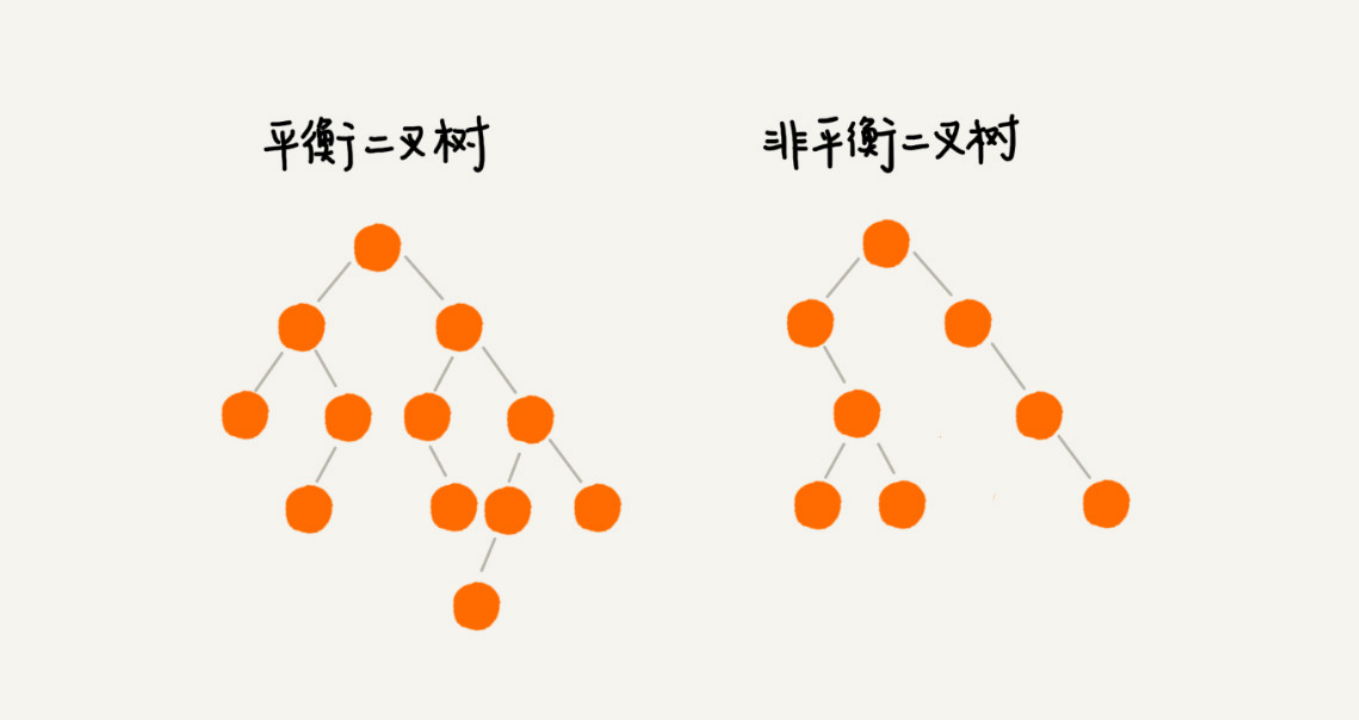
红黑树上：为什么工程中都用红黑树这种二叉树

上两节我们依次讲解了，树，二叉树，二叉查找树，理想情况下，时间复杂度是O(logn)。

为什么工程中都喜欢用红黑树，而不是其他平衡二叉查找树呢?

什么是“平衡二叉查找树”?

平衡二叉树的严格定义是这样的，二叉树中任意一个节点的左右子树高度相差不能大于1 。



平衡二叉查找树不仅满足平衡二叉树的条件，还满足二叉查找树的特点。最先被方面的平衡二叉查找树是AVL树。它严格复核相关定义。

但是很多平衡二叉查找树其实并没有严格复核上面的定义，比如下面说的红黑树，他从根节点到叶子节点的最长路径，有可能比最短路径大一倍。

平衡二叉查找树中，平衡的意思，其实就是让整棵树左右看起来比较对称。比较平衡，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。

如何定义一颗“红黑树”?

平衡二叉树有很多，比如，Splay Tree（伸展树）、Treap（树堆)等，但是我们提到的平衡二叉查找树，听到的基本都是红黑树，有时候甚至默认平衡二叉查找树就是红黑树。

红黑树，英文“Red-Black Tree”，简称R-B Tree。它是一种不严格的平衡二叉查找树。

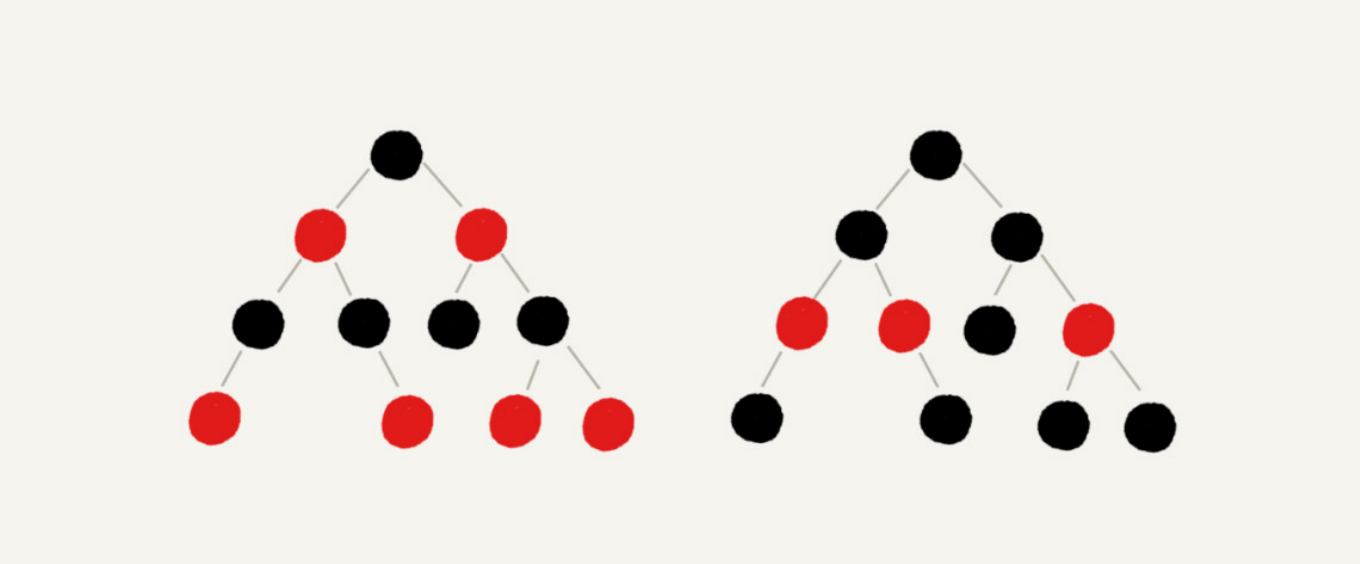
红黑树定义： 红黑树中节点，一类被标记为黑色，一类被标记为红色。除此之外还需要满足这样几个要求：

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；

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，从该结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；

这里的第二点要求“叶子节点都是黑色的空节点”，稍微有些奇怪，他主要是为了简化红黑树的代码实现而设置的。（这一节的画图和讲解的时候，就将黑色的，空的叶子节点都省略掉了）

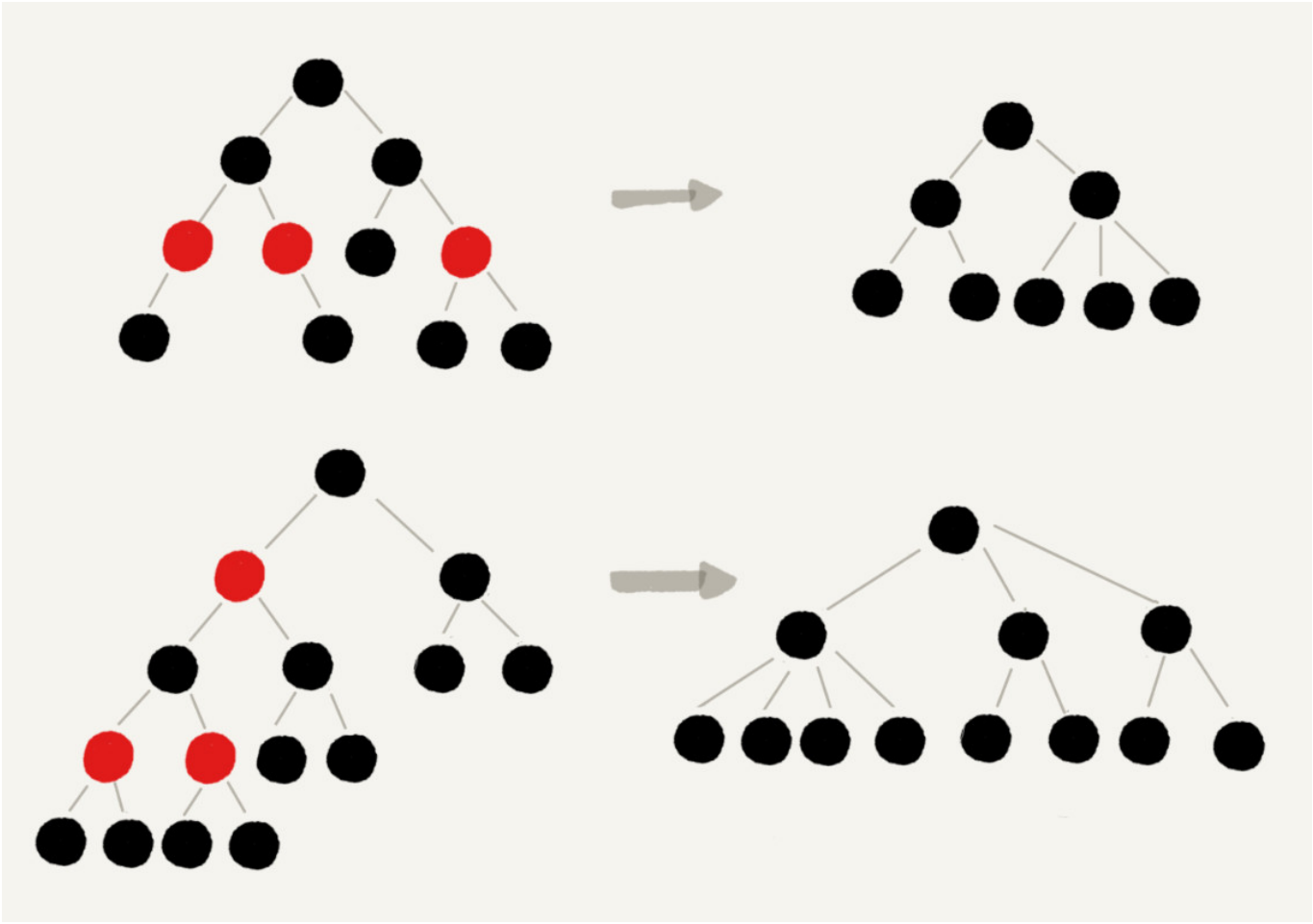
如图：



为什么说红黑树是近似平衡的呢？

平衡二叉查找树的初衷，是为了解决二叉查找树因为动态更新导致的性能退化问题。**“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的很严重。**

二叉查找树很多性能都跟树的高度成正比。一颗机器平衡的二叉树的高度大约是logn，所以如果证明红黑树近似平衡，只需分析，红黑树的高度是否比较稳定的趋近于logn就好了。



如图: 首先，如果我们将红色节点从红黑树中去掉，那么单纯包含黑色节点的红黑树的高度是多少呢？ 红色节点删除，有些节点没有父节点了，他们会直接拿这些节点的祖父节点作为父节点。所以之前的二叉树就变成了四叉树。

之前的定义中有一条：从任意节点到可达的叶子节点的每个路径中包含相同数目的黑色节点，我们从四叉树中取出某些节点，放到叶子节点的位置，四叉树就变成了完全二叉树，所以，仅包含黑色节点的四叉树的高度，比包含相同节点个数的完全二叉树高度要小。

我们现在知道只包含黑色节点的黑树的高度，那我们把红色节点加回去，高度会变成多少呢？

红黑树中红色节点不能相邻，也就是说，有一个红色节点就至少有一个黑色节点，将他与其他红色节点隔开，红黑树中包含最多黑色节点的路径不会超过logn，所以加入红色节点之后，最长路径不会超过2logn，也就是说，红黑树的高度近似2logn。

所以红黑树的高度之比高度平衡的AVL树仅仅大一倍。在性能中下降并不多。

解答开篇

为什么工程中大家都喜欢用红黑树呢？ 我们前面提到Treap、Splay Tree，绝大多数情况下，他们操作的效率都很高，但是极端情况下，无法避免时间复杂度的退化。尽管这种情况出现的概率不大，但是对于单次操作时间免肝的场景，他们并不适用。

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，有利就有弊，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，就比较复杂，耗时，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。所以，红黑树的插入、删除、查找各种操作性能都比较稳定。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。

内容小结

红黑树的重点不应该所有的学习侧重点放到他的实现上。 **我们学习数据结构和算法，要学习它的由来、特性、适用场景以及它能解决的问题。对于红黑树，也不例外。你如果能搞懂这几个问题，其实就已经足够了**

红黑树是一种平衡二叉查找树，它是为了解决普通二叉查找树在数据更新的过程中，复杂度退化的问题而产生的。红黑树的高度近似logn，所以，他是近似平衡，插入、删除、查找操作的时间复杂度都是O(logn) 。

课后思考

动态数据结构支撑动态的数据插入、删除、查找操作，处理红黑树，我们前面还学习过哪些呢？对比一下各自的优势、劣势，以及应用场景？

散列表：插入删除查找O(1),是最常用的，但其缺点是不能顺序遍历以及扩容缩容的性能损耗。适用于哪些不需要顺序遍历，数据更新不那么频繁的。

跳表：插入删除查找都是O(logn)，并且能顺序遍历。确定是空间复杂度O(n),适用于不那么在意内存空间的，其顺序遍历和区间查找非常方便。

红黑树：插入删除查找都是O(logn)，中序遍历就是顺序遍历，稳定。缺点是难以实现，去查找不方便，其实跳表更加，但红黑树已经用于很多地方了。