Size Balanced Tree

**来自"NOCOW"**

跳转到: [导航](http://www.nocow.cn/index.php/Size_Balanced_Tree#column-one), [搜索](http://www.nocow.cn/index.php/Size_Balanced_Tree#searchInput)

**Size Balanced Tree**(**SBT**)是一种[平衡二叉查找树](http://www.nocow.cn/index.php/%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91)。它的论文由中国广东中山纪念中学的陈启峰于2006年底完成，并在Winter Camp 2007中发表。由于SBT的拼写很容易找到中文谐音，它常被中国的[OIer](http://www.nocow.cn/index.php/OIer)们戏称为“傻X树”、“Super BT”等。但它的性能并不SB，编写起来也并不BT。恰恰相反，SBT易于实现，且据陈启峰论文中所言，“这是目前为止速度最快的高级[二叉搜索树](http://www.nocow.cn/index.php/%E4%BA%8C%E5%8F%89%E6%90%9C%E7%B4%A2%E6%A0%91)”。它能在O(logn)的时间内完成所有[BST](http://www.nocow.cn/index.php/BST)的相关操作。而且由于SBT赖以保持平衡的是Size域而不是其他“无用”的域，它可以很方便地实现动态顺序统计中的select和rank。

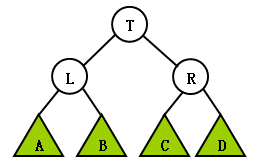
|  |
| --- |
| **目录**  [[隐藏](javascript:toggleToc())]   * [1 性质](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.80.A7.E8.B4.A8) * [2 旋转](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.97.8B.E8.BD.AC)   + [2.1 左旋转](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.B7.A6.E6.97.8B.E8.BD.AC)   + [2.2 右旋转](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.8F.B3.E6.97.8B.E8.BD.AC) * [3 保持性质(Maintain)](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E4.BF.9D.E6.8C.81.E6.80.A7.E8.B4.A8.28Maintain.29) * [4 基本操作](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.9F.BA.E6.9C.AC.E6.93.8D.E4.BD.9C)   + [4.1 查找](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.9F.A5.E6.89.BE)   + [4.2 取大/取小](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.8F.96.E5.A4.A7.2F.E5.8F.96.E5.B0.8F)   + [4.3 后继](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.90.8E.E7.BB.A7)   + [4.4 前趋](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.89.8D.E8.B6.8B)   + [4.5 插入](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.8F.92.E5.85.A5)   + [4.6 删除](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.88.A0.E9.99.A4) * [5 动态顺序统计操作](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.8A.A8.E6.80.81.E9.A1.BA.E5.BA.8F.E7.BB.9F.E8.AE.A1.E6.93.8D.E4.BD.9C)   + [5.1 检索具有给定排序的元素](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.A3.80.E7.B4.A2.E5.85.B7.E6.9C.89.E7.BB.99.E5.AE.9A.E6.8E.92.E5.BA.8F.E7.9A.84.E5.85.83.E7.B4.A0)   + [5.2 求元素的秩](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.B1.82.E5.85.83.E7.B4.A0.E7.9A.84.E7.A7.A9) * [6 性能分析](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.80.A7.E8.83.BD.E5.88.86.E6.9E.90) * [7 源码](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E6.BA.90.E7.A0.81) * [8 参考资料](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E5.8F.82.E8.80.83.E8.B5.84.E6.96.99) * [9 例题:NOI2004郁闷的出纳员](http://www.nocow.cn/index.php/Size_Balanced_Tree#.E4.BE.8B.E9.A2.98:NOI2004.E9.83.81.E9.97.B7.E7.9A.84.E5.87.BA.E7.BA.B3.E5.91.98) |

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=1)] 性质

Size Balanced Tree（SBT）是一种通过大小（Size）域来保持平衡的二叉搜索树，它也因此得名。它总是满足：  
对于SBT的每一个结点 t：

1. 性质(a) s[right[t] ]≥s[left[left[t]]]，s[right[left[t]]]
2. 性质(b) s[left[t] ]≥s[right[right[t]]]，s[left[right[t]]]

即每棵子树的大小不小于其兄弟的子树大小。

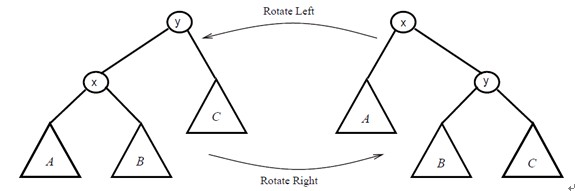
[](http://www.nocow.cn/index.php/Image:Sbt1.PNG)  
图1

如图（圈代表结点，三角代表SBT，下同）：

1. s[R] ≥ s[A]，s[B]
2. s[L] ≥ s[C]，s[D]

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=2)] 旋转

SBT的旋转（Rotations）与其他许多高级BST相同。它是下面提到的Maintain操作的基础。

[](http://www.nocow.cn/index.php/Image:Sbt2.PNG)  
图2

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=3)] **左旋转**

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

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=4)] **右旋转**

Right-Rotate(t)

1 k ← left[t]

2 left[t] ← right[k]

3 right[k] ← t

4 s[k] ← s[t]

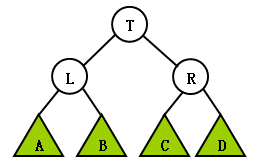
5 s[t] ← s[left[t]] + s[right[t]] + 1

6 t ← k

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=5)] 保持性质(Maintain)

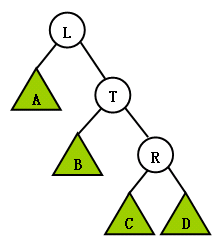
当我们插入或删除一个结点后，SBT的大小就发生了改变。这种改变有可能导致性质(a)或(b)被破坏。这时，我们需要用**Maintain**操作来修复这棵树。**Maintain**操作是SBT中最具活力的一个独特过程；Maintain(T)用于修复以T为根的 SBT。调用Maintain(T)的前提条件是T的子树都已经是SBT了。  
我们需要讨论的有4种情况。由于性质a和性质b是对称的，所以我们仅仅详细的讨论性质a。

1. **第一种情况：s[left[left[t]]>s[right[t]]**

[](http://www.nocow.cn/index.php/Image:Sbt1.PNG)  
图3（同图1）

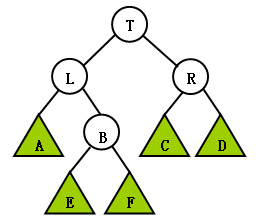
如图3，执行完Insert(left[t],v)后发生S[A]>S[R]，我们可以执行以下的指令来修复SBT：

* 1. 首先执行Right-Ratote(t)，这个操作让图3变成图4；

[](http://www.nocow.cn/index.php/Image:Sbt4.PNG)  
图4

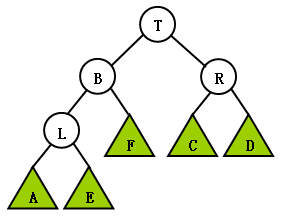
* 1. 在这之后，有时候这棵树还仍然不是一棵SBT，因为 s[C]>s[B] 或者 s[D]>s[B] 也是可能发生的。所以就有必要继续调用Maintian(t)。
  2. 结点L的右子树有可能被连续调整，因为有可能由于性质的破坏需要再一次运行Maintain(L)。

1. **第二种情况：s[right[left[t]]>s[right[t]]**

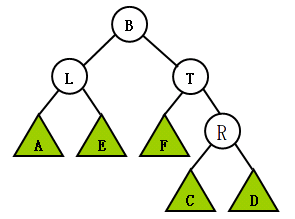
[](http://www.nocow.cn/index.php/Image:Sbt5.PNG)  
图5

在执行完Insert(left[t],v)后发生s[B]>s[R]，如图5，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：

* 1. 在执行完Left-Ratote(L)后，图5就会变成下面图6那样了。

[](http://www.nocow.cn/index.php/Image:Sbt6.PNG)  
图6

* 1. 然后执行Right-Ratote(T)，最后的结果就会由图6转变成为下面的图7。

[](http://www.nocow.cn/index.php/Image:Sbt7.PNG)  
图7

* 1. 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7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。
  2. 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)。

1. **第三种情况：s[right[right[t]]>s[left[t]]**  
   与情况1对称。
2. **第四种情况：s[left[right[t]]>s[left[t]]**  
   与情况2对称。

通过前面的分析，很容易写出一个普通的Maintain。

Maintain (t)

01 If s[left[left[t]]>s[right[t]] then //case1

02 Right-Rotate(t)

03 Maintain(right[t])

04 Maintain(t)

05 Exit

06 If s[right[left[t]]>s[right[t]] then //case2

07 Left-Rotate(left[t])

08 Right-Rotate(t)

09 Maintain(left[t])

10 Maintain(right[t])

11 Maintain(t)

12 Exit

13 If s[right[right[t]]>s[left[t]] then //case1'

14 Left-Rotate(t)

15 Maintain(left[t])

16 Maintain(t)

17 Exit

18 If s[left[right[t]]>s[left[t]] then //case2'

19 Right-Rotate(right[t])

20 Left-Rotate(t)

21 Maintain(left[t])

22 Maintain(right[t])

23 Maintain(t)

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，来避免毫无意义的判断。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)

01 If flag=false then

02 If s[left[left[t]]>s[right[t]] then //case1

03 Right-Rotate(t)

04 Else

05 If s[right[left[t]]>s[right[t]] then //case2

06 Left-Rotate(left[t])

07 Right-Rotate(t)

08 Else //needn’t repair

09 Exit

10 Else

11 If s[right[right[t]]>s[left[t]] then //case1'

12 Left-Rotate(t)

13 Else

14 If s[left[right[t]]>s[left[t]] then //case2'

15 Right-Rotate(right[t])

16 Left-Rotate(t)

17 Else //needn’t repair

18 Exit

19 Maintain(left[t],false) //repair the left subtree

20 Maintain(right[t],true) //repair the right subtree

21 Maintain(t,false) //repair the whole tree

22 Maintain(t,true) //repair the whole tree

为什么Maintain(left[t],true)和Maintain(right[t],false)被省略了呢？您可以在陈启峰论文第六部分的分析中找到答案。  
其他可以从论文中获得的信息：每次SBT后树的总深度递减的证明；Maintain的平摊运行时间是O(1)的证明（也就是说你不必担心Maintain这个递归过程是否会永不停止）等。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=6)] 基本操作

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=7)] **查找**

SBT的查找操作与普通BST完全相同。下面的过程将返回指向目标节点的指针。

Search(t,k)

1 if x=NIL or k=key[t]//t=nil???不明白……

2 then return x

3 if k<key[x]

4 then return Search(left[x],k)

5 else return Search(right[x],k)

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=8)] **取大/取小**

由于SBT本身已经维护了size，因此这两项可用Select操作完成。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=9)] **后继**

SBT的后继操作与普通BST完全相同。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=10)] **前趋**

SBT的前趋操作与普通BST完全相同。它与上面的后继操作对称。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=11)] **插入**

SBT的插入操作很简单。它仅仅比普通BST的多出了一个Maintain操作和对s的简单维护。下面这个过程将一个节点v插入SBT中。

Insert (t,v)

1 If t=0 then

2 t ← v

3 Else

4 s[t] ← s[t]+1

5 If v<key[t] then

6 Insert(left[t],v)

7 Else

8 Insert(right[t],v)

9 Maintain(t,v≥key[t])

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=12)] **删除**

与普通维护size域的BST删除相同。  
关于无需Maintain的说明by sqybi：  
在删除之前，可以保证整棵树是一棵SBT。当删除之后，虽然不能保证这棵树还是SBT，但是这时整棵树的最大深度并没有改变，所以时间复杂度也不会增加。这时，Maintain就显得是多余的了。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=13)] 动态顺序统计操作

由于SBT本来就是靠着size域来维持平衡的，当我们进行动态顺序统计操作时，我们就无需去“额外”维护一个size域来进行数据结构的扩张。这样，以下操作就与其他高级BST扩张后的动态顺序统计操作完全一样了。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=14)] **检索具有给定排序的元素**

下面这个过程将返回一个指向以x为根的子树中包含第i小关键字的节点的指针。

Select(x,i)

1 r ← size[left[x]] + 1

2 if(i=r)

3 then return x

4 else if i<r

5 then return Select(left[x],i)

6 else return Select(right[x],i-r)

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=15)] **求元素的秩**

SBT的rank操作与普通BST完全相同。

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=16)] 性能分析

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

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=17)] 源码

[C](http://www.nocow.cn/index.php/Code:SBT_C)  
[C++](http://www.nocow.cn/index.php/Code:SBT_C%2B%2B)  
[Pascal](http://www.nocow.cn/index.php/Code:SBT_Pascal)

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=18)] 参考资料

* [Size Balanced Tree], 陈启峰
* [Size Balanced Tree](http://combooleaf.blog.hexun.com/8039549_d.html), 陈启峰, Translated by BambooLeaf
* [Introduction to Algorithms](http://www.nocow.cn/index.php/Introduction_to_Algorithms)

[[编辑](http://www.nocow.cn/index.php?title=Size_Balanced_Tree&action=edit&section=19)] 例题:NOI2004郁闷的出纳员

【问题描述】

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

\_（下划线）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】

9 10

I 60

I 70

S 50

F 2

I 30

S 15

A 5

F 1

F 2

【样例输出】

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

数组实现的 Size Balanced Tree

#include <fstream>

using namespace std;

ifstream fin("cashier.in");

ofstream fout("cashier.out");

const unsigned int MAX\_N=100001;

int ZUISHAO; *//最低工资*

int ADD\_PAY=0;

unsigned int C\_NUM=0;

int front = 0;

struct node

{

int key;

int size, llink, rlink;

}OIER[MAX\_N];

void LeftRotate(int &x)*//左旋*

{

int y = OIER[x].rlink;

if (y == 0) return;

OIER[x].rlink = OIER[y].llink;

OIER[y].llink = x;

OIER[y].size = OIER[x].size;

OIER[x].size = OIER[OIER[x].llink].size+OIER[OIER[x].rlink].size+1;

x = y;

}

void RightRotate(int &x)*//右旋*

{

int y = OIER[x].llink;

if (y == 0) return;

OIER[x].llink = OIER[y].rlink;

OIER[y].rlink = x;

OIER[y].size = OIER[x].size;

OIER[x].size = OIER[OIER[x].llink].size+OIER[OIER[x].rlink].size+1;

x = y;

}

void Maintain(int &root, bool flag)*//维护 SBT 树*

{

if (!root) return;

if (flag)

{

if(OIER[root].llink && OIER[OIER[root].llink].llink

&& (!OIER[root].rlink || OIER[OIER[OIER[root].llink].llink].size > OIER[OIER[root].rlink].size))

RightRotate(root);

else if(OIER[root].llink && OIER[OIER[root].llink].rlink

&& (!OIER[root].rlink || OIER[OIER[OIER[root].llink].rlink].size > OIER[OIER[root].rlink].size))

{

LeftRotate(OIER[root].llink);

RightRotate(root);

}

else return;

}

else

{

if (OIER[root].rlink && OIER[OIER[root].rlink].rlink

&& (!OIER[root].llink || OIER[OIER[OIER[root].rlink].rlink].size > OIER[OIER[root].llink].size))

LeftRotate(root);

else if (OIER[root].rlink && OIER[OIER[root].rlink].llink

&& (!OIER[root].llink || OIER[OIER[OIER[root].rlink].llink].size > OIER[OIER[root].llink].size))

{

RightRotate(OIER[root].rlink);

LeftRotate(root);

}

else return;

}

Maintain(OIER[root].llink, **true**);

Maintain(OIER[root].rlink, **false**);

Maintain(root, **true**);

Maintain(root, **false**);

}

void Insert(int &root, int x)*//插入关键字 x*

{

if (!root)

{

root = ++front;

OIER[root].key = x;

OIER[root].size = 1;

}

else

{

++OIER[root].size;

Insert(x <= OIER[root].key ? OIER[root].llink : OIER[root].rlink, x);

Maintain(root, x <= OIER[root].key);

}

}

int Delete(int &root)*//删除*

{

int t=0,sum=0;

if(!root) return root;

if(OIER[root].key+ADD\_PAY<ZUISHAO) {

sum+=OIER[OIER[root].llink].size+1;

OIER[root].size-=sum;

OIER[root].llink=0;

t=Delete(OIER[root].rlink);

sum+=t;

OIER[root].size-=t;

OIER[OIER[root].rlink].size=OIER[root].size;

root=OIER[root].rlink;

}

else {

t=Delete(OIER[root].llink);

sum=t;

OIER[root].size-=t;

}

return sum;

}

int Select(int R, int x)*//返回第 x 大的元素*

{

if(OIER[R].rlink==0)OIER[OIER[R].rlink].size=0;

int r = OIER[OIER[R].rlink].size+1;

if (x<r) return Select(OIER[R].rlink, x);

else

if (x>r) return Select(OIER[R].llink, x-r);

if(x==r) return OIER[R].key;

}

int main(void)

{

unsigned int N;

char command;

int pay,root=0;

int i;

fin>>N>>ZUISHAO;

for(i=1;i<=N;i++)

{

fin>>command>>pay;

if(command=='I'){

if(pay>=ZUISHAO)Insert(root,pay-ADD\_PAY);

}

if(command=='F'){

if(pay>OIER[root].size) fout<<-1<<endl;

else fout<<Select(root, pay)+ADD\_PAY<<endl;

}

if(command=='A')ADD\_PAY+=pay;

if(command=='S'){

ADD\_PAY-=pay;

C\_NUM+=Delete(root);

}

}

fout<<C\_NUM<<endl;

return 0;

}