平衡二叉查找树

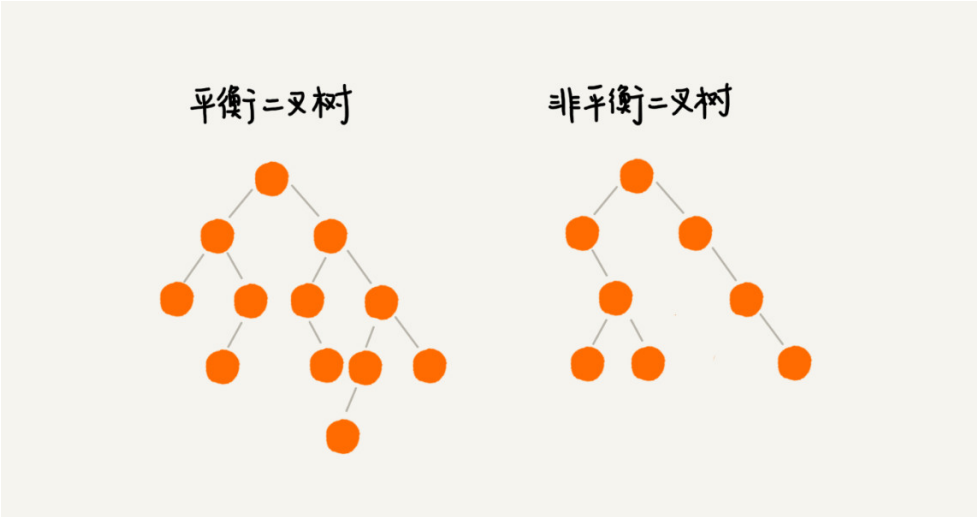
二叉查找树在频繁的动态更新过程中，可能会出现树的高度远大于 $\log_2 n$ 的情况，从而导致各个操作的效率下降。极端情况下，二叉树会退化为链表，时间复杂度会退化到 O(n)。为了解决这个复杂度退化的问题，平衡二叉查找树就应运而生了。

严格定义

平衡二叉树：

二叉树中任意一个节点的左右子树的高度相差不能大于 1。

完全二叉树、满二叉树其实都是平衡二叉树，但是非完全二叉树也有可能是平衡二叉树。



平衡二叉查找树：

不仅满足上面平衡二叉树的定义，还要满足二叉查找树的特点。

最先被发明的是"AVL树"，严格符合平衡二叉查找树的定义，是一种高度平衡的二叉查找树。

但是很多平衡二叉查找树其实并没有严格符合平衡二叉树的定义，比如：红黑树，它从根节点到各个叶子节点的最长路径，有可能会比最短路径大一倍。

平衡二叉查找树中"平衡"的意思，就是让整棵树左右看起来比较"对称""平衡"，不要出现左子树很高、右子树很矮的情况。这样就能让整棵树的高度相对来说低一些，相应的插入、删除、查找等操作的效率高一些。并不需要严格符合定义。

2-3树

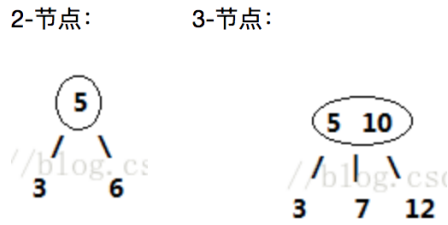
二叉查找树的变种，红黑树背后的逻辑是基于2-3树的，理解好2-3树对理解红黑树有很大作用。

- 节点含义

  树中的2和3代表两种节点。

2-节点即普通节点：包含一个元素，两条子链接。

3-节点则是扩充版，包含2个元素和三条链接：两个元素A、B，左边的链接指向小于A的节点，中间的链接指向介于A、B值之间的节点，右边的链接指向大于B的节点。



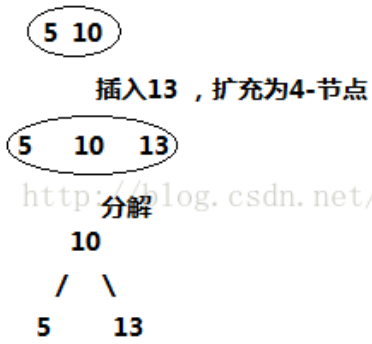
在这两种节点的配合下，**2-3树可以保证在插入值过程中，任意叶子节点到根节点的距离都是相同的**。完全实现了矮胖矮胖的目标。

- 构造过程

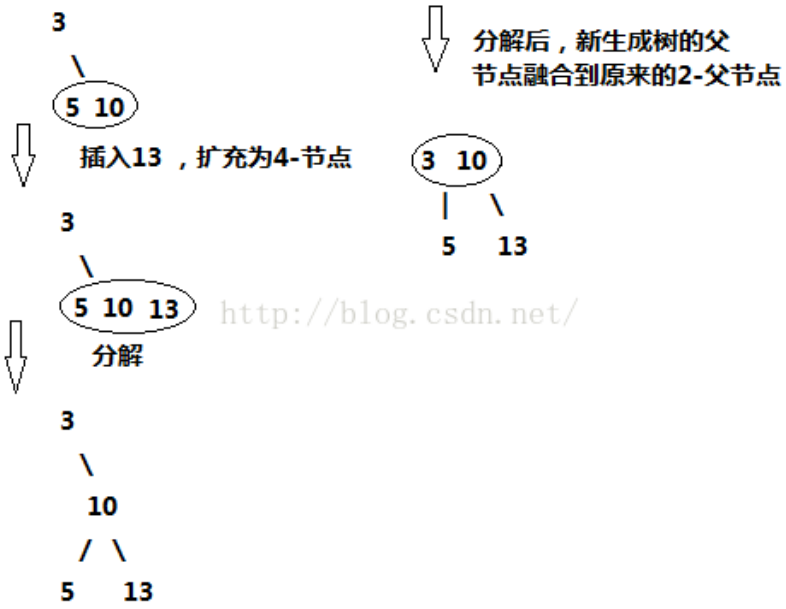
  如果将值插入一个2-节点，则将2-节点扩充为一个3-节点。

  如果将值插入一个3-节点，分为以下几种情况。

  1. 3-节点没有父节点，即整棵树就只有它一个三节点。此时，将3-节点扩充为一个4-节点，即包含三个元素的节点，然后将其分解，变成一棵二叉树。此时二叉树依然保持平衡。

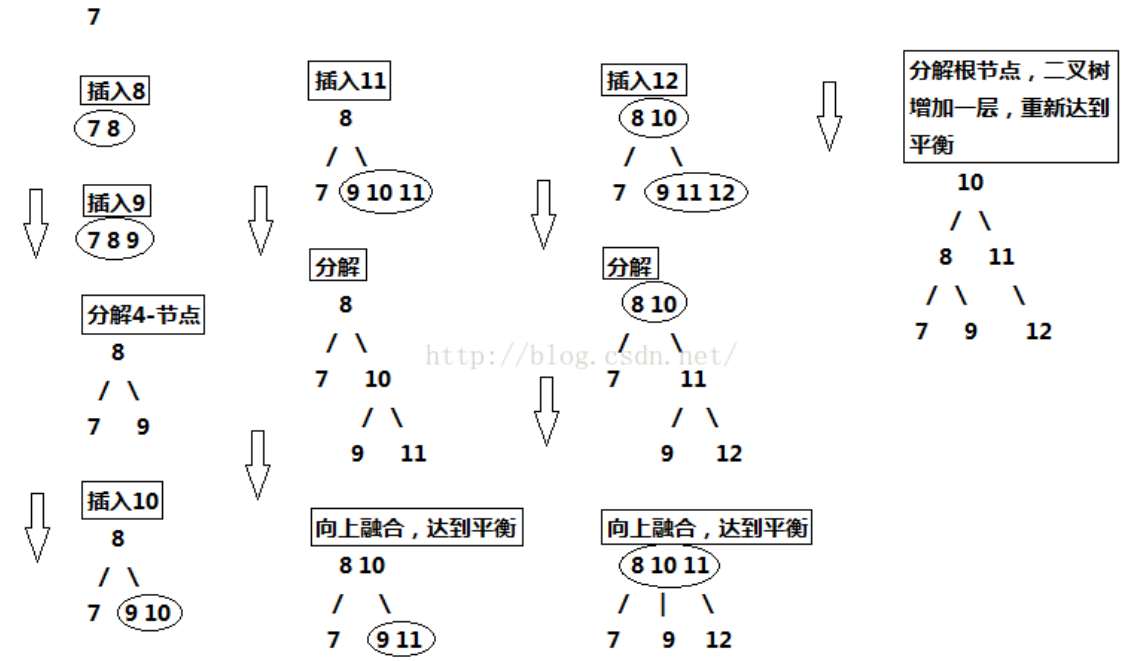


  2. 3-节点有一个2-节点的父节点，此时的操作是，3-节点扩充为4-节点，然后分解4-节点，然后将分解后的新树的父节点融入到2-节点的父节点中去。



3. 3-节点有一个3-节点的父节点，此时操作是：3-节点扩充为4-节点，然后分解4-节点，新树父节点向上融合，上面的3-节点继续扩充，融合，分解，新树继续向上融合，直到父节点为2-节点为止，如果向上到根节点都是3-节点，将根节点扩充为4-节点，然后分解为新树，至此，整个树增加一层，仍然保持平衡。

   如：将{7,8,9,10,11,12}中的数值依次插入2-3树



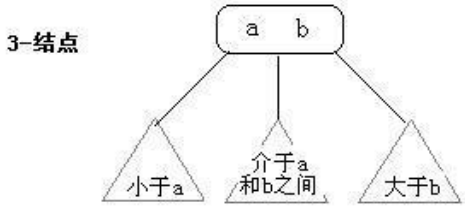
2-3树的设计完全可以保证二叉树保持矮矮胖胖的状态，保持其性能良好。但是，将这种直白的表述写成代码实现起来并不方便，因为要处理的情况太多。需要维护两种不同类型的节点，将链接和其他信息从一个节点复制到另一个节点，将节点从一种类型转换为另一种类型等等。因此，红黑树出现了，由于用红黑作为标记这个小技巧，最后实现的代码量并不大。

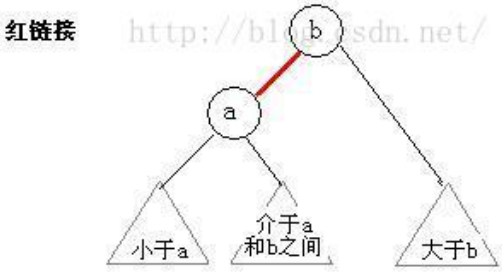
- 红黑树和2-3树的关联

  1. 红和黑的含义：

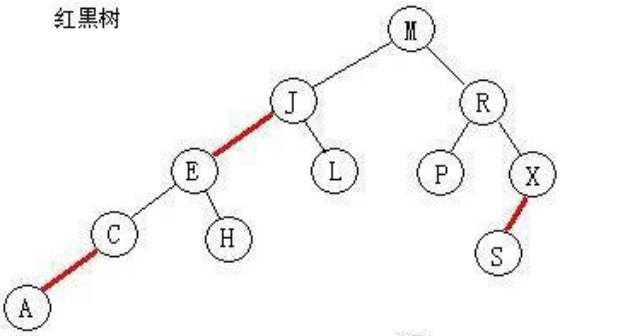
     红黑树中，所有的节点都是标准的2-节点，为了体现出3-节点，这里将3-节点的两个元素用左斜红色的链接连接起来，即连接了两个2-节点来表示一个3-节点。这里**红色节点标记就代表指向其的链接是红链接，黑色标记的节点就是普通的节点**。

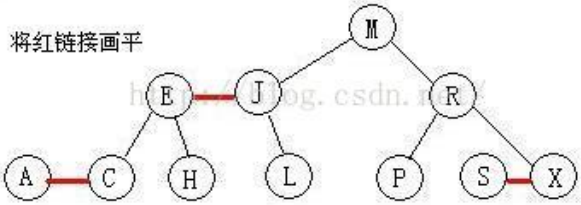
     所以才会有那样一条定义，叫"从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点"，因为红色节点是可以与其父节点合并为一个3-节点的，红黑树实现的其实是一个**完美的黑色平衡，**如果你将红黑树中所有的红色链接放平，那么它所有的叶子节点到根节点的距离都是相同的。所以它并不是一个严格的平衡二叉树，但是它的综合性能已经很优秀了。

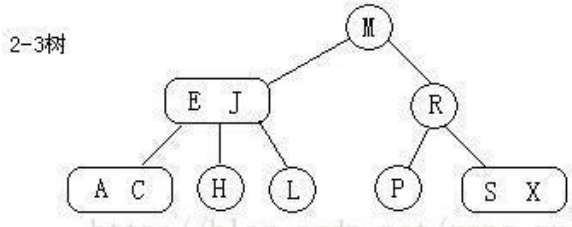




红链接只能是左链接，且由于a小于b，故b在上 为a的父结点







2. 所以，红黑树的另一种定义是满足下列条件的二叉查找树：

   红链接均为左链接。（根据资源显示左连接右链接，两种都是可用的，这种左倾红黑树考虑情况少一些，实现会比较简单，当然旋转的次数也会多一些）

   没有任何一个结点同时和两条红链接相连。(这样会出现4-节点)。

   该树是完美黑色平衡的，即任意空链接到根结点的路径上的黑链接数量相同。

3. 旋转

旋转是一项非常重要的操作。我们在不改变树的有序性的情况下，将某个红链接从左链接变成右链接，或者从右链接变成左链接，这在处理一些情况比如对应于2-3树中向3节点插入元素的时候，更新整个树是很有用的。

比如：在2-3树中向2节点插入非常简单，直接合并成一个3节点就行。但是红黑树具体实现时，因为相对于父节点可能有大有小，那么在插入的时候就可能在父节点的左边或者右边，而红链接只能是左链接，那么当在右边插入的时候，就需要进行旋转操作将右链接变成左链接。

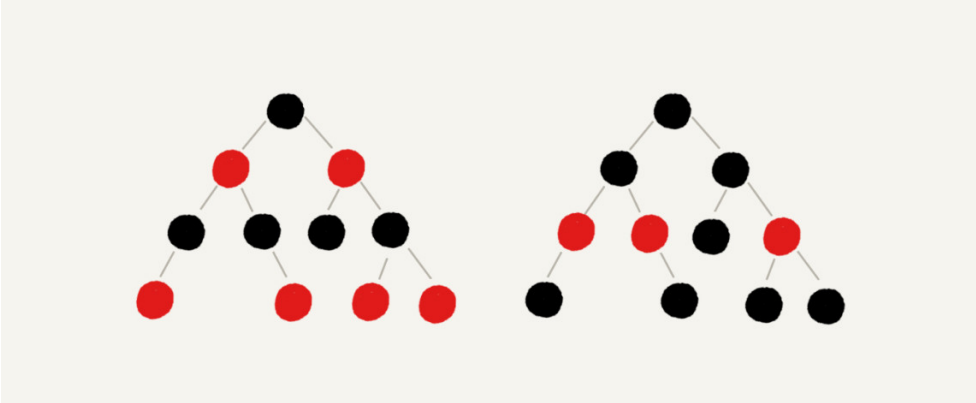
红黑树

- 定义

  红黑树中的节点，一类被标记为黑色，一类被标记为红色。要求：

  1. 根节点是黑色的；
  2. 每个叶子节点都是黑色的空节点（NIL），也就是说，叶子节点不存储数据；

     主要是为了简化红黑树的代码实现而设置的。
  3. 任何相邻的节点都不能同时为红色，也就是说，红色节点是被黑色节点隔开的；
  4. 每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点；



- 为什么工程中喜欢用红黑树这种平衡二叉查找树?

  红黑树是近似平衡的二叉树。

  （“平衡”的意思可以等价为性能不退化。“近似平衡”就等价为性能不会退化得太严重。）

  Treap、Splay Tree，绝大部分情况下，它们操作的效率都很高，但是也无法避免极端情况下时间复杂度的退化。尽管这种情况出现的概率不大，但是对于单次操作时间非常敏感的场景来说，它们并不适用。

  AVL 树是一种高度平衡的二叉树，所以查找的效率非常高，但是，AVL 树为了维持这种高度的平衡，就要付出更多的代价。每次插入、删除都要做调整。所以，对于有频繁的插入、删除操作的数据集合，使用 AVL 树的代价就有点高了。

  红黑树只是做到了近似平衡，并不是严格的平衡，所以在维护平衡的成本上，要比 AVL 树要低。红黑树的插入、删除、查找各种操作性能都比较稳定。红黑树的高度只比高度平衡的 AVL 树的高度（$\log_2 n$）大了一倍，在性能上，下降得并不多，时间复杂度$\log_2 n$。对于工程应用来说，要面对各种异常情况，为了支撑这种工业级的应用，我们更倾向于这种性能稳定的平衡二叉查找树。

- 实现思路

调整的过程包含两种基础的操作：左右旋转和改变颜色。

（注意，2-3树是左倾树，红节点都在左边，但是红黑树还有其他实现方式，比如：2-3-4树，允许右倾等）

插入过程：

红黑树规定插入的节点都必须是红节点，可以根据"二叉查找树新插入节点都在子节点"和"2-3插入过程"比较容易理解这项规则。

对照2-3树，理解树的转换流程：