14.1 红黑树概述及插入

1. 红黑树概述

1.1 什么是红黑树

- 红黑树(Red-Black Tree，简称R-B Tree)，它一种特殊的二叉查找树。

> 红黑树是特殊的二叉查找树，意味着它满足二叉查找树的特征：任意一个节点所包含的键值，大于等于左孩子的键值，小于等于右孩子的键值。
>
> 除了具备该特性之外，红黑树还包括许多额外的信息。

- 红黑树的特性:

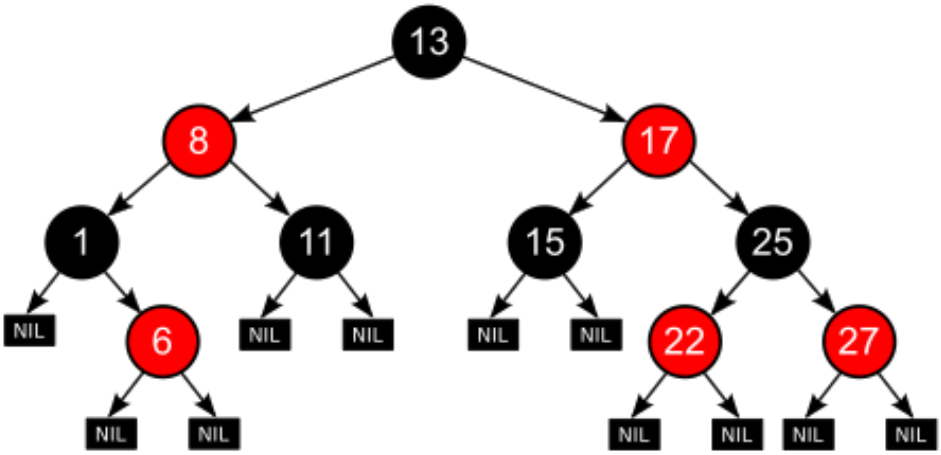
  (1) 每个节点或者是黑色，或者是红色。

  (2) 根节点是黑色。

  (3) 每个叶子节点是黑色。[注意：这里叶子节点，是指为空的叶子节点！]

  (4) 如果一个节点是红色的，则它的子节点必须是黑色的。[不存在两个相邻的红色节点]

  (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。[黑高相同]



1.2 为什么要有红黑树

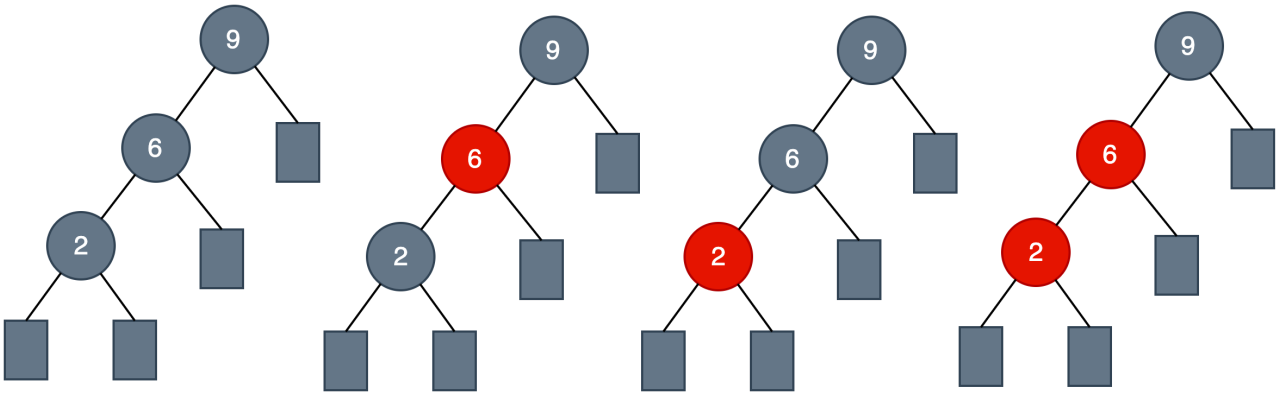
大多数二叉排序树BST的操作（查找、最大值、最小值、插入、删除等等）都是O(h)的时间复杂度，h 为树的高度。但对于斜树而言（BST极端情况下出现），BST的这些操作的时间复杂度将达到O(N)。为了保证BST的所有操作的时间复杂度的上限为O(logN)，就要想办法把一颗BST树的高度一直维持在logN。

红黑树RBT与平衡二叉树AVL比较：

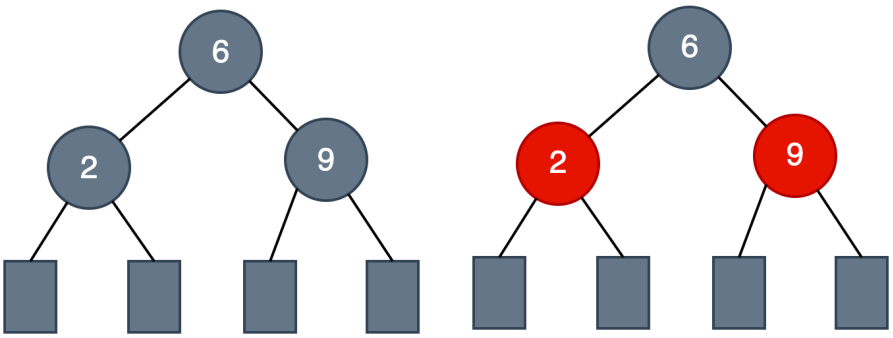
AVL 树比红黑树更加平衡，但AVL树在插入和删除的时候也会存在大量的旋转操作。所以当你的应用涉及到频繁的插入和删除操作，切记放弃AVL树，选择性能更好的红黑树；当然，如果你的应用中涉及的插入和删除操作并不频繁，而是查找操作相对更频繁，那么就优先选择 AVL 树进行实现。

1.3 一颗红黑树是如何保持平衡的

假设有3个节点的序列，那么在红黑树当中，不可能出现3个节点的单链表，如图：



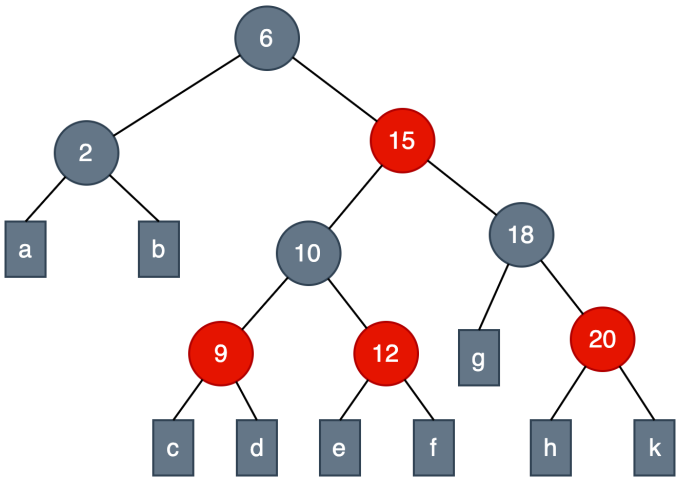
将根结点 9 涂黑色，其他结点分四种情况着色，结果都不满足红黑树的性质要求。唯一的办法就是调整树的高度和染色。



1.4 什么是一颗红黑树的黑高

在一颗红黑树中，从某个结点 x 出发（不包含该结点）到达一个叶结点的任意一条简单路径上包含的黑色结点的数目称为黑高 ， 记为 bh(x) 。

例如下图的节点6的黑高和红色节点15的黑高是一样的，都是2。



计算结点 6 的黑高，从结点 6 到结点 c 的路径是 6→15→10→9→c ，其中黑色结点为 6、10、c ，但是在计算黑高时，并不包含结点本身，所以从结点 6 到结点 c 的路径上的黑色结点个数为 2 ，那么 bh(6)=2 ；从结点 15 到结点 c 的路径为 15→10→9→c ，其中黑色结点为 10、c ，所以从结点 15 到结点 c 的路径上黑色结点数目为 2 ，bh(15)=2 。

红黑树的黑高则为其根结点的黑高。根据红黑树的性质 4、5，一颗红黑树的黑高 bh >= h/2。

> Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.
>
> 从一个结点到其最远的后代叶结点的顶点数目不会超过从该结点到其最近的叶结点的结点数目的两倍。

这句话的意思就是公式 bh >= h/2，其中黑高 bh 就表示从根结点 6 到离它最近的叶结点 2 包含的结点数 2 ，而 h 则表示从根结点 6 到其最远的叶结点 9 所包含的结点数目 4 ，显然这一公式是合理的。

引理：一棵有n个内部结点的红黑树的高度 h <= 2lg(n+1)。

1.5 红黑树的应用

- 大多数自平衡BST(self-balancing BST) 库函数都是用红黑树实现的，比如C++中的map 和 set （或者 Java 中的 TreeSet 和 TreeMap） 。
- 红黑树也用于实现 Linux 操作系统的 CPU 调度。完全公平调度（Completely Fair Scheduler）使用的就是红黑树。
- 红黑树也用于Linux提供的epoll多路复用的底层结构，便于快速增加、删除和查找网络连接的节点。

2. 红黑树的插入操作思路

将一个节点插入到红黑树中，需要执行哪些步骤呢?

2.1 将红黑树当作一颗二叉查找树，将节点插入

红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

那接下来，我们就来想方设法的旋转以及重新着色，使这颗树重新成为红黑树！

2.2 将插入的节点着色为红色

为什么着色成红色，而不是黑色呢？为什么呢？再看红黑树的性质：

> (1) 每个节点或者是黑色，或者是红色。
>
> (2) 根节点是黑色。
>
> (3) 每个叶子节点是黑色。[注意：这里叶子节点，是指为空的叶子节点！]
>
> (4) 如果一个节点是红色的，则它的子节点必须是黑色的。
>
> (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它就又是一颗红黑树了。

如果插入节点是根节点，将该节点转换为黑色。

2.3 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树

第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢?

对于"特性(1)"，显然不会违背了。因为我们已经将它涂成红色了。

对于"特性(2)"，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。

对于"特性(3)"，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。

对于"特性(4)"，是有可能违背的！

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

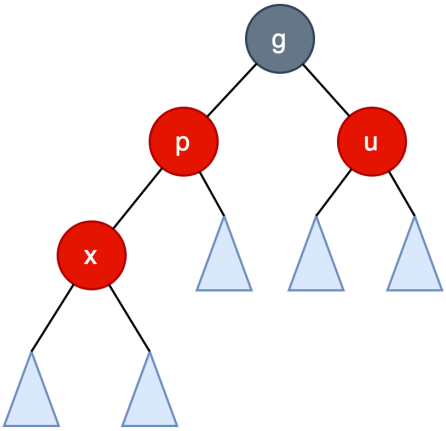
3. 红黑树红红节点的处理

插入操作需要调整的情况，变为了性质4的违背后的调整了，那么，如何进行调整那?

红红节点一定有叔叔节点，调整的核心就是看叔叔节点的颜色，分情况讨论。

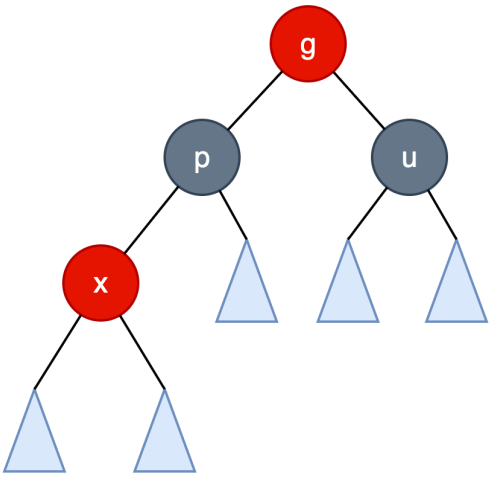
一个是叔叔节点是红色，一个是叔叔节点是黑色。当叔叔节点是黑色时，从LL,LR,RR,RL中进行考虑。

3.1 叔叔节点是红色



第一步：将 p 和 u 的颜色设置为 黑色，第二步：将 g 的颜色设置为红色，第三步：将 x = g，对 g 重复执行算法框架。

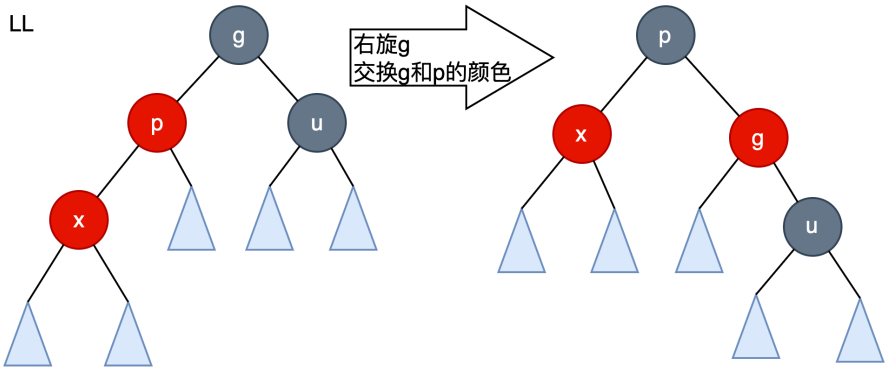
> (01) 将“父节点”设为黑色。
> (02) 将“叔叔节点”设为黑色。
> (03) 将“祖父节点”设为“红色”。
> (04) 将“祖父节点”设为“当前节点”；



如果新的x节点（即g节点）是根节点，将其设置为黑色。

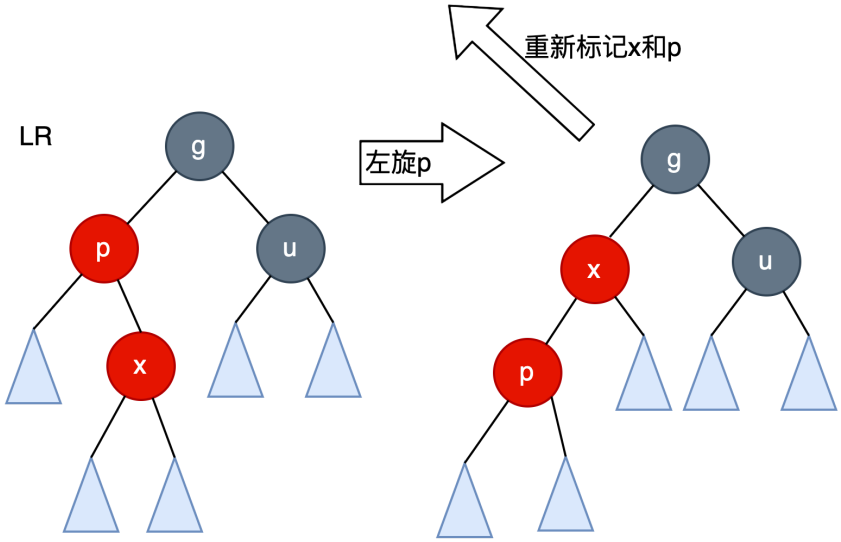
3.2 叔叔节点是黑色

- LL



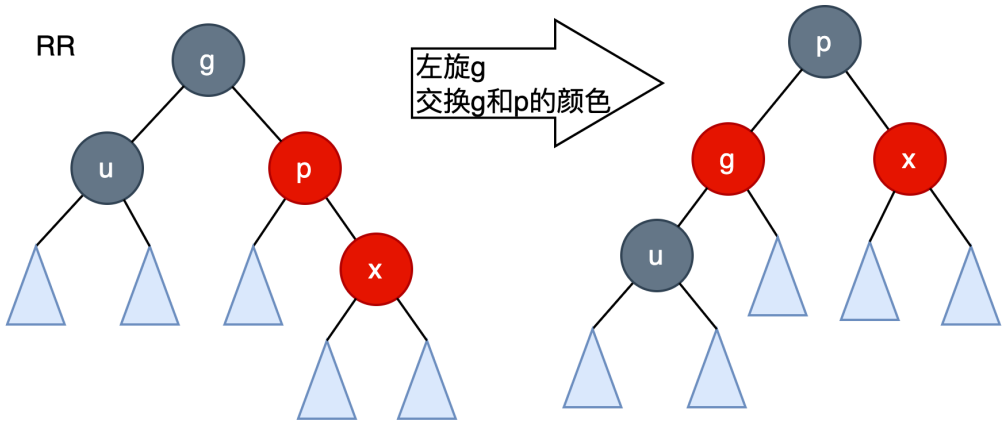
> (01) 将“父节点”设为“黑色”。
>
> (02) 将“祖父节点”设为“红色”。
>
> (03) 以“祖父节点”为支点进行右旋。

- LR



> (01) 左旋p
>
> (02) 重新标记x和p，就成为LL的情况

- RR



> (01) 将“父节点”设为“黑色”。
>
> (02) 将“祖父节点”设为“红色”。
>
> (03) 以“祖父节点”为支点进行左旋。

- RL