重量平衡树和后缀平衡树在信息学奥赛中的应用

杭州外国语学校 陈立杰

摘 要

重量平衡树是一种具有研究和实际价值的平衡树概念，后缀平衡树是一种基于重量平衡树的非常有价值的后缀数据结构，然而在近几年的OI活动中很少看到他们的身影，本文作者为了改善这一状况，从多个角度详细介绍了重量平衡树的概念和各个种类，并且选取了一些经典的应用加以深入探讨。

1 本文结构

本文先简单的介绍重量平衡树的概念，并且将介绍一些重量平衡树。

[1] Treap

[2] Skip-List(跳表)

[3] Scapegoat Tree(替罪羊树)

之后本文将介绍几个重量平衡树的应用

[1] 动态区间k大询问

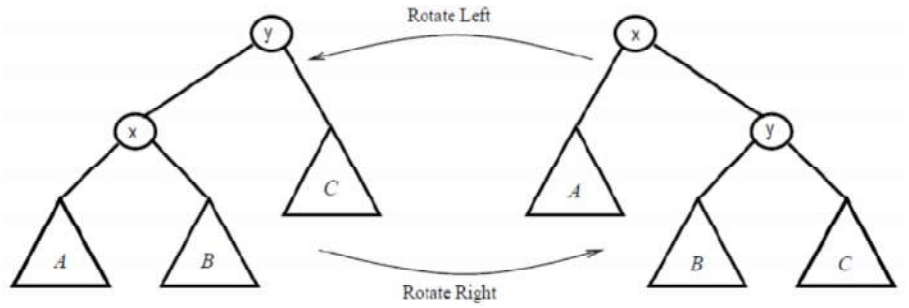
[2] 序列顺序维护问题

接下来本文将介绍可持久化重量平衡树。

最后是本文的重点，介绍使用重量平衡树来维护后缀平衡树这一数据结构，并解决一些问题。

2 重量平衡树的概念

一般的平衡树依赖于一个旋转操作，旋转操作在统计一些可以快速合并的信息的时候是没有问题的，但是对于更加复杂的信息，就会遇到复杂度的问题。



2.1 旋转机制的弊端

考虑一个简单的例子，我们想在一个平衡树的每个节点上维护一个集合，存储子树内部所有的数，此时一次旋转操作的代价可能会达到$O(n)$，传统的旋转平衡树就无法发挥作用了。

而重量平衡树则不同，他每次操作影响的最大子树的大小是最坏或者均摊或者期望的$O(\log n)$。这样即使按照上面所说的方式维护，复杂度也是可以接受的。

2.2 不采用旋转机制的几种重量平衡树

有一些平衡树(或者能实现传统平衡树操作的数据结构)并不依赖于旋转机制，因此也就不会受到旋转机制弊端的影响。

本文选取在OI中有一定实际价值的两者加以介绍。

2.2.1 跳表

关于跳表，已经有国家集训队的前辈在论文中做了详细的讲述，因此本文也不再赘述，有兴趣可以参考附录中的”线段跳表——跳表的一个拓展”。

2.2.2 替罪羊树

替罪羊树依赖于一种暴力重构的操作。

我们定义一个平衡因子a，对替罪羊树的每个节点t，我们都需要满足$\max(size_l, size_r) \leq a \cdot size_t$($l, r$分别表示左右子树)。

他的实现方式也非常的简洁明了，每次我们进行操作之后，找出往上最高的不满足平衡性质的节点，将它暴力重构成完全二叉树。

2.2.3 替罪羊树的复杂度

让我们考虑一个已经被重建过了的子树t，经过计算我们可以发现至少要在里面插入$\Omega(|t|)$ 个节点，他才会被重构（整体的重构，子孙的重构不考虑）。

同时重构的代价是$|t|$的，不妨每次插入一个点的同时，顺便给它的所有祖先都添上1的势能，那么当每个节点被重构的时候，它上面就会有足够的势能来进行重构。

那么插入的均摊复杂度就是$O(\log n)$了。

删除的分析于此基本一样，就留给读者思考了。

同时由于平衡性质，我们可以证明替罪羊树的高度是$\log_{\frac{1}{a}} n$的。那么查找操作自然也是$O(\log n)$ 的了。

2.2.4 平衡因子a的选取

我们可以注意到平衡因子a在略大的情况下，重构操作会变少，因此插入的时间会有所降低，但是树高也因此变大，查找时间会增大。

在小的情况下，重构操作会增多，因此插入的时间会增大，但是树高因此变小，查找时间也变少了。

一般情况下取0.6到0.7左右的平衡因子就能满足大部分的需求了。

2.2.5 替罪羊树的优缺点

替罪羊树虽然不基于旋转机制，但是其思路非常清晰，代码量非常的小，在速度上也不慢，无论是时间复杂度，实现复杂度和思维复杂度都不输给Treap以及Splay这样的传统平衡树。在实际比赛中使用是很明智的选择。

当然替罪羊树由于其平衡机制的限制，并不能支持一些非常复杂的操作，比如Splay中常见的提取一个区间等等。

同时由于他是一个势能的均摊结构，也无法简单的进行可持久化。

并且他在实现中也需要记录$size$域来表示子树大小，当然这在OI中并不是什么大问题。

总体来说替罪羊树在简单的平衡树应用上非常出色。

2.3 采用旋转机制的重量平衡树

即使采用旋转机制，也有很多平衡树是重量平衡的。可以这么理解，大部分旋转都是所谓的"微调"，只在底下发生，所以子树不会太大，高处的旋转则较少发生。所以总体均摊或期望复杂度任然是$O(\log n)$。

2.3.1 Treap

Treap是一个非常常见的数据结构。

顾名思义，Treap=Tree+Heap，它对每个点维护一个随机权值，同时使得这棵树的形状就跟按随机权值顺序从小到大插入的普通BST一样。

我们都知道一个随机数据的情况下，普通BST的期望高度是$O(\log n)$的。那么Treap的期望高度也是$O(\log n)$的，并且跟输入无关。

Treap的一系列操作都是很经典的算法，再次就不再赘述了。

2.3.2 Treap的重量平衡

当我们插入一个数的时候，可能会造成一些旋转，不妨设插入点t转到了祖先k的位置，那么我们需要使用$size_k$的时间来重构，但这种情况发生的条件是点t的随机权值是k子树中最小的一个，那么概率就是$\frac{1}{size_k}$。那么期望的代价就是1。同时一个点最多有$O(\log n)$个祖先，那么总共一次插入的期望代价就是$O(\log n)$。

接下来我们考虑删除一个数x。我们不妨直接重构以该树为根的子树。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)关系总共会有$O(n \log n)$ 对，也就是说每个点的子树的大小的和期望是$O(n \log n)$。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$O(\log n)$。

3 重量平衡树的应用

那么我们来看重量平衡树的几个应用。以此来强调重量平衡树的作用。

3.1 序列顺序维护问题

3.1.1 问题描述

我们现在有一个节点序列，要支持如下两种操作：

[1] 在节点x的后面插入一个新节点y

[2] 询问节点a,b的前后关系。

3.1.2 解法1

我们可以使用简单的平衡树来维护这个序列，插入就直接插入，询问大小关系只要求出两个节点各自的$rank$就能简单实现了。

插入复杂度$O(\log n)$。

询问复杂度$O(\log n)$。

3.1.3 解法2

我们仍然使用平衡树来维护这个序列，不同的是我们对每个点维护一个标记tag_i，询问a,b前后关系时只需要比较tag_a,tag_b的大小就行了。

如何维护这个标记呢，我们不妨让每个点对应一个实数区间，让根对应实数区间$(0,1)$，对于对应实数区间(l,r)的节点i，它的tag_i是$\frac{l+r}{2}$，它的左子树的实数区间是(l,tag_i)，右子树是(tag_i,r)，容易发现这样的tag_i是满足要求的。

那么我们比较的时候只需要比较tag_i就可以了，同时注意到在这里每个点tag_i值的分母是2^{depth_i+1},$depth_i$表示i的深度也就是到根的距离。

那么只要树是平衡的，也就是最深深度不大，精度是有保证的。使用double就可以了。

那么在插入的时候，我们可能要对一整个子树的tag值进行重新计算，只要使用重量平衡树，就能做到$O(\log n)$的复杂度。

询问复杂度是$O(1)$。

3.1.4 解法3

本问题是一个经典问题，黄嘉泰同学查阅了论文并发现存在一个插入和询问都是$O(1)$的优秀算法，不过跟本文主题无关，有兴趣的同学参加附录中的“Two Simplied Algorithms for Maintaining Order in a List”。

3.2 动态区间第k大

3.2.1 题目描述

我们现在有一个长度为n的序列s，要求支持几个操作：

[1] 在第i个数的后面插入数x

[2] 修改s_i的值为v

[3] 删除第i个数

[4] 询问区间$[l,r]$内的第k大数是多少。

3.2.2 题目解法1

我们使用重量平衡树进行维护，每个节点用一颗平衡树保存其子树的中的所有数。

那么插入删除修改都可以在$O(\log^2 n)$的时间内解决。

对于询问操作我们先二分答案a，然后就可以在$O(\log^2 n)$的时间内回答区间中比a小的数有几个，总复杂度就是$O(\log^3 n)$。

3.2.3 题目解法2

我们可以在每个点维护一个权值线段树，从而将复杂度改进到$O(\log^2 n)$。

关于权值线段树，可以参考我去年的论文，附录中的”可持久化数据结构研究”，里面有详细的讲解。

3.2.4 题目解法3

我们另外提出一个有意思的解法，我们不妨对所有权值整体开一个线段树。在权值区间$[l,r)$保存所有权值在$[l,r)$之间的数的位置组成的平衡树。

回答询问的时候就在权值线段树上走一遍，复杂度是$O(\log^2 n)$。

插入数的时候，我们需要在序列中插入该数，同时线段树中包含该数的节点中插入它。

此时我们需要注意到，因为插入了一个数，它后面所有数的位置都+1了，我们显然无法直接修改它们的位置，但是注意到我们并不需要的它们的确切位置！在平衡树中维护只需要能快速比较两个位置前后顺序就可以了！

因此我们使用序列顺序维护问题的解法来维护这个序列，这样我们就能快速比较两个数的前后关系，从而完成平衡树的各种操作。

那么其它操作的复杂度也是$O(\log^2 n)$。

3.3 动态K-D树

K-D树是一个很经典的几何数据结构并支持很多操作。

首先简单回顾一下K-D树的概念。

K-D树依赖于对平面所有点的不断划分，我们每次将所有点按照X或Y轴排序，按照左右或者上下分成尽量均匀的两部分，然后递归划分。

划分的轴的顺序是交替的，首先X轴然后Y轴然后依次类推。

假设我们现在有了个K-D树，我们需要支持插入一个点的操作。

如果我们直接暴力查找点的位置然后插入，K-D树的性质就会被破坏。

我们可以仿照替罪羊树的思想，设定一个平衡因子a，令一个划分最多的部分不能超过a*总大小，这样就能保证插入和询问的复杂度了。

3.4 MAXPATH

这是一道我的原创题

3.4.1 题目描述

平面上有n个点，两个点p_i,p_j之间有有向边的条件是$i<j,|p_ip_j|<R_j$。

求一条最长的路径。

$n \leq 50000$。

3.4.2 解法1

此题有一个基于分治和V图的$O(n \log^3 n)$解法，常数和代码量都十分惊人。

由于跟本文主题无关，就不再赘述，感兴趣的同学可以去翻阅WC2013顾昱洲的讲课课件。

3.4.3 解法2

首先我们进行DP，令dp_i表示以i为结尾的最长链的长度是多少。

那么可以写出DP方程$dp_i = \max dp_j (j < i, |p_i p_j| < R_i)$。

考虑这个式子，第一个想法是维护一个数据结构，支持插入一个带权点和询问一个圆内部最大的点权。

但是这是非常困难的问题。我们不妨换一个思路，我们将之前所有的$j < i$按照dp_j值的顺序排序。

然后我们依次从大到小找出第一个与i的距离小于R_i的。

这样复杂度还是$O(n^2)$的，不过我们注意到此时问题就变得可以套用之前提到过的数据结构了。

我们使用重量平衡树来维护所有j按照dp_j的排序，在节点上维护一个K-D树保存节点内部所有的点。

那么回答询问时，我们在K-D树上询问其中离i最近的点的距离是否小于R_i，就能知道该往哪个孩子走了。

算出dp_i之后我们就直接插入平衡树中就可以了，由于也需要在K-D树中插入，我们还需要使用前面提到的动态K-D树。

K-D树询问最近点的代价可以看作是$O(\sqrt{n})$的，那么总共的复杂度就是$O(n^{1.5} \log n)$。虽然复杂度相较解法1变高了，但是常数和代码复杂度就变少了很多(V图的代码量和代码复杂度相比K-D树非常大)。

4 可持久化重量平衡树

我们来考虑可持久化重量平衡树。由于Treap本身就是重量平衡的，我们只需要实现一个可持久化的Treap就可以了。

关于可持久化Treap的总总细节，可以参考我去年的论文“可持久化数据结构研究”，在此就不再赘述了。

时间复杂度和空间复杂度都是每部期望$O(\log n)$。

替罪羊树由于它是一个均摊的数据结构，如果我们只需要部分可持久化(只能修改当前结构，对历史结构只进行询问)，是可以通过简单的可持久化平衡树方法实现的。

但是如果要实现完全可持久化(对历史结构也要支持修改)，则因为他是均摊$O(\log n)$的，如果一个高代价的操作进行多次，复杂度就被破坏了。

5 后缀平衡树

5.1 后缀平衡树的定义

考虑一个长度为n的字符串s，定义s^i为其由$s_i, s_{i+1}, \cdots, s_n$组成的后缀。

后缀之间的大小由字典序定义，后缀平衡树就是一个平衡树并维护这些后缀的顺序。

5.2 后缀平衡树的构造

接下来我们从在线或离线的角度分别考虑如何构造后缀平衡树。

5.2.1 离线算法

给定一个字符串s，我们先求出它的后缀数组也就是所有后缀的排序，然后根据后缀数组很容易就能构造后缀平衡树。

复杂度$O(n)$或$O(n \log n)$,取决于你使用的求后缀数组的算法。

5.2.2 在线算法

跟后缀树和后缀自动机相同，后缀平衡树也有一个一个个添加字符的在线算法，也因此能解决一些单靠后缀数组无法解决的问题。

跟后缀树和后缀自动机相反的是，后缀平衡树的在线算法不是在字符串最后，而是在字符串开头添加字符。

让我们考虑当前字符串s，并且在其开头加入一个字符c。

那么可以发现实际上就等于在后缀平衡树中插入了一个新的后缀cS。

如果我们能快速比较两个后缀的大小，那么直接套用平衡树的插入算法就行了。

5.2.3 哈希

首先我们可以使用最经典的哈希法，我们预处理一下字符串的哈希，然后就能通过二分在$O(\log n)$ 的时间内求出两个后缀的LCP，从而比较它们的大小。

此时时间复杂度是$O(n \log^2 n)$的，代码复杂度很小，比较好写。

5.2.4 套用序列顺序问题

注意到如果我们使用之前提到的序列顺序问题的解法，就能在$O(1)$的时间内比较两个后缀的大小了。

这样复杂度就变成了$O(n \log n)$。较哈希法有了很大的改进。

5.2.5 可持久化

我们考虑对后缀平衡树进行可持久化，只要我们使用Treap，可以很容易的将算法扩展成为可持久化算法，从而实现可持久化后缀平衡树。

此时时间复杂度不变，空间复杂度变为$O(n \log n)$。

5.3 后缀平衡树的优势

首先后缀平衡树在概念上只是后缀数组和平衡树的一个综合应用，相比复杂的后缀自动机和后缀树更加好理解。

代码复杂度则相对大一些，但是思路很清楚，不容易写错。

同时它相比后缀自动机和后缀树，还支持一些其它的操作。

比如后缀自动机和后缀树虽然都可以在末尾加入一个字符，但是它们无法支持在末尾删除一个字符的操作，相比之下后缀平衡树就可以实现这个操作。

同时后缀平衡树还可以支持动态维护一个Trie的所有后缀，这是后缀自动机不能高效实现的。

还有就是后缀平衡树完全不依赖于字符集的大小，而后缀自动机和后缀树的简单实现都需要考虑这点。

最后，后缀平衡树可以支持可持久化，这是后缀自动机和后缀数望尘莫及的。

接下来我们就通过几个例题来表明后缀平衡树的用处。

6 后缀平衡树的应用

6.1 COT4 online

此题改编自黄嘉泰大神的经典题目COT4。(相对于原题这可以说是一个弱化)。

6.1.1 题目大意

给你一个字符串集合S，一开始里面只有一个空串。你需要支持一些操作：

[1] 取出一个$s \in S$，考虑一个字符c将sc和s都放入集合S。

[2] 对于一个询问串q和一个集合中的字符串$s \in S$，输出q作为s的子串出现了几次。

操作次数$n \leq 200000$,询问串长度和$Q \leq 500000$。

6.1.2 题目解法

其实我们注意到，将sc放入集合S，实际上就是在s后面添加了一个字符，那么我们使用可持久化后缀平衡树维护s的所有后缀，添加时进行可持久化插入即可。

然后是询问一个字符串q出现了几次，注意到这个实际上就是所有字典序比$q\#$小的后缀的数量减去比q小的后缀的数量，$\#$代表一个虚拟的最大字符。

那么我们在平衡树上二分比较查找即可，复杂度是$O(|q| \log n)$，可以接受。

通过维护一些附加信息可以将询问复杂度改进到$O(|q| + \log n)$，不过较为复杂。

操作复杂度是$O(\log n)$，总体复杂度就是$O((n+Q)\log n)$。

6.2 STRQUERY

此题是我的原创题。

6.2.1 题目大意

我们有一个字符串s，我们需要支持如下4个操作:

[1] 在最左端插入或删除一个字符

[2] 在最右端插入或删除一个字符

[3] 在正中间($\frac{|s|}{2}$处)插入或删除一个字符。

[4] 询问字符串q出现了几次。

一共有$n(n \leq 150000)$个操作，所有询问的长度和≤ 1500000。

6.2.2 题目解法

首先，我们的后缀平衡树可以支持在最左端插入或删除一个字符，令这个为结构TL。

那么我们将这个整个反过来，并且将询问串也反过来，就能支持在最右端插入或删除一个字符，令这个为结构TR。

我们现在考虑来实现一个可以在两端删除或插入的结构，我们不妨维护一个结构TL，一个结构TR，并让结构$TB = TL + TR$，那么插入删除分别在TL和TR做，如果其中一个被删到了0个，那么我们可以把另外一个均匀的分成两半，从均摊意义上这个操作不影响复杂度。

同时在TB上询问，我们可以先询问q在TL和TR中出现了几次，然后再询问它在跨越的部分出现了几次，注意到跨越部分长度最多为$2|q|-2$，直接取出进行KMP即可。

然后我们来实现一个可以在正中间插入的结构TM，我们维护两个TB : l, r，每次插入之后在l和r之间交换几个字符，使得l后面那个位置始终是正中间。这样就能在正中间插入了。

询问跟在TB上询问一样。

那么我们就得出了一个支持此题的结构TM，复杂度是$O(n \log n)$。

感谢

黄嘉泰同学对我的帮助。
CCF为我们提供的这次宝贵的机会。
以及教练们的指点。

参考文献

[1] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[2] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

[3] Michael A. Bender, Richard Cole, Erik D. Demaine,Martin Farach-Colton, and Jack Zito, “Two Simplied Algorithms for Maintaining Order in a List”.

[4] 李骥扬,《线段跳表——跳表的一个拓展》, 国家集训队2009论文。

[5] 陈立杰,《可持久化数据结构研究》, 国家集训队2012作业。