由二叉查找树到容均树

From Binary Search Tree to Size Balanced Tree

田劲锋

2011 年 7 月 9 日

Abstract

本文从二叉查找树开始，介绍了 2006 年由陈启峰发明的 Size Balanced Tree ，我译作容均树。文章给出了容均树的各种操作的伪代码以及相应 C++ 代码，并对其运行效率作了证明，最后介绍了其在竞赛题目中的应用。

关键字：二叉查找树, BST, Binary Search Tree, 容均树, SBT, Size Balanced Tree, 节点大小平衡树

Contents

1 引子

在计算机科学和数学中，**排序** (*sort*)算法无疑是最经典、最常用的了。我们可以设想一下，倘若字典中的词不是以字母的顺序排列，那你还敢使用字典吗？你固然要疯掉的。同样，在计算机中存储的数据，也必须有序排列。我们定义：

> 一个**排序算法** (*Sorting algorithm*)是一种能将一串资料依照特定排序方式的一种算法。

最常用到的排序方式是数值顺序以及字典顺序。有效的排序算法在一些算法（例如搜寻算法与合并算法）中是重要的，如此这些算法才能得到正确解答。排序算法也用在处理文字资料以及产生人类可读的输出结果。基本上，排序算法的输出必须遵守下列两个原则：

- 输出结果为递增序列（递增是针对所需的排序顺序而言）
- 输出结果是原输入的一种排列、或是重组

虽然排序算法是一个简单的问题，但是从计算机科学发展以来，已经有大量的研究在此问题上。举例而言，冒泡排序在 1956 年就已经被研究。虽然大部分人认为这是一个已经被解决的问题，有用的新算法仍在不断的被发明。

我们主要讨论的，不是排序，也就是说，我们不去讨论**选择排序** (*Selection sort*)或**冒泡排序** (*Bubble sort*)的执行过程，不研究大大小小的**堆排序** (*Heap sort*)、**合并排序** (*Merge sort*)或者**插入排序** (*Insertion sort*)，甚至不关心均摊 $O(\lg n)$ 的**快速排序** (*Quick sort*)。我们要研究的，是一种用于排序的**数据结构** (*Data structure*)，它不仅有序，而且可以快速查找、删除、插入、修改，这不仅在很多竞赛题目中有广泛应用，而且我们也会知道计算机存储数据的方式也基于此。

这样，我们的研究对象从排序转移到了**查找** (*searching*)，准确而言，是“信息的存储与检索”[七卷本]，也可以简单叫做“查表”。有序序列为查找提供了方便，所以查找和排序通常是紧密相关的。顺序查找我们就不再讨论了，这里，我们从一棵“树”说起。

2 二叉查找树

2.1 神马是二叉查找树

我们来研究一棵叫做“二叉查找”的“树”，即**二叉查找树** (*Binary Search Tree*)：

> 二叉查找树，是一棵空树，或是具有下列性质的二叉树：
>
> - 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
> - 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
> - 它的左、右子树也分别为二叉查找树。

上述性质被称为**二叉查找树性质**。

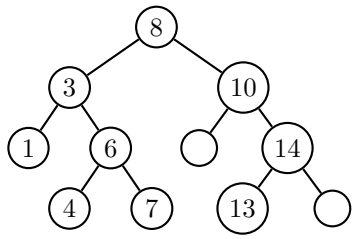


Figure 1. 一棵三层的二叉查找树

二叉查找树可简称为 BST（鄙视它？），它又翻译作“二叉排序树”“二元搜尋樹”等等。二叉查找树是递归定义的，如图1是一棵二叉查找树。

我们用链表结构来存储二叉查找树，其中每一个节点都是一个对象。有 key 存储其值，$left$ 和 $right$ 指向其左右儿子。

对于一个已知的二叉查找树，从小到大输出其节点的值，只需对其进行二叉树的中序遍历，即递归地先输出其左子树，再输出其本身，然后输出其右子树。遍历的时间复杂度为 $O(n)$。

这里，我们给出这一算法的伪代码，类似于《算法导论》，稍有差异。具体程序实现方法很多，实在写不出来，可以参见2.4。

```
PRINT(x)
1  if x ≠ NIL
2        PRINT(x.left)
3        print x.key
4        PRINT(x.right)
```

2.2 查询二叉查找树

2.2.1 查找

对于一个已知的二叉查找树 x，在其中查找特定的值 k，函数 SEARCH 返回指向值为 k 的节点指针，若找不到则返回 NIL。算法时间复杂度为 $O(h)$， h 为树的高度，理想情况下等于 $\lg n$[1]。

```
SEARCH(x, k)
1  if x = NIL or k = x.key
2        return x
3  if k < x.key
4        return SEARCH(x.left, k)
5  else
6        return SEARCH(x.right, k)
```

[1]本文中定义 $\lg n = \log_2 n$。

当然，非递归版本运行得更快些，不至于栈溢出：

ITERATIVE-SEARCH(x, k)

```
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else
5              x = x.right
6  return x
```

2.2.2 最大值和最小值

要查找二叉查找树中具有最小值的元素，只要从根节点开始，沿着左子树找到最左边的节点就可以了。

MINIMUM(x)

```
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

最大值的查找则非常类似，找最右边即可。

MAXIMUM(x)

```
1  while x.right ≠ NIL
2        x = x.right
3  return x
```

算法的时间复杂度依然是 $O(h)$。

2.2.3 前驱和后继

给定一个二叉查找树中的节点，有时需要找到其在树中的后继。对于节点 x，其后继为值大于 $x.key$ 的最小节点。由于二叉查找树的特殊性质，不必进行比较就可以找到。下面的算法对于给定的节点指针 x，返回其后继的指针[2] y。

[2]注意，是**指针**！不是其值，不要在这里犯错误！

SUCCESSOR(x)

```
1  if x.right ≠ NIL
2        return MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x = y.right
5        x = y
6        y = y.p
7  return y
```

查找前驱则很对称了。

PREDECESSOR(x)

```
1  if x.left ≠ NIL
2        return MAXIMUM(x.left)
3  y = x.p
4  while y ≠ NIL and x = y.left
5        x = y
6        y = y.p
7  return y
```

两者的时间复杂度依然为 $O(h)$。

2.3 插入和删除

插入 (INSERT)和**删除** (DELETE)操作涉及到了对数据结构的修改，在修改中又要保持二叉查找树性质。所以，代码就有些复杂了。

2.3.1 插入

将一个新值 v 插入到树 T 中，要建立在查找的基础上。基本方法是类似于线性表中的二分查找，不断地在树中缩小范围定位, 最终找到一个合适的位置插入。具体方法如下所述：

- 从根节点开始插入；
- 如果要插入的值小于等于当前节点的值，在当前节点的左子树中插入；
- 如果要插入的值大于当前节点的值，在当前节点的右子树中插入；
- 如果当前节点为空节点，在此建立新的节点，该节点的值为要插入的值，左右子树为空，插入成功。

对于相同的元素，一种方法我们规定把它插入左边，另一种方法是我们在节点上再加一个域，记录重复节点的个数。上述方法为前者。插入的时间复杂度仍为 $O(h)$。

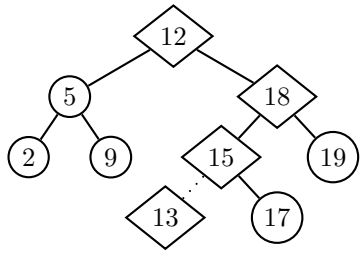


Figure 2. 将 13 插入一棵二叉查找树。菱形表示从根节点开始向下到插入位置的路径，虚线为在树中插入该节点而产生的链接。

```
INSERT(T, v)
1  if T = NIL
2      new T.key = v // 新建节点并插入
3  elseif v ≤ T.key
4      T.left.p = T
5      INSERT(T.left, v)
6  else
7      T.right.p = T
8      INSERT(T.right, v)
```

下面是非递归版本，取自《算法导论》。

```
ITERATIVE-INSERT(T, v)
 1  y = NIL
 2  x = T.root
 3  z.key = v
 4  while x ≠ NIL
 5      y = x
 6      if z.key < x.key
 7          x = x.left
 8      else
 9          x = x.right
10  z.p = y
11  if y = NIL
12      T.root = z // 树 T 为空
13  elseif z.key < y.key
14      y.left = z
15  else
16      y.right = z
```

2.3.2 删除

从二叉查找树中删除一个节点，要分三种情况分别讨论，而且每一种都要大量的技巧：

1. 如果它没有子女，直接删除；
2. 如果它只有一个子女，则删除它，将其子女的父亲改为它的父亲；
3. 如果它有两个子女，先用其后继替换该节点，其后继的卫星数据一并加在其后。

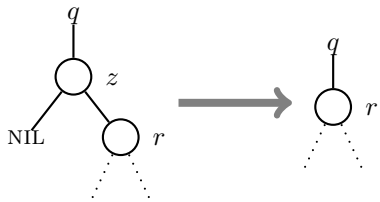
下面的过程描述了给出树 T 和待删除节点 z，几种不同的删除情况。见图3。

- 如果 z 没有左孩子（如图3(a)），则用 z 的右孩子替换它，当然可以为空或不为空。当 z 的右孩子为空，则表示 z 没有孩子；若不为空，就是 z 只有一个孩子的情况了。
- 如果 z 只有一个孩子，且为左孩子（如图3(b)），则用其左孩子替换 z 即可。
- 其他情况，就是 z 有左右两个孩子。我们找到 z 的后继 y，它就在 z 的右子树上且没有左孩子。我们想把 y 及其子树拼接替换到 z。
  - 如果 y 是 z 的右孩子（如图3(c)），就用 y 替换 z，把 y 的右孩子单独留在那里。
  - 如果 y 在 z 的右子树里但不是 z 的右孩子（如图3(d)），先要用 y 自己的右孩子替换 y，再用 y 替换 z。

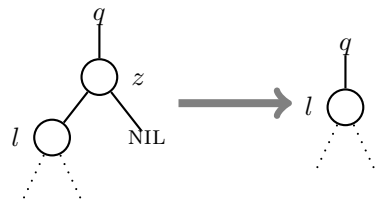
为了在二叉查找树中移动一棵子树，我们定义一个**移植** (Transplant)过程，把一棵子树 u 归并到另一棵子树 v 中， u 的父亲变为 v 的父亲， u 的父亲就有了 v 作为其孩子。

```
TRANSPLANT(T, u, v)
1  if u.p = NIL
2      T.root = v
3  elseif u = u.p.left // u 是左孩子
4      u.p.left = v
5  else
6      u.p.right = v
7  if v ≠ NIL
8      v.p = u.p
```

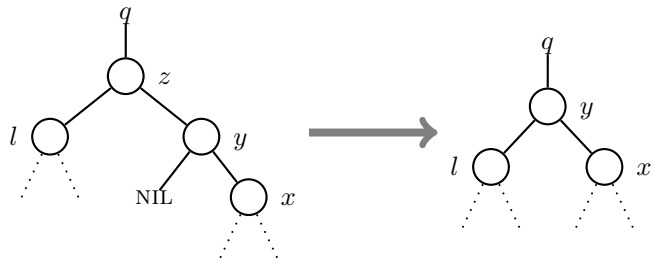
有了移植过程，就可以很简单地写出从树 T 中删除节点 z 的算法了。



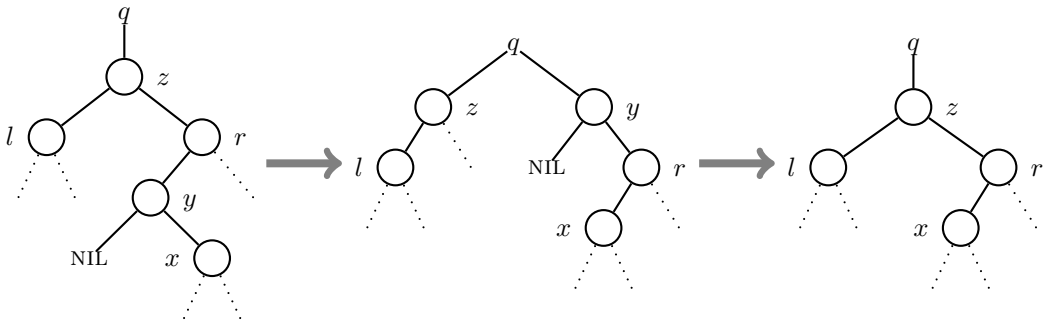
(a) 节点 z 没有左孩子。我们用 z 的右孩子 r 替换它，当然可以为或不为 NIL。



(b) 节点 z 只有一个左孩子 l，则用 l 替换 z。



(c) 节点 z 有两个孩子，左孩子为 l，右孩子是其后继 y，且 y 的右孩子为 x。我们用 y 替换 z，更新 y 的左孩子为 l，留下 y 的右孩子 x。



(d) 节点 z 有两个孩子（左孩子为 l，右孩子为 r），其后继 $y \neq r$ 在以 r 为根的子树中。我们用 y 自己的右孩子 x 替换 y，设置 y 为 r 的父亲。然后，设置 y 为 q 的孩子，为 l 的父亲。

Figure 3. 二叉查找树几种不同的形态下的删除操作，其中 z 为根节点 q 的孩子。

DELETE(T, z)

```
1  if z.left = NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right = NIL
4      TRANSPLANT(T, z, z.left)
5  else
6      y = MINIMUM(z.right)
7      if y.p ≠ z
8          TRANSPLANT(T, y, y.right)
9          y.right = z.right
10         y.right.p = y
11     TRANSPLANT(T, z, y)
12     y.left = z.left
13     y.left.p = y
```

算上移植过程，整个删除算法的时间复杂度为 $O(h)$。

2.4 具体代码实现

有些书 [算法果壳] 上说，实际的代码要比伪代码好。究竟哪个好？这就是“公说公有理，婆说婆有理”了。实际上，这样的争论是有很多典型例子的，比如 Gnome 与 KDE 之争，Vim 和 Emacs 的大战， TEX 用户对 Word 的口诛笔伐……真是数不胜数。我是一个折衷取共的人，所以将二叉查找树的 C++ 代码列在下面，就不“伪”了。

```
#include <cstdio>
#define nil 0
const int MAX = 100000;
int key[MAX], left[MAX], right[MAX], p[MAX];
int root, node;
void Print(int x) {
    if(x != nil) {
        Print(left[x]);
        printf("%d␣", key[x]);
        Print(right[x]);
    }
}
int Search(int x, int k) {
    if(x == nil || k == key[x])
        return x;
    if(k < key[x])
        return Search(left[x], k);
```

```
18      else
19          return Search(right[x], k);
20  }
21  int Iterative_Search(int x, int k) {
22      while(x != nil && k != key[x])
23          if(k < key[x])
24              x = left[x];
25          else
26              x = right[x];
27      return x;
28  }
29  int Minimum(int x) {
30      while(left[x] != nil)
31          x = left[x];
32      return x;
33  }
34  int Maximum(int x) {
35      while(right[x] != nil)
36          x = right[x];
37      return x;
38  }
39  int Successor(int x) {
40      if(right[x] != nil)
41          return Minimum(right[x]);
42      int y = p[x];
43      while(y != nil && x == right[y]) {
44          x = y;
45          y = p[y];
46      }
47      return y;
48  }
49  int Predecessor(int x) {
50      if(left[x] != nil)
51          return Maximum(left[x]);
52      int y = p[x];
53      while(y != nil && x == left[y]) {
54          x = y;
55          y = p[y];
56      }
57      return y;
58  }
59  void Insert(int &T, int v) {
```

```
60      if(T == nil)
61          key[T = ++node] = v;
62      else if(v <= key[T]) {
63          p[left[T]] = T;
64          Insert(left[T], v);
65      } else {
66          p[right[T]] = T;
67          Insert(right[T], v);
68      }
69  }
70  void Iterative_Insert(int T, int v) {
71      int y = nil;
72      int x = T;
73      int z = ++node;
74      key[z] = v;
75      while(x != nil) {
76          y = x;
77          if(key[z] < key[x])
78              x = left[x];
79          else
80              x = right[x];
81      }
82      p[z] = y;
83      if(y == nil)
84          key[T] = z;
85      else if(key[z] < key[y])
86          left[y] = z;
87      else
88          right[y] = z;
89  }
90  void Transplant(int T, int u, int v) {
91      if(p[u] == nil)
92          T = v;
93      else if(u == left[p[u]])
94          left[p[u]] = v;
95      else
96          right[p[u]] = v;
97      if(v != nil)
98          p[v] = p[u];
99  }
100 void Delete(int T, int z) {
101     if(left[z] == nil)
```

```
102             Transplant(T, z, right[z]);
103         else if(right[z] == nil)
104             Transplant(T, z, left[z]);
105         else {
106             int y = Minimum(right[z]);
107             if(p[y] != z) {
108                 Transplant(T, y, right[y]);
109                 right[y] = right[z];
110                 p[right[y]] = y;
111             }
112             Transplant(T, z, y);
113             left[y] = left[z];
114             p[left[y]] = y;
115         }
116     }
117     void printarrs() {
118         printf("\ti\tkey\tleft\tright\tp\n");
119         for(int i=1; i<=node; i++)
120             printf("\t%d\t%d\t%d\t%d\t%d\n",
121                     i, key[i], left[i], right[i], p[i]);
122     }
123     int main() {
124         int m[]={3, 4, 5, 6, 7, 1};
125         for(int i=0; i<6; i++) {
126             //Iterative_Insert(root, m[i]);
127             Insert(root, m[i]);
128             printf("Insert␣%: d", m[i]);
129             printarrs();
130             Print(root); printf("\n");
131         }
132         printf("Delete␣: 4");
133         Delete(root, Search(root, 4));
134         printarrs();
135         Print(root); printf("\n");
136         printf("5's␣Succssor's␣:
                pointer%d\n", Successor(Search(root, 5)));
137         printf("5's␣Predecessor's␣:
                pointer%d\n", Predecessor(Search(root, 5)));
138         return 0;
139     }
```

3 容均树

3.1 几何原本

此“几何原本”非彼《几何原本》。准确地说，**欧几里德** (*Euclid*)在写 $\Sigma\tau o\iota\chi\varepsilon\tilde{\iota}\alpha$ 这本书时并没有发明**几何** (*geometria*)这个词，几何一词源于《几何原本》的翻译。《几何原本》英文叫做 *Elements*，意为“元素”，是世界的基本量。“几何”意指“万事万物”，而“原本”意指“原来本貌”，合起来就是“世间万物的本来面目”，即“元素”之意。[3]

好了，扯远了。其实我借用它，一是“引起读者兴趣”，二是“为下文作铺垫”。怎么铺垫呢？前面我们知道，事物都要有其“本来面目”，那我们就探询一下容均树的几何原本。

在前面一节，我们描述了二叉查找树这一计算机科学中的一种重要的查找结构。我们看到这些算法的时间复杂度都是 $O(h)$ 的，其中 h 是树的高度近似等于 $\lg n$。这是为什么呢？我们知道，一个满二叉树的高度一定是 $h = \lg(n+1)$，但二叉查找树并不能保持这个性质（我们有 $h \geq \lfloor \lg n \rfloor$），在输入序列有序时，它会退化成一条链，这时它的高度就成了 $n-1$，时间复杂度随之提高了一个很大的数量级。为了解决这个问题，可以自动保持平衡的**平衡树** (*Balanced Tree*)应运而生了，它的目的就是使树的高度尽可能趋于 $\lg n$，降低平摊时间复杂度。

AVL 树 (*AVL Tree*)是最先发明的自平衡二叉查找树。在 AVL 树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是 $O(\lg n)$。插入和删除可能需要通过一次或多次树旋转来重新平衡这个树。AVL 树得名于它的发明者 G.M. Adelson-Velsky 和 E.M. Landis，他们在 1962 年的论文 *An algorithm for the organization of information* 中发表了它。AVL 树的基本操作一般涉及运作同在不平衡的二叉查找树所运作的同样的算法，但是要进行预先或随后做一次或多次所谓的“AVL 旋转”，这就导致它难以实现。虽然它很经典，效率也非常高，但是在快速应用中，它令人望而却步了。

红黑树 (*Red-Black Tree*)是令一种自平衡二叉查找树，它是在 1972 年由 Rudolf Bayer 发明的，他称之为“对称二叉 B 树”，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文中获得的。伟大、光荣、正确的《算法导论》非常严谨地讨论了红黑树 $O(\lg n)$ 的高效算法。同样，算法虽经典，实现相当难。但它优异的性能，至今还在 STL 中发挥巨大作用。

伸展树 (*Splay Tree*)由 Daniel Sleator 和 Robert Tarjan 发明，不需要记录用于平衡树的冗余信息，一般操作都基于伸展操作。它同**堆树** (*Treap*)一样，都是竞赛选手喜闻乐见的高效数据结构。$Treap = Tree + Heap$，然而它的发明人 Cecilia R. Aragon 和 Raimund G. Seidel 却将其称为**随机查找树** (*Randomized Search Trees*)，因为它利用了随机数和堆结构，使得效率提高。它实现简单，易于使用，效率取决于随机数，值得推荐。

[3]然而大部分人 [维基] 认为，“几何”的原文是“geometria”，徐光启和利玛窦在翻译时，取“geo”的音为“几何”（明朝音: gi-ho），而“几何”二字中文原意又有“衡量大小”的意思，用“几何”译“geometria”，音义兼顾，确是神来之笔……

在 2006–2007 的冬天，广东纪念中学的**陈启峰**神牛，创造了惊世绝伦的神级数据结构——*Size Balanced Tree*，全文 [陈启峰] 英语写成，论述详细，易于实现，自称是“目前为止速度最快的高级二叉搜索树”。由于作者本人对其推广不力，致使只有在中国才有人研究，甚至在英文维基百科上都没有该词条。但它实在太优美了，我实在忍不住这种艺术的诱惑，下面就是我的膜拜过程。

3.2 定义

很奇怪这么优秀的中国人发明的数据结构竟然没有中文名字，维基百科翻译为“节点大小平衡树”，不仅啰嗦，也体现不出中华文化的优越性。思考许久，我命名为**容均树**，“容”表示“大小”，“均”表示“平衡”，也算不错。当然这不太确切，盼望某大神再赐美名。

容均树的缩写很有意思——SBT。傻逼树？超级变态树？其实它很优异的。我们定义一个容均树节点 x 有四个域：值 key，左孩子 $left$，右孩子 $right$，以及最重要的、保持平衡的 $size$。对于容均树中任一节点 x，我们有如下**容均树性质**：

(a)

$$x.right.size \geq x.left.left.size, x.left.right.size$$

(b)

$$x.left.size \geq x.right.right.size, x.right.left.size$$

即每棵子树的大小不小于其兄弟的子树大小。

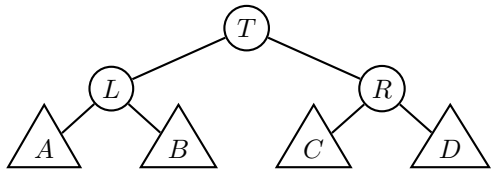


Figure 4. 结点 L 和 R 分别是结点 T 的左右儿子。子树 A, B, C, D 分别是结点 L 和 R 各自的左右子树。

若图4是一棵容均树，则有

$$A.size, B.size \leq R.size; \quad C.size, D.size \leq L.size \text{。}$$

3.3 旋转

为了保持容均树平衡（而不是退化成为链表），我们通常通过**旋转** (Rotate)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查找树性质的本地运算[4]。

旋转分为**左旋** (Left-Rotate)和**右旋** (Right-Rotate)，它们是互逆操作，如图5。

下面给出在节点 x 处左旋的算法。

[4] **本地运算** (*Local Operation*)，也叫**原地运算**，就是基本不依赖其他附加结构、空间的运算。

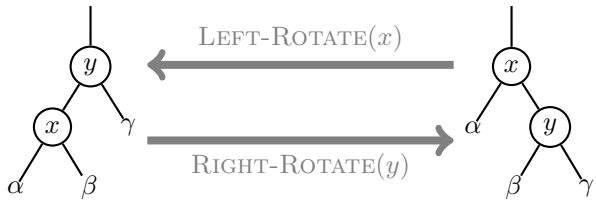


Figure 5. 左旋操作通过更改两个常数指针将左边两个结点的结构转变成右边的结构，右边的结构也可以通过相反的操作——右旋，来转变成左边的结构。

LEFT-ROTATE(x)

1 $k = x.right$
2 $x.right = k.left$
3 $k.left = x$
4 $k.size = x.size$
5 $x.size = x.left.size + x.right.size + 1$
6 $x = k$

右旋亦然。

RIGHT-ROTATE(y)

1 $k = y.left$
2 $y.left = k.right$
3 $k.right = y$
4 $k.size = y.size$
5 $y.size = y.left.size + y.right.size + 1$
6 $y = k$

3.4 插入

二叉查找树的**插入** (INSERT)比较简单，但是容均树因为其特殊的性质，我们不能简单地插入。插入算法先简单插入节点，然后调用一个维护过程以保持性质。维护操作在第3.6节。

INSERT(T, v)

```
1  if T = NIL
2      new T.key = v
3      T.size = 1
4  else
5      T.size++
6      if v < T.key
7          INSERT(T.left, v)
8      else
9          INSERT(T.right, v)
10     MAINTAIN(T, v ≥ T.key)
```

3.5 删除

容均树的**删除** (*Delete*)操作与普通维护 $size$ 域的二叉查找树相同，我们使用了《算法导论》第三版之前没有移植过程的算法，可以对比一下。

DELETE(T, v)

```
1  if T.size ≤ 2
2      record = T.key // record 为全局变量
3      T = T.left + T.right
4      return
5  T.size--
6  if v = T.key
7      DELETE(T.left, v + 1)
8      T.key = record
9      MAINTAIN(T, TRUE)
10 else
11     if v < T.key
12         DELETE(T.left, v)
13     else
14         DELETE(T.right, v)
15     MAINTAIN(T, v < T.key)
```

因为在删除之前，我们保证了容均树的性质；又因为删除后，虽然不能保证其还是容均树，但是这时整棵树的高度[5]并没有改变，所以时间复杂度也不会增加，所以就没有必要调用维护操作了。于是有了如下较高效的算法。这里，我们在没有找到待删除节点时，则删除最后搜索到的节点。

[5]可以证明，此时树的高度是 $O(\lg n^*)$，其中 n^* 是插入节点的个数。

DELETE(T, v)

```
1  T.size − −
2  if (v = T.key) or (v < T.key and T.left = NIL) or (v > T.key and T.right = NIL)
3      r = T.key
4      if T.left = NIL or T.right = NIL
5          T = T.left + T.right // T 变为其左子或右子
6      else
7          T.key = DELETE(T.left, T.key + 1) // 找不到则删除最后找到的节点
8      return r
9  else
10     if v < T.key
11         return DELETE(T.left, v)
12     else
13         return DELETE(T.right, v)
```

3.6 维护

当我们插入或删除一个结点后，容均树的大小就发生了改变。这种改变有可能导致性质 (a) 或 (b) 被破坏。这时，我们需要用维护操作来修复这棵树。**维护** (MAINTAIN)操作是容均树中最具活力的一个独特过程。维护有四种情况，我们注意讨论。由于性质 (a) 和性质 (b) 是对称的，所以我们仅详细讨论性质 (a)。对于以 T 为根的伪容均树，有：

3.6.1 维护的四种情况

第一种情况： $t.left.left.size > t.right.size$

假设图4中的树是执行了 INSERT$(T.left, v)$，发生了 $A.size > R.size$，那么我们可以这样修复：

- 如图6，执行 RIGHT-ROTATE(T)。

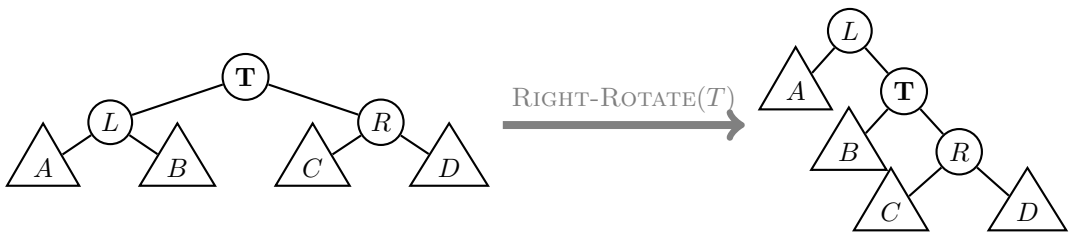


Figure 6. 节点描述同图4。

- 在这之后，树可能还不是容均树，可能出现 $C.size > B.size$ 或 $D.size > B.size$，所以有必要维护一下。

- 节点 L 的右子树有可能连续调整，需要再执行维护操作。

第二种情况： $T.left.right.size > T.right.size$

假设图4中的树是执行了 INSERT$(T.left, v)$，发生了 $B.size > R.size$，如图7。这种情况要复杂一些。

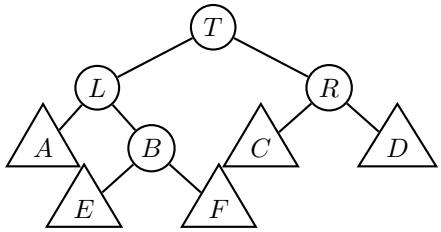


Figure 7. E 和 F 是结点 B 的左右子树。

- 如图8，执行 LEFT-ROTATE(L)。

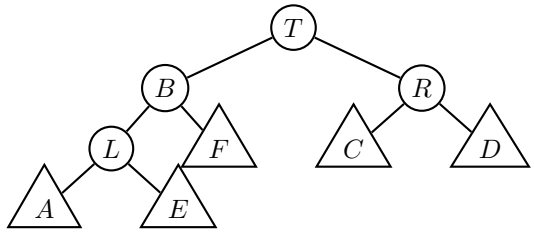


Figure 8. 执行 LEFT-ROTATE(L)。

- 然后执行 RIGHT-ROTATE(T)，就变成了图9。

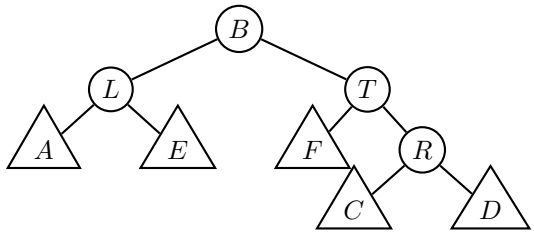


Figure 9. 执行 RIGHT-ROTATE(T)。

- 在经过两个巨大的旋转之后，整棵树就变得非常不可预料了。万幸的是，子树 A, E, F, R 依然是容均树，所以要依次修复 L 和 T。
- 子树都已经是容均树了，但 B 可能还有问题，所以还要修复一下。

第三种情况： $T.right.right.size > T.left.size$

与第一种情况正好相反，不再讨论。

第四种情况： $T.right.left.size > T.left.size$

与第二种情况正好相反，也不再讨论。

3.6.2 维护算法

通过一大堆分析，我们有下面的维护算法，用于修复以 T 为根的容均树。

```
MAINTAIN(T)
 1  if T.left.left.size > T.right.size
 2      RIGHT-ROTATE(T)
 3      MAINTAIN(T.right)
 4      MAINTAIN(T)
 5  elseif T.left.right.size > T.right.size
 6      LEFT-ROTATE(T.left)
 7      RIGHT-ROTATE(T)
 8      MAINTAIN(T.left)
 9      MAINTAIN(T.right)
10      MAINTAIN(T)
11  elseif T.right.right.size > T.left.size
12      LEFT-ROTATE(T)
13      MAINTAIN(T.left)
14      MAINTAIN(T)
15  elseif T.right.left.size > T.left.size
16      RIGHT-ROTATE(T.right)
17      LEFT-ROTATE(T)
18      MAINTAIN(T.left)
19      MAINTAIN(T.right)
20      MAINTAIN(T)
```

代码易懂，但是它在程序设计上并不是优秀的。它不仅重复编写了一半代码，也造成了算法效率低下。通常，在保证 T 的子树满足容均树性质的情况下，我们只需要检查两种情况。这里使用一个逻辑参数 $flag$ 来避免多次判断，当 $flag =$ FALSE 时，则检查情况一和二，反之检查情况三四。下面即是这一算法，注意每个判断语句的作用范围（已用缩进标出）。

```
MAINTAIN(T, flag)
 1  if flag = FALSE
 2      if T.left.left.size > T.right.size
 3          RIGHT-ROTATE(T)
 4      else
 5          if T.left.right.size > T.right.size
 6              LEFT-ROTATE(T.left)
 7              RIGHT-ROTATE(T)
 8          else return // 无需维护
 9  else // 即 flag = TRUE
10      if T.right.right.size > T.left.size
11          LEFT-ROTATE(T)
12      else
13          if T.right.left.size > T.left.size
14              RIGHT-ROTATE(T.right)
15              LEFT-ROTATE(T)
16          else return // 无需维护
17  MAINTAIN(T.left, FALSE)
18  MAINTAIN(T.right, TRUE)
19  MAINTAIN(T, TRUE)
20  MAINTAIN(T, FALSE)
```

细心会发现，我们省略了 MAINTAIN($T.left$, TRUE) 和 MAINTAIN($T.right$, FALSE)，这会不会出现问题？答案是不会。在4.1中，我们证明了这一点，并且证明了整个 MAINTAIN操作的均摊复杂度只有 $O(1)$。

3.7 查询与统计

3.7.1 查找

神马？还要写代码？这个操作可与二叉查找树的完全相同啊！不信请看第2.2.1节。

3.7.2 最大值和最小值

其实这个题目又是危言耸听。由于容均树本身已经维护了 $size$ 域，那么我们只需要 SELECT($T, 1$) 来取最大值，SELECT($T, T.size$) 取最小值。我们可爱的 SELECT(T, k) 函数返回树 T 在第 k 位置上的节点值。

```
SELECT(T, k)
1  r = T.left.size + 1
2  if k = r
3        return T.key
4  elseif k < r
5        return SELECT(T.left, k)
6  else
7        return SELECT(T.right, k − r)
```

3.7.3 前驱和后继

由于我们在容均树中没有定义与维护父亲域 p，所以这里给出递归版本的两个算法。此时的传递参数是根节点 T 与给定值 k。

```
SUCCESSOR(T, k)
 1  if T = NIL
 2        return k
 3  if T.key ≤ k
 4        return SUCCESSOR(T.right, k)
 5  else
 6        r = SUCCESSOR(T.left, k)
 7        if r = k
 8              return T.key
 9        else
10              return r
```

查找前驱亦然。

```
PREDECESSOR(T, k)
 1  if T = NIL
 2        return k
 3  if T.key ≥ k
 4        return PREDECESSOR(T.left, k)
 5  else
 6        r = PREDECESSOR(T.right, k)
 7        if r = k
 8              return T.key
 9        else
10              return r
```

3.7.4 排名（秩）

所谓的**排名** (*rank*)（也叫**秩**）就是求整棵树中从大到小排序的第 k 位元素。在普通二叉查找树中，执行这样一个操作需要维护 $size$ 域，然而在容均树中，我们已经用它来保持平衡了，无需“额外”维护这样的数据结构扩张。算法如此简单，让堆树情何以堪！

RANK(T, k)

1 **if** $T = \text{NIL}$
2 **return** 1
3 **if** $T.key \geq k$
4 **return** RANK$(T.left, k)$
5 **else**
6 **return** $T.left.size +$ RANK$(T.right, k) + 1$

嗯，和找前驱很类似。这是从大到小，那么从小到大就不言自明了。这里不再给出代码。

3.8 代码实现

嗯，实现了上面所有操作的 C++ 代码，值得一看：

```
#include <cstdio>
#define nil 0
const int MAX = 100000;
int key[MAX], left[MAX], right[MAX], size[MAX];
int T, node;
int record; // This is used for the commented Delete
inline void Left_Rotate(int &x) {
    int k = right[x];
    right[x] = left[k];
    left[k] = x;
    size[k] = size[x];
    size[x] = size[left[x]] + size[right[x]] + 1;
    x = k;
}
inline void Right_Rotate(int &y) {
    int k = left[y];
    left[y] = right[k];
    right[k] = y;
    size[k] = size[y];
    size[y] = size[left[y]] + size[right[y]] + 1;
    y = k;
}
void Maintain(int &T, bool flag);
```

```
24  void Insert(int &T, int v) {
25      if(T == nil) {
26          key[T = ++node] = v;
27          size[T] = 1;
28      } else {
29          size[T]++;
30          if(v < key[T])
31              Insert(left[T], v);
32          else
33              Insert(right[T], v);
34          Maintain(T, v >= key[T]);
35      }
36  }
37  /*
38  void Delete(int &T, int v) {
39      if(size[T] <= 2) {
40          record = key[T];
41          T = left[T] + right[T];
42          return;
43      }
44      size[T]--;
45      if(v == key[T]) {
46          Delete(left[T], v+1);
47          key[T] = record;
48          Maintain(T, true);
49      } else {
50          if(v < key[T])
51              Delete(left[T], v);
52          else
53              Delete(right[T], v);
54          Maintain(T, v < key[T]);
55      }
56  }
57  */
58  int Delete(int &T, int v) {
59      size[T]--;
60      if( (v == key[T]) || (v < key[T] && left[T] == nil) || (v > key
            [T] && right[T] == nil) ) {
61          int r = key[T];
62          if(left[T] == nil || right[T] == nil)
63              T = left[T] + right[T];
64          else
```

```
65              key[T] = Delete(left[T], key[T] + 1);
66          return r;
67      } else {
68          if(v < key[T])
69              return Delete(left[T], v);
70          else
71              return Delete(right[T], v);
72      }
73  }
74  /*
75  void Maintain(int &T) {
76      if(size[left[left[T]]] > size[right[T]]) {
77          Right_Rotate(T);
78          Maintain(right[T]);
79          Maintain(T);
80      } else if(size[right[left[T]]] > size[right[T]]) {
81          Left_Rotate(left[T]);
82          Right_Rotate(T);
83          Maintain(left[T]);
84          Maintain(right[T]);
85          Maintain(T);
86      } else if(size[right[right[T]]] > size[left[T]]) {
87          Left_Rotate(T);
88          Maintain(left[T]);
89          Maintain(T);
90      } else if(size[left[right[T]]] > size[left[T]]) {
91          Right_Rotate(right[T]);
92          Left_Rotate(T);
93          Maintain(left[T]);
94          Maintain(right[T]);
95          Maintain(T);
96      }
97  }
98  */
99  void Maintain(int &T, bool flag) {
100     if(flag == false) {
101         if(size[left[left[T]]] > size[right[T]])
102             Right_Rotate(T);
103         else {
104             if(size[right[left[T]]] > size[right[T]]) {
105                 Left_Rotate(left[T]);
106                 Right_Rotate(T);
```

```
107             } else return;
108         }
109     } else {
110         if(size[right[right[T]]] > size[left[T]])
111             Left_Rotate(T);
112         else {
113             if(size[left[right[T]]] > size[left[T]]) {
114                 Right_Rotate(right[T]);
115                 Left_Rotate(T);
116             } else return;
117         }
118     }
119     Maintain(left[T], false);
120     Maintain(right[T], true);
121     Maintain(T, true);
122     Maintain(T, false);
123 }
124 int Search(int x, int k) {
125     if(x == nil || k == key[x])
126         return x;
127     if(k < key[x])
128         return Search(left[x], k);
129     else
130         return Search(right[x], k);
131 }
132 int Select(int T, int k) {
133     int r = size[left[T]] + 1;
134     if(k == r)
135         return key[T];
136     else if(k < r)
137         return Select(left[T], k);
138     else
139         return Select(right[T], k - r);
140 }
141 int Succ(int T, int k) {
142     if(T == nil)
143         return k;
144     if(key[T] <= k)
145         return Succ(right[T], k);
146     else {
147         int r = Succ(left[T], k);
148         if(r == k)
```

```
149             return key[T];
150         else
151             return r;
152     }
153 }
154 int Pred(int T, int k) {
155     if(T == nil)
156         return k;
157     if(key[T] >= k)
158         return Pred(left[T], k);
159     else {
160         int r = Pred(right[T], k);
161         if(r == k)
162             return key[T];
163         else
164             return r;
165     }
166 }
167 int Rank(int T, int k) {
168     if(T == nil)
169         return 1;
170     if(key[T] >= k)
171         return Rank(left[T], k);
172     else
173         return size[left[T]] + Rank(right[T], k) + 1;
174 }
175 int main() {
176     int q, a, b;
177     scanf("%d", &q);
178     for(int i=0; i<q; i++) {
179         scanf("%d%d", &a, &b);
180         switch(a) {
181             case 1: Insert(T, b); break;
182             case 2: Delete(T, b); break;
183             case 3: printf("%d\n", Search(T, b)); break;
184             case 4: printf("%d\n", Rank(T, b)); break;
185             case 5: printf("%d\n", Select(T, b)); break;
186             case 6: printf("%d\n", Pred(T, b)); break;
187             case 7: printf("%d\n", Succ(T, b)); break;
188         }
189     }
190     return 0;
```

```
191 }
```

4 分析应用

4.1 数学推倒

嗯？“推倒”？不是啦。是“推导”。很多人认为，数学证明对于信息学竞赛来说，真是不如没有——直接告诉我们时间复杂度就好，那一堆数学式子都要让人疯掉了。我也曾经这样想过，然而，在计算机科学中，数学证明是非常重要的。我们学习的每一个算法，每一种数据结构，它们的时空效率都是要进行严格推导的，这样才能保证其正确性。《算法导论》作为这一类的“圣经”，其中证明十分丰富、严谨。在**高德纳** (*Donald E. Knuth*)的[七卷本] 巨著《计算机程序设计艺术》中，严格的数学证明也是处处可见。更可怕的是，为了排版出精美的数学式子，高德纳用了十几年的时间，开发了世界上最伟大的排版软件——TEX——这篇文章就是用其衍生产品 XeLaTeX 排版而成。他的疯狂不限于此，从 TEX 第三版开始，之后的升级是在小数点后加入一个新数位，使之越来越接近圆周率 π，当前版本 3.1415926 显示了这款软件的稳定性。同样的，用于字体输出的 METAFONT则趋于常数 e。

数学是计算机的灵魂！不用举例了。下面是我们的推导。

4.1.1 容均树的高度

设 $f_h(h \in \mathbb{N}^*)$ 是高度为 h 的节点个数最少的容均树节点个数，我们有：

$$f_h = \begin{cases} 1, & h = 0, \\ 2, & h = 1, \\ f_{h-1} + f_{h-2} + 1, & h > 1. \end{cases}$$

证明: 易证

$$f_0 = 1, f_1 = 2.$$

对于每个 $h > 1$，设 T 指向一棵高度为 h 的容均树，它包含一个高度为 $h-1$ 的子树。不妨设其就是左子树。由 f_h，我们有

$$T.left.size \geq f_{h-1}.$$

且在左子树上有一棵高为 $h-2$ 的子树，由容均树性质 (b) 可得

$$T.right.size \geq f_{h-2}.$$

所以，

$$f_h \geq f_{h-1} + f_{h-2} + 1 (h > 1).$$

构造一棵有 f_h 个节点的容均树，其高度为 h，这棵树记为 R_h。我们定义

$$R_h = \begin{cases} \text{只有一个节点的容均树}, & h = 0, \\ \text{有两个节点的容均树}, & h = 1, \\ \text{以}R_{h-1}\text{和}R_{h-2}\text{为其子树的容均树}, & h > 1. \end{cases}$$

显然，我们得到

$$f_h = f_{h-1} + f_{h-2} + 1 (h > 1).$$

□

我们对 f_h 的定义无疑是一个类**斐波纳契数列** (*Fibonacci Sequence*)的数列，于是我们得到

$$\begin{aligned} f_h &= \sum_{i=0}^{h} F_i \\ &= F_{h+2} - 1 \\ &= \left\| \frac{\phi^{h+3}}{\sqrt{5}} \right\| - 1 \\ &= \frac{\phi^{h+3} - \hat{\phi}^{h+3}}{\sqrt{5}} - 1 \end{aligned}$$

其中， $\phi = \dfrac{1+\sqrt{5}}{2}$ 是黄金比， $\hat{\phi} = \dfrac{1-\sqrt{5}}{2}$， $\|x\|$ 即取 x 的**范数** (*Norm*)。

定理：一棵有 n 个节点的容均树，在最坏情况下高度是满足 $f_h \le n$ 的最大 h。

设 $\max h$ 是有 n 个节点的容均树最大高度，由上定理，近似有

$$\begin{aligned} f_{\max h} = \left\| \frac{\phi^{h+3}}{\sqrt{5}} \right\| - 1 &\le n &&\Rightarrow \\ \frac{\phi^{h+3}}{\sqrt{5}} &< n + 1.5 &&\Rightarrow \\ \max h &< \log_{\phi}^{\sqrt{5}(n+1.5)} - 3 &&\Rightarrow \\ \max h &< 1.44 \lg^{n+1.5} - 1.33 \end{aligned}$$

我们得到容均树的高度为 $O(\lg n)$。

4.1.2 维护操作的正确性与时间复杂度

有了前面的结论，我们可以很容易证明 MAINTAIN过程是 $O(1)$ 的。

评价一棵二叉查找树有一个非常重要的参数，即**节点平均深度**，它定义为节点深度和与节点个数之比。通常，这个值越小，二叉查找树就会越优秀。对于节点数 n 为常数，我们期望节点深度和 (SD) 尽可能小。

我们来探讨 SD，其意义在它能约束 MAINTAIN的运行时间。回顾每次 MAINTAIN中执行旋转操作的条件，会发现每次旋转后都会使 SD 减小。

在第一种情况下，如图6，我们看到旋转前后，

$$\Delta SD = T.right.size - T.left.left.size < 0.$$

在第二种情况下，图7旋转后变成图8，

$$\Delta SD = T.right.size - T.left.right.size - 1 < 1.$$

所以，对于高度为 $O(\lg n)$ 的树， SB 总为 $n \lg n$。在执行插入操作后， SD 只增加了 $O(\lg n)$。因此有

$$\begin{aligned} SD &= nO(\lg n) = O(\lg n) \quad \Rightarrow \\ T &= O(\lg n) \end{aligned}$$

其中， T 是 MAINTAIN中旋转操作的执行次数。所以，MAINTAIN操作的平摊运行时间就是

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

由于容均树的高度为 $O(\lg n)$，MAINTAIN是 $O(1)$，所以所有主要操作的时间复杂度都是 $O(\lg n)$。

简化后的 MAINTAIN省略了 MAINTAIN$(T.left, \text{TRUE})$ 和 MAINTAIN$(T.right, \text{FALSE})$，这并无问题。

对于第二种情况，如图7，我们有

$$\begin{aligned} L.size &\le 2 \cdot R.size + 1 & & \Rightarrow \\ B.size &\le \frac{2 \cdot L.size - 1}{3} & \le \frac{4 \cdot R.size + 1}{3} & \Rightarrow \\ E.size, F.size &\le \frac{2 \cdot B.size - 1}{3} & \le \frac{8 \cdot R.size + 3}{9} & \Rightarrow \\ E.size, F.size &\le \left\lfloor \frac{8 \cdot R.size + 3}{9} \right\rfloor & \le R.size & \end{aligned}$$

因此 MAINTAIN$(T.right, \text{FALSE})$ 的效果与 MAINTAIN(T, FALSE) 等价，予以省略。同样地，MAINTAIN$(T.left, \text{TRUE})$ 也可以省略。

如图8，我们有

$$\begin{cases} A.size \ge E.size \\ F.size \le R.size \end{cases}$$

这些不平衡也意味着 $E.left.size, E.right.size < A.size$， $F.left.size, F.right.size < R.size$，又一次证明了这两个操作都可以省略。

这就是整个维护 (MAINTAIN) 过程。

4.1.3 删除操作的正确性

对第一个删除操作无需证明，因为它是个普通算法。对于简化后的 DELETE，我们需要证明其正确性。

声明命题 $P(n^*)$，对于使用简化删除和插入了 n^* 个节点的容均树，高度为 $O(\lg n^*)$。我们使用数学归纳法，证明对于 $\forall n^* \in \mathbb{N}^*$， $P(n^*)$ 成立。

证明: 假设节点 T 已经被 MAINTAIN(T, FALSE) 维护过，则有

$$\begin{array}{ll} \frac{T.left.size-1}{2} & \leq T.right.size \\ T.left.size & \leq \frac{2 \cdot T.size-1}{3} \end{array} \Rightarrow$$

因此，如果一个节点到根的路径上的所有节点都已经维护过，那么这个节点的深度为 $O(\lg n)$。

(1) 对于 $n^* = 1$，显然 $P(n^*)$ 为真；
(2) 假设对于 $n^* < k$， $P(n^*)$ 为真。对于 $n^* = k$，在最后的连续插入之后，所有维护过的节点一定连成一棵树。对于在树中的每个叶节点，由它指向的子树也不会在维护中改变。故子树中节点深度不会大于 $O(\lg n^*) + O(\lg n) = O(\lg n^*)$；
(3) 因此，当 $n^* = 1, 2, 3, ...$ 时， $P(n^*)$ 为真。

□

这样，既证明了简化 DELETE的正确性，又再一次证明了 MAINTAIN的平摊为 $O(1)$。

4.2 优势和局限性

陈启峰在论文中提到了容均树的七大优点：

1. 速度快。随机插入删除，用时少。
2. 性能高。随机插入，平均深度和高度、旋转次数少。
3. 调试易。我们可以简单输入一棵容均树来测试，然后加入 MAINTAIN操作，此后只要调试 MAINTAIN便可。不像堆树，容均树没有随机性，稳定性更高，更易调试。
4. 代码短。仅仅较普通二叉查找树多了一个 MAINTAIN操作和两个对称的旋转操作。
5. 空间少。容均树的 $size$ 域不仅用于维护，还可以用于 SELECT和 RANK操作，不像堆树、红黑树、AVL 树等还要维护附加域。
6. 功能强。插入、删除、查找、顺序统计一应俱全。
7. 应用广。可以解决多数平衡树问题。

在陈启峰的演示文稿中，写到了容均树的缺点。对于“人”字形数据（如图10），容均树会退化成 $O(n)$。但这种数据比有序队列还要少见，也就可以忽视了。

4.3 竞赛应用

毫无疑问，平衡树在竞赛中应用广泛。比如：

郁闷的出纳员 (NOI 2006)

【问题描述】

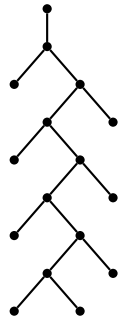


Figure 10. 极为少见的使容均树退化的“人”字形数据形成的容均树。

OIER 公司是一家大型专业化软件公司，有着数以万计的员工。作为一名出纳员，我的任务之一便是统计每位员工的工资。这本来是一份不错的工作，但是令人郁闷的是，我们的老板反复无常，经常调整员工的工资。如果他心情好，就可能把每位员工的工资加上一个相同的量。反之，如果心情不好，就可能把他们的工资扣除一个相同的量。我真不知道除了调工资他还做什么其它事情。

工资的频繁调整很让员工反感，尤其是集体扣除工资的时候，一旦某位员工发现自己的工资已经低于了合同规定的工资下界，他就会立刻气愤地离开公司，并且再也不会回来了。每位员工的工资下界都是统一规定的。每当一个人离开公司，我就要从电脑中把他的工资档案删去，同样，每当公司招聘了一位新员工，我就得为他新建一个工资档案。

老板经常到我这边来询问工资情况，他并不问具体某位员工的工资情况，而是问现在工资第 k 多的员工拿多少工资。每当这时，我就不得不对数万个员工进行一次漫长的排序，然后告诉他答案。

好了，现在你已经对我的工作了解不少了。正如你猜的那样，我想请你编一个工资统计程序。怎么样，不是很困难吧？

【输入文件】

`cashier.in` 第一行有两个非负整数 n 和 min。n 表示下面有多少条命令，min 表示工资下界。

接下来的 n 行，每行表示一条命令。命令可以是以下四种之一：

名称	格式	作用
I 命令	`I␣k`	新建一个工资档案，初始工资为 k。如果某员工的初始工资低于工资下界，他将立刻离开公司。
A 命令	`A␣k`	把每位员工的工资加上 k。
S 命令	`S␣k`	把每位员工的工资扣除 k。
F 命令	`F␣k`	查询第 k 多的工资。

␣表示一个空格，I 命令、A 命令、S 命令中的 k 是一个非负整数，F 命令中的 k 是一个正整数。

在初始时，可以认为公司里一个员工也没有。

【输出文件】

输出文件 cashier.out 的行数为 F 命令的条数加一。

对于每条 F 命令，你的程序要输出一行，仅包含一个整数，为当前工资第 k 多的员工所拿的工资数，如果 k 大于目前员工的数目，则输出 -1。

输出文件的最后一行包含一个整数，为离开公司的员工的总数。

【样例输入输出】

cashier.in	cashier.out
9 10 I 60 I 70 S 50 F 2 I 30 S 15 A 5 F 1 F 2	10 20 -1 2

【约定】

- I 命令的条数不超过 100000
- A 命令和 S 命令的总条数不超过 100
- F 命令的条数不超过 100000
- 每次工资调整的调整量不超过 1000
- 新员工的工资不超过 100000

【评分方法】

对于每个测试点，如果你输出文件的行数不正确，或者输出文件中含有非法字符，得分为 0。

否则你的得分按如下方法计算：如果对于所有的 F 命令，你都输出了正确的答案，并且最后输出的离开公司的人数也是正确的，你将得到 10 分；如果你只对所有的 F 命令输出了正确答案，得 6 分；如果只有离开公司的人数是正确的，得 4 分；否则得 0 分。

【题目分析】

根据题目，我们可以发现它其实就是对数据结构的基本操作。我们使用容均树，可以直接实现插入和查找操作。对于 S 命令，我们编写如下擦除算法遍历删除员工。

```
ERASE(T, k)
 1  if T = NIL
 2      return T
 3  if T.key < k
 4      r = T.left.size + ERASE(T.right, k)
 5      CLEAR(T.left)
 6      T.size− = r
 7      DELETE(T, T.key)
 8      return r + 1
 9  else
10      r = ERASE(T.left, k)
11      T.size− = r
12      return r
```

其中，我们定义清理过程如下：

```
CLEAR(T)
1  if T = NIL
2      return
3  CLEAR(T.left)
4  CLEAR(T.right)
5  T.size = 0
6  T = NIL
7  T.key = T.left = T.right = 0
```

在主程序中，我们对于每次插入操作对计数器 T 加一，这样最后离开公司员工总数就是 $I - T.size$。我们定义 add 为工资的增减量，用以避免大规模修改数据，所以 A 操作和 S 操作只要对 add 进行增减即可。由于 S 操作会导致有员工离开，所以调用 ERASE$(T, min - add)$，并对容均树进行维护。主程序算法如下：

MAIN()

```
1  input n, min
2  for i = 1 to n
3      input c, b
4      if c ="I"
5          if b ≥ min
6              INSERT(T, b − add)
7              I + +
8      elseif c ="A"
9          add+ = b
10     elseif c ="S"
11         add− = b;
12         ERASE(T, min − add)
13         MAINTAIN(T, TRUE)
14         MAINTAIN(T, FALSE)
15     elseif c ="F"
16         if b > T.size
17             print −1
18         else
19             print SELECT(T, T.size − b + 1) + add
20 print I − T.size
```

实际代码如斯：

```
#include <cstdio>
#define nil 0
const int MAX = 100000;
int key[MAX], left[MAX], right[MAX], size[MAX];
int T, node;
int n, min, add, I;
inline void Left_Rotate(int &x) {
    int k = right[x];
    right[x] = left[k];
    left[k] = x;
    size[k] = size[x];
    size[x] = size[left[x]] + size[right[x]] + 1;
    x = k;
}
inline void Right_Rotate(int &y) {
    int k = left[y];
    left[y] = right[k];
```

```
18      right[k] = y;
19      size[k] = size[y];
20      size[y] = size[left[y]] + size[right[y]] + 1;
21      y = k;
22  }
23  void Maintain(int &T, bool flag);
24  void Insert(int &T, int v) {
25      if(T == nil) {
26          key[T = ++node] = v;
27          size[T] = 1;
28          left[T] = right[T] = nil;
29      } else {
30          size[T]++;
31          if(v < key[T])
32              Insert(left[T], v);
33          else
34              Insert(right[T], v);
35          Maintain(T, v >= key[T]);
36      }
37  }
38  int Delete(int &T, int v) {
39      size[T]--;
40      if( (v == key[T]) || (v < key[T] && left[T] == nil) || (v > key
            [T] && right[T] == nil) ) {
41          int r = key[T];
42          if(left[T] == nil || right[T] == nil)
43              T = left[T] + right[T];
44          else
45              key[T] = Delete(left[T], key[T] + 1);
46          return r;
47      } else {
48          if(v < key[T])
49              return Delete(left[T], v);
50          else
51              return Delete(right[T], v);
52      }
53  }
54  void Clear(int &T) {
55      if(T == nil)
56          return;
57      Clear(left[T]);
58      Clear(right[T]);
```

```
59      size[T] = 0; T = nil;
60      key[T] = left[T] = right[T] = 0;
61  }
62  int Erase(int &T, int k) {
63      if(T == nil)
64          return T;
65      int r = 0;
66      if(key[T] < k) {
67          r = size[left[T]] + Erase(right[T], k);
68          Clear(left[T]);
69          size[T] -= r;
70          Delete(T, key[T]);
71          return r+1;
72      } else {
73          r = Erase(left[T], k);
74          size[T] -= r;
75          return r;
76      }
77  }
78  void Maintain(int &T, bool flag) {
79      if(flag == false) {
80          if(size[left[left[T]]] > size[right[T]])
81              Right_Rotate(T);
82          else {
83              if(size[right[left[T]]] > size[right[T]]) {
84                  Left_Rotate(left[T]);
85                  Right_Rotate(T);
86              } else return;
87          }
88      } else {
89          if(size[right[right[T]]] > size[left[T]])
90              Left_Rotate(T);
91          else {
92              if(size[left[right[T]]] > size[left[T]]) {
93                  Right_Rotate(right[T]);
94                  Left_Rotate(T);
95              } else return;
96          }
97      }
98      Maintain(left[T], false);
99      Maintain(right[T], true);
100     Maintain(T, true);
```

```
101     Maintain(T, false);
102 }
103 int Select(int T, int k) {
104     int r = size[left[T]] + 1;
105     if(k == r)
106         return key[T];
107     else if(k < r)
108         return Select(left[T], k);
109     else
110         return Select(right[T], k - r);
111 }
112 int main() {
113     freopen("cashier.in", "r", stdin);
114     freopen("cashier.out", "w", stdout);
115     scanf("%d␣%d\n", &n, &min);
116     char c; int b;
117     for(int i=0; i<n; i++) {
118         scanf("%c␣%d\n", &c, &b);
119         switch(c) {
120             case 'I':
121                 if(b >= min) {
122                     Insert(T, b-add);
123                     I++;
124                 }
125                 break;
126             case 'A':
127                 add += b;
128                 break;
129             case 'S':
130                 add -= b;
131                 Erase(T, min-add);
132                 Maintain(T, true);
133                 Maintain(T, false);
134                 break;
135             case 'F':
136                 if(b>size[T])
137                     printf("-1\n");
138                 else
139                     printf("%d\n", Select(T, size[T]-b+1) + add);
140                 break;
141         }
142     }
```

```
143     printf("%d\n", I-size[T]);
144     return 0;
145 }
```

其他题目

其他相关题目如《生日快乐》(NOI 2006)、《排名系统》(HAOI 2008+) 等。这里就不列出了。

5 总结

当然，平衡树不止这一种，像线段树、B 树，树状数组等都是在竞赛中经常用到的数据结构。在一些搜索问题、动态规划问题中，平衡的二叉查找树也大有用途。优先队列的实现有时也用到平衡树。我们应该积极努力学习，膜拜神牛，在竞赛的道路上越走越远，越陷越深，越……

The End.

References

[陈启峰] Size Balanced Tree, 陈启峰, 2006-12-29

[郭家宝] 随机平衡二叉查找树 Treap 的分析与应用, 郭家宝, 2010

[维基] 维基百科, 自由的百科全书, 2000–?

[算导] 算法导论, 可门, 雷瑟桑, 罗纳德, 克里夫特斯坦, 2001–2009

[七卷本] 计算机程序设计艺术, 高德纳, 1968–?

[算法果壳] 算法技术手册, 海涅曼, 波利切, 塞克欧, 2009–2010