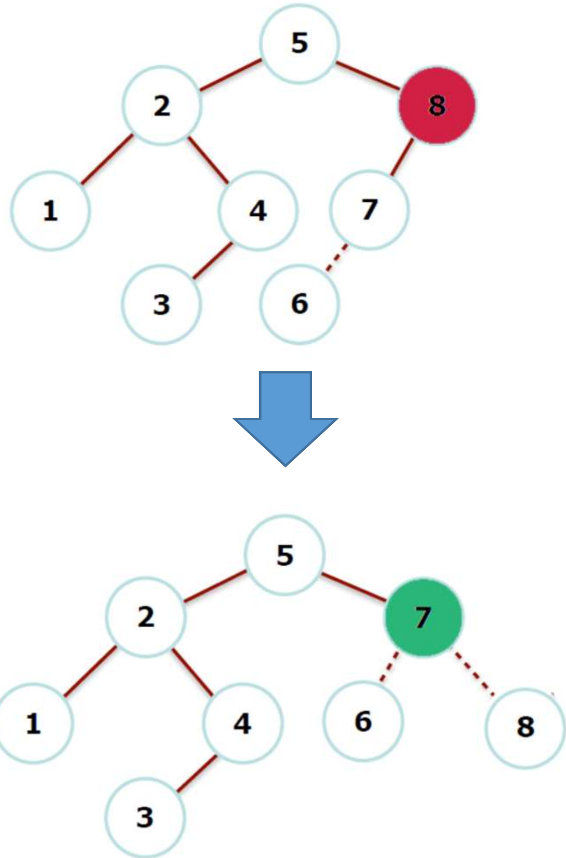
情况1

- 如果在新节点插入到靠外边的子树后，AVL树失去平衡了
- 那么X的高度一定比Y高1，比Z高1



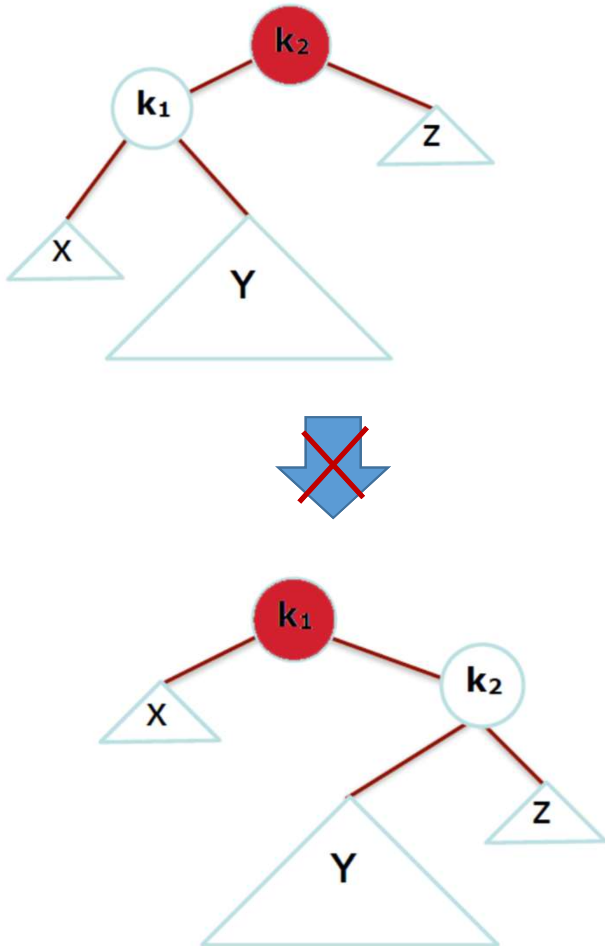
回到之前的例子

- 插入6后，
- 节点8违反了规则，不是节点5
- 属于情形1



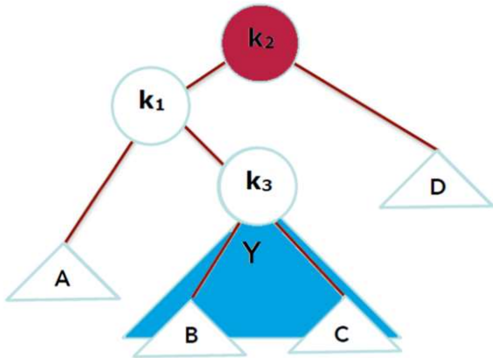
情形2

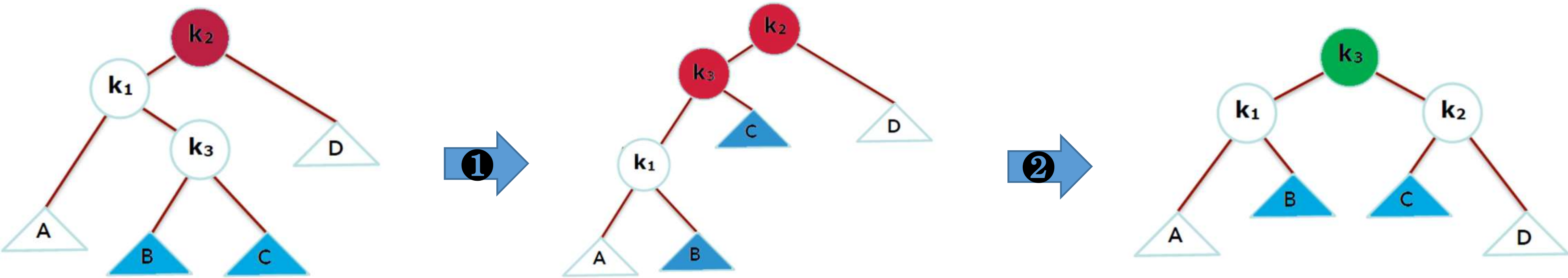
- 如果在新节点插入到靠内部的子树后，AVL树失去平衡了
- 之前的类似情形1的旋转不行



情形2

- 如果在新节点插入到靠内部的子树后，AVL树失去平衡了
- 对Y子树再细分



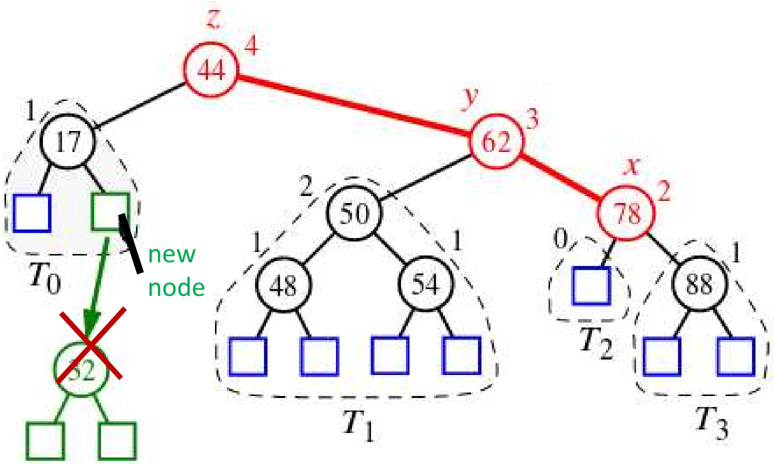


AVL插入算法

- 先按BST插入
- 如果子树高度没有变化，就结束
- 要不然，看哪个子树需要重新平衡（旋转）
- 旋转后还需要更新高度

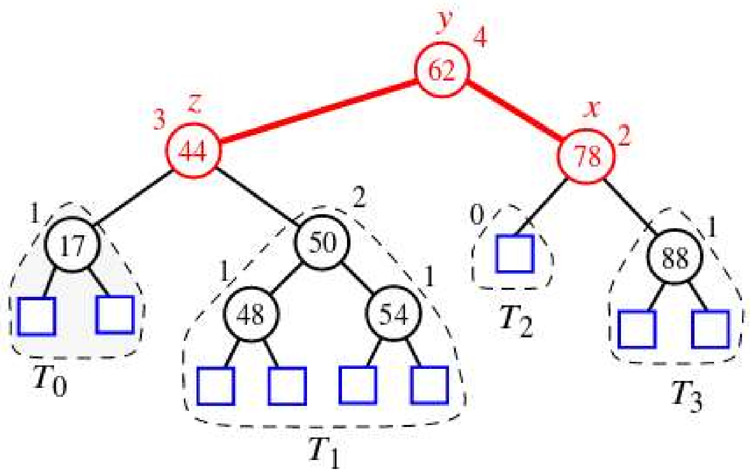
- 删除算法也是类似，多了重新平衡的逻辑

删除例子



z
44
4
1
17
y
62
3
x
78
2
2
50
1
1
48
54
0
88
1
T_0
new
node
32
T_1
T_2
T_3

绕y左旋



y
62
4
3
z
44
x
78
2
1
17
2
50
0
1
88
1
48
1
54
T_0
T_1
T_2
T_3

红黑树

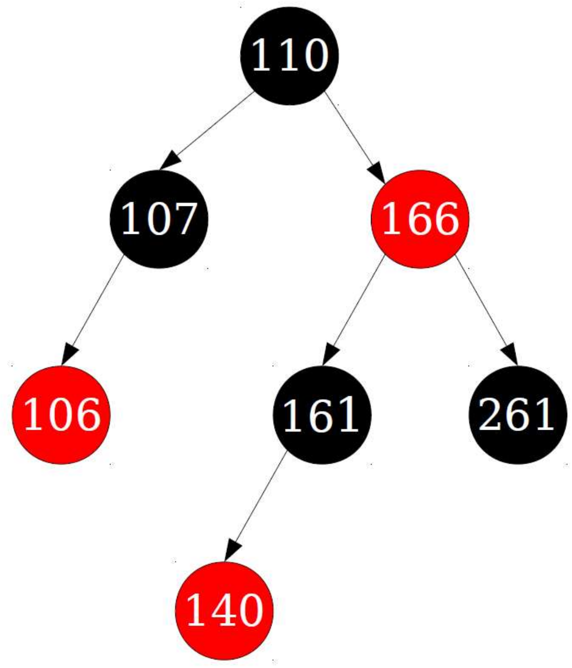
Red Black Tree

什么是红黑树

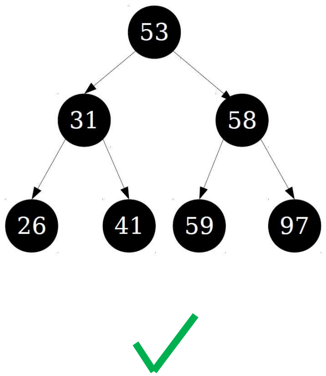
同样是个二叉查找树，用如下规则保持平衡：

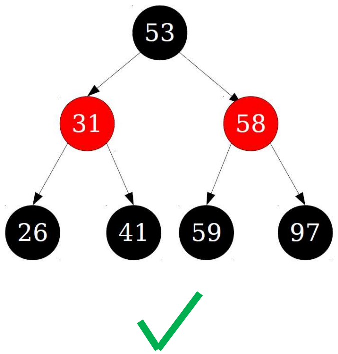
1. 每个节点要么**红色**，要么**黑色**
2. 根节点是**黑色**
3. Nil节点算**黑色**
4. **红色**节点的子节点必须是**黑色**
5. 从任意节点开始，到每个叶节点的路径上的**黑色**节点的个数必须一样

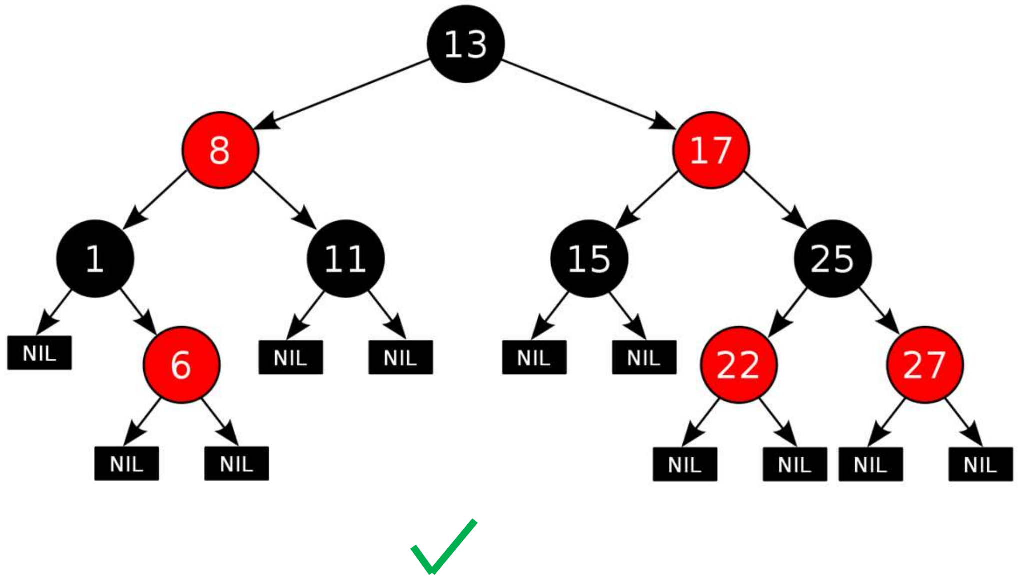
用**黑色**节点来保持平衡，**红色**节点不见得多

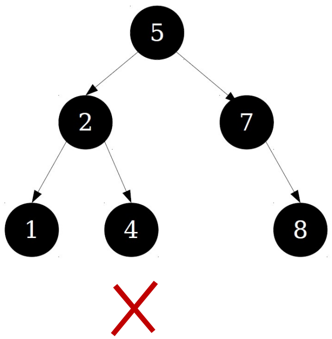


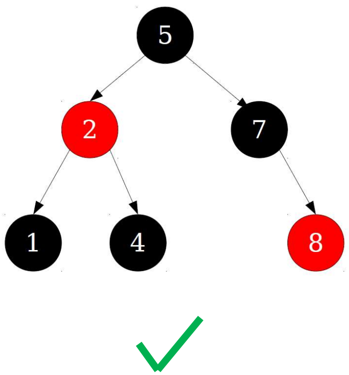
例子


53
31
58
26
41
59
97


53
31
58
26
41
59
97

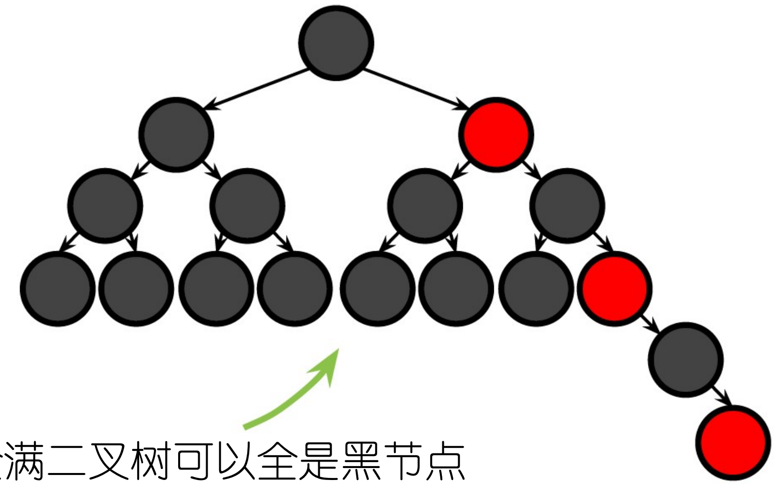

13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL


5
2
7
1
4
8


5
2
7
1
4
8

定理

- 红黑树的高度最大为$2\log(n+1)$



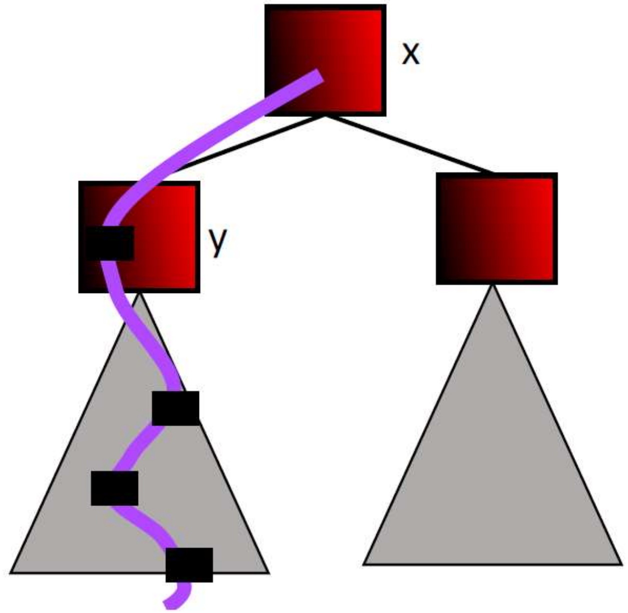
完全满二叉树可以全是黑节点
沿着一条路径最多能放一半红节点

证明

- 假定$b(x)$表示节点x到任意一个叶节点的路径上的黑节点的个数
- 那么可以证明以x为根节点的子树里至少有$2^{b(x)}-1$个节点
- 那么

$$
\begin{aligned}
n &\geq 2^{b(root)} - 1 \\
&\geq 2^{height/2} - 1
\end{aligned}
$$

- 即 $n + 1 \geq 2^{height/2} \Rightarrow height \leq 2\log(n+1)$

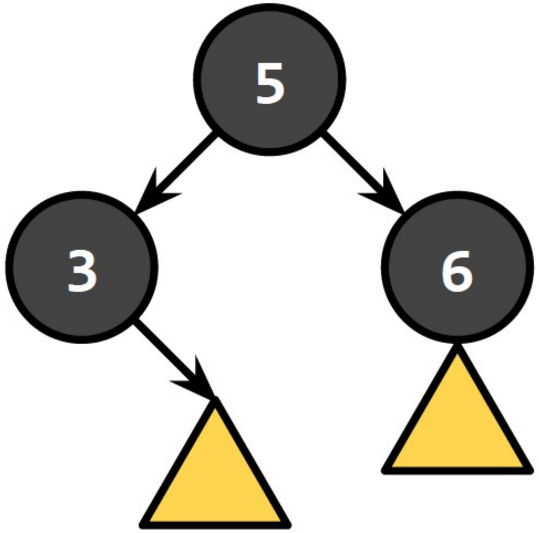


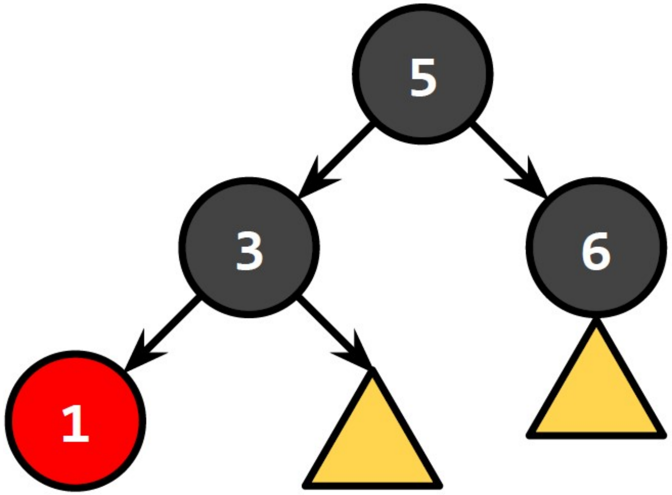
红黑树能保持平衡

- 怎么来插入/删除？

插入算法

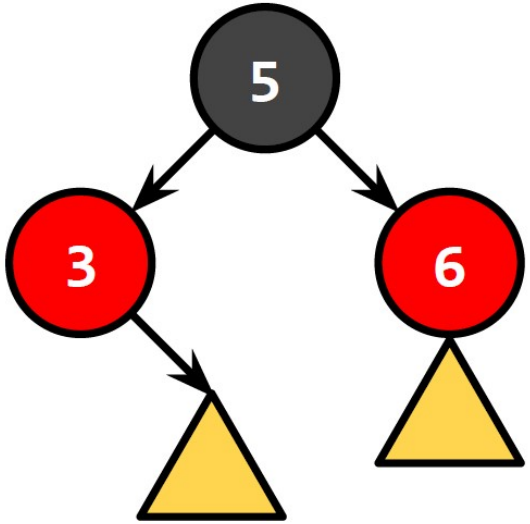
- 如果插入节点的父节点是黑色，直接以红色节点插入，Done

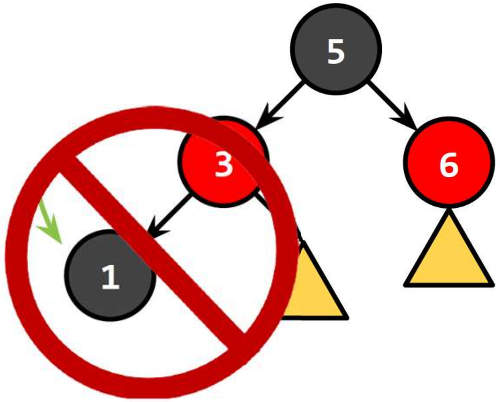


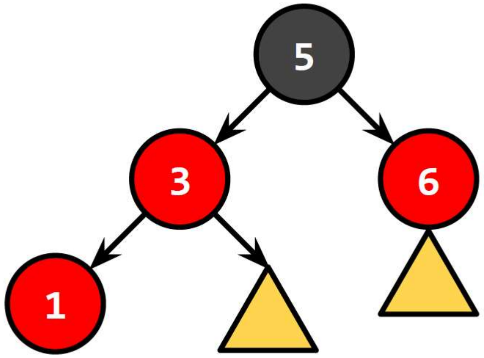


插入算法

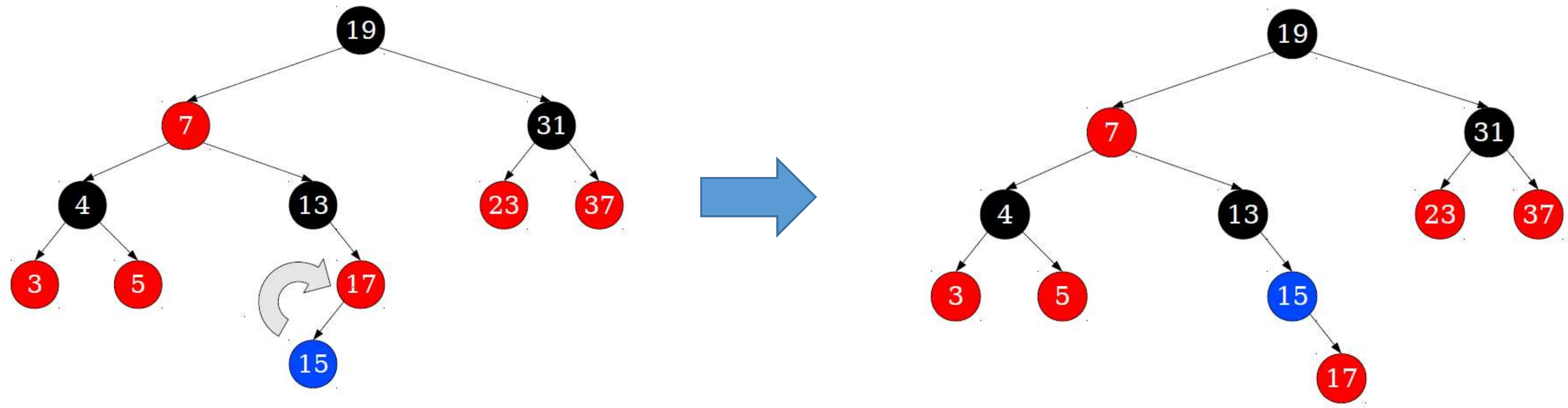
- 如果插入节点的父节点是红色，不太好办！



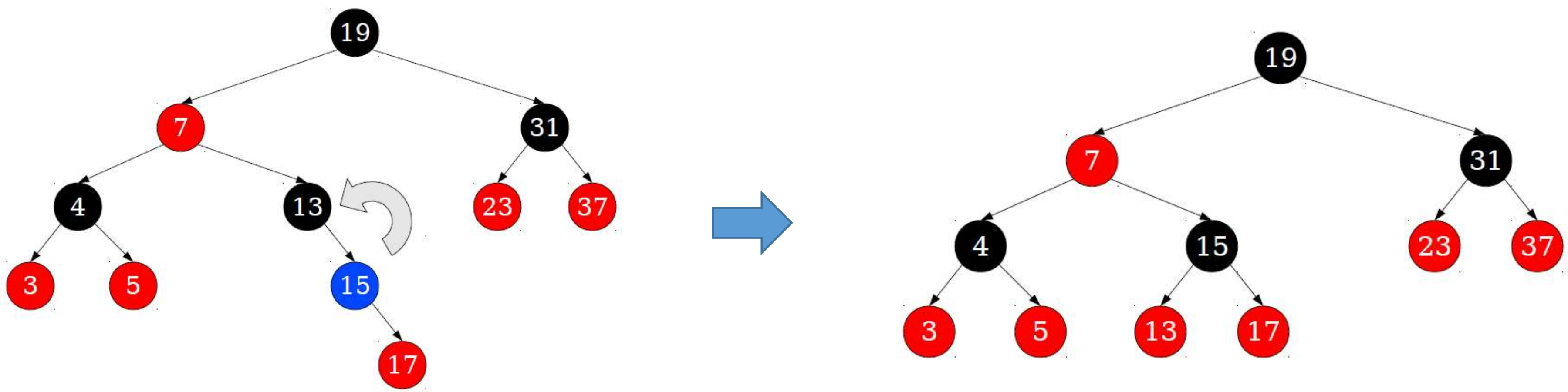




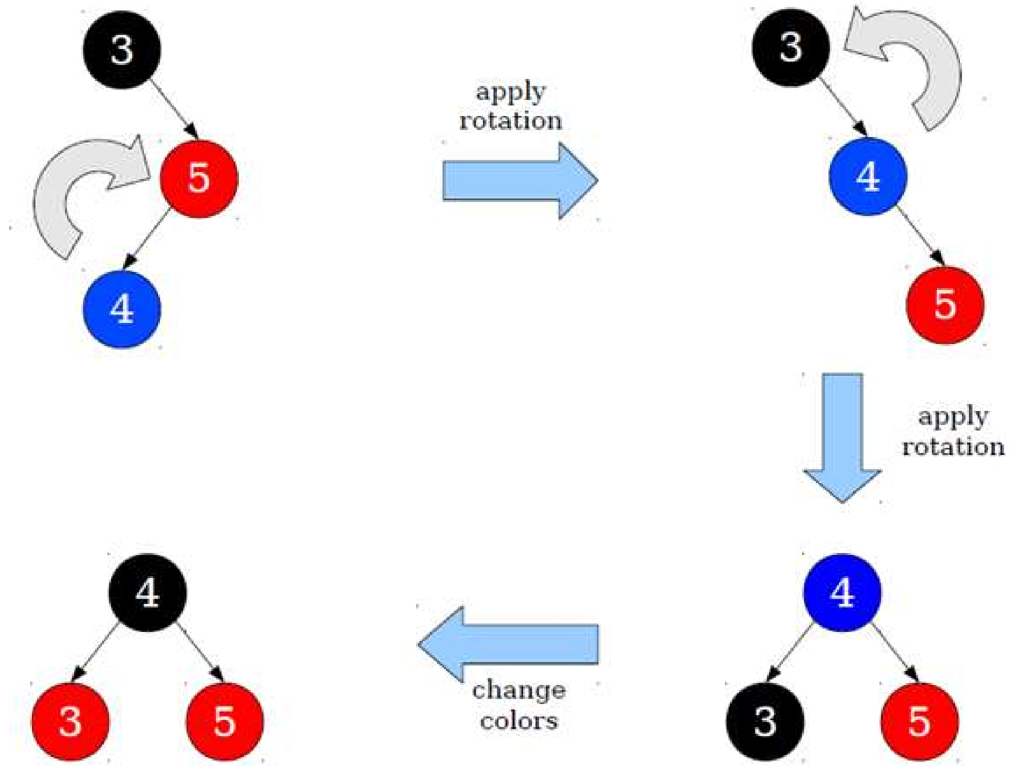
例子1：父节点是红色，没有兄弟节点



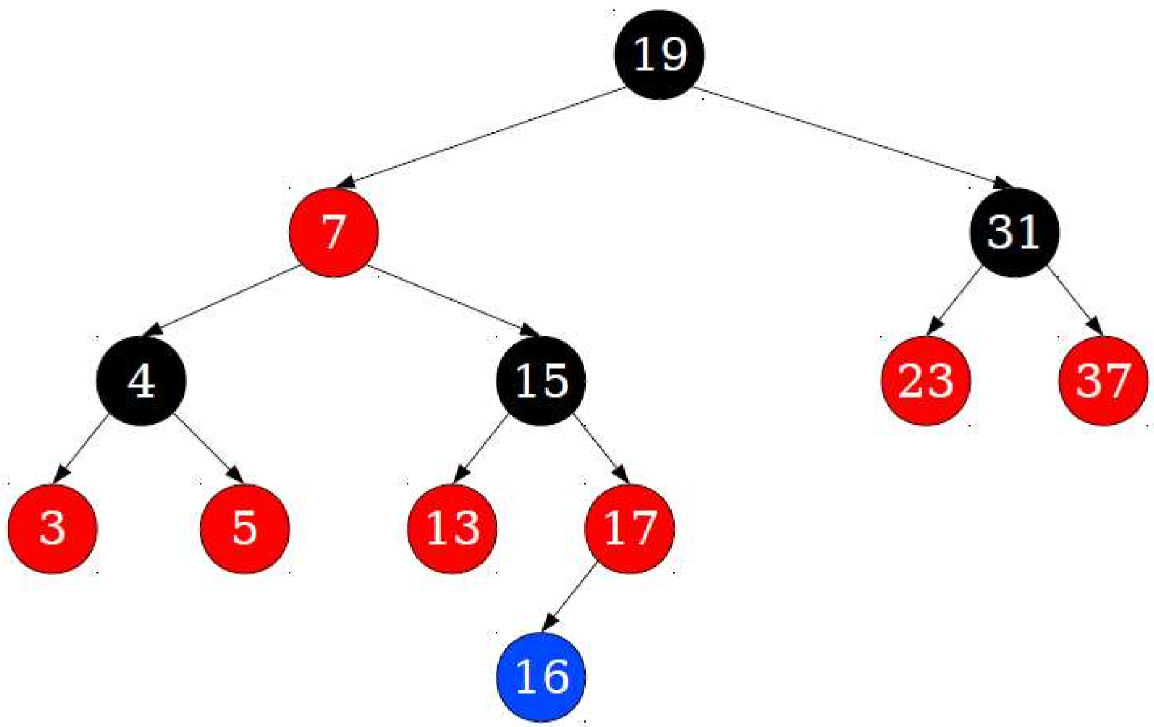
例子1：父节点是红色，没有兄弟节点



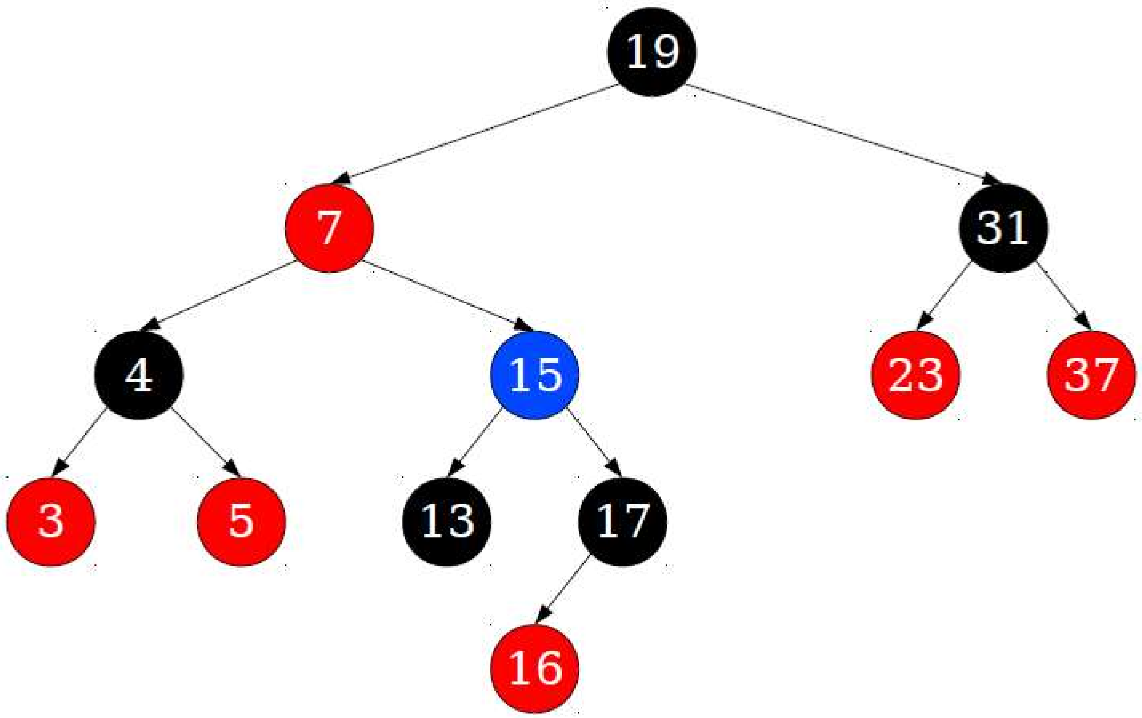
例子1：


3
5
4
apply
rotation
3
4
5
apply
rotation
4
3
5
change
colors
4
3
5

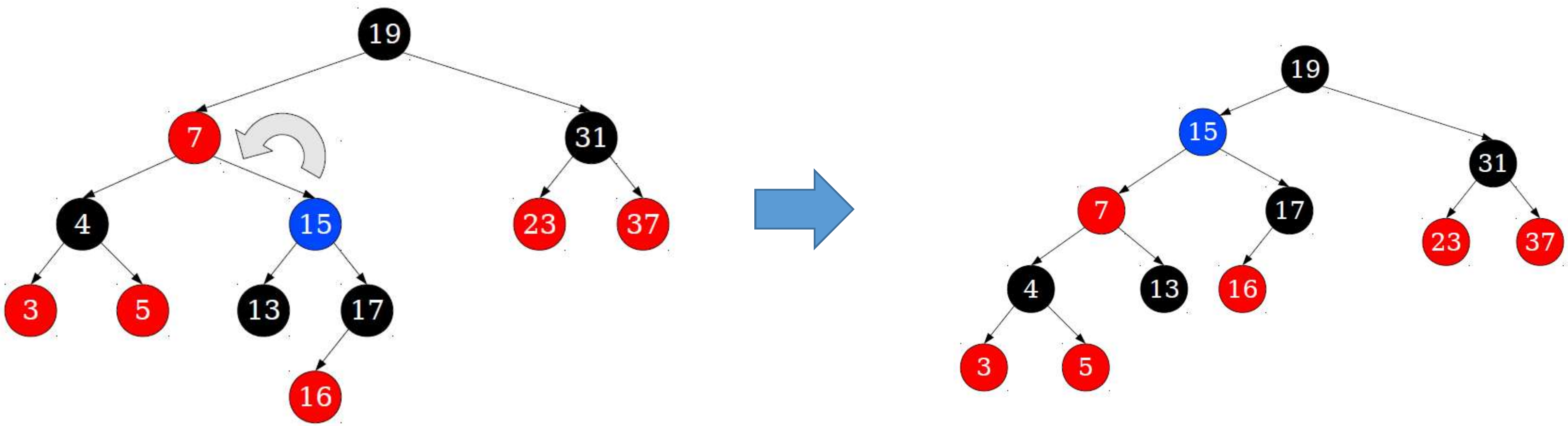
例子2： 父节点是红色，有红兄弟节点



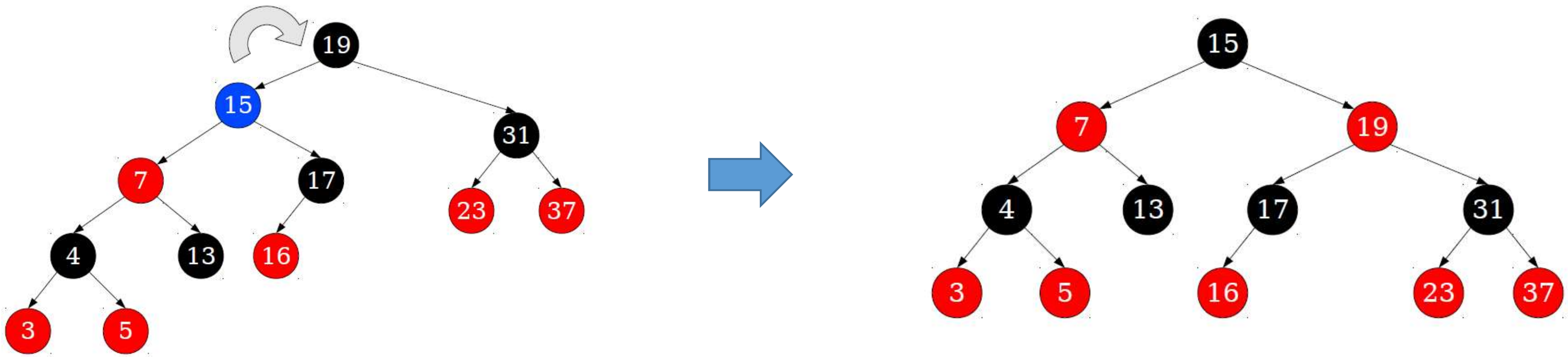
例子2： 父节点是红色，有红兄弟节点



例子2： 父节点是红色，有黑兄弟节点



例子2： 父节点是红色，有黑兄弟节点



插入算法

主要场景是

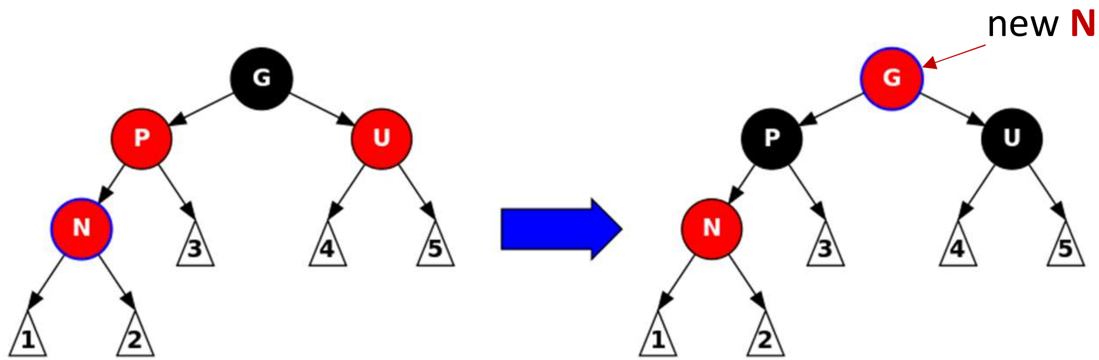
- 插入**红**结点N，但是父节点P也是**红色**

  需要按叔叔结点U的颜色分情况讨论

一般情形（1）

N的叔叔节点是**红色**

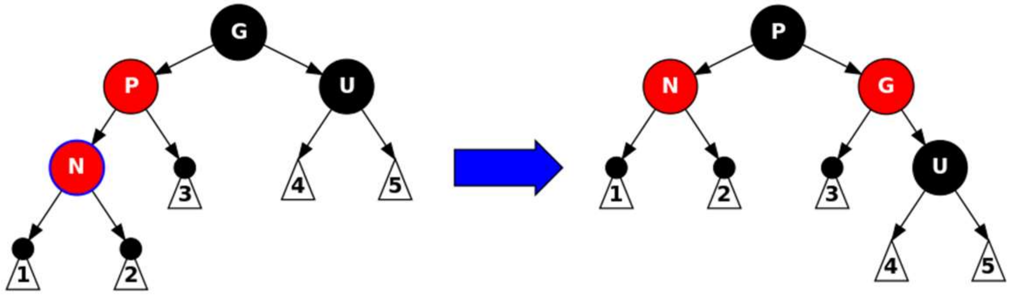
- N是插入红节点
- P是N的父红节点
- U是N的叔叔红节点
- G是N的爷爷黑节点



方案：换颜色，G变成了新的N，
可以看作N的上浮

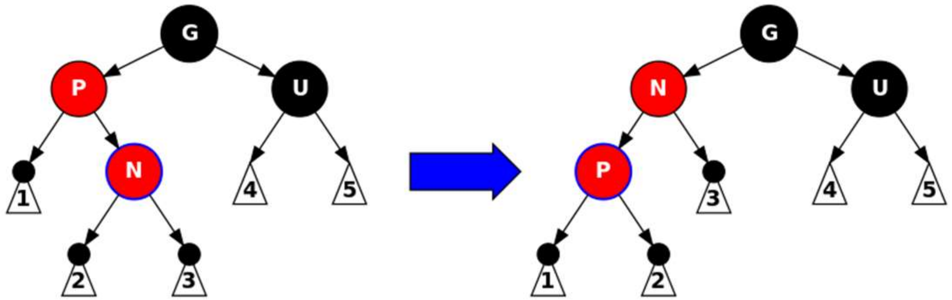
一般情形（2）

1）N的叔叔节点U是**黑色**， N是P的左子节点，P是G的左子节点



方案：绕G右旋转，PG换颜色

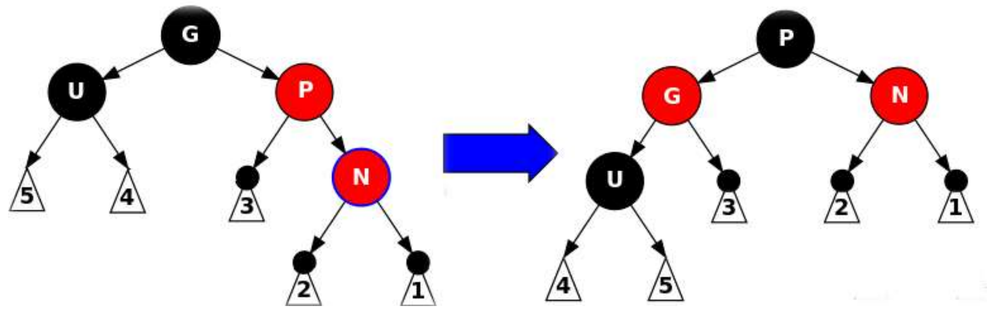
2）N的叔叔节点U是**黑色**， N是P的右子节点，P是G的左子节点



方案：绕P左旋转成上面情形

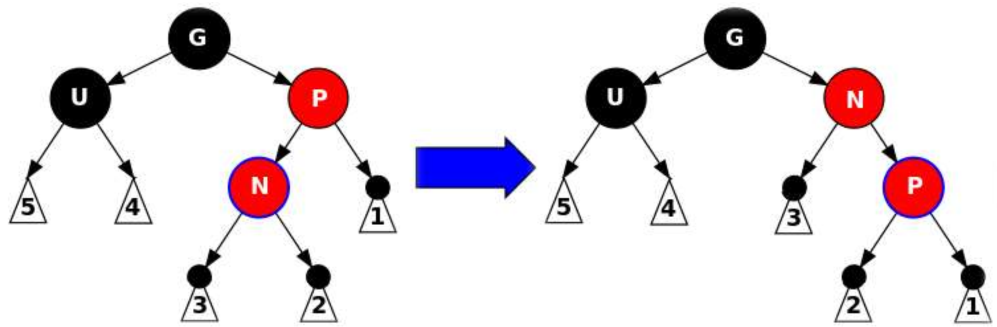
一般情形（3）（情形2的对称）

1）N的叔叔节点U是**黑色**， N是P的右子节点，P是G的右子节点



方案：绕G左旋转，PG换颜色

2）N的叔叔节点U是**黑色**， N是P的左子节点，P是G的右子节点



方案：绕P右旋转成上面情形

插入算法小结

- 总是以红色插入
- 需要解决父节点也是红色的问题
- 叔叔节点是红色时候，相当于上浮
- 叔叔节点是黑色时候，一两次旋转，类似于AVL插入场景

删除算法

- 先按BST删除算法
  - 如果被删节点X有两个子节点，先用后继节点的key替换
  - 然后删除后继节点原来的节点，至多有一个子节点

- 所以考虑最多有一个子节点的删除情形

删除算法：

有一个子节点

- 被删节点X一定是一个黑节点，其子节点C一定是红节点
  方案：只需要用C替代X，并换成黑色

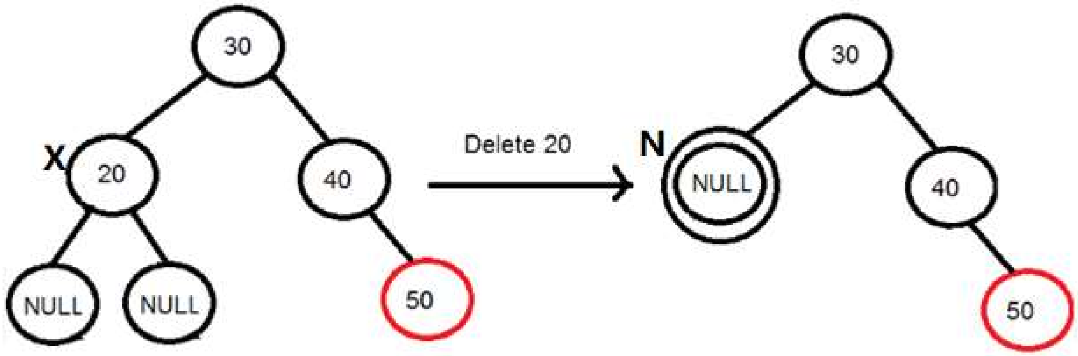
没有子节点

- 被删节点X是红色
  方案：直接删除

- 被删节点X是黑色
  方案：复杂情形，因为这个分支少了一个黑色

Double black notation

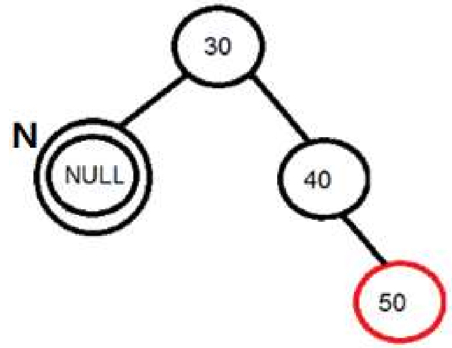
- 当一个黑色节点被删除，并用其黑色子节点代替时候，这个代替的子节点称为double black节点
- 用这个表示这个子树欠了一个黑节点，需要上浮到某个位置补



这里X是黑色节点，被删除后用Double Black空节点N代替

删除算法续

- 起始状态：一个double black空叶节点N
- 分情况讨论，如何从父辈节点那里借
  - N的兄弟节点是红节点
  - N的兄弟节点是黑节点



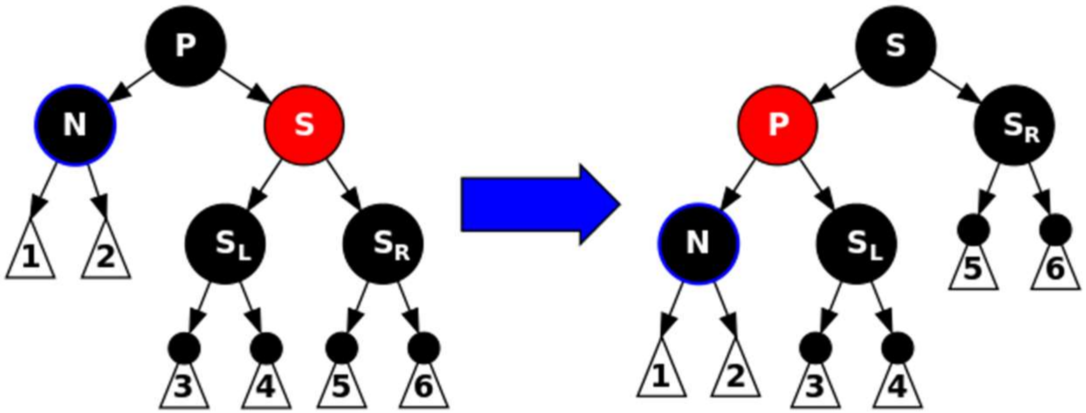
- 注意这里N一定有兄弟节点， 为什么？

- 情况分类按：N的兄弟节点S→S的两个子节点→S的父节点

情形1： N的兄弟节点S是红节点

- 方案：左旋转，SP换颜色
- 转化为兄弟节点为黑节点的情形

- 注意double black节点N用蓝色圈表示

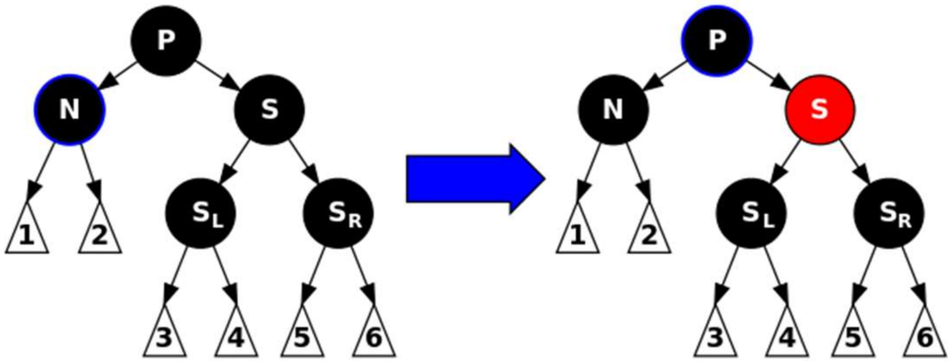


情形2.1： N的兄弟节点S是黑节点， S的两个子节点都是黑色

1）父节点P是**黑色**

方案：S换红颜色，N变成single black，P变成double black节点

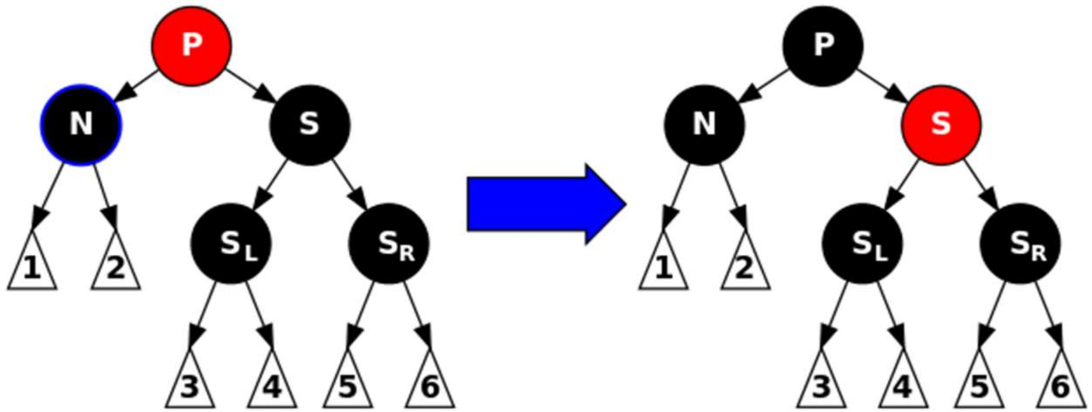
→ 相当于N上浮到P



情形2.2： N的兄弟节点S是黑节点， S的两个子节点都是黑色

2）父节点P是**红色**

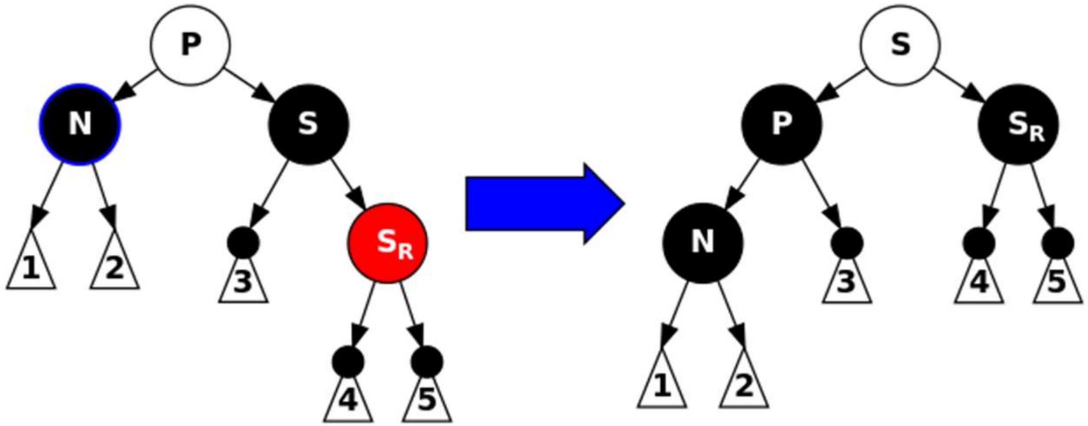
方案：P和S/N换颜色后，N变成single black。因为N缺的一个黑色，被P弥补了（验证第5条性质）



情形3.1: N的兄弟节点S是黑节点， S的两个子节点至少有一个红色

- S的右子节点S_R是**红色**

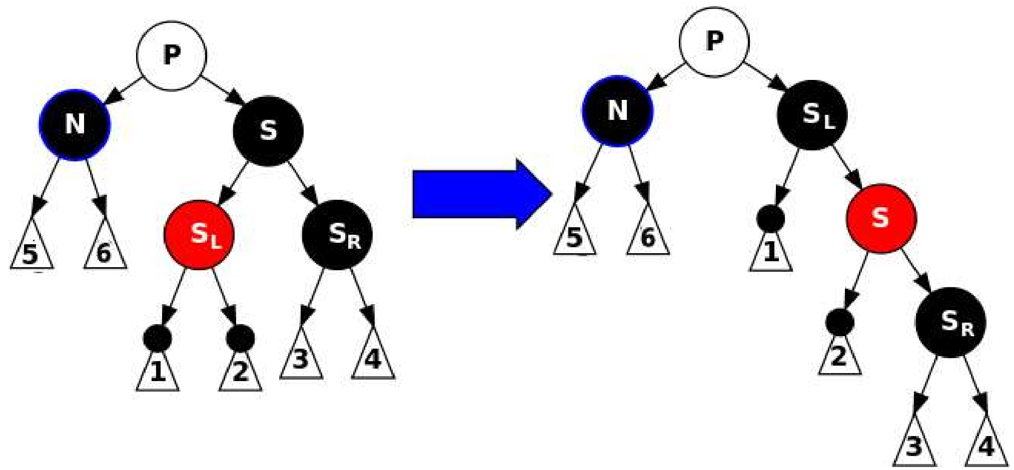
方案：左旋转， S_R换成黑色，3子树直接连到P。
这样N变成single black，因为缺的一个被P弥补



情形3.2: N的兄弟节点S是黑节点， S的两个子节点至少有一个红色

S的右子节点S_R是**黑色**，S_L**红色**

方案：在S子树右旋转，S和S_L换颜色，转换成情形3.1



删除算法小结

- 删除黑色叶节点时候有问题
- 转化为把double black节点展开的问题

1) 兄弟节点是红色时候，通过一次旋转转化为黑兄弟节点情形

2) 黑兄弟节点的两个子节点都是黑色

3) 黑兄弟节点至少有一个子节点是红色

总结

- 红黑树在保持平衡的同时确保动态操作时间复杂度$logn$
- 不过操作比较复杂，又很多种情形要做不同处理
- AVL树的平衡要求更严格，实际插入/删除操作比红黑树要慢，不过查找会快点。

有些问题：

- 怎么记住不同情形的旋转
- 谁发明的红黑树

结论

	Sorted Arrays	Linked Lists	Balanced Binary Search Trees
Search	O(log(n))	O(n)	O(log(n))
Insert/Delete	O(n)	O(1)	O(log(n))

Q&A

Thanks!