重量平衡树和后缀平衡树在信息学奥赛中的应用

陈立杰

杭州外国语学校

2013 年 4 月 8 日

为什么要研究重量平衡树

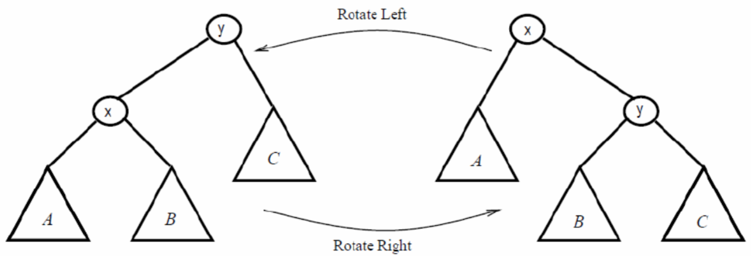
说起重量平衡树的优点，我们就先要了解传统平衡树的弊端。

传统平衡树的弊端

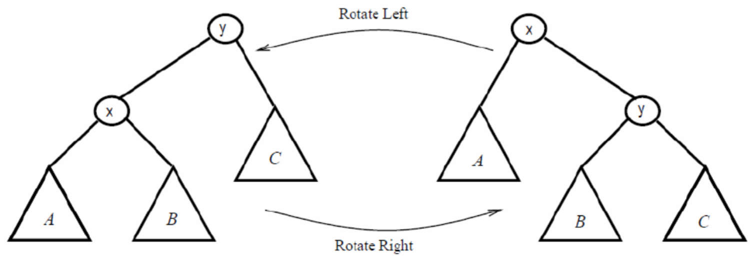
在传统平衡树中，我们需要使用旋转操作来维护平衡的性质。

同时我们经常在平衡树上维护一些信息来解决问题，比如子树内的最小值或子树的大小。

考虑一个经典的旋转操作。



传统平衡树的弊端



在完成上图的旋转后，我们需要重新计算x, y这两个节点维护的信息。

对于一般的信息，都可以使用常数时间由孩子的信息计算得出，比如内部最小值和子树大小，但是如果我们需要维护更加复杂的信息，比如我们要在每个节点维护一个平衡树存储子树内部所有点。那么一次旋转之后我们为了得出新的信息，只能暴力合并孩子的平衡树，复杂度会达到$O(n)$。

重量平衡树的概念

为了解决这一问题，我们提出重量平衡树的概念，重量平衡树的每次操作，涉及到的子树的总大小是在$O(\log n)$级别的。

有了这一理论担保，我们就可以使用重量平衡树解决一些复杂问题。

有哪些平衡树是重量平衡的

考虑到在OI比赛中实际应用的可行性，我们仅列出以下两种实现复杂度不高的平衡树。

替罪羊树　平摊$O(\log n)$

Treap　期望$O(\log n)$

其中Treap的实用性最高，下面将介绍Treap。

什么是Treap

Treap中每个节点维护一个随机权值key，在是一般平衡树的同时，要求父亲的key值始终小于等于孩子的key值。

那么插入一个数后，如果key值比父亲小就一直往上旋转，就能维护这个性质了。

可以证明所有操作都是期望$O(\log n)$的。

为什么Treap是重量平衡的

我们可以简单的理解一下，Treap中插入导致的旋转大部分都是微调，所需要的时间不多。因此平均的时间也不多。

下面来详细的证明一下。

Treap重量平衡的证明

让我们来考虑我们插入了一个数x。

插入Treap之后，根据Treap的算法，我们要一路往上进行旋转。

假设我们旋转到了祖先t的上面，那么这意味着x的key值，在t的子树中是最小的。

令t大小为n，那么此时的概率是$\frac{1}{n}$，那么涉及的子树大小期望就是$\frac{1}{n} \cdot n = 1$。

由于x有期望$O(\log n)$个祖先，那么此时涉及的子树的大小和期望就是$O(\log n)$。

所以插入一个数的复杂度是期望$O(\log n)$。

Treap重量平衡的证明

接下来我们考虑删除一个数x。

我们不妨直接重构以该数为根的子树。

由于一个点期望有$O(\log n)$个祖先，那么我们可以知道(祖先，孩子)这样的关系总共会有$O(n \log n)$对。因此一个点的子树大小期望是$O(\log n)$。

所以删除一个数的复杂度是期望$O(\log n)$。

所以Treap是一个重量平衡树。

实际应用:后缀平衡树

接下来我们来看重量平衡树的一个很有价值的应用：后缀平衡树。

考虑一个长度为n的字符串s。

所谓的后缀平衡树就是我们用一个平衡树来维护s的所有后缀按照字典序排序的顺序，也就是说用平衡树来维护一个字符串的后缀数组。

后缀平衡树的构造

跟后缀自动机和后缀树一样，后缀平衡树有一个在线的构造算法。

跟后缀自动机和后缀树不同，后缀平衡树的在线算法依赖于在最前面加入一个字符。

比如S，在最前面插入字符c之后就变成了cS。

后缀平衡树的构造

当我们在最前面插入c之后，所有S的后缀还是后缀，我们所做的一切只是在后缀集合中增加了一个新后缀cS而已。

我们在插入过程中，需要多次比较cS和S的一个后缀p的大小关系。

我们先比较c和p的第一个字符c',如果不一样比较就完成了，不然就比较S和p去掉第一个字符的后缀p'。

快速比较两个后缀的大小

由于S和p'都是已经在后缀平衡树里的后缀，只要比较他们在树里的先后关系就可以了。

一个简单的方法是分别求出两者在后缀平衡树里的排名(rank)然后比较，但是这样需要$O(\log n)$的时间，由于插入需要比较$O(\log n)$次。总复杂度就达到了$O(\log^2 n)$。

快速比较两个后缀的大小

而我们只需要知道S和p'的前后关系，求出rank显然显得有些浪费。

我们可以使用标记法，我们给每个后缀i打上一个实数的标记tag_i，如果后缀i小于后缀j，那么$tag_i < tag_j$。

也就是说，如果有这个标记的话，我们只需要比较S和p'的标记，就能比较出它们的大小关系了。

对标记的维护

我们对树中的每个节点t，让他对应一个实数区间(l, r)，表示它内部的点的tag都在该区间内。

我们可以令$tag_t = \frac{l+r}{2}$,令它的左孩子的对应区间为(l, tag_t)，右孩子为(tag_t, r)。容易发现这样的tag是满足条件的。

插入一个新节点的时候，我们只需要根据新节点的父亲，算出对应区间和tag值就行了。

同时为了方便，令根节点的权值范围为$(0, 1)$就行了。

对标记的维护

　　这种做法看似很完美，但是我们需要注意到，根节点*tag*是0.5，在最后插入一个数是0.75,再插入一个是0.875，插入100个的话，可以算出*tag*值是0.9999999999999999999999999999996055695473894940972941357 1735860688516339678244548849761486053466796875。

　　这显然已经超过了double类型的表示精度，会造成精度问题。

　　不过我们可以注意到，如果一个节点的深度(到根的距离)是dep，那么它tag值的分母就是2^{dep+1}。也就是说，只要点的深度是$O(\log n)$级别，分母就不会太大，就能避免精度问题。

　　因此我们需要使用一个平衡树来维护标记。

标记的重构

考虑使用Treap，如果我们将新插入点x旋转几步往上到了一个新的位置，根据x的新父亲得出x的新的对应区间，由于x的区间改变了，整个以x为根的子树的标记也需要重算。

由于Treap是一个重量平衡树，所以插入只需要$O(\log n)$的时间。

后缀平衡树的构造

那么我们使用Treap来维护所有后缀，并且维护一个tag用来比较后缀的大小，就能做到在最前面插入一个字符$O(\log n)$的复杂度。

这一复杂度是十分优秀的。

例题：SUBSTRING II

我们有一个字符串S，现在需要你支持一下3种操作：

在字符串S的最后插入一个字符c

删去字符串S的最后一个字符

询问字符串q在字符串S中出现了几次

例题：SUBSTRING II

首先注意到该题是在字符串的最后插入或删除一个字符，不过这影响不大，我们可以将整个字符串S反过来，就变成在前面插入或删除一个字符了，此时询问串q也要反过来。

那么插入操作照旧，删除操作也不难实现，我们简单的套用之前提到的Treap的删除算法并维护tag值即可。

考虑询问操作，对于字符串q，出现次数等价于有几个后缀以q开头，我们可以在平衡树上二分算出有几个后缀$< q*$(*符号表示一个虚拟字符，比任何字符都大)，然后再减去有几个后缀$< q$ 就行了。

这样询问的复杂度就是$O(|q|\log n)$。

总结

后缀平衡树相较于其他一些经典的后缀数据结构，拥有几乎是最大的灵活性(可以在末尾插入或者删除字符)，以及最大的可持久化性(可以支持完全可持久化)，对Trie的支持也十分完美。虽然在代码复杂度和时间复杂度上有一定劣势，但可以解决更多的问题。

同时，后缀平衡树最大的优势，在于非常易于理解。让一位同学在10分钟内学会后缀树或者后缀自动机是相当不现实的，因为他们有许多复杂的概念，但是相信大家听了我的答辩之后，已经能够实现一个后缀平衡树了。

总结

而重量平衡树则相较于传统的平衡树，能在节点上维护更加复杂的信息，从而解决一些原来只能使用块状链表解决的问题，大大降低时间复杂度。

本次对于两者的介绍，旨在抛砖引玉，希望能够引发广大同学们的兴趣，进而让这些优秀的数据结构得以发扬光大。

感谢

感谢黄嘉泰同学对我的帮助。

感谢CCF为我们提供的这次宝贵的机会。

感谢教练们的指点。

感谢你们的聆听。