Size Balanced Tree

陈启峰 (Farmer John)

中国广东纪念中学

Email:344368722@QQ.com

2007.12.29

摘要

这篇论文将展现一个独特巧妙的策略，动态地维护二叉搜索树（Binay Search Trees，缩写为 BST），并且它在最坏的情况下也有着良好的期望运行速度。Size Balanced Tree，顾名思义，这是一棵通过**大小（Size）域**来维持平衡的二叉搜索树。

这是一种简单、高效并且在各方面都通用的数据结构。

这也是一种很容易被语言工具表述的数据结构，它有着简单明了的定义，和令人惊叹的运行速度，而且你会惊讶于它简单的证明。

这是目前为止速度最快的高级二叉搜索树[1]。

此外，它比其它一些知名的高级二叉搜索树要快得多，并且在实践中趋于完美。

它不仅支持典型的二叉搜索树操作，而且也支持 Select 和 Rank。

关键字

Size Balanced Tree
SBT
Maintain

翻译 By 竹子的叶子

文档经过处理，可以使用“ 视图 ”——“ 文档结构图 ”来方便阅读。

希望大家不要随便修改，谢谢！

1 介绍

在展现 Size Balanced Tree 之前，有必要详细说明一下二叉搜索树的旋转，**左旋**（Left-Ratote）和**右旋**（Right-Ratote）。

1.1 二叉搜索树

二叉搜索树是一种非常重要的高级数据结构，它支持很多动态操作，其中包括搜索（Search），取小（Minimum），取大（Maximun），前驱（Predecessor），后继（Successor），插入（Insert）和删除（Delete）。它能够同时被当作字典和优先队列使用。

二叉搜索树是按照二叉树结构来组织的，树中的每个节点最多只有两个儿子，二叉搜索树中关键字的储存方式总是满足以下**二叉搜索树**的性质：

设 *x* 是二叉搜索树中的一个结点，那么 *x* 的关键字不小于左子树中的关键字，并且不大于右子树中的关键字。

对于每一个结点 ***t***，我们使用 ***left[t]***和 ***right[t]***来储存它两个儿子的指针[2]，并且我们定义 ***key[t]***来表示结点 ***t*** 用来做比较的值。另外，我们增加 ***s[t]***，表示以 ***t*** 为根的子树的大小（Size），维持它成为这棵树上结点的个数。特别地，当我们使用 ***0*** 时，指针指向一棵空树，并且 ***s[0]=0***。

1.2 旋转

为了保持二叉搜索树的平衡（而不是退化成为**链表**），我们通常通过旋转改变指针结构，从而改变这种情况。并且，这种旋转是一种可以保持二叉搜索树特性的本地运算[3]。

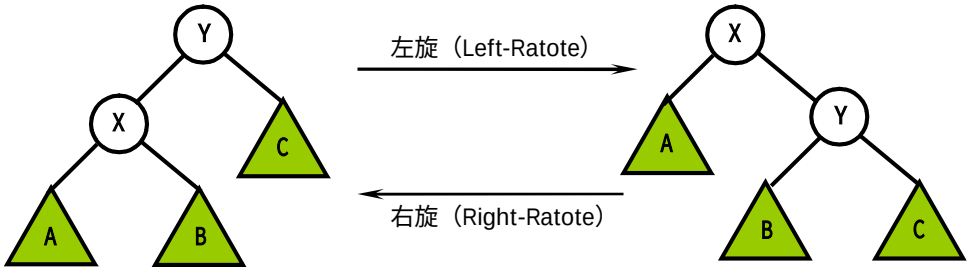


图 1.1

图 1.1：左旋 Left-Ratote(x)操作通过更改两个常数指针将左边两个结点的结构转变成右边的结构，右边的结构也可以通过相反的操作 Right-Ratote(x)来转变成左边的结构。

1.2.1 右旋的伪代码

右旋操作假定左儿子存在。

```
Right-Rotate(t)
1 k ← left[t]
2 left[t] ← right[k]
3 right[k] ← t
4 s[k] ← s[t]
5 s[t] ← s[left[t]] + s[right[t]] + 1
6 t ← k
```

1.2.2 左旋的伪代码

左旋操作假定右儿子存在。

```
Left-Rotate (t)
1 k ← right[t]
2 right[t] ← left[k]
3 left[k] ← t
4 s[k] ← s[t]
5 s[t] ← s[left[t]] + s[right[t]] + 1
6 t ← k
```

2 Size Balanced Tree

Size Balanced Tree（SBT）是一种通过**大小（Size）域**来保持平衡的二叉搜索树。它支持许多运算时间级别为 O(logn)的主要操作：

Insert(t,v)	在以 ***t*** 为根的 SBT 中插入一个关键字为 ***v*** 的结点。
Delete(t,v)	从以 ***t*** 为根的 SBT 中删除一个关键字为 ***v*** 的结点，如果树中没有一个这样的结点，删除搜索到的最后一个结点。
Find(t,v)	查找并返回结点关键字为 ***v*** 的结点。
Rank(t,v)	返回 ***v*** 在以 ***t*** 为根的树中的排名，也就是比 ***v*** 小的那棵树的大小（Size）加一。
Select(t,k)	返回在第 ***k*** 位置上的结点。显然它包括了**取大**（Maximum）和**取小**（Minimun），取大等价于 Select(t,1)，取小等价于 Select(t,s[t])。
Pred(t,v)	返回比 ***v*** 小的最大的数。
Succ(t,v)	返回比 ***v*** 大的最小的数。

通常 SBT 的每一个结点包含 ***key***，***left***，***right*** 和 ***size*** 等域。***size*** 是一个额外但是十分有用的数据域，它一直在更新，它在前面已经定义了。

每一个在 SBT 中的结点 *t*，我们保证：

性质 a：

$$s[right\ [t]] \geq s[left\ [left\ [t]]],\ s[right\ [left\ [t]]]$$

性质 b：

$$s[left\,[t]] \geq s[right\,[right\,[t]]],\ s[left\,[right\,[t]]]$$

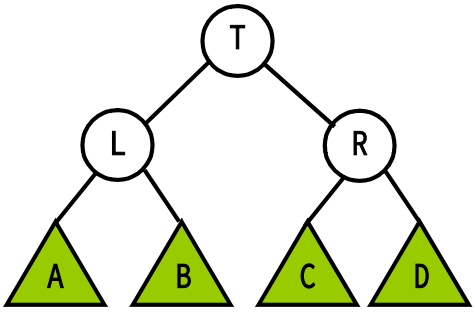


图 2.1

图 2.1：结点 L 和 R 分别是结点 T 的左右儿子。子树 A、B、C 和 D 分别是结点 L 和 R 各自的左右子树。

符合性质 a 和性质 b，$s[A], s[B] \leq s[R]\ \&\ s[C], s[D] \leq s[L]$

3 Maintain

如果我们要在一个 BST 插入一个关键字为 ***v*** 的结点，通常我们使用下列过程来完成任务。

```
Simple-Insert (t,v)
1 If t=0 then
2   t ← NEW-NODE(v)
3 Else
4   s[t] ← s[t]+1
5   If v<key[t] then
6     Simple-Insert(left[t],v)
7   Else
8    Simple-Insert(right[t],v)
```

在执行完简单的插入之后，性质 a 或性质 b 可能就不满足了，于是我们需要调整 SBT。

SBT 中最具活力的操作是一个独特的过程，Maintain。

Maintain(t)用来调整以 ***t*** 为根的 SBT。假设 ***t*** 的子树在使用之前已经都是 SBT。

由于性质 a 和性质 b 是对称的，所以我们仅仅详细的讨论性质 a。

情况 1：$s[left[left[t]]] > s[right[t]]$

插入后发生 $s[A] > s[R]$ 正如**图** 2.1，我们可以执行以下的指令来修复 SBT。

（1）首先执行 Right-Ratote(t)，这个操作让**图** 2.1 变成**图** 3.1；

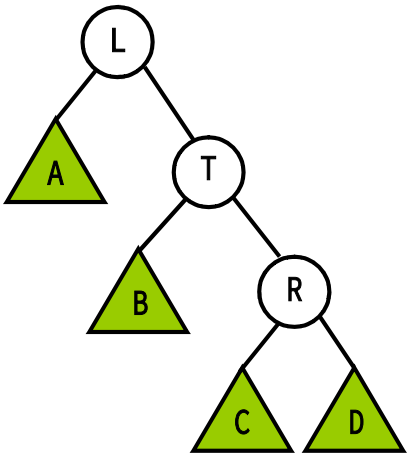


图 3.1

图 3.1：所有结点的描述都和图 2.1 一样

（2）在这之后，有时候这棵树还仍然不是一棵 SBT，因为 $s[C] > s[B]$ 或者 $s[D] > s[B]$ 也是可能发生的。所以就有必要继续调用 Maintian(t)。

（3）结点 L 的右子树有可能被连续调整，因为有可能由于性质的破坏需要再一次运行 Maintain(t)。

情况 2：$s[right\,[left\,[t]]] > s[right\,[t]]$

在执行完 Insert(left[t],v)后发生 $s[B] > s[R]$，如**图** 3.2，这种调整要比情况 1 复杂一些。我们可以执行下面的操作来修复。

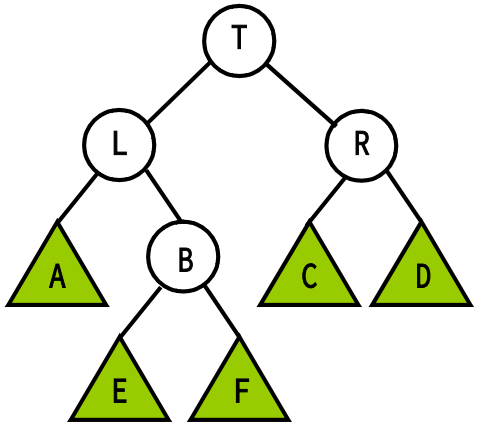


图 3.2

图 3.2：除了 E、B、F 以外，其他结点都和图 2.1 种的定义一样。E、F 是结点 B 的子树。

（1）在执行完 Left-Ratote(L)后，图 3.2 就会变成下面图 3.3 那样了。

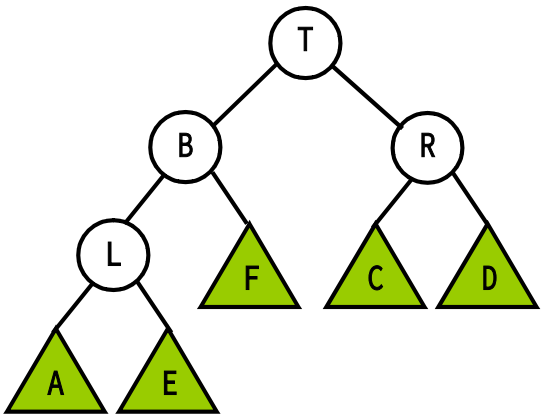


图 3.3

图 3.3：所有结点的定义都和图 3.2 相同

（2）然后执行 Right-Ratote(T)，最后的结果就会由图 3.3 转变成为下面的图 3.4。

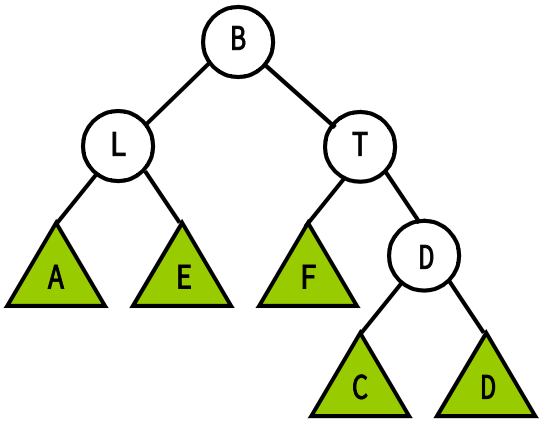


图 3.4

图 3.4：所有结点的定义都和图 3.2 相同

（3）在第（1）步和第（2）步过后，整棵树就变得非常不可预料了。万幸的是，在图 3.4 中，子树 A、E、F 和 R 仍就是 SBT，所以我们可以调用 Maintain(L)和 Maintain(T)来修复结点 B 的子树。

(4)在第(3)步之后,子树都已经是 SBT 了,但是在结点 B 上还可能不满足性质 a 或性质 b,因此我们需要再一次调用 Maintain(B)。

情况 3：$s[right[right[t]]] > s[left[t]]$

与情况 1 正好相反。

情况 4：$s[left[right[t]]] > s[left[t]]$

与情况 2 正好相反。

3.1 标准 Maintain 的伪代码

通过前面的分析，很容易写出一个普通的 Maintain。

```
Maintain (t)
1 If s[left[left[t]]>s[right[t]] then
2   Right-Rotate(t)
3   Maintain(right[t])
4   Maintain(t)
5   Exit
6 If s[right[left[t]]>s[right[t]] then
7   Left-Rotate(left[t])
8   Right-Rotate(t)
9   Maintain(left[t])
10   Maintain(right[t])
11   Maintain(t)
12   Exit
13 If s[right[right[t]]>s[left[t]] then
14   Left-Rotate(t)
15   Maintain(left[t])
16   Maintain(t)
17   Exit
18 If s[left[right[t]]>s[left[t]] then
19   Right-Rotate(right[t])
20   Left-Rotate(t)
21   Maintain(left[t])
22   Maintain(right[t])
23   Maintain(t)
```

3.2 更快更简单的 Maintain 的伪代码

前面的标准过程的伪代码有一点复杂和缓慢。通常我们可以保证性质 a 和性质 b 的满足，因此我们只需要检查情况 1 和情况 2 或者情况 3 和情况 4，这样可以提高速度。所以在那种情况下，我们需要增加一个布尔（boolean）型变量，flag，来避免毫无疑义的判断。

如果 flag 是 false，那么检查情况 1 和情况 2；否则检查情况 3 和情况 4。

```
Maintain (t,flag)
1  If flag=false then
2    If s[left[left[t]]>s[right[t]] then
3     Right-Rotate(t)
4    Else
5      If s[right[left[t]]>s[right[t]] then
6        Left-Rotate(left[t])
7        Right-Rotate(t)
8      Else
9        Exit
10 Else
11   If s[right[right[t]]>s[left[t]] then
12     Left-Rotate(t)
13   Else
14     If s[left[right[t]]>s[left[t]] then
15       Right-Rotate(right[t])
16       Left-Rotate(t)
17     Else
18       Exit
19 Maintain(left[t],false)
20 Maintain(right[t],true)
21 Maintain(t,false)
22 Maintain(t,true)
```

为什么 Maintain(left[t],true)和 Maintain(right[t],false)被省略了呢？Maintain 操作的运行时间是多少呢？你可以在第 6 部分 **分析** 中找到答案。

4 插入

SBT 中的插入很简单，下面是 SBT 中插入的伪代码。

4.1 插入的伪代码

```
Insert (t,v)
1 If t=0 then
2   t ← NEW-NODE(v)
3 Else
4   s[t] ← s[t]+1
5   If v<key[t] then
6     Simple-Insert(left[t],v)
7   Else
8     Simple-Insert(right[t],v)
9   Maintain(t,v≥key[t])
```

5 删除

我增加了删除的简易程度，如果在 SBT 中没有这么一个值让我们删除，我们就删除搜索到的最后一个结点，并且记录它。下面是标准删除过程的伪代码。

5.1 删除的伪代码

```
Delete (t,v)
1  s[t] ← s[t]－1
2  If (v=key[t])or(v<key[t])and(left[t]=0)or(v>key[t])and(right[t]=0) then
3    Delete ← key[t]
4    If (left[t]=0)or(right[t]=0) then
```

```
5     t ← left[t]+right[t]
6   Else
7     key[t] ← Delete(left[t],v[t]+1)
8  Else
9   If v<key[t] then
10    Delete(left[t],v)
11  Else
12    Delete(right[t],v)
13 Maintain(t,false)
14 Maintain(t,true)
```

5.2 更快更简单的删除伪代码

实际上这是没有任何其他功能的，最简单的删除。这里的 Delete(t,v)是函数，它的返回值是被删除的结点的值。虽然他会破坏 SBT 的结构，但是使用上面的插入，它还是一棵高度为 O(logn*)的 BST。这里的 *n**是所有插入结点的个数，而不是当前结点的个数！

```
Delete (t,v)
1  s[t] ← s[t]−1
2  If (v=key[t])or(v<key[t])and(left[t]=0)or(v>key[t])and(right[t]=0) then
3    Delete ← key[t]
4    If (left[t]=0)or(right[t]=0) then
5      t ← left[t]+right[t]
6    Else
7      key[t] ← Delete(left[t],v[t]+1)
8  Else
9    If v<key[t] then
10     Delete(left[t],v)
11   Else
12     Delete(right[t],v)
```

6 分析

很明显，Maintain 是一个递归过程，也许你会担心它是否能够停止。其实不用担心，因为已经能够证明 Maintain 过程的平摊时间时间是 O(1)。

6.1 高度分析

设 *f[h]*是高度为 h 的结点个数最少的 SBT 的结点个数。则我们有：

$$f[h]\begin{cases}1 & (h=0)\\ 2 & (h=1)\\ f[h-1]+f[h-2]+1 & (h>1)\end{cases}$$

6.1.1 证明

(1) 易证 $f[0]=1$ 和 $f[1]=2$。

(2) 首先，$f[h]\geq f[h-1]+f[h-2]+1(h>1)$。

对于每个 $h>1$，设 ***t*** 指向一个高度为 ***h*** 的 SBT，然后这个 SBT 包含一个高度为 ***h-1*** 的子树。不妨设它就是左子树。

通过前面对于 ***f[h]***的定义，我们得到 $s[left[t]]\geq f[h-1]$。

并且在左子树上有一棵高为 ***h-2*** 的子树，或者说有一棵大小（size）至少为 ***f[h-1]***的子树在左子树上。由性质 b 我们可得 $s[right[t]]\geq f[h-2]$。

因此我们得出结论：$s[t]\geq f[h-1]+f[h-2]+1$ 。

我们可以构造一个有 ***f[h]***个结点的 SBT，它的高度是 ***h***。我们把这种 SBT 叫做 ***tree[h]***。

$$tree[h]\begin{cases}\text{只有一个结点的}BST & (h=0)\\ \text{由两个结点的}BST & (h=1)\\ \text{包含}tree[h-1]\text{和}tree[h-2]\text{两棵子树的}BST & (h>1)\end{cases}$$

因此，宗上二点所述可得：$f[h]=f[h-1]+f[h-2]+1(h>1)$。

6.1.2 最坏高度

实际上 ***f[h]***是指数级函数，它的准确值能够被递归的计算。

$$f[h]=\frac{\alpha^{h+3}-\beta^{h+3}}{\sqrt{5}}-1=\left\|\frac{\alpha^{h+3}}{\sqrt{5}}\right\|=Fibonacci[h+2]-1=\sum_{i=0}^{h}Fibonacci[i]$$

其中 $\alpha=\frac{1+\sqrt{5}}{2}$，$\beta=\frac{1-\sqrt{5}}{2}$

一些 ***f[h]***的常数值

H	13	15	17	19	21	23	25	27	29	31
F[h]	986	2583	6764	17710	46367	121392	317810	832039	2178308	5702886

定理

一个有 *n* 个结点的 SBT，它在最坏情况下的高度是满足 $f[h]\le n$ 的最大 *h*。

假设 ***Maxh*** 是有 ***n*** 个结点的 SBT 的最坏高度，通过上面的定理，我们有

$$f[Maxh]=\left\|\frac{\alpha^{Maxh+3}}{\sqrt{5}}\right\|-1\le n\Rightarrow$$

$$\frac{\alpha^{Maxh+3}}{\sqrt{5}}\le n+1.5\Rightarrow$$

$$Maxh\le\log_{\alpha}^{\sqrt{5}(n+1.5)}-3\Rightarrow$$

$$Maxh\le 1.44\log_2^{n+1.5}-1.33$$

现在可以很清楚地看到 SBT 的高度是 O(logn)。

6.2 分析 Maintain

通过前面的结论，我们可以很容易的证明 Maintain 过程是非常有效的过程。

评价一棵 BST 时有一个非常重要的值，那就是结点的平均深度。它是通过所有结点深度和除以总结点个数 ***n*** 获得的。通常它会很小，而 BST 会更小[4]。因为对于每个常数 ***n***，我们都期望结点深度和（缩写为 SD）尽可能的小。

现在我们的目的是削减结点深度和，而它就是用来约束 Maintain 的次数[5]。

回顾一下 Maintain 中执行旋转的条件，会惊奇的发现结点深度和在旋转后总是在减小。

在情况 1 中，举个例子来说，比较图 2.1 和图 3.1，深度和增加的是一个负数，$s[right[t]]-s[left[left[t]]]$。再举个例子，比较图 3.2 和图 3.4，深度和增加的值比 1 小，是 $s[right[t]]-s[right[left[t]]]-1$。

所以高度为 O(logn)的树，深度和总是保持在 O(nlogn)。而且在 SBT 中插入后，深度和仅仅只增加 O(logn)。因此

$$SD=n\times O(\log n)-T=O(\log n)\Rightarrow$$

$$T=O(\log n)$$

在这里，T 是 Maintain 中旋转的次数。Maintain 的执行总次数就是 T 加上除去旋转的 Maintain 次数。所以 Maintain 的平摊运行时间是：

$$\frac{O(T)+O(n\log n)+O(T)}{n\log n}=O(1)$$

6.3 分析其它操作

现在 SBT 的高度是 O(logn)，Maintain 是 O(1)，所有主要操作都是 O(logn)。

6.4 分析更快更简单的删除

我们声明命题 P(n*)，若有一棵已经插入了 n*个结点并且有**快速简单删除**方法的 BST，则它的高度为 O(logn)[6]。我们通过数学方法来证明，对于任意整数 n*，P(n*)都是成立的。

6.4.1 证明

这里我只给出大概的证明。
假设结点 t 已经被 Maintain(t,false)检查过，则有

$$\frac{s[left[t]]-1}{2} \le s[right[t]] \Rightarrow$$
$$s[left[t]] \le \frac{2s[t]-1}{3}$$

因此如果一个结点到根的路径上的所有结点都已经被 Maintain 检查过，那么这个结点的深度就是 O(logn)。
（1）对于 n*=1，很明显 P(n*)是真的；
（2）假设对于 P(n*)在 n*<k 的情况下为真。对于 n*=k，在最后的连续插入之后，所有被 Maintain 检查过的结点一定会连接成一棵树。对于树中的每一个叶子，由它指向的子树不会在 Maintain 中被改变[7]。所以子树中结点的深度不会比 O(logn*)+O(logn*)=O(logn*)大。
（3）因此，当 n*=1,2,3...时，P(n*)是正确的。

这种方法证明出来的 Maintain 平摊时间依旧是 O(1)。

6.5 分析更快更简单的 Maintain

这里讨论有关为什么 Maintain(left[t],true)和 Maintain(right[t],false)被省略。

在情况 1 的图 3.1[8]中，我们有

$$s[L] \le 2s[R]+1 \Rightarrow$$
$$s[B] \le \frac{2s[L]-1}{3} \le \frac{4s[R]+1}{3} \Rightarrow$$
$$s[E], s[F] \le \frac{2s[B]-1}{3} \le \frac{8s[R]+3}{9} \Rightarrow$$
$$s[E], s[F] \le \left\lfloor \frac{8s[R]+3}{9} \right\rfloor \le s[R]$$

因此 Maintain(right[t],false)相当于图 3.1 中的 Maintain(T,false)，能够被省略。同样的，Maintain(left[t],true)明显的也不需要。

在情况 2 的图 3.2 中，我们有

$$\begin{cases} s[A] \ge s[E] \\ s[F] \le s[R] \end{cases}$$

这些不平衡也意味着 E 的子树大小要比 s[A]小，F 的子树大小要比 s[R]小。因而 Maintain(right[t],false)和 Maintain(left[t],true)可以被省略。

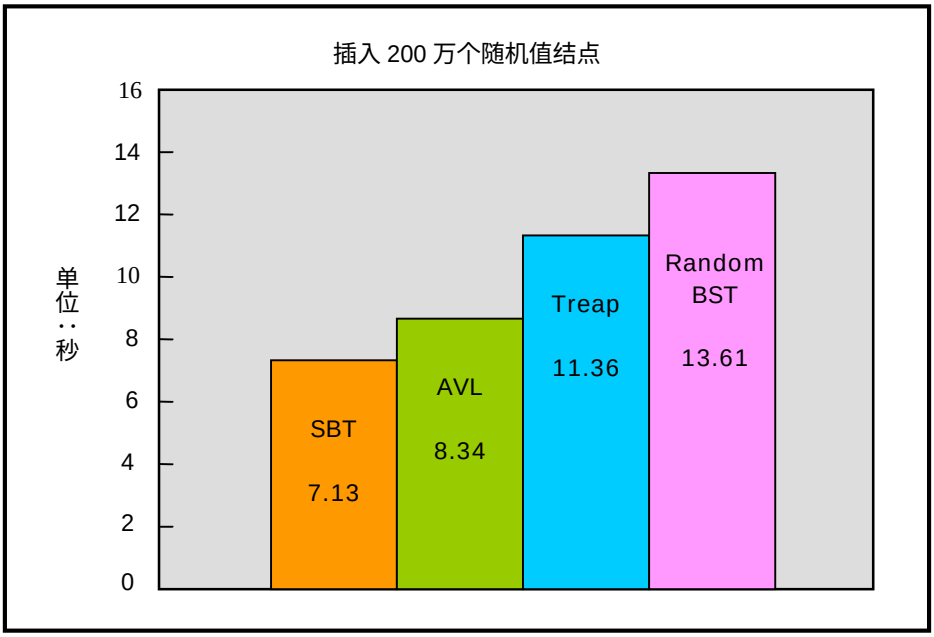
7 优点

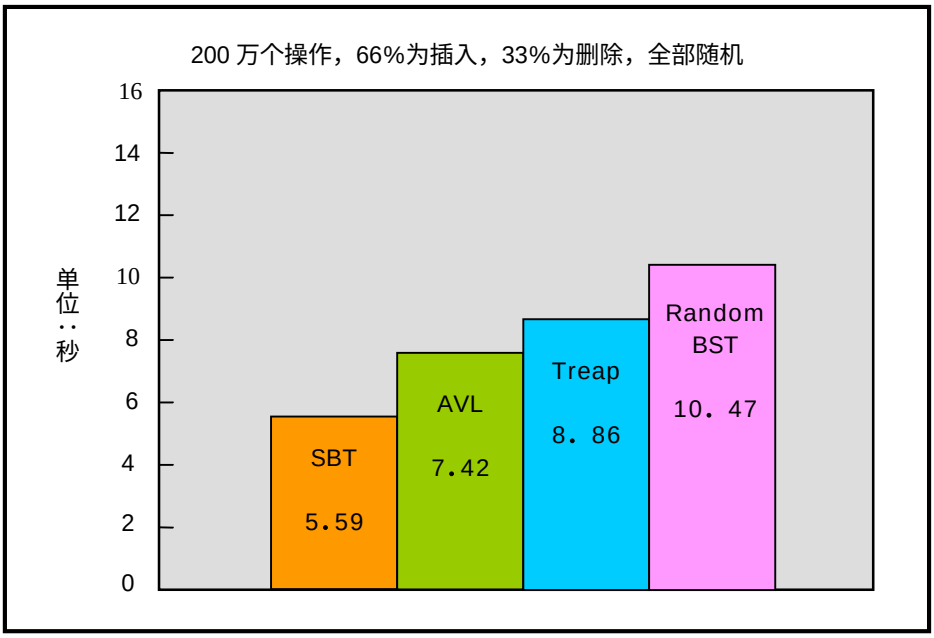
7.1 SBT 跑得快

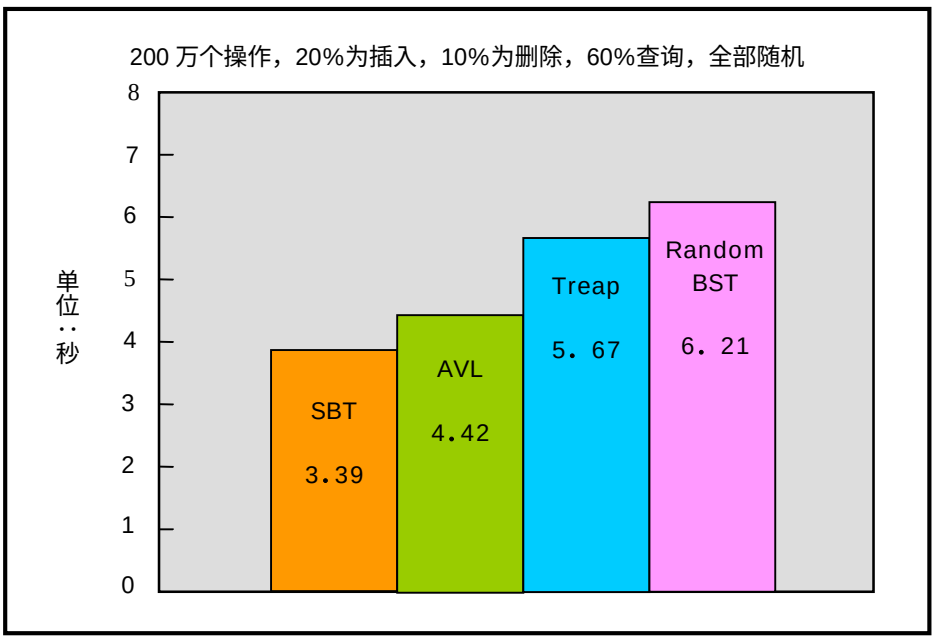
7.1.2 典型问题

写一个执行 n 个由输入给定的操作，他们分别是：
1） 在有序集合中插入一个给定的数字；
2） 从有序集合中删除一个给定的数字；
3） 返回一个给定的数字是否在有序集合中；
4） 返回一个给定的数字在有序集合中的排名；
5） 返回有序集合中第 k 大的数字；
6） 返回有序集合中一个给定数字前面的数字；
7） 返回有序集合中一个给定数字后面的数字。

7.1.2 统计


插入 200 万个随机值结点
单位：秒
16
14
12
10
8
6
4
2
0
SBT
7.13
AVL
8.34
Treap
11.36
Random BST
13.61


200 万个操作，66%为插入，33%为删除，全部随机
单位：秒
16
14
12
10
8
6
4
2
0
SBT
5.59
AVL
7.42
Treap
8. 86
Random BST
10. 47


200 万个操作，20%为插入，10%为删除，60%查询，全部随机
单位：秒
8
7
6
5
4
3
2
1
0
SBT
3.39
AVL
4.42
Treap
5. 67
Random BST
6. 21

在现实中，Size Balanced Tree 运行优秀，从上面的图表就能看出，同样都在随机数据的情况下，SBT 比其它平衡 BST 要快得多。此外，如果是有序数据，SBT 将会是意想不到的快速。它仅仅花费 2s 就将 200 万个有序数据结点插入到 SBT 中。

7.2 SBT 运行高效

当 Maintain 运行的时候平均深度一点也不会增加，因为 SBT 总是趋近于一个完美的 BST。

插入 200 万个随机值结点

类型	SBT	AVL	Treap	随机 BST	Splay	完美的 BST
平均深度	19.2415	19.3285	26.5062	25.5303	37.1953	18.9514
高度	24	24	50	53	78	20
旋转次数	1568017	1395900	3993887	3997477	25151532	?

插入 200 万个有序值结点

类型	SBT	AVL	Treap	随机 BST	Splay	完美的 BST
平均深度	18.9514	18.9514	25.6528	26.2860	999999.5	18.9514
高度	20	20	51	53	1999999	20
旋转次数	1999979	1999979	1999985	1999991	0	?

7.3 SBT 调试简单

首先我们可以输入一个简单的 BST 来保证不会出错，然后我们在插入过程中加入 Maintain，并调试。如果发生错误也只需要调试和修改 Maintain。此外，SBT 不是基于随机性的数据结构，所以它要比 Treap，跳跃表（Skip List），随机 BST 等更稳定。

7.4 SBT 书写简单

SBT 几乎是和 BST 同样简单。仅仅在插入过程中有一个附加的 Maintain，它也仅仅比 BST 先旋转[9]。而且 Maintain 也是同样的相当容易。

7.5 SBT 小巧玲珑

许许多多的平衡二叉搜索树，例如 SBT，AVL，Treap，红黑树等等都需要额外的域去保持平衡。但是他们中的很多都有着毫无用处的域，像高度（height），随机因子（random factor）和颜色（color）。而相反的是 SBT 包含一个十分有用的额外域，大小（size）域。通过它我们可以让 BST 支持选择（Select）操作和排名（Rank）操作。

7.5 SBT 用途广泛

现在 SBT 的高度是 O(logn)，即便在最坏情况下我们也可以在 O(logn)时间内完成 Select 过程。但是这一点伸展树（Splay）却不能很好的支持。因为它的高度很容易退化成 O(n)。上面的图表已经显示出这一点[10]。

感谢

作者感谢他的英语老师 Fiona 热情的帮助。

参考

[1] G.M.Adelson-Velskii and E.M.Landis, “An algorithm for the Organization of Information”, Soviet.Mat.Doklady (1962)
[2] L.J.Guibas and R.Sedgewick, “A dichromatic Framework for Balanced Trees”, Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science (1978)
[3] D. D. Sleator and R. E. Tarjan, ‘Self-adjusting binary search trees’, *JACM*, 32, 652–686 (1985).
[4] S.W.Bent and J.R.Driscoll, Randomly balanced searchtrees. Manuscript (1991).
[5] Raimund Seidel and Cecilia R. Aragony, Randomized Search Trees.(1996).
[6] M.A.Weiss, ‘‘Data Structures and Algorithm Analysis in C++’’, Third Edition, Addison Wesley, 2006.
[7] R.E. Tarjan, ‘‘Sequential access in splay trees takes linear time’’, Combinatorica 5(4), 1985, pp. 367-378.
[8] J. Nievergelt and E. M. Reingold, ‘Binary search trees of bounded balance’, *SIAM J. Computing*, 2, 33–43 (1973).
[9] K.Mehlhorn, and A.Tsakalidis, “Data Structures,” in *Handbook of Theoretical Computer Science,* Chapter 6, Vol.A,

pp.303–341, Elsevier Science Publishers,(1990)

注释

以下是译者在文章中的注释。

[1] 高级二叉搜索树，Advance Binary Search Tree，或者也可以叫做附加二叉搜索树，都是在 BST 基础上作了一定改进的数据结构，譬如 AVL，Treap，伸展树等。[回去 1]

[2] 指针，在本文中作者使用的都是静态指针，也就是数组的下标。[回去 2]

[3] 本地运算，local operation，也较原地运算，就是基本不依赖其他附加结构、空间的运算。[回去 3]

[4] 原文为：“ Generally the less it is, the better a BST is. ” [回去 4]

[5] 原文为：“ Now we need to concentrate on SD. Its significance is the ability to restrict the times of Maintain. ”SD 是“ the sum of the depths of all nodes ”，即结点深度和。[回去 5]

[6] 原文为：“ We call the statement P(n*) that a BST with the faster and simpler Delete and n* standard Inserts is at the height of O(logn*). ” [回去 6]

[7] 原文为：“ For every leaf of this tree, the subtree pointed by it does not be changed by Maintain. ” [回去 7]

[8] 原文中这里是图 3.2，可是情况 1 中只有图 3.1，图 3.2 是在情况 2 后面，于是将它改为了“ 图 3.1 ”。原文为：“ In case 1 in Figure 3.2 ” [回去 8]

[9] 原文为：“ Removing the only additional Maintain in Insert the former turns to be the latter. ” [回去 9]

[10] 原文为：“ which is exposed by the char above ”，其中 char 个人认为是输入错误，应该是“ chart ”。[回去 10]

译者的话

因为网上和朋友那里四处都找不到陈启峰 Size Balanced Tree 中文版的论文，当然也就不知道怎么翻译 SBT 更好一些了，于是就这么写上去了，相信有点 OI 常识的人看了都知道 SBT 是怎么一回事。

关于这篇翻译过来的中文版的论文，目前还没有给陈启峰本人看过，希望他和各位大牛看过之后不要鄙视，能尽量挑出一些错误来，我就很感激了。还有其中的一些语言，特别是分析证明的那一部分，由于叶子我英语和 OI 水平有限，怕翻译出错，把可能认为理解有误，或者不好用中文表达，或者自己表达得不够清晰的部分在后面加了注释。

翻译陈启峰这篇论文的原动力是因为英语版看着太费劲，我是这样，相信大多数人都应该是这样，都是中国人，推广以下新东西，是十分必要的，当然最好能够语言本土化，于是我就花了两三天功夫翻译了一下。

还有论文中原来的图片、图表，我都经过了再加工，相信看起来更漂亮（^-^）。关于文字大小，小五是宋体看得清楚的最小字体，也很清秀，如果看不清楚，可以放大，但希望您能不要修改，都是排好版的，很容易乱的。

尽管是翻译，也是我的劳动成果，如果发现翻译错误，或文字错误，请联系我。
也请您不要修改本文，Word 文档是为了大家更好的使用、查看。
我的 E-Mail：combooleaf@21cn.com
我的 QQ：541206837
我的 Blog：竹叶小店 http://combooleaf.blog.hexun.com 欢迎观光

最后再崇拜一下陈启峰大牛，谢谢。