

Leafy Tree 及其实现的加权平衡树

成都市第七中学 王思齐

摘要

二叉树与平衡树是信息学竞赛中重要的数据结构。本文介绍了 Leafy Tree，这是一种能实现大多数基于二叉树的数据结构的数据结构。本文描述了其实现二叉搜索树和加权平衡树的方法，并给出了这两种数据结构与其他数据结构的比较。

1 什么是 Leafy Tree

Leafy Tree 是一种二叉树，其每个节点要么为叶子，要么有两个儿子。其信息完全储存在叶子上面，每个非叶节点存储的信息是其儿子的信息的合并。它的结构和线段树类似，同时也可以将其看作是二叉搜索树的 *kruskal* 重构树。

这是一个纯函数化的数据结构，可以方便高效地实现其他的函数化数据结构，同时也可以方便的实现可持久化。

2 Leafy Tree 实现二叉搜索树

2.1 定义

考虑用一棵 Leafy Tree 维护一个集合。每个叶子节点存储集合中一个值，每个非叶节点保存它右儿子的值（即子树最大值）。同时满足每个非叶节点左儿子的最大值不大于右儿子的最小值，即这棵树的中序遍历中，叶节点的值是有序的。我们把这样的一棵 Leafy Tree 称为 Leafy Tree 实现的二叉搜索树。

2.2 基本操作

为了方便描述，我们用 $A.left, A.right, A.value$ 表示节点 A 的左儿子、右儿子和值。对于叶节点 X ，定义 $X.left = X.right = null$ 。同时定义函数 $newNode(x)$ 用来新建一个值为 x 的节点，并将新建节点的左右儿子设为 $null$ ， $deleteNode(x)$ 用来回收节点 x 。

2.2.1 查询

在节点 T 的子树中查找值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果值相等，则查找成功，否则查找失败；

否则若 x 不大于 T 的左子树的值，搜索左子树；

否则搜索右子树。

伪代码见下面算法 1。

算法 1 Leafy Tree 实现二叉搜索树的查询操作

Find 函数参数为节点 T 、查询的数 x ，返回查询结果。

```
1: function Find( $T, x$ )
2:   if  $T.left = null$  then
3:     if  $T.value = x$  then
4:       return  $True$ 
5:     else
6:       return  $False$ 
7:     end if
8:   else
9:     if  $x \leq T.left.value$  then
10:      return Find( $T.left, x$ )
11:    else
12:      return Find( $T.right, x$ )
13:    end if
14:  end if
15: end function
```

2.2.2 插入

在节点 T 的子树中插入值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果 x 大于 T 的值，则新建节点 A, B 来存放 T 的值，并将 T 的左右儿子设为 A 和 B ； x 不大于 T 的值时操作类似；

否则若 x 不大于 T 的左子树的值，将 x 插入左子树；

否则将 x 插入右子树。

伪代码见下面算法 2。

算法 2 Leafy Tree 实现二叉搜索树的插入操作

Insert 函数参数为节点 T 、插入的数 x 。

```
1: function Insert( $T, x$ )
```

```

2:  if  $T.left = null$  then
3:       $A \leftarrow \text{newNode}(T.value)$ 
4:       $B \leftarrow \text{newNode}(x)$ 
5:      if  $A.value > B.value$  then
6:           $\text{swap}(A, B)$ 
7:      end if
8:       $T.left \leftarrow A$ 
9:       $T.right \leftarrow B$ 
10:      $T.value \leftarrow B.value$ 
11:  else
12:      if  $x \leq T.left.value$  then
13:           $\text{Insert}(T.left, x)$ 
14:      else
15:           $\text{Insert}(T.right, x)$ 
16:      end if
17:       $T.value \leftarrow T.right.value$ 
18:  end if
19: end function

```

2.2.3 删除

在节点 T 的子树中删除值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果 x 与 T 的值相等，则删除成功，并将 T 父亲的所有属性设为 T 兄弟的，然后删除 T 的兄弟节点与 T 本身；否则删除失败；

否则若 x 不大于 T 的左子树的值，在左子树删除；

否则在右子树删除。

伪代码见下面算法 3。

算法 3 Leafy Tree 实现二叉搜索树的删除操作

Remove 函数参数为节点 T 、删除的数 x ，返回是否删除成功。这个函数要求 T 为非叶节点。

```

1: function Remove( $T, x$ )
2:   if  $x \leq T.left.value$  then
3:       if  $T.left.size = 1$  then
4:           if  $T.left.value = x$  then
5:                $temp \leftarrow T.right$ 
6:                $T = T.right$ 

```

```

7:         deleteNode(temp)
8:         deleteNode(T.left)
9:         return True
10:    else
11:        return False
12:    end if
13: else
14:     return Remove(T.left, x)
15: end if
16: else
17:     if T.right.size = 1 then
18:         if T.right.value = x then
19:             temp  $\leftarrow$  T.left
20:             T = T.left
21:             deleteNode(temp)
22:             deleteNode(T.right)
23:             return True
24:         else
25:             return False
26:         end if
27:     else
28:         temp  $\leftarrow$  Remove(T.right, x)
29:         T.value  $\leftarrow$  T.right.value
30:         return temp
31:     end if
32: end if
33: end function

```

虽然这里伪代码较长，但是实际实现上，如果保证被删除数存在，并用数组来存储每个节点的两个儿子，可以节省大量代码。

2.3 和二叉搜索树的比较

可以发现，由于用到了只有叶子节点维护信息的性质，Leafy Tree 进行插入删除相当方便。每次插入删除的节点实际上都是叶子节点，大量减少了普通二叉搜索树结构中对插入删除的节点类型的繁杂讨论（即这个节点是叶子节点，还是有一个儿子，还是有两个儿子）。同时由于每个非叶子节点维护了区间信息，通过左儿子的区间信息来进行定位，本质

上和二叉搜索树通过与树根节点的值比较进行定位相同，所以是以更加简洁的实现达到了同样的效果。

3 Leafy Tree 实现加权平衡树

3.1 定义

加权平衡树（Weight Balanced Tree，也叫 $BB[\alpha]$ 树，重量平衡树）是一种储存子树大小的二叉搜索树。即一个结点包含以下字段：值（value）、左儿子（left）、右儿子（right）、子树大小（size）。

在 Leafy Tree 实现的加权平衡树中，子树大小为子树中叶节点的个数，非叶节点的值为其右儿子的值。即对于一个叶节点 x ， $x.size = 1$ ；对于一个非叶节点 x ， $x.size = x.left.size + x.right.size$ ， $x.value = x.right.value$ 。

一个结点的权重 $weight$ 取决于它的子树大小或者等于它的子树大小，在 Leafy Tree 实现的加权平衡树中，对于一个节点 x ， $weight[x] = x.size$ 。如果一个节点 x 满足 $\min(weight[x.left], weight[x.right]) \geq \alpha \cdot weight[x]$ ，则称这个节点是 α 加权平衡的，显然 $0 < \alpha \leq \frac{1}{2}$ 。一棵含有 n 个元素的加权平衡树的高度 h 满足 $h \leq \log_{\frac{1}{1-\alpha}} n = O(\log n)$ 。

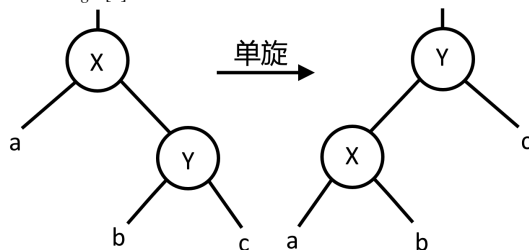
3.2 实现方式

加权平衡树有两种实现方式，重构和旋转。这里只介绍旋转平衡。

对于一棵满足 α 加权平衡的树，在插入或删除一个节点后，不满足 α 加权平衡的节点一定在一条链上。考虑依次处理这条链上的节点。

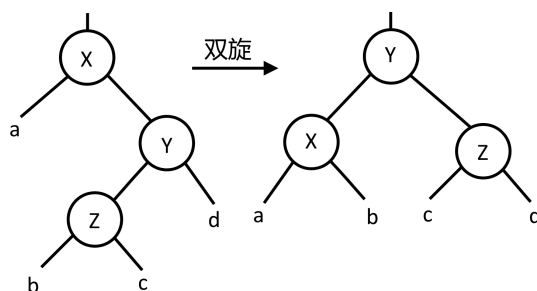
假设这棵树的一个子树 T 中，除根节点外的所有节点均满足 α 加权平衡。我们可以用一次单旋或双旋操作使 T 满足 α 加权平衡。

定义 x 的平衡度表示 $\frac{weight[x.left]}{weight[x]}$ 。



如图，考虑一次单旋，设 X, Y 旋转前平衡度为 ρ_1, ρ_2 ，旋转后为 γ_1, γ_2 。

可以得到 $\gamma_1 = \frac{\rho_1}{\rho_1 + (1 - \rho_1)\rho_2}$ ， $\gamma_2 = \rho_1 + (1 - \rho_1)\rho_2$ 。



如图, 考虑一次双旋, 设 X, Y 旋转前平衡度为 ρ_1, ρ_2, ρ_3 , 旋转后为 $\gamma_1, \gamma_2, \gamma_3$ 。

可以得到 $\gamma_1 = \frac{\rho_1}{\rho_1 + (1 - \rho_1)\rho_2\rho_3}$, $\gamma_2 = \rho_1 + (1 - \rho_1)\rho_2\rho_3$, $\gamma_3 = \frac{\rho_2(1 - \rho_3)}{1 - \rho_2\rho_3}$ 。

假设 $\rho_1 < \alpha$, 在只插入或删除一个节点的情况下, ρ_1 最小为 $\frac{\alpha}{2 - \alpha}$, 可在子树 a 中原有 2 个节点, 现在删除其中一个时取到。

在给定的限制下, 可以证明, 当 $\alpha \leq 1 - \frac{\sqrt{2}}{2}$ 时, 通过这两种操作, 可以使平衡度改变的这几个节点, 其新平衡度在 $[\alpha, 1 - \alpha]$ 内。证明时可能还需要进一步对 ρ_1 进行放缩, 以及小范围的分类讨论。具体证明由于篇幅所限, 不再赘述。

具体操作为, 当 $\rho_2 < \frac{1 - 2\alpha}{1 - \alpha}$ 时, 进行一次单旋, 否则进行一次双旋。

3.3 基本操作

为了方便描述, 在 *left* 和 *right* 之外, 我们同时用 $A.child[0], A.child[1]$ 表示节点 A 的左儿子和右儿子。对于叶节点 X , 定义 $X.child[0] = X.child[1] = null$ 。同时定义函数 $newNode(x)$ 用来新建一个值为 x 的节点, 并将新建节点的左右儿子设为 $null$, $deleteNode(x)$ 用来回收节点 x 。

3.3.1 合并信息

一个节点经过操作后, 需要重新计算其子树大小及值。

伪代码见下面算法 4。

算法 4 Leafy Tree 实现的加权平衡树的信息合并

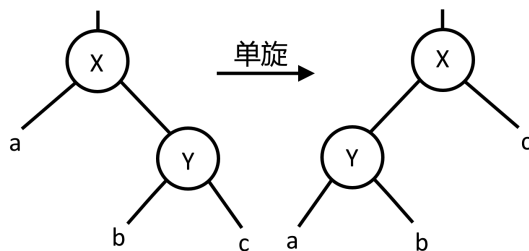
Pushup 函数参数为节点 T 。

```

1: function Pushup( $T$ )
2:   if not  $T.left = null$  then
3:      $T.size \leftarrow T.left.size + T.right.size$ 
4:      $T.value \leftarrow T.right.value$ 
5:   end if
6: end function

```

3.3.2 单旋



如图，这个操作是将 Y 单旋至 X 的位置。

伪代码见下面算法 5。其中 \oplus 表示异或。

算法 5 Leafy Tree 实现的加权平衡树的单旋

Rotate 函数参数为节点 T 和一个 bool 变量 d ，表示是将 $child[d]$ 旋转到 T 的位置。

```

1: function Rotate( $T, d$ )
2:    $temp \leftarrow T.child[d \oplus 1]$ 
3:    $T.child[d \oplus 1] \leftarrow T.child[d]$ 
4:    $T.child[d] \leftarrow T.child[d \oplus 1].child[d]$ 
5:    $T.child[d \oplus 1].child[d] \leftarrow T.child[d \oplus 1].child[d \oplus 1]$ 
6:    $T.child[d \oplus 1].child[d \oplus 1] \leftarrow temp$ 
7:   Pushup( $T.child[d \oplus 1]$ )
8:   Pushup( $T$ )
9: end function

```

3.3.3 维护平衡

一个节点经过操作后，需要一次单旋或双旋来维护平衡。

伪代码见下面算法 6。

算法 6 Leafy Tree 实现的加权平衡树的维护平衡

Maintain 函数参数为节点 T 。

```

1: function Maintain( $T$ )
2:   if not  $T.left = null$  then
3:     if  $T.left.size < T.size \cdot \alpha$  then
4:        $d \leftarrow 1$ 
5:     else if  $T.right.size < T.size \cdot \alpha$  then
6:        $d \leftarrow 0$ 
7:     else  $T.right.size < T.size \cdot \alpha$ 
8:       return
9:     end if

```

```

8:   if  $T.child[d].child[d \oplus 1].size \geq T.child[d].size \cdot \frac{1-2\alpha}{1-\alpha}$  then
9:       Rotate( $T.child[d], d \oplus 1$ )
10:  end if
11:  Rotate( $T, d$ )
12:  end if
13: end function

```

3.3.4 插入

与 Leafy Tree 实现的二叉搜索树类似。

伪代码见下面算法 7。

算法 7 Leafy Tree 实现的加权平衡树的插入操作

Insert 函数参数为节点 T 、插入的值 x

```

1: function Insert( $T, x$ )
2:   if  $T.size = 1$  then
3:        $T.left \leftarrow newNode(x)$ 
4:        $T.right \leftarrow newNode(T.value)$ 
5:   if  $x > T.value$  then
6:       swap( $T.left, T.right$ )
7:   end if
8:   else
9:       Insert( $T.child[x > T.left.value], x$ )
10:  end if
11:  Pushup( $T$ )
12:  Maintain( $T$ )
13: end function

```

3.3.5 删除

与 Leafy Tree 实现的二叉搜索树类似。

伪代码见下面算法 7。

算法 7 Leafy Tree 实现的加权平衡树的删除操作

Remove 函数参数为节点 T 、删除的值 x 。这个函数要求 T 为非叶节点。

```

1: function Remove( $T, x$ )
2:    $d \leftarrow x > T.left.value$ 
3:   if  $T.child[d].size = 1$  then

```

```
4:   if  $T.child[d].value = x$  then
5:       deleteNode( $T.child[d]$ )
6:        $temp \leftarrow T.child[d \oplus 1]$ 
7:        $T.left = T.child[d \oplus 1].left$ 
8:        $T.right = T.child[d \oplus 1].right$ 
9:        $T.value = T.child[d \oplus 1].value$ 
10:      deleteNode( $temp$ )
11:   else
12:       return
13:   end if
14: else
15:     Remove( $T.child[d]$ )
16: end if
17: Pushup( $T$ )
18: Maintain( $T$ )
19: end function
```

3.3.6 查询

与 Leafy Tree 实现的二叉搜索树相同。

伪代码见上面算法 1。

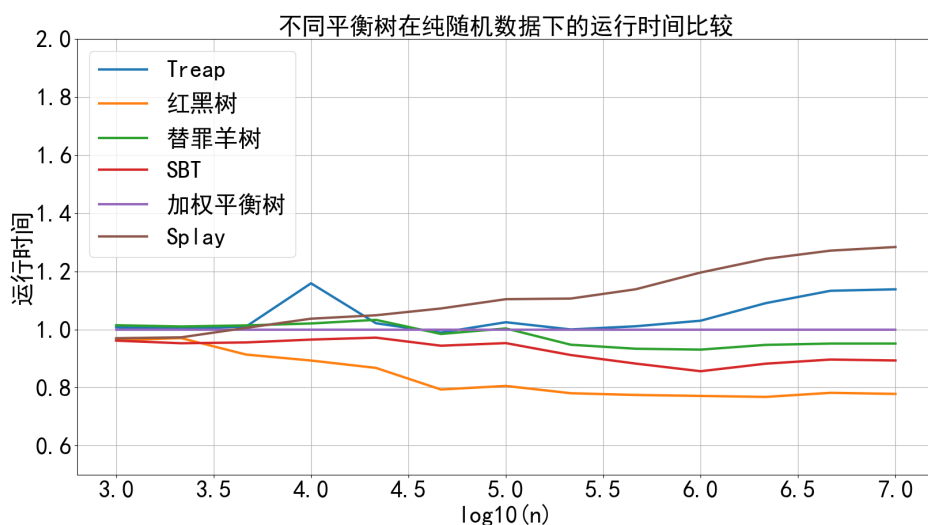
3.4 运行速度

这里选取了 LOJ #104 普通平衡树一题中，运行较快的一些代码进行测试。

由于不同的数据生成器的测试结果大致相同，这里只展示纯随机数据的测试结果。

测试代码及原始数据可以在 <https://mcfx.us/ctsc> 下载。

这里的结果可能有一定误差，并且代码也并不是保证常数最优秀的，但是能大致反应出这几种平衡树的运行速度。



如图，这里运行时间为该平衡树实际运行时间与加权平衡树实际运行时间的比值。

可以发现，加权平衡树虽然比红黑树和 SBT 慢，但是比 Treap 和 Splay 快，和替罪羊树运行速度相近。

4 总结

本文介绍了 Leafy Tree 及其实现的加权平衡树。

Leafy Tree 和二叉搜索树相比，其插入删除操作更容易实现，但是会使用两倍的节点。

加权平衡树的常数较小，实现较简单，同时还可以方便的实现可持久化。与常数更小的红黑树相比，可以不用记录额外信息（节点大小可以用来截取区间或求第 K 小值），并且实现方便得多。与实现简单的 *Splay* 相比，代码量基本相同，而常数更小，可以可持久化。与 *Treap* 相比，不用记录额外信息且常数更小。与替罪羊树相比，复杂度不是均摊的，可以实现可持久化。

而将这两者结合起来的加权平衡的 Leafy Tree，则可以在较短的代码中，实现平衡树的大多数功能，而且常数较小，可以可持久化。

除此之外，Leafy Tree 还可以实现各种基于二叉树的数据结构，如其他平衡树，线段树，堆等。但由于篇幅所限，本文不能一一介绍。

感谢

感谢中国计算机学会提供学习和交流的平台。

感谢成都七中张君亮、蔺洋老师多年以来的关心和指导。

感谢成都七中李欣隆、安师大附中赵雨扬同学为本文提供思路。
感谢其他帮助过我的老师、同学们。

参考文献

- [1] Nievergelt, J. and Reingold, E. M. (1972) Binary search trees of bounded balance. In *Proceedings of the Fourth Annual Acm Symposium on Theory of Computing*. ACM, pp. 137 – 142.
- [2] Wikipedia, Weight-balanced tree
- [3] Wikipedia, Binary search tree
- [4] LOJ #104 普通平衡树, <https://loj.ac/problem/104>