CS2045M: 高级数据结构与算法分析 **2025-2026 学年秋冬学期**

Lecture 1: AVL 树、Splay 树与摊还分析

编写人：吴一航 yhwu_is@zju.edu.cn

1.1 引言

第一节课将介绍经典的两种平衡搜索树。作为第一节课，个人认为简单回顾上一学期的内容以引入本学期的内容是非常重要的。在课程简介中已经提到，数据结构的重要性在于可以通过有效管理数据，减少算法中对输入数据或中间数据的搜索、插入、删除、合并等操作的时间。因此，学习数据结构的基本要点就是了解其能支持的操作以及对时间复杂度的改进，从而可以在应用中面对合适的场景选择正确的数据结构。

在数据结构基础中，我们主要学习了如下数据结构以及它们对应的操作的时间复杂度分析，也学习了它们适用的场景：

1. 栈：实现常数级头部添加或删除对象，可用于深度优先搜索；
2. 队列：实现常数级尾部添加和头部删除对象，可用于广度优先搜索。

栈和队列称为线性结构（链表太简单不再重复），以上提到的关键例子（深搜和广搜）只需要用到首尾插入和删除的功能，因此简单的栈与队列是合适的选择。但在数据存储的场景中，数据的更新没有数据的搜索频繁，因此我们更重视搜索的效率，而线性结构往往需要线性时间完成这一点，除非使用有序数组可以用二分查找降低复杂度，但这样插入和删除的操作又变为线性。我们希望搜索、插入、删除的效率都比线性好，那么如何实现呢？

1. 二叉搜索树：遵循左小右大的原则进行插入，因此可以实现在树结构上的“二分”查找。设树高为 h，则搜索、插入和删除都可以在 $O(h)$ 时间完成（还记得如何实现吗？），如果运气不差，$h = O(\log n)$，则实现了我们的理想。
2. 优先队列（二叉堆）：为经常要选取最小值或最大值的算法服务，例如操作系统进程调度，Dijkstra 算法（很多贪心算法都是如此）等。它并不支持高效的任意查找，因为只要求结点的儿子一定要比自己大（或小）即可，但可以支持 $O(1)$ 的 FindMin 或 FindMax，$O(\log n)$ 的插入（还记得吗？可以用数组实现，直接插入在最后一个位置，然后 percolate up）和 DeleteMin（用最后一个元素覆盖根结点，然后 percolate down）。

除此之外，为了支持更快的查询，我们介绍了散列表（hash table）的概念，可以在散列函数选择得较好的时候实现 $O(1)$ 级别的查询、插入和删除速度，但如果设计不佳则查询和删除也是线性级别的时

间。这是很现实的问题，事实上前面的例子都告诉我们，要想实现一些特定功能的加速，那必然需要施加一些约束，这些约束可能不好实现，增加设计的复杂程度，也可能会影响其它操作的速度，这其中的 trade off 也使得数据结构的研究充满挑战和趣味。

本讲将介绍平衡搜索树，事实上设计平衡搜索树的目的是非常明确的，因为一般的二叉搜索树查询、插入和删除都可以在 $O(h)$ 时间完成（树高为 h），但存在最差的情况使得 h 和 n 较为接近，这导致搜索树所有操作效率的降低。因此最简单的想法就是，在二叉搜索树左小右大的大小要求之上，再添加一些有关于树的平衡的要求，使得 $h = O(\log n)$ 总是成立的，便可以真正实现搜索、插入和删除速度的提升。

在开始正式内容之前推荐一个数据结构可视化网站，包括了我们讲到的所有数据结构的基本操作可视化功能：https://www.cs.usfca.edu/ galles/visualization/Algorithms.html

1.2 AVL 树

最简单的想法便是强制要求每次插入、删除之后都保证树的“绝对平衡”，这就是 AVL（Adelson-Velskii-Landis）树的思想。AVL 树的平衡要求每个结点的左右子树高度差的绝对值不能大于 1，然后便可以证明以下引理.

引理 1.1 *设 AVL 树的结点数量为 n，则树高 $h = O(\log n)$.*

证明时将给定结点个数求最高的树高问题，转化为给定树高，求最小结点个数的对偶问题，然后转化为斐波那契数列即可。

下面简要讨论搜索、插入和删除操作的具体实现，从而能得到其时间复杂度。搜索与普通的二叉搜索树完全一致，利用左小右大的性质递归或循环均可。因此时间复杂度显然就是 $O(h) = O(\log n)$.

对于插入，需要 $O(\log n)$ 的时间找到插入的位置，接下来的问题是，插入可能会破坏树的平衡结构。需要注意的是，被破坏平衡性的结点显然只能在新加入的结点到根结点的路径上，因为只有这些结点的子树高度变化了。因此需要从插入的结点开始向上走找到第一个被破坏平衡性质的结点。此时，平衡被破坏的原因可以分为四种情况：LL，LR，RL，RR。其中 LL 和 RR 只需一次旋转即可，LR 和 RL 需要两次旋转。此处读者应当结合 PPT 的动态图像，参考 *Data Structures and Algorithm Analyses in C* 理解具体代码实现，包括如何记录和更新结点信息，如何判断平衡条件被破坏，以及如何具体写 rotation 操作（2022 年期末考试编程题直接要求写 AVL 树的实现，得分情况很糟糕）。

讨论：回答以下两个问题，（1）请举出合适的例子肯定或给出证明反驳，（2）请举出合适的例子反驳或给出肯定的证明（证明无需教科书式的完整，但需要思路清晰，可以有图示辅助）：

1. 在插入的时候是不是有可能很多个结点的平衡性质都被破坏?

2. 如果是的话，一次旋转（Double rotation 也视为一个整体）操作能让所有平衡受到破坏的结点恢复吗?

上面的问题的意义在于，如果一条线上 $O(\log n)$ 个结点平衡性质都被破坏，且需要每个顶点都旋转的话，操作代价是比较大的。事实上，两个问题的答案都是肯定的。第一个在课件第 6 页就可以找到例子，第二个问题，回忆只有新加入的结点到根结点的路径上的结点可能平衡属性被破坏，因为此时是插入操作，所以平衡被破坏是因为新加的结点太深了，但如果观察可以发现，无论是 LL、RR 还是 LR、RL，事实上都会使得插入的结点深度减少 1，所以一定能使得路径上所有结点平衡属性恢复（建议结合图示思考）。

至于删除，课件上并未作出要求，感兴趣的同学可以参考维基百科学习，事实上也是破坏平衡性质然后进行旋转。需要注意的是，删除不再有上面插入只需一次旋转即可恢复平衡的性质，具体可以参考这个知乎回答。

根据引理以及插入、删除的具体操作（每次旋转是常数级别的，删除最多 $O(\log n)$ 次的旋转），我们很容易得到如下定理：

定理 1.2 *AVL 树的搜索、插入和删除操作的时间复杂度为* $O(\log n)$.

1.3 Splay 树

Splay 树的想法一方面来源于希望可以不像 AVL 那样保持严格的平衡约束，但也能保证某种层面（均摊）的对数时间复杂度，另一方面 Splay 树在访问（特别注意访问包括搜索、插入和删除）时都需要将元素移动到根结点，这非常符合程序局部性的要求，即刚刚访问的数据很有可能再次被访问，因此在实现缓存和垃圾收集算法中有一定的应用。

接下来讨论具体操作，简而言之 Splay 树的操作就是在原先操作的基础上加上 splay 操作，而所谓 splay 操作就是通过一系列旋转将访问的结点移动到根结点的位置。因此可以将搜索、插入、删除描述如下：

- 搜索：使用普通二叉搜索树的方法找到结点，然后通过 splay 操作经过一系列旋转将搜索的结点移动到根结点的位置；
- 插入：使用普通二叉搜索树的方法找到要插入的位置进行插入，然后把刚刚插入的结点通过 splay 操作经过一系列旋转移动到根结点的位置；
- 删除：使用普通二叉搜索树的方法找到要删除的结点，然后通过 splay 操作经过一系列旋转将要删除的结点移动到根结点的位置，然后删除根结点（现在根结点就是要删除的点），然后和普通二叉搜索树的删除一样进行合理的 merge 即可。

于是接下来的问题就是 splay 操作的具体实现。PPT 的 12 页给出了一种非常 naive 的想法，就是不断地把访问的结点与其父结点更换父子关系，事实上就是不断用 SingleRotation 翻到根结点的位置。然

而这个例子告诉你，这么做之后虽然把要访问的结点放到了根结点，但其它有的结点被移动到了很深的位置，这是不好的（对比 PPT 的 15 页）。事实上可以论证这样的旋转无法保证从空树开始的连续 M 个操作是 $O(M\log n)$ 的，这就是 PPT 第 13 页的例子的作用，这个例子从空树开始，通过 N 次插入（每次都是 $O(1)$）和 N 次查找（总共是 $N+\cdots+1=O(N^2)$ 的操作数）构造了长度为 $2N$ 但操作 $O(N^2)$ 的序列，不符合目标中从空树开始连续 M 个操作是 $O(M\log n)$ 的要求，因此我们要放弃这种 naive 的、不断把访问的结点与其父结点更换父子关系，从而翻到根结点的方法（可以与 PPT 的 16 页正确的方法比较）。

因此在 PPT 的 14 页给出了合理的旋转方法，注意：

1. case 1 就是简单的交换 X 和 P 的父子关系，然后调整子树满足搜索树性质即可；
2. zig-zag 的操作方法与 AVL 树的 LR 或者 RL 是一致的，实际上与之前的 naive 的方法也是一样的，三者等价；
3. zig-zig 才是与 naive 的方法不一样的地方！特别注意 naive 的方法先交换 X 和 P 的位置关系，然后交换 X 和 G 的位置关系，但是 zig-zig 的标准操作方式是，先交换 P 和 G 的位置关系，再交换 X 和 P 的位置关系！这个区别就是它与 naive 方法的唯一区别，却能实现最终均摊的目标；
4. 注意虽然 zig-zig 在 PPT 上写的是 single rotation，但实际上转了两次，这里的 single 大概就是为了表示和 AVL 的 double rotation 不一样吧，zig-zag 有底气称为 double rotation 是因为它和 AVL 一样。

在此再次强调，Splay 树的访问包括搜索、插入和删除，因此 PPT 的 16 页连续插入 1, 2, 3, 4, 5, 6, 7 是会得到图示结果的，因为插入 2 就会把 2 翻到根结点，插入 3 就会把 3 翻到根结点，以此类推，最后 7 在根结点的位置，排成一条斜线。这里特别建议诸位动手推一遍这些操作，防止考试眼高手低。

具体的代码实现以及更加详细的分解动作讲解同样见维基百科。

1.4 摊还分析

1.4.1 概述

摊还分析的想法来源于我们希望估计一种数据结构经过一系列操作的平均花费时间。然而，平均时间非常难计算，因为每一步都有非常多的选择，连续 m 个操作，可能的操作路径是指数级别的。并且有时候平均涉及概率分布等，但我们并不知道确切的分布，因此比较难以计算。

一种最简单的估计方法就是用最差情况分析作为平均情况的上界，例如 Splay 树，每个操作最差都是 $O(n)$（n 为树中结点个数）的，因此平均不会比最差情况差，所以也是 $O(n)$ 的。然而这样的估计显然是放得太宽了，我们对这个复杂度是非常不满意的，因此需要进行摊还分析。事实上，在最差情况的分

析中，我们忽略了一件事情，就是有的序列是不可能出现的，例如直接在空的树上用 $O(n)$ 时间删除。摊还分析则是希望排除掉最差情况分析中把所有不管可能不可能的情况，最差的路径挑出来的这种无脑行为，转而分析所有可能的从空结构开始的操作路径中，最差的平均时间，那么这一时间一定比最差情况分析好，因为排除掉了一些不可能出现的所谓最差序列，但又会大于等于平均时间，因为取的是所有可能序列中最差的那一种。因此摊还分析的时间复杂度一定是平均时间的上界，同时这个上界会比最差情况分析好，这也是 PPT 第 18 页不等式的内在含义。

注意摊还分析要从空结构开始，如果不从空结构开始，则必须要求连续的操作数量足够大，从而抵消初始步骤中可能出现的消耗较大的操作。否则可以思考从一个已经有很多元素的栈里面一次性 Multipop 出所有元素，这一步操作的复杂度显然不再是 $O(1 \cdot 1)$ 的. 回到 PPT 的 13 和 16 页的例子，我们构造的序列必须从空的开始插入然后才能 find，然后 13 页的 naive 的方法是能构造出很差的例子的，但 16 页重复 13 页的操作并不能构成反例。

接下来介绍三种方法：聚合分析、核算法以及势能法。聚合分析直接使用了上面提到的思想："摊还分析是考虑可能出现的操作序列中的最差序列"（再次强调，这里的最差不是最差情况分析的最差，最差情况分析的最差会包括不可能出现的序列，但摊还分析要排除不可能的序列），而后两种方法则是基于另一种理解，具体展开时再介绍。

1.4.2 聚合分析

根据前面的介绍可知，"摊还分析是考虑可能出现的操作序列中的最差序列"，此时再审视 PPT 第 19 页的例子。我们希望计算出这个数据结构操作的平均时间，但连续 n 次操作的序列有 3^n 种（每一步都有 3 种操作选择），其中还有一些不可能的要剔除，而且我们也没有假定几种操作出现的概率分布，所以很难计算平均情况，因此需要其它方法。最差情况分析非常暴力，认为最差的操作就是一次 MultiPop n 个元素，因此得到连续 n 次操作最差时间为 $O(n^2)$，然而怎么可能从空栈开始每次都 Pop n 个元素呢？这种分析明显是严重放大了估计的上界，因此采用摊还分析，只考虑可能序列遇到的最差情况，排除掉最差情况分析臆想的不可能的序列。

于是需要思考这一支持 MultiPop 操作的栈，在从空栈开始的连续 n 次操作中，最差的情况是什么：实际上就是先 Push $n-1$ 个然后最后一次操作一次性 Multipop 出所有元素。为什么这样是最差的呢？事实上简单想想就能理解，无法理解或者希望严谨一些可以看下面的解释：因为 n 次操作是固定的，所以目标是固定 n 的情况使得下总的代价最大，普通的 Push 和 Pop 都是 1 次操作对应 1 个单位代价，所以必须寄希望于序列中单次 MultiPop 代价最大，那代价最大的情况就是只 MultiPop 一次，且就在最后一次，因为如果只 MultiPop 一次，不在最后一次，显然这次 MultiPop 代价比 $n-1$ 小；如果 MultiPop 多次，那么 Push 操作的个数少了，所以 MultiPop 能弹出的比 $n-1$ 少，所以代价也少。

综上，从空栈开始的连续 n 次操作中，最差的操作代价是 $2n-2$，因此摊还分析复杂度为 $O(1)$.

1.4.3 核算法

核算法的思想来源于，我们希望计算平均成本，完美的平均成本就是截长补短，即把时间长的操作成本摊到时间短的操作上，但完美的截长补短很难做到，那么我们尽力做到，且要保证从长成本大的操作截取的部分都要分摊出去，时间不能变少了，否则会导致总时长变短，从而比平均时间短，那么摊还时间复杂度会成为平均时间的下界，就失去意义了。说起来有些抽象，直接写下来表达式吧。所谓截长补短，设第 i 种操作的真实成本是 c_i，截长补短的摊还成本是

$$\hat{c_i} = c_i + \Delta_i,$$

其中 Δ_i 就是截的长（负值），或者补的短（正值），并且要保证摊还成本比平均成本大，这样分析出来的摊还成本才是平均成本的上界，即要求

$$\sum_{i=1}^{n} \hat{c_i} \geqslant \sum_{i=1}^{n} c_i,$$

也即 $\sum\limits_{i=1}^{n} \Delta_i \geqslant 0$。有了这一基础之后再来审视 PPT 第 21 页的例子。因为希望一次操作摊还成本为 $O(1)$，所以希望这三种操作的摊还成本都是常数级别的，这样只要使得 $\sum\limits_{i=1}^{n} \Delta_i \geqslant 0$，或者说 $\sum\limits_{i=1}^{n} \hat{c_i} \geqslant \sum\limits_{i=1}^{n} c_i$，那就直接证明了结论。但是我们知道，MultiPop 的代价比较大，把它调整为常数需要对应的 $\Delta < 0$，故必然需要代价小的操作 $\Delta > 0$，所以才有了例子中把 Push 操作代价调整为 2，然后我们可以利用 $\text{size}(S) \geqslant 0$ 这一约束证明 $\sum\limits_{i=1}^{n} \Delta_i \geqslant 0$。

事实上，核算法这一名字也非常形象。这就像你的银行卡，余额不可以为负数，所以你必须在花钱之前先存钱，你知道 MultiPop 是个花钱的事情，花钱之前必须存钱。当然你也可以说，你希望你的平均绩点保持在 4.0 以上，你知道 ADS 这门课你可能会挂科，所以你之前的微积分、线性代数、FDS 等就需要为这次挂科存下足够高的均绩。

1.4.4 势能法

事实上核算法尽管非常形象，但你要为每个操作设计一个摊还代价 $\hat{c_i} = c_i + \Delta_i$ 是不一定像上面的例子那么简单的，况且你还要保证 $\sum\limits_{i=1}^{n} \Delta_i \geqslant 0$（比如思考 Splay 树这种复杂的结构）。所以希望有一个更统一的方法解决这个问题：我们不再把目光局限于每个操作，而是给整个结构定义一个势函数，这个势函数描述了这个结构不同状态。基于这一思想，形式化而言，规定第 i 次操作的摊还代价为

$$\hat{c_i} = c_i + (\Phi(D_i) - \Phi(D_{i-1})), \tag{1.1}$$

注意这里的 i 不再是核算法中表示第 i 种操作，而是 n 个操作组成的序列中的第 i 个。因此这里的含义是，每一步的摊还代价等于真实操作的代价加上势函数的变化，于是我们求和有

$$\begin{aligned}\sum_{i=1}^{n}\hat{c}_i &= \sum_{i=1}^{n}(c_i+\Phi(D_i)-\Phi(D_{i-1}))\\ &= \left(\sum_{i=1}^{n}c_i\right)+\Phi(D_n)-\Phi(D_0)\end{aligned}$$

为了使得摊还成本是平均成本的上界，仍然需要满足 $\sum\limits_{i=1}^{n}\hat{c}_i \geqslant \sum\limits_{i=1}^{n}c_i$，因此需要 $\Phi(D_n)\geqslant\Phi(D_0)$，这一点在设计函数的时候很容易满足，因为可以通过调整使得 $\Phi(D_0)=0$，即初始状态势能为 0，则只需要其它任何状态的势能都不会小于 0 就可以了。

基于上述定义分析支持 MultiPop 操作的栈，自然地，我们希望真实操作成本高的 MultiPop 在势能法下的的摊还代价（式1.1）回归到常数，从而达到总复杂度 $O(1)$ 的目标。这就要求势能函数在经过 MultiPop 操作后有较大的下降，抵消 MultiPop 的真实操作代价。事实上在 MultiPop 后，栈中的元素个数下降了很多，因此一个自然的想法就是将势能函数定义为栈中的元素（并且这一定义符合 $\Phi(D_n)\geqslant\Phi(D_0)$），从而在这一势能函数定义下，三种操作的摊还代价都是常数，这也就是 PPT 第 23 页的思路。

读者不难发现势能法和核算法有些类似，都是通过对真实操作代价进行一些改变，实现最后证明的目标。只是核算法是对每一类操作添加 Δ_i（从而要求 $\sum\limits_{i=1}^{n}\Delta_i\geqslant 0$），势能法是定义一个统一的势能函数来决定每一步对真实代价的改变量（从而要求 $\Phi(D_n)\geqslant\Phi(D_0)$）。

此外，从上面的例子中可以得到一个经验：为了实现摊还分析得到比较好的复杂度结果，在核算法和势能法中，必然需要对真实代价比较高的操作添加一个比较大的负数项，从而得到一个符合要求的摊还代价。特别地，在势能法中，这就要求**势能函数在代价较高的操作后会有较大的下降**，读者在后续章节的分析以及完成相关习题时都可以回忆这一基本的摊还分析原则。最后，其实势能法就是借用了物理里面的势能的思想，给数据结构定义一个与结构本身特点相关的量。比如重力势能是质量和高度的函数，但树的势能可能是树高的函数，也可能是树的结点总数的函数甚至更复杂等等。

1.4.5 综合应用

总结一下上面的三种方法，第一种聚合分析建立在寻找可能出现的操作序列中的最差情况的想法，后两种建立在截长补短的思想，这两种思想是描述摊还分析等价的两种思想。然后要注意，在核算法中是对每个操作直接定义摊还代价，而势能法是利用势能函数对每一步操作间接定义摊还代价。还需要注意的是，聚合分析中所有操作的摊还代价是统一的，就是最后的平均值，因为是无差别的分析。但核算法不同操作的代价是直接定义的，可以是不同的，势能法也是可以不同的（栈的例子就可以看出）。

如果希望看到更多的例子应用这一美妙的思想的读者可以参考《算法导论》（第三版）17.4 节动态表的经典例子。这里给出一些考试可能出现的题目作为例子：

例 1.3 *A queue can be implemented by using two stacks S_A and S_B as follows:*

- *To enqueue x, we push a onto S_A.*
- *To dequeue from the queue, we pop and return the top item from S_B. However, if S_B is empty, we first fill it (and empty S_A) by popping the top item from S_A, pushing this item onto S_B, and repeat until S_A is empty.*

Assuming that push and pop operations take $O(1)$ worst-case time, please select a potential function Φ which can help us prove that enqueue and dequeue operations take $O(1)$ amortized time(when starting from an empty queue).

A. $\Phi = 2|S_A|$

B. $\Phi = |S_A|$

C. $\Phi = 2|S_B|$

D. $\Phi = |S_B|$

例 1.4 *A sum list L is a data structure that can support the following operations:*

- *Insert(x, L): insert the item x into the list L. The cost is 1 dollar.*
- *Sum(L): sum all items in the list L, and replace the list with a list containing one item that is the sum. The cost is the length of the list $|L|$ dollars.*

Now we would like to show that any sequence of Insert and Sum operations can be performed in $O(1)$ amortized cost per insert and $O(1)$ amortized cost per Sum. Which of the following statement is TRUE?

A. We use the accounting method that charges an amortized cost of 2 dollars to Insert and 0 dollar for Sum.

B. We use the potential function to be the number of elements in the list.

C. We use the potential function to be the opposite number of elements in the list.

D. Neither method can show the amortized cost for Insert and Sum is $O(1)$.

1.4.6 Splay 树的摊还分析

接下来将完成本讲的最后一个使命：证明 Splay 树的每个操作的摊还代价是 $O(\log n)$ 的。PPT 第 24 页给出了设计合理势函数的思想：

1. 回忆
$$\sum_{i=1}^{n}\hat{c}_i=\left(\sum_{i=1}^{n}c_i\right)+\Phi(D_n)-\Phi(D_0),$$
我们心里知道，最后合理的摊还分析结果是 $O(\log n)$ 的，所以 $\Phi(D_n)$ 应当是 $O(n\log n)$ 量级，所以应该带个对数函数。

2. 我们还希望在计算的时候，旋转导致的势能的变化尽量简单，这需要两个状态之间势函数相减时很多项之间可以互相消去，因此定义树 T 的势函数为
$$\Phi(T)=\sum_{i\in T}\log S(i),$$
即把所有结点的后代数取对数然后求和，这样在旋转过后大部分结点后代数不变，只有少量结点发生变化，于是减法时可以消去。

总而言之这两点可以说是一些直觉，但实际构造的时候必然要经过一系列的尝试，不可能一蹴而就。此外，在证明过程中需要用到一个引理，具体分析 PPT 或者维基百科已经比较详尽，此处不再赘述：

引理 1.5 *若 $a+b\leqslant c$，且 a 和 b 均为正整数，则*
$$\log a+\log b\leqslant 2\log c-2.$$

最后一个问题是，为什么不用每个结点对应子树的树高求和作为势函数？实际上代入后面的分析，会发现因为树高并非 $O(\log n)$ 的，因此最后一步无法得到 $O(\log n)$ 的摊还代价。总而言之，基于一系列的分析（具体见 PPT），可以得到如下结论：

定理 1.6 *记 $R(X)=\log S(X)$，对于结点 X 在第 i 次操作时，有*

1. *zig 操作的摊还代价 $\hat{c}_i\leqslant 1+(R_i(X)-R_{i-1}(X))\leqslant 1+3(R_i(X)-R_{i-1}(X))$；*
2. *zig-zag 操作的摊还代价 $\hat{c}_i\leqslant 2(R_i(X)-R_{i-1}(X))\leqslant 3(R_i(X)-R_{i-1}(X))$（这一步用到了前面的引理）；*
3. *zig-zig 操作的摊还代价 $\hat{c}_i\leqslant 3(R_i(X)-R_{i-1}(X))$；*

然而这还不够，因为搜索、插入和删除操作需要经过一系列的旋转操作才能实现。于是需要进一步分析将 X 从它所在的位置连续移动到根结点总共需要的摊还代价。实际上，假设 X 的高度为 $H(X)$，则可能的旋转次数为
$$k=\begin{cases}H(X)/2 & H(X)\text{为偶数}\\ (H(X)-1)/2+1 & H(X)\text{为奇数}\end{cases},$$

其中偶数的情况都是 zig-zig 或 zig-zag，奇数会先进行 $(H(X)-1)/2$ 次 zig-zig 或 zig-zag，最后进行一次 zig。放缩成最差的情况，就是都要有 k 次 zig-zig 或 zig-zag，加上最后一次 zig，则将 X 旋转到根结点的操作摊还代价为

$$\begin{aligned}\sum_{i=1}^{k+1}\hat{c}_i &\leqslant 1+3(R_{k+1}(X)-R_k(X))+\sum_{i=1}^{k}3(R_i(X)-R_{i-1}(X))\\ &= 1+3(R_{k+1}(X)-R_0(X))=O(\log n).\end{aligned}$$

回忆搜索算法需要先找到 X，然后将 X 旋转到根结点，事实上寻找 X 不可能比旋转更差，因为只需要一路下行，不需要指针换来换去，而二者路径总长度还是一样的，所以寻找 X 的摊还复杂度也必定是 $O(\log n)$，因此整个搜索 X 的算法复杂度就是 $O(\log n)$。至于插入和删除，用同样的分析方式，结合它们的具体算法就可以发现也是 $O(\log n)$。但是要注意的是，插入的时候插入的结点到根结点路径上所有的结点的势函数值会增大，如果要严谨证明必须还要说明这些结点势函数的增量也是 $O(\log n)$ 的，这里不给出详细证明，有兴趣的同学可以自行搜索资料或者自己尝试。总而言之，可以总结出如下定理：

定理 1.7 *Splay 树的搜索、插入和删除操作的摊还复杂度均为 $O(\log n)$。*