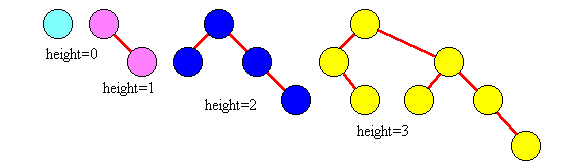
自平衡二叉查找树（Self-Balancing Binary Search Tree）

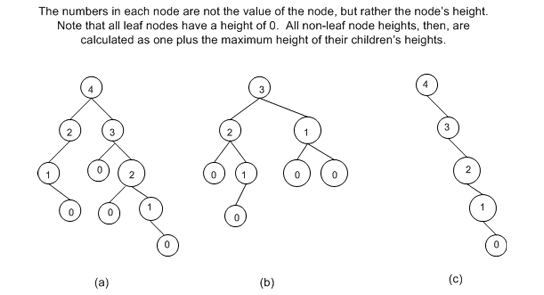
实际上，BST 操作的运行时间与树的高度（Height）是有关系的。一个树的高度指的是从树的根开始所能到达的最长的路径长度。树的高度可被递归性地定义为：

* 如果节点没有子节点，则高度为 0；
* 如果节点只有一个子节点，则高度为该子节点的高度加 1；
* 如果节点有两个子节点，则高度为两个子节点中高度较高的加 1；

计算树的高度要从叶子节点开始，首先将叶子节点的高度置为 0，然后根据上面的规则向上计算父节点的高度。以此类推直到树中所有的节点高度都被标注后，则根节点的高度就是树的高度。



下图显示了几棵已经计算好高度的 BST 树。



**如果树中节点的数量为 n，则一棵满足O(log2n) 渐进运行时间的 BST 树的高度应接近于比 log2n 小的最大整数。**

上图中的三棵 BST 树中，树（b）拥有最好的高度与节点数量的比例。树（b）的高度为 3 ，节点数量为 8，所以 log28 = 3，结果正好与树的高度相等。

树（a）的节点数量为 10，而高度为 4，log210 = 3.3219，比 3.3219 小的最大整数是 3，所以树（a）最理想的高度应该为 3。我们可以通过移动距离最远的节点到中间的某个非叶子节点，以减少数的高度，以使该树的高度与节点数量的比例达到最优。

树（c）的情况是最差的，它的节点数量是 5，所以log25 = 2.3219，则理想高度为 2，但实际上是 4。

实际上我们真正面对的问题是如何保证 BST 的拓扑结构始终保持树高度与节点数量的最佳比例。因为 BST 的拓扑结构与节点的插入顺序息息相关:

1. 一种方式是通过数据的乱序来保证。如果在向树中插入节点前就可以得到数据还好说，而如果我们无法掌控数据的来源呢？比如，数据来自用户的输入，或者来自传感器的实时数据等，基本上要保证数据乱序是没希望了。
2. 另一种方案就是在不试图让数据源决定数据顺序的情况下，新的节点插入后仍然可以保持 BST 树的平衡（balanced）。这种能够始终维持树平衡状态的数据结构称为自平衡二叉查找树（self-balancing binary search tree）。

一棵平衡树指的是树能够保持其高度与广度保持预先定义的比例。不同的数据结构可以定义不同的比例以保持平衡，但所有的比例都趋向于log2n。那么，一颗自平衡的 BST 也同样呈现出 O(log2n) 的渐进运行时间。

有许多种不同的自平衡 BST 数据结构，例如 AVL 树、红黑树（Red-Black Tree）、2-3 树、2-3-4 树、伸展树（Splay Tree）、B 树等等。本文中我们将简要介绍其中的两种：AVL 树和红黑树。

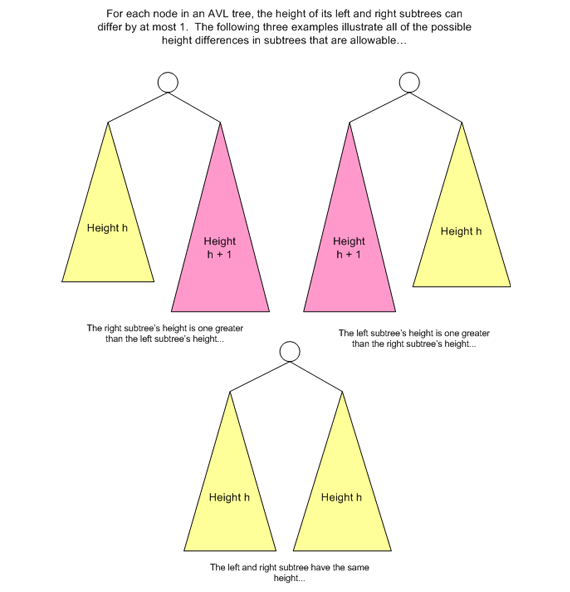
AVL 树

在 1962 年，俄罗斯数学家 G. M. Andel'son-Vel-skii 和 E. M. Landis 发明了第一种自平衡二叉查找树，叫做 AVL 树。AVL 树必须维持如下平衡条件，对每个节点 n：

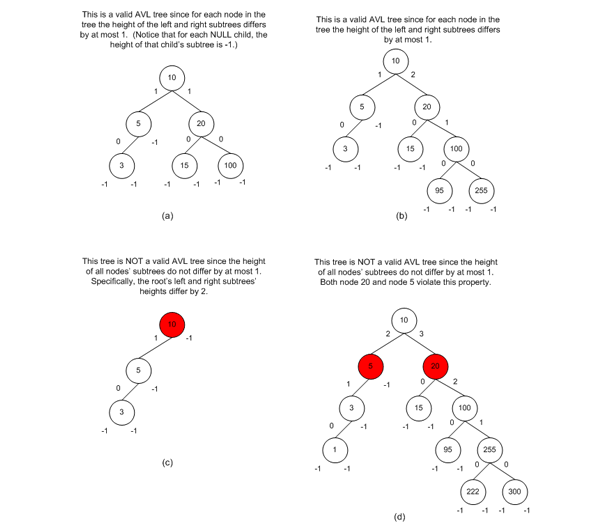
* 节点 n 的左子树的高度与右子树的高度的差至多是 1。

节点的左子树或者右子树的高度可以通过上面描述的步骤来计算。如果节点仅有一个子节点，则无子节点侧的高度为 -1。

下图展示了概念上 AVL 树节点的两侧子树高度需要保持的关系。



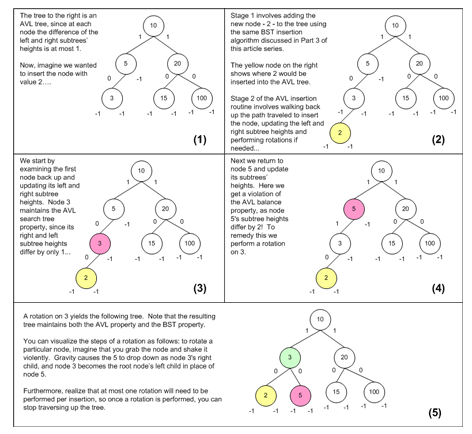
下面是一些 BST 树。节点中的数字代表着节点的值，左右两侧的数字代表着左右子树的高度。其中树（a）和树（b）是合法的 AVL 树，而树（c）和树（d）则不合法，因为树中不是所有的节点都满足 AVL 的平衡性质要求。



当创建一棵 AVL 树时，难点在于如何保证 AVL 的平衡性质要求，而不用关注对树的具体操作。也就是说，无论是向树添加节点还是删除节点，最重要的事情就是保持树的平衡。AVL 树通过 "旋转操作（rotations）" 来保持树的平衡。旋转操作可以重塑树的拓扑结构来恢复树的平衡，更重要的是，重塑后的树依然符合二叉查找树的性质要求。

当向一棵 AVL 树中插入一个新的节点时，需要经过两阶段的过程。首先，插入新节点的操作将使用与向 BST 树中插入新节点时使用的相同的查找算法。新的节点将做为一个叶子节点被添加到树中合适的位置，以满足 BST 的性质要求。在添加完节点后，将导致树的结构可能已经违背 AVL 树的性质要求。所以在第二个阶段中，将遍历访问路径，来检查每个节点左右子树高度。如果存在某节点的左右子树的高度差大于 1 时，则需要使用旋转操作来处理。

下图阐述了对节点 3 进行旋转操作的步骤。在阶段一插入新节点 2 后，在节点 5 处的 AVL 树的平衡性质已经被破坏，因为节点 5 的左右子树的差为 2，大于 AVL 树要求的 1。为了解决这个问题，需要在节点 5 的左子树的根节点，也就是节点 3 处做旋转操作。这个旋转操作不仅恢复了 AVL 树的平衡要求，而且也保持了 BST 的性质要求。

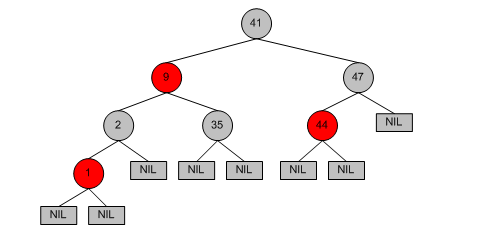


有时除了像上图中描述的简单的旋转操作之外，可能还需要进行多次旋转操作。对于成组的旋转操作的深入讨论已经超出了本篇文章的范畴，这里就不再赘述了。最重要的就是要意识到插入操作和删除操作都会破坏 AVL 树的平衡，而旋转操作就是解决这些问题的法宝。

通过确保所有节点的左右子树的差小于等于 1，AVL 树保证了插入、删除和查找操作将始终保持 O(log2n) 的渐进运行时间，而与插入或删除节点的顺序无关。

红黑树（Red-Black Tree）

在 1972 年，慕尼黑理工大学（Technical University of Munich）的计算机科学家 Rudolf Bayer 创造了红黑树（Red-Black Tree）数据结构。除了包含数据和左右孩子节点之外，红黑树的节点还包含了一项特别的信息 -- 颜色。这个颜色只包含两种颜色，即红色和黑色。并且，红黑树还添加了一种特殊类型的节点，称为 NIL 节点。NIL 节点将做为红黑树的伪叶子节点出现。也就是说，所有带有关键数据的节点称为内节点，而所有其他的外节点则均指向 NIL 节点。这个概念可能理解起来有些费劲，希望下面这张图有所帮助。



红黑树（R-B Tree）需要满足如下性质：

* 节点的颜色只能是红色或者黑色；
* 根节点是黑色的；（根性质）
* NIL 节点的颜色是黑色；
* 如果节点的颜色是红色，则其子节点均为黑色；（红性质）
* 从任一节点到其后代任一叶子节点的路径上的黑色节点的数量相同；（黑性质）

前面几条性质都很好解释，只有最后一条最难理解。简单的说，从树中任意一个节点开始，从该节点到其后代的任意一个 NIL 节点的路径上的黑色节点的数量必须相同。比如上图中，以根节点为例，从节点 41 到任意一个 NIL 节点的路径上，黑色节点的数量都是相同的，也就是 3 个。如从节点 41 到左下角的 NIL 节点的路径上，黑色节点包括 41, 2, NIL，所以黑色节点数量是 3 个。

类似于 AVL 树，红黑树也是一种自平衡二叉查找树。AVL 树的平衡性质是通过限制节点的左右子树的高度来达成，而红黑树则是通过更形象化的方式来保证树的平衡。如果一棵树满足红黑树的性质，其节点的总数量为 n，则它的高度将始终小于 2 \* log2(n+1) 。鉴于这个原因，致使红黑树保证了对树的所有操作都能在 O(log2n) 渐进运行时间范围内。

同样是和 AVL 树一样，当对红黑树进行节点的插入和删除时，最终要的就是使其仍然符合红黑树的性质。AVL 树通过使用旋转操作（rotations）来恢复树的平衡。而红黑树则是通过重新着色（recoloring）和旋转两种操作共同来完成。这不仅需要判断节点的父节点的颜色，还需要对比叔父节点的颜色，使得红黑树的恢复过程变得更加复杂。

向红黑树中插入新的节点时，需要考虑很多种情况。假设已存在红黑树 T，即将被插入的新节点为 K。

首先一种特殊情况就是如果树 T 为空，则可直接将节点 K 设置为根节点，并且将颜色标为黑色，这样即可满足 R-B 树的所有要求。

如果树 T 不为空，则需要遵循如下步骤：

1.使用 BST 插入算法将节点 K 插入到树 T 中；

2.将节点 K 着色为红色；

3.如果需要，则重塑 R-B 树的性质；

我们知道 BST 树总是将新节点添加为叶子节点，所以将节点 K 插入到树 T 中不会破坏根性质。而添加一个红色的叶子节点也不会影响树 T 的黑性质。实际上，添加一个红色的叶子节点仅有可能影响树 T 的红性质，所以我们仅需检查树的红性质，如果红性质被违背，则需要重塑树结构以重新满足红黑树性质。

我们将节点 K 的父节点称为节点 P（parent node），将节点 P 的父节点称为节点 G（grandparent node），将节点 P 的兄弟节点称为节点 S（sibling node）。

当向非空树 T 中插入节点 K 时，将直接受到父节点 P 的颜色的影响，可能会遇到如下多种情况。

情况1：节点 P 是黑色。

如果 P 为黑色，而节点 K 为红色，所以实际上不会违背红性质，则树 T 已经满足所有红黑树性质条件。

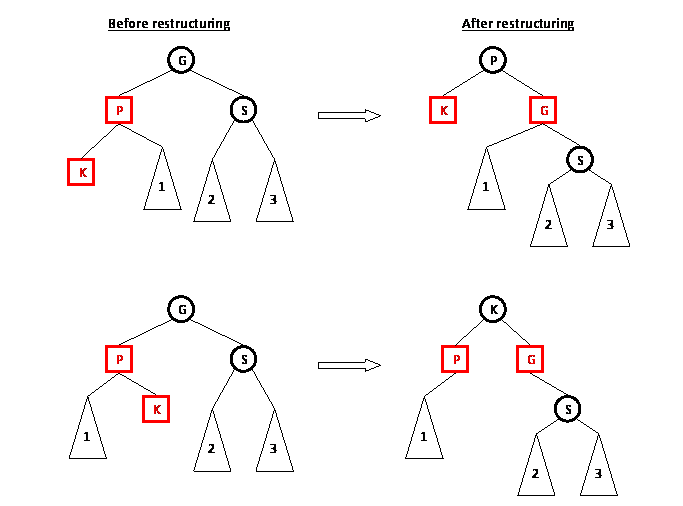
情况2：节点 P 是红色。

如果节点 P 为红色，那么 P 现在有了新的子节点 K，并且 K 也为红色，所以已经违背了红性质。为了处理这种两个红节点的情况，我们需要考虑节点 G 的其他子节点，也就是节点 P 的兄弟节点 S。此时，会有两种情况发生：

情况 2a：节点 S 是黑色或者为空。

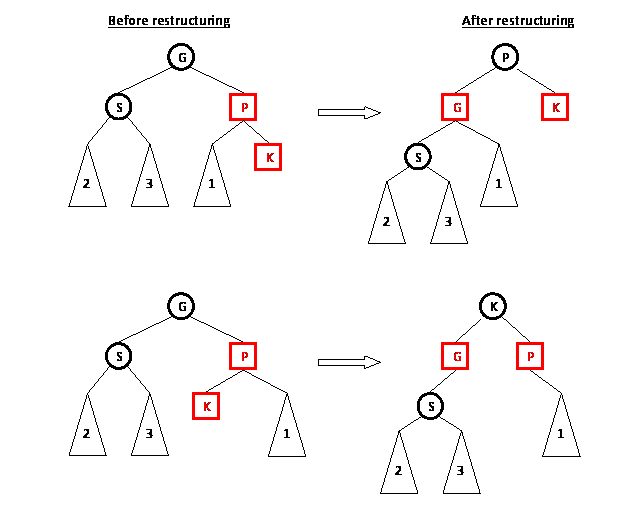
如果节点 S 是黑色或者为空，则需要对节点 K、P、G 进行旋转。根据 K、P、G 顺序的不同，旋转操作可能存在四种可能性。

前两种可能性为当 P 为 G 的左子节点时。



如果 S 为空，则上图中直接将 S 删除即可。

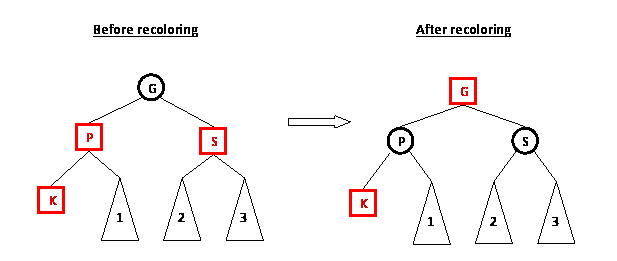
另两种可能性为当 P 为 G 的右子节点时，正好与上面图中的过程相反。



旋转操作过程结束后，双红节点情况已经被合理的解决了。

情况 2b：节点 S 是红色。

如果 P 的兄弟节点 S 是红色，则需要对 P、S、G 进行重新着色：将 P 和 S 着色为黑色，将 G 着色为红色。



重新着色操作不会影响树 T 的黑性质，因为当 P、G 的颜色更改时，所有路径上的黑色节点数量并没有改变。但是，重新着色可能会使 G 和 G 的父节点产生双红情况。这种情况下，则需要从 G 和 G 的父节点开始继续遵循处理 K 和 K 的父节点的方式递归式地解决双红问题。