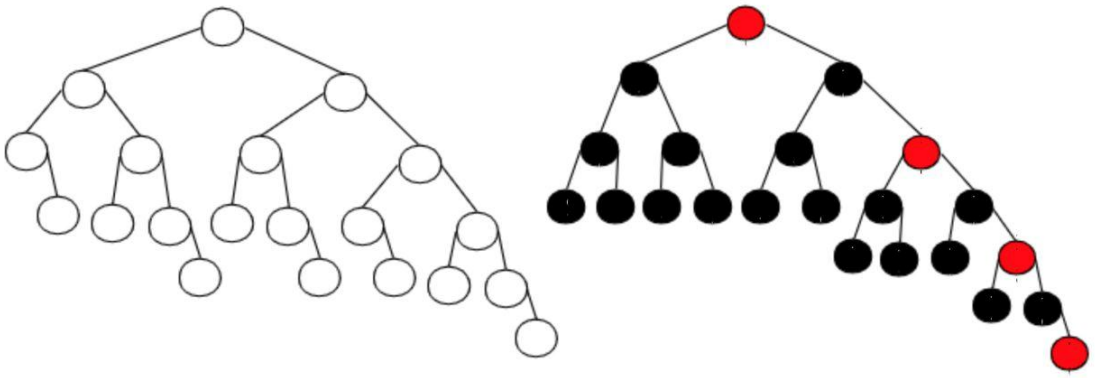


关于AVL树和红黑树的一点看法

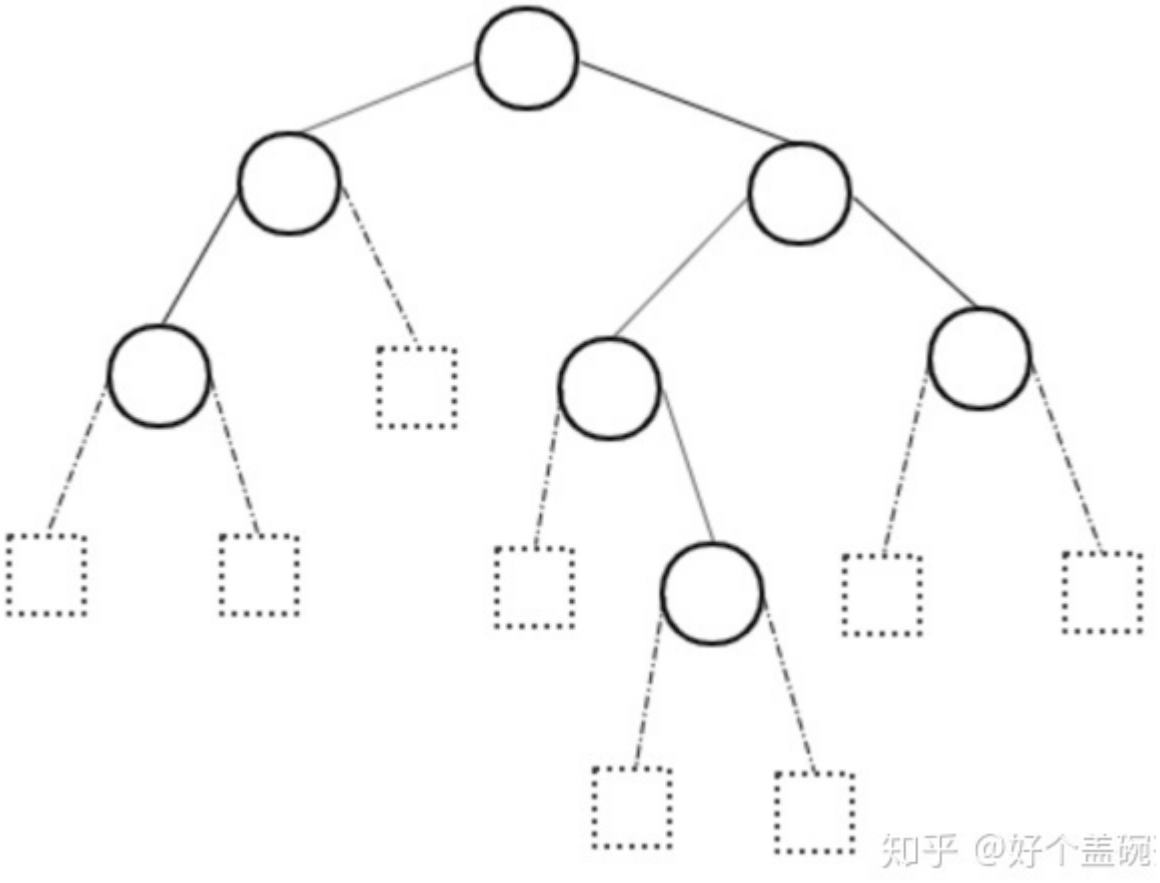
AVL树和红黑树是两类重要的二叉查找树，关于它们孰优孰劣，人们往往众说纷纭莫衷一是从空间开销、实现难度和时间开销(最坏/均摊/期望)等多个维度进行了细致而具体的比较，出结论：**AVL树和红黑树在不同的场景下均有各自的微弱优势，但几乎不可能形成碾压。**工本身是个充满了 trade-off 的过程，与其武断(却毫无底气)地说哪一个更好，深入探究它们藏的奥秘恐怕会更有趣一些。

准备阶段：二叉树高度的定义

AVL树和红黑树的自身定义以及操作算法，教材或者网上都有非常多的资料，由于本文不是文，所以就不花大量篇幅去复制粘贴这些大家都能很方便查到的东西了。不过有一点特殊的出，二叉树的高度总的来说能见到两种定义(以及若干种表述方式)：

- 根节点到其所有叶节点的最长路径包含的边数。
- 根节点到其所有叶节点的最长路径包含的节点。

这些定义(和表述)无本质区别也都没错，只是数值上可能会相差一点，但在讨论前最好能达认识。我比较相信 Knuth 的审美，因此采用他在 TAOCP 里面推荐的描述方式。



上图中一棵二叉树由圆形的真实节点构成，我们将所有真实节点缺少孩子的地方均补上方形
节点，构成一棵扩充的二叉树。在这种语境下我们又称圆形节点为内部节点(数量为 n)，方
为外部节点(数量为 s)。顺带说一下它们之间有个显著的关系： s = n + 1 。在后面的讨论
就是针对这种扩充的二叉树来讨论，而且很多时候外部节点都可以省略不画。在这基础上重
树的高度(记为 h)：**根节点到其所有外部节点的最长路径包含的边数。**

准备阶段：二叉树高度的下界

高度为 h 的二叉树内部节点数为 n ， 假设 n <= 2^h-1 ， 利用数学归纳法：

- 当 h = 0 时，有 n = 0 <= 2^h-1 = 2^0-1 = 0 成立;
- 当 h > 0 时，考虑根节点最多只能拥有两棵高度为 h-1 的子树，所以它的节点数必然满足
  <= 1 + 2^(h-1)-1 + 2^(h-1)-1 = 2^h-1 。

由此可以得到二叉树高度的下界：

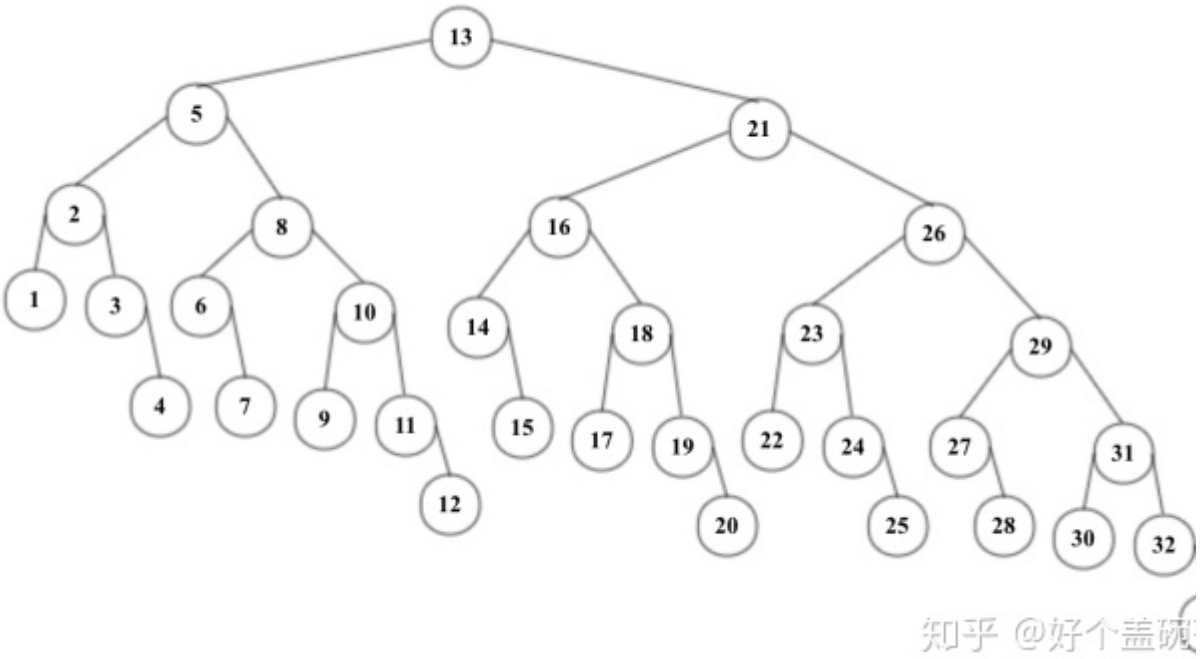
$$h \geq \lceil \log(n+1) \rceil \approx \log(n)$$

准备阶段：AVL树高度的上界

假设高度为 h 的AVL树最少包含 f(h) 个内部节点，因为AVL树本身的定义决定了左右子树高
不超过1，所以容易得出关系 f(h) = 1 + f(h-1) + f(h-2)，再结合初始条件 f(0) = 0, f(1) =
反解出AVL树高度的上界：

$$h \leq f^{-1}(n) \approx 1.44\log(n)$$

下图就是这种极端情况：



准备阶段：红黑树高度的上界

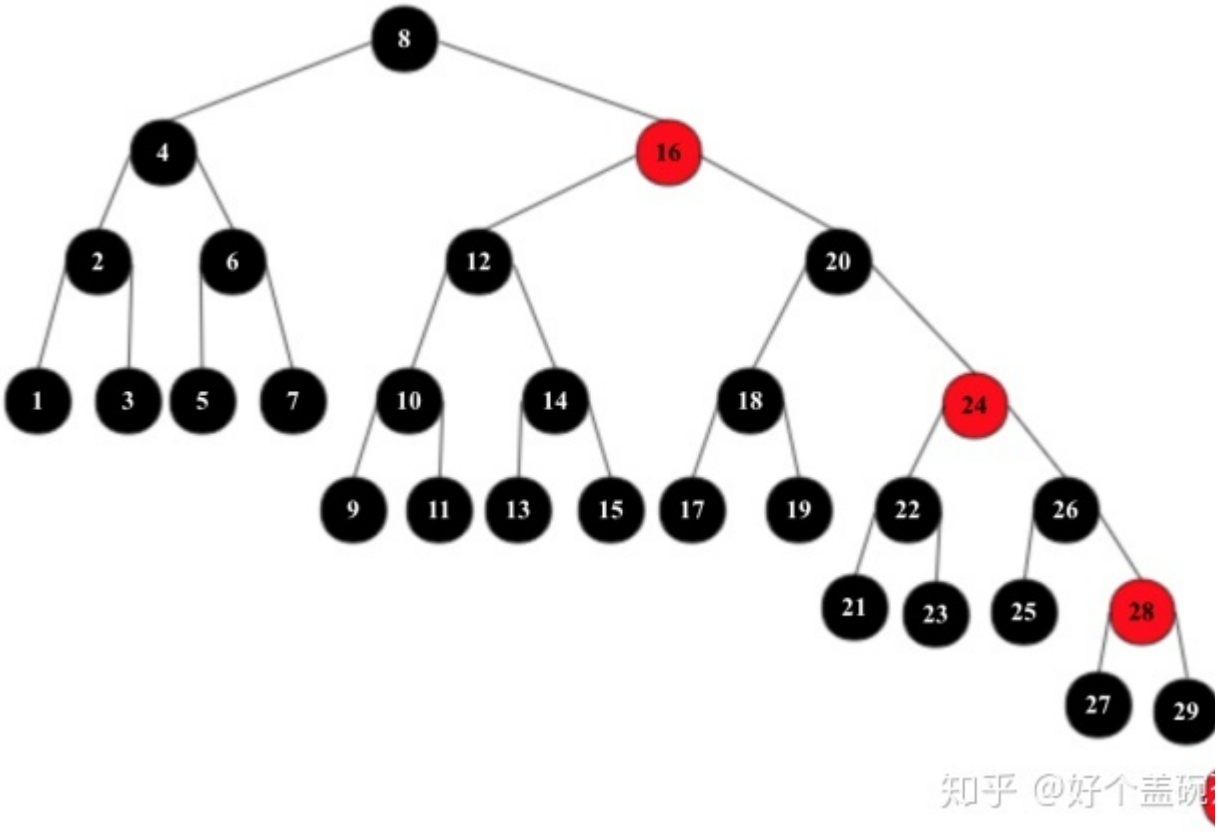
黑高为 b 的红黑树内部节点数为 n，假设 n >= 2^b-1，利用数学归纳法：

- 当 b = 0 时，有 n = 0 >= 2^b-1 = 2^0-1 = 0 成立；
- 当 b > 0 时，考虑根节点最少也要拥有两棵黑高为 b-1 的子树，所以它的节点数必然满足
  >= 1 + 2^(b-1)-1 + 2^(b-1)-1 = 2^b-1 。

由于红节点的孩子必须是黑节点，故而得到红黑树的高度上界：

$$h \leq 2\lfloor \log{(n+1)} \rfloor \approx 2\log(n)$$

下图就是这种极端情况：



AVL树和红黑树的比较

经常在网上看到关于AVL树和红黑树的讨论，讨论的双方往往各执一词，都在试图证明到底加优越。并且似乎都可以给出充足的理论依据，但最后的结果往往是谁也不能说服谁。我认的根源在于没有对齐讨论对象，当我们在比较AVL树和红黑树时，首先需要明确的是：我们比较什么？

- **空间开销**。AVL树的每个节点需要额外两比特来表示左斜、平衡、右斜三种状态，而红黑个节点只需要额外一比特来表示红、黑两种颜色，看起来是红黑树占据了优势。但是结构的分配不可能是以比特为单位来进行的，因此在以字节为单位分配内存的情况下红黑树的没有了。从另一个角度来讲，不管是两比特还是一比特，都可以把它编码到某个指针域的位，理由是内存对齐使得指针域的最低两位必然为零。在这种骚操作的情况下，AVL树和的空间开销也是一模一样的。顺带提一点不太相关的，父指针域要不要都可以，区别是不针域可以使空间开销更小，然而代价是循环的时候需要维护一个栈结构，因此主流的实现这一点微弱的空间开销以获取更快的运行速度。
- **实现难度**。有一种观点认为红黑树插入和删除后的调整过程需要考虑太多的场景了，而AV需要比较左右子树的高度决定如何旋转即可，因此AVL树的实现难度要低于红黑树。事实前学习的时候在网上看过不下十份的AVL树代码，几乎没有正确的(有的版本不保存平衡因存高度，有的版本甚至每一次都用递归来求高度)。这两种树我都亲自实现过，就以自己的

看，在考虑各种Corner case、常数时间返回最值元素、指针向前向后迭代、对重复元素
持等等条件的限制下，要无误且优雅地实现这两者之间的任意一个都是很困难的。鉴于实
是个比较主观的东西，这里就不做过多的评价了。
- **时间开销**。也是大家通常说的时间复杂度，这个恐怕才是争议的核心，后面所有的篇幅都
它来讨论。

一般说来，不管是AVL树还是红黑树，不管是插入还是删除(我们不专门另讨论查找，它包含
插入和删除中)，操作的开销大致都可以分解成如下两部分：

- **查找开销**。插入前总是需要查找到具体的位置才行，需要不断向下查找直至外节点。删除
也要查找到对应元素，虽然这不是必要的，但是将查找和删除合在一起讨论显得更加方便
- **调整开销**。插入和删除都有可能打破原来的平衡约束，因此需要一层一层地向上调整。每
调整开销里面具体又会包含：a. **变色开销**，即修改节点的颜色(或者平衡因子)带来的开销
转开销，即旋转操作中修改各种指针指向带来的开销。这两种开销的系数是不一样的，在
确讨论的时候应该严格区分而不能把它们混为一谈。

最坏时间开销的比较

明确了时间开销的衡量指标以后，我们开始对AVL树和红黑树做最简单的比较：最坏情况下
销的比较。

操作类型	查找次数	调整次数	变色次数	旋转次
AVL插入	$1.44\log(n)$	$1.44\log(n)$	$1.44\log(n)$	2
R-B插入	$2\log(n)$	$2\log(n)$	$3\log(n)$	2
AVL删除	$1.44\log(n)$	$1.44\log(n)$	$1.44\log(n)$	$0.72\log($
R-B删除	$2\log(n)$	$\log(n)$	$\log(n)$	3

上表是两者在最坏情况下各项开销的汇总，这些数值是比较容易分析出来的：

- AVL树和红黑树的插入查找次数和删除查找次数，最坏情况等于高度上界。
- AVL树插入旋转、红黑树插入旋转、红黑树删除旋转，看算法就一目了然。

- AVL树删除旋转，考虑极端情况图中删除1号节点，最左路径长度刚好是最右路径长度的-
- 调整次数指的是在调整过程中不断向上访问了父节点多少次，并不能粗暴地说调整次数就调整开销，在每一次调整中总是伴随着若干次变色或者旋转的。
- 对于AVL树插入调整和变色，考虑极端情况图中一开始没有33号节点，这时候插入它就会上修改平衡因子直到根节点，每上一层修改一次平衡因子。
- 对于红黑树插入调整和变色，考虑极端情况图中在30号节点下再插入一个红色节点，并且路径上的每一个红色节点都补充一个红色的左孩子，就会导致逐层向上修改颜色直到根节上两层修改三次颜色。
- 对于AVL树删除调整和变色，考虑极端情况图中删除33号节点，这时候就需要逐层向上修因子直到根节点，每上一层修改一次平衡因子。
- 红黑树要形成逐层向上调整的局面则要困难一些，最坏的情况考虑一棵全是黑节点的满二删掉最边缘的叶节点，每上一层修改一次颜色。

简单证明了上表的正确性以后我们就可以分析AVL树和红黑树在最坏情况下各自的优劣了：

1. **就查找而言，两者都是对数级别的，但红黑树的系数更大一点。举个可以感知的例子，在个节点的时候，AVL树保证最多需要比较34次一定能查出结果，而红黑树则最多需要比次。**
2. **插入调整(及变色)和查找类似，AVL树拥有着系数上的优势。然而删除调整时构造的红黑是一颗满二叉树，这种情况下两者的优势又反转了，可谓是各有千秋。**
3. **至于插入旋转大家最多都是两次，而删除旋转红黑树保证最多三次，而AVL树则可能是对别的。**

因此有些人基于此结果便得出“结论”：“最坏情况下AVL树和红黑树的查找次数都是对数的，虽然红黑树的系数更高一些，但是没有本质区别可以容忍。而AVL树最致命的地方在于旋转次数也是对数级别的，这是导致了红黑树应用广泛而AVL树无人问津的原因。”

但我觉得这并不能让人信服，因为最坏情况的分析(Big O)具有很恶心的传染性。我举个例子C++的std::vector在执行push back操作的时候如果内存不够会重新申请一片更大的内存，当前的数据拷贝过去，这个操作明显是 O(n) 的。那是不是就能证明如果我们push back n总复杂度就成了 O(n^2) 呢？显然不是的，你可以算算对于 n 次操作的总复杂度依然是 O(

均摊时间开销的比较

所以虽然最坏情况下的分析已经相对明白了，但是它的参考意义并没有想象中的那么大，相情况下的分析显得更有意思一些。

操作类型	查找次数	调整次数	变色次数	旋转次
AVL插入	-	O(1)	O(1)	O(1)
R-B插入	-	O(1)	O(1)	O(1)
AVL删除	-	O(1)	O(1)	O(1)
R-B删除	-	O(1)	O(1)	O(1)

上表是两者在均摊情况下各项开销的汇总，考虑的是**以任意序列连续插入 n 次**的最坏平均开
以任意序列连续删除 n 次的最坏平均开销。其中AVL树还有一个更强的上界，插入后的均摊
次数不大于3.618，删除后的均摊调整次数不大于2.618。相关文献：

- Mehlhorn, K., & Tsakalidis, A. (1986). An amortized analysis of insertions into AVL-
  SIAM Journal on Computing, 15(1), 22-33.
- Tsakalidis, A. K. (1985). Rebalancing operations for deletions in AVL-trees. RAIRO.
  Informatique théorique, 19(4), 323-329.
- Amani, M., Lai, K. A., & Tarjan, R. E. (2015). Amortized rotation cost in AVL trees. a
  preprint arXiv:1506.03528.
- Tarjan, R. E. (1985). Amortized computational complexity. SIAM Journal on Algebr
  Discrete Methods, 6(2), 306-318.
- Amortized Analysis: web.stanford.edu/class/...

注意这是一个非常强的结论，远比分析最坏时间开销得出的结论更有意义。该结论指出：**无
AVL树还是红黑树，它们以任意序列连续插入的平均开销和以任意序列连续多次删除的平均
是常数的，跟树的大小无关！**也就是说无论大小是一百万、一千万还是一亿的树，当你以任
连续插入或者以任意序列连续删除，其导致的调整(以及变色和旋转)开销都是微乎其微的。
查找的开销才是对数复杂度，会随着树的大小变化而变化。

以上的均摊分析给出了可喜的结果，但条件是连续插入或者连续删除。如果把条件放宽松一
允许任意序列的插入或者删除混合进行，这时候的均摊结果将会是怎样？结果就是：**红黑树
意序列插入或者删除混合进行的情况下，均摊复杂度依然保持在 O(1) 。而AVL树就惨了，
明存在精心构造出的特定操作序列，让它的均摊复杂度退化到 O(log(n)) 。**我觉得这个结果

对AVL树的致命打击，因为真实环境中往往是插入和删除混合进行的，特别是在恶意攻击的下。

期望时间开销的比较

分析完了最坏复杂度和均摊复杂度，我们最后再来分析期望复杂度。注意期望和均摊的区别它们都是衡量操作序列的平均开销，但均摊是针对任意操作序列的最坏情况而言的，而期望对随机操作序列的平均情况。真实环境中充斥着大量的随机事件，因此针对这种随机性进行间开销分析也是很有必要的。

操作类型	查找次数	调整次数	变色次数	旋转次
AVL插入	$\log(n)$	2.79	3.34	0.70
R-B插入	$\log(n)$	1.80	2.32	0.58
AVL删除	$\log(n)$	1.92	2.16	0.30
R-B删除	$\log(n)$	0.73	1.99	0.38

上表是两者在随机情况下各项开销的汇总，数据都是实验得出的(其中大部分有文献依据)。大程度上是支持了均摊时间开销的分析结果，而且比均摊分析中给出的上界要好很多。**单从果来看无论是插入还是删除，无论是调整、变色还是旋转(红黑树删除旋转除外)，红黑树相AVL树都要略胜一筹，但优势仅限于这么可见的一点点。**然而我们比较的是操作次数，实现的差异也可能使得红黑树即便是调整次数上占优，但如果每次调整带来的开销更大一些，也这个次数上的微弱优势消耗殆尽。

为了验证真实的情况，我只好用 Go 重新徒手撸了一发AVL树和红黑树，在程序输出结果的下，给出一些自己的看法。顺便说明一下，AVL树的实现参考了维基百科，红黑树的实现参法导论，数据结构的定义参考了STL源码。为了保证最高效，采用了父指针方式并且杜绝一衡因子或者颜色编码到指针等骚操作。项目地址在 github.com/dploop/avl-v...

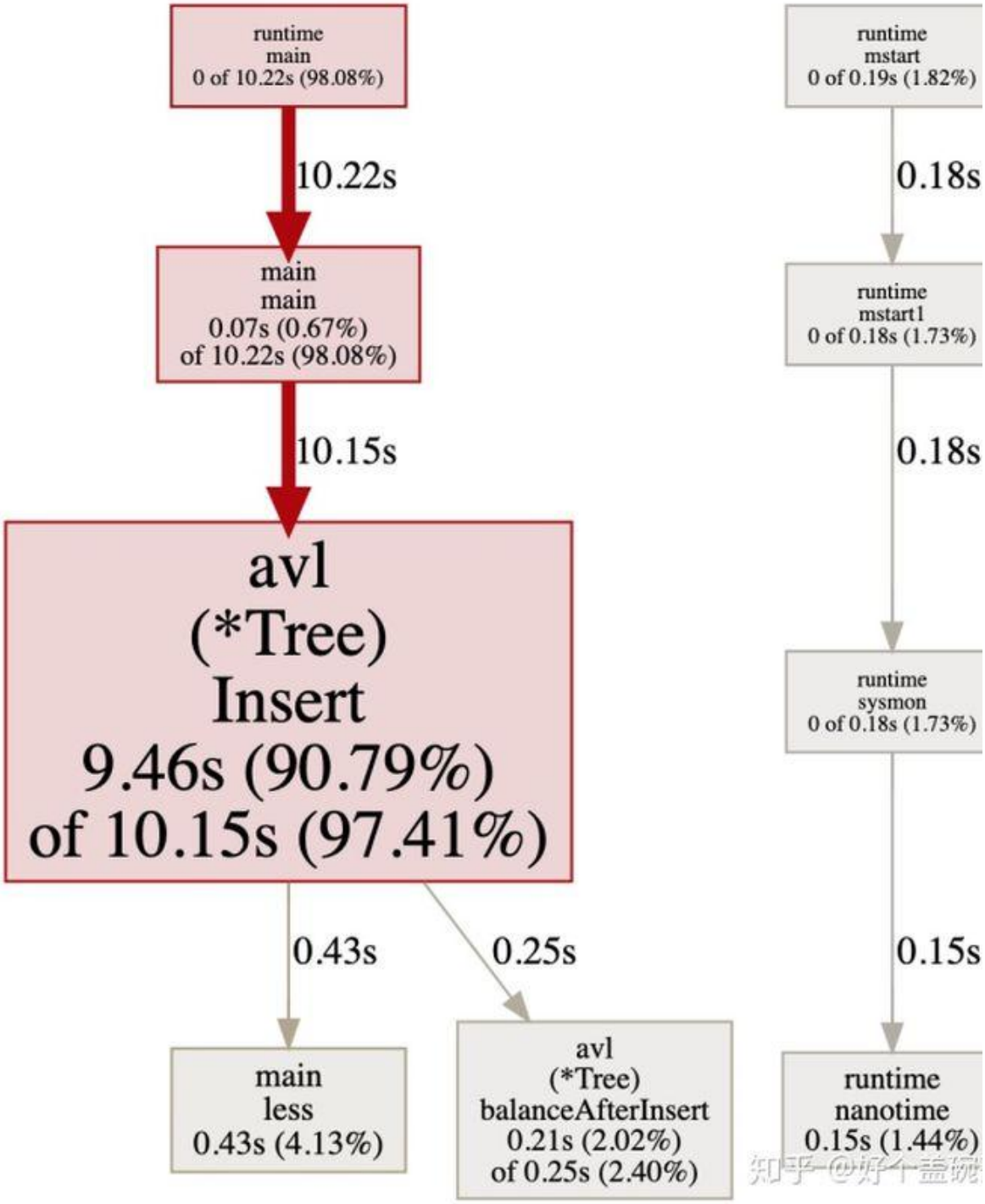
操作类型	查找次数	调整次数	变色次数	旋转次数	耗时(r
AVL插入	22.22	2.79	3.34	0.70	1181
R-B插入	22.25	1.80	2.32	0.58	1171
AVL删除	21.21	1.92	2.16	0.30	1192
R-B删除	21.23	0.73	1.99	0.38	1179

上图是一千万个节点的插入和删除分析结果，我的测试方案(为了和大部分论文的实验方案保致)是：

1. 预分配好一千万个节点，这样做的目的是避免运行时内存分配带来的开销扰乱结果。
2. 将这一千万个节点完全随机打乱。
3. 按照打乱的顺序将节点逐个插入，然后统计插入累计消耗时间。
4. 将这一千万个节点再次完全随机打乱。
5. 按照再次打乱的顺序逐个查找并且删除，然后统计累计消耗时间。

从实验的结果看来，AVL树和红黑树在插入和删除的耗时几乎没有任何差距(2%以内)，我怀疑操作系统任何一点轻微的扰动带来的差距都有可能比这大。

为了进一步揭示实验结果的含义，我对AVL树的插入过程做了profiling(其实删除情况以及统也做了，结果是类似的)：



这个调用时间占比图很直观地体现了什么叫做实践是检验真理的唯一标准，我们前面做了一于插入调整(及变色和旋转)的分析，似乎都很有理有据。到最后真正测试的时候发现：**在AV Insert函数累计消耗的10.15秒里面，它们只占用了0.25秒，时间占比2.4%。**

但同时这个图也给我带来了新的疑问，按理说一个插入平均查找22次，其间的操作非常简单调整过程中变色和旋转加起来好歹也有3次，而且旋转操作还各种指针换来换去。这么估算好歹给我个10%以上的时间占比我也算有点面子啊，这些调整操作不要钱的吗！于是乎我又更进一步的profiling，具体到每一行代码看它的开销是多少：

这下更加明白了，当需要插入一个节点z时，需要将它从根节点开始逐层向下比较。对于当
到的节点y，如果z的数据比y的数据小，则将y指向它的左孩子，否则右孩子。就是这么一个
单的操作，在10.15s的总时间占据了7.94s，绝对的大头。前面那个7.51s又代表什么呢，它
是真正发生在本函数调用中的时间，反过来说中间0.43s的差值是t.Less()这个函数调用的总
也就是z的数据和y的数据进行比较的时间。

我一开始以为是t.Less()这个函数调用会占据比较多的时间，现在看起来错了。那么这一行(
面的几行)真正慢的原因是：

1. **缓存miss**。在z的数据和y的数据比较之前，需要加载y的数据。因为是随机插入，局部性
再起作用，缓存miss以后需要到主存去加载y的数据，这个带来的开销远远高于之前讨论
整、变色和旋转。
2. **分支预测**。如果是接近有序的情况下，if语句的分支能够被很好地预测进而极大地提高运
度。然而在随机插入的情况下，分支预测达到了最坏的情况(50%命中率，跟瞎猜没什么
这使该条语句的开销进一步加大到了不可思议的程度。

关于上述两个原因，都能很简单地验证，譬如你可以使用C++的std::map顺序插入键值对比
插入键值，它们之间的效率差距真不是一点半点。另外针对本文，也可以在运行我的代码时
参数，用来动态调整节点顺序打乱的随机程度(0是完全有序1是完全随机)，有奇效。

最后给出我实验时的运行输出：

```
➜  avl-vs-rb git:(master) make
go build -o avl-vs-rb ./main
➜  avl-vs-rb git:(master) ./avl-vs-rb -n=10000000 -t=avl -r=1
==========>>>>> Input Arguments
type: avl, size: 10000000, rand: 1, seed: 1
pause for 5 seconds...
pause for 5 seconds...
==========>>>>> Insert Results
insert elapse: 11815ms
insert search: 22.22
insert  fixup: 2.79
insert  extra: 3.34
insert rotate: 0.70
pause for 5 seconds...
pause for 5 seconds...
==========>>>>> Delete Results
delete elapse: 11922ms
delete search: 21.21
delete  fixup: 1.92
delete  extra: 2.16
delete rotate: 0.30
pause for 5 seconds...
^C
==========>>>>> Input Arguments
type: rb, size: 10000000, rand: 1, seed: 1
pause for 5 seconds...
pause for 5 seconds...
==========>>>>> Insert Results
insert elapse: 11719ms
insert search: 22.25
insert  fixup: 1.80
insert  extra: 2.32
insert rotate: 0.58
```

```
pause for 5 seconds...
pause for 5 seconds...
==========>>>>> Delete Results
delete elapse: 11793ms
delete search: 21.23
delete  fixup: 0.73
delete  extra: 1.99
delete rotate: 0.38
pause for 5 seconds...
^C
```