对无旋转操作的平衡树的一些探究

武汉市第二中学 吕凯风

February 18, 2013

1 简介

一般使用的平衡树，比如Splay树、红黑树，Treap，还有SBT等，都是通过旋转来保持平衡。为什么一定是用旋转操作呢？本文从不使用旋转操作出发，探究了朝鲜树和替罪羊树两种数据结构的性质与应用。

2 符号

为了说话方便，我们定义一些符号作为辅助。

2.1 对于二叉树中的结点v

$left(v)$: 结点v的左孩子
$right(v)$: 结点v的右孩子
$size(v)$: 结点v为根的子树包含的结点数
$depth(v)$: 结点v的深度。即从树根到v的距离
$height(v)$: 结点v为根的子树的高度。即最长的一条从v到叶子的路径的长度

2.2 对于二叉树

n: 整棵树包含的结点数
$root$: 树的根结点
h: 整棵树的高度。即$height(root)$

3 朝鲜树

3.1 引子

一棵裸的BST当然是不用旋转的，可是效率是很低的。如何避免旋转操作来维持BST的平衡呢？

我的同学金正中，想出了一个解决方案。由于没有搜到类似的数据结构，暂且称之为“朝鲜树”。

3.2 不带删除操作的情况

由于删除操作有点复杂，我们先不考虑删除操作。

朝鲜树的查询和裸BST是一样的，只有插入元素的时候略有不同。

设定一个值l，如果插入后产生的新结点的深度大于l，则把整棵树暴力重建成平衡的树结构。

简单的分析可知，任何时候一定有$h \leq l$，否则就会导致一次重建。所以每次查询操作的时间复杂度为$O(l)$。最主要的是插入的复杂度。如果没有导致重建，则时间复杂度为$O(l)$，而如果导致重建，则有额外的重建时间复杂度$O(n)$。最坏情况下，上一次重建至这一次重建，插入的结点形成了一条链，其间经过了约$l - log(n')$次插入。这里的n'表示的是上一次重建时的n。这$l - log(n')$次插入中有一次重建，均摊下来每次插入的时间复杂度增加了$O(\frac{n}{l-log(n')})$。

如果取$l = \sqrt{n}$，则能获得较好的时间复杂度：

查询 ：$O(\sqrt{n})$

插入 ：均摊$O(\sqrt{n})$

3.3 删除操作

如果有删除操作，则执行删除操作后l会减小，导致朝鲜树进行一些不必要的重建。处理的方法是再开一个域$maxn$记录自上次重建以来n的最大值，在每次插入时更新$maxn$的值。并将l定为$\sqrt{maxn}$，这样我们前面的复杂度分析依然适用。所以删除的时候就像裸BST一样删除就行了。

删除 ：$O(\sqrt{n})$

3.4 实际测试

虽然朝鲜树不像其它的平衡树那样所有操作都是$O(log(n))$的，严格来说朝鲜树甚至不能被称为“平衡树”。但在实际测试中，朝鲜树的表现并不坏。因为对于随机数据，朝鲜树就等同于一棵裸BST，树高期望是$O(log(n))$。只有碰见单调插入这类极端数据，朝鲜树才会显露出它带根号的一面。

4 替罪羊树

4.1 引子

通过查阅资料[1]，我了解到了替罪羊树这一数据结构。替罪羊树是无旋转操作的平衡树，但是它的所有操作都是$O(log(n))$的。这不由得使我感到好奇：朝鲜树的l取到$\sqrt{maxn}$已经做到极致了，进一步优化的余地在哪里？耐心阅读论文，我明白了替罪羊树的精髓所在：部分重建。

4.2 定义平衡

为了介绍替罪羊树，我们首先来重新看待平衡。有两种平衡：

α高度平衡 如果一棵二叉树T的高度h满足：

$$h \leq log_{1/\alpha}(n) \tag{1}$$

则称T是α高度平衡的。

α大小平衡 一个结点v满足：

$$size(left(v)) \leq \alpha \cdot size(v) \tag{2}$$
$$size(right(v)) \leq \alpha \cdot size(v) \tag{3}$$

则称v是α大小平衡的。若一棵二叉树T的任意一个结点v都是α大小平衡的，则称T是α大小平衡的。

当然，由定义可知，当$\alpha < \frac{1}{2}$ 或 $\alpha \geq 1$时是毫无意义的，故规定：

$$\frac{1}{2} \leq \alpha < 1 \tag{4}$$

4.3 不带删除操作的情况

替罪羊树的查询和裸BST是一样的，也只有插入元素的时候略有不同。

设定一个值α，插入时像BST一样递归进行插入，如果插入得到的结点v满足：

$$depth(v) > log_{1/\alpha}(n) \tag{5}$$

则说明这棵树失去了α高度平衡。那么在回溯的时候，找到第一个非α大小平衡结点u，将以u为根的子树重建成$1/2$大小平衡的。结点u被称为替罪羊结点。

为什么一定会存在一个替罪羊结点呢？ 我们可以使用反证法证明。如果某个结点x有一个孩子结点y，且满足x是α大小平衡结点。有定义可知$size(y) \leq \alpha \cdot size(x)$。通过归纳法可知，如果一个结点$v$到$root$的所有结点都是$\alpha$大小平衡结点，那么一定有：

$$size(v) \leq \alpha^{depth(v)} \cdot n \tag{6}$$

所以如果v是叶节点，那么必有：

$$depth(v) \leq log_{1/\alpha}(n) \tag{7}$$

与假设矛盾，故一定存在替罪羊结点。

由此，替罪羊总能保持$h \leq log_{1/\alpha}(n)$。查询操作毋庸置疑是$O(log(n))$的，可是插入操作该如何分析呢？

我们可以对一个结点x定义势能函数：

$$\Delta(x) = \begin{cases} |size(left(x)) - size(right(x))| & \text{if } |size(left(x)) - size(right(x))| > 2 \\ 0 & \text{else.} \end{cases} \tag{8}$$

而一棵树的势能定义为所有结点的势能函数之和。

那么可以推得，递归插入时结点使得所有祖先的势能都增加或减少了一个常数。如果发生重建，则重建前替罪羊结点u为根的子树的势能是$\Omega(size(u))$的。而发生重建后的这棵子树的势能为0，故所释放的势能能够支付重建的代价$O(size(u))$。由此证明了插入的时间复杂度为$O(log(n))$。

查询 ：$O(log(n))$

插入 ：均摊$O(log(n))$

4.4 删除操作

删除操作我们仍然和处理朝鲜树一样遇到了麻烦，不过没关系，我们可以如法炮制解决方案。处理的方法仍然是再开一个域$maxn$记录自上次重建以来n的最大值，在每次插入时更新$maxn$的值。而上文所述的所有$log_{1/\alpha}(n)$都用$log_{1/\alpha}(maxn)$替代即可。

删除 ：$O(log(n))$

5 重建操作

朝鲜树和替罪羊树都有一个核心操作：重建。那么如何快速重建呢？

5.1 简单方法

对这棵树进行中序遍历，求出中序遍历序列，然后分治建树即可。这个需要一个辅助的数组记录中序遍历序列。

空间 ：$O(n)$

5.2 拍扁重建法

下面主要要介绍的是拍扁重建法。拍扁重建法一共分两步，第一步称为拍扁，即将这棵树拍扁成为链表，链表的各个结点通过原结点的$right$连接起来。如下伪代码展示了拍扁的过程。其中$flatten(x, y)$表示将以x为根的子树拍扁，并在最后添加一个节点y，返回这条链的第一个结点。

```
flatten(x, y)
{
    if (x == null)
        return y;
    right(x) = flatten(right(x), y);
    return flatten(left(x), x);
}
```

第二步为建树，即将链表建成一棵树。如下伪代码展示了建树的过程。其中$build(x,n)$表示将x及后面的结点，一共n个，重建成一棵树，返回x后面第$n+1$个结点z，并使$left(z)=$这棵树的根结点。

```
build(x, n)
{
    if (n == 0)
    {
        left(x) = null;
        return x;
    }
    y = build(x, n / 2);
    z = build(right(y), (n - 1) - n / 2);
    right(y) = left(z);
    left(z) = y;

    return z;
}
```

这样我们就能得到重建操作。如下伪代码展示了重建的过程，其中$rebuild(x)$表示将以x为根的子树重建。由于$build(head,n)$会将重建后的树的根结点保存在链上最后一个结点的左孩子处，故先在$flatten$时插入额外结点t，则最后树根就会保存在$left(t)$处了。

```
rebuild(x)
{
    n = size(x);
    t = new node;
    head = flatten(x, t);

    build(head, n);
    x = left(t);
}
```

可以发现除了递归栈外，我们耗费的其它空间都是$O(1)$的。由于替罪羊树树高是$O(log(n))$的，所以递归栈耗费的空间也是$O(log(n))$。可见拍扁重建法是很令人满意的重建算法。

空间 ：$O(log(n))$

6 应用

利用替罪羊树的思想，可以优化无法旋转的树形结构的时间复杂度。

6.1 k-d树

k-d树无法旋转，但是它能在$O(h)$时间内插入，在$O(k \cdot Llog(L))$时间内重建大小为L的子树。我们可以利用替罪羊树的思想将其优化到均摊$O(log^2(n))$插入，而单点查询仍是$O(log(n))$。

6.2 平衡树套线段树

平衡树套线段树通常被认为是不可写的，因为树套树导致平衡树无法旋转。但是我们可以写替罪羊树套函数式线段树。每个替罪羊结点上都带有一棵函数式线段树，故如果n是替罪羊树的结点个数，m是线段树维护的序列长度，它就能在$O(log(n)log(m))$时间内插入，利用线段树合并[2]我们可以在$O(Llog(m))$时间内进行重建。从而我们可以利用替罪羊树的思想将其优化到均摊$O(log(n)log(m))$插入，而查询仍是$O(log(n)log(m))$。

参考文献

[1] 《Scapegoat Trees》 Igal alperin, Ronald L.Rivest

http://www.akira.ruc.dk/~keld/teaching/algoritmedesign_f07/Artikler/03/Galperin93.pdf

[2] 《线段树合并》 杭州二中 黄嘉泰

http://pan.baidu.com/share/link?shareid=249380&uk=4161988545