**[SBT!](http://www.cnblogs.com/zgmf_x20a/archive/2008/11/14/1333205.html)**

在今年的信息学冬令营上，陈启峰提出了一个自己创造的BST数据结构—Size Balanced Tree。这个平衡二叉树被全世界内的许多网站所讨论，大家讨论的主题也只有一个—SBT能够取代Treap吗？本文详细介绍SBT树的性质，以及一些常用的操作，最后证明SBT是一颗高度平衡的二分查找树。

**一．        介绍**

众所周知，BST能够快速的实现查找等动态操作。但是在某些情况下，比如将一个有序的序列依次插入到BST中，则BST会退化成为一条链，效率非常之低。由此引申出来很多平衡BST，比如AVL树，红黑树，treap树等。这些数据结构都是通过引入其他一些性质来保证BST的高度在最坏的情况下都保持在O(log n)。其中,AVL树和红黑树的很多操作都非常麻烦，因此实际应用不是很多。而treap树加入了一些随机化堆的性质，实际应用效果非常好，实现起来很简单，一直以来受到很多人的青睐。本文介绍一种新的平衡BST树，实现起来也是非常之简单，并且能够支持更多的操作，实际评测效率跟treap也不差上下。

在介绍SBT之前，先介绍一下BST以及在BST上的旋转操作。

**1.      Binary Search Tree**

BST是一种高级的数据结构，它支持很多动态操作，包括查找，求最小值，最大值，前驱，后继，插入和删除，能够用于字典以及优先队列。

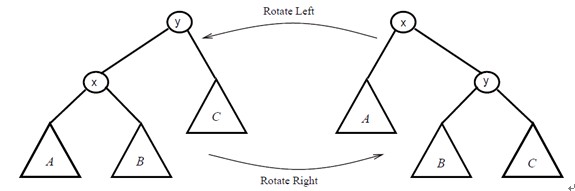
       BST是一棵二叉树，每个结点最多有2个儿子。每个结点都有个键值，并且键值必须满足下面的条件：

       如果x是BST中的一个结点。那么x的键值不小于其左儿子的键值，并且不大于其右儿子的键值。

       对于每个结点t，用left[t]和right[t]分别来存放它的两个儿子，ket[t]存放该结点的键值。另外，在SBT中，要增加s[t],用来保存以t为根的子树中结点的个数。

**2.      旋转**

为了保证BST的平衡(不会退化成为一条链)，通常通过旋转操作来改变BST的结构。旋转操作不会影响binary-search-tree的性质！



**2.1右旋操作的伪代码**

       右旋操作必须保证左儿子存在

       Right-Rotate(t)

              k←left[t]

              left[t]←right[k]

              right[k]←t

              s[k]←s[t]

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

              t←k

**2.2 左旋操作的伪代码**

       左旋操作必须保证右儿子存在

       Left-Rotate(t)

              k←right[t]

              right[t]←left[k]

              left[k]←t

              s[k]←s[t]

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

              t←k

**二．Size Balanced Tree**

Size Balanced Tree(简称SBT)是一种平衡二叉搜索树，它通过子树的大小s[t]来维持平衡性质。它支持很多动态操作，并且都能够在O(log n)的时间内完成。

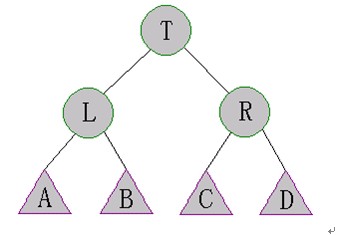
|  |  |
| --- | --- |
| Insert(t,v) | 将键值为v的结点插入到根为t的树中 |
| Delete(t,v) | 在根为t的树中删除键值为v的结点 |
| Find(t,v) | 在根为t的树中查找键值为v的结点 |
| Rank(t,v) | 返回根为t的树中键值v的排名。也就是树中键值比v小的结点数+1 |
| Select(t,k) | 返回根为t的树中排名为k的结点。同时该操作能够实现Get-min,Get-max,因为Get-min等于Select(t,1),Get-max等于Select(t,s[t]) |
| Pred(t,v) | 返回根为t的树中比v小的最大的键值 |
| Succ(t,v) | 返回根为t的树中比v大的最小的键值 |

SBT树中的每个结点都有left，right，key以及前面提到的size域。SBT能够保持平衡性质是因为其必须满足下面两个条件：

对于SBT中的每个结点t，有性质(a)(b)：

(a). s[right[t]]≥s[left[left[t]]],s[right[left[t]]]

(b). s[left[t]]≥s[right[right[t]]],s[left[right[t]]]



即在上图中，有s[A],s[B]≤s[R]&s[C],s[D] ≤s[L]

**三.              Maintain**

假设我们要在BST中插入一个键值为v的结点，一般是用下面这个过程：

Simple-Insert(t,v)

        If t=0 then

            t←NEW-NODE(v)

              Else

                     s[t]←s[t]+1

                     If v<key[t] then

                            Simple-Insert(left[t],v)

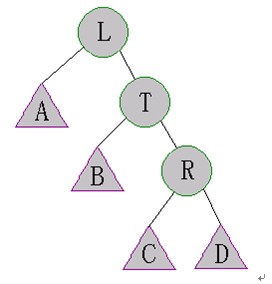
                     Else

                            Simple-Insert(right[t],v)

执行完操作Simple-Insert后，SBT的性质(a)和(b)就有可能不满足了，这是我们就需要修复(Maintain)SBT。

Maintain(t)用来修复根为t的SBT,使其满足SBT性质。由于性质(a)和(b)是对称的，下面仅讨论对性质(a)的修复。

**Case 1**：s[left[left[t]]]>s[right[t]]



这种情况下可以执行下面的操作来修复SBT

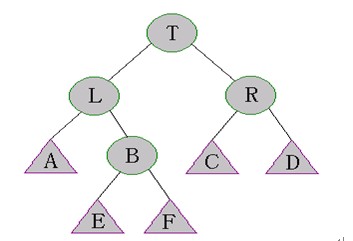
执行Right-Rotate(T)

有可能旋转后的树仍然不是SBT,需要再次执行Maintain(T)

由于L的右儿子发生了变化，因此需要执行Maintain(L)

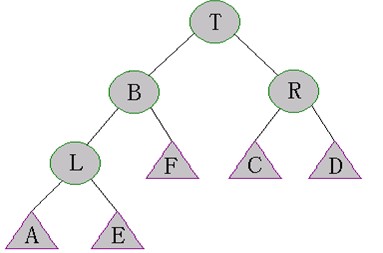
**Case 2**：s[right[left[t]]]>s[right[t]]

这种情况如下图所示：

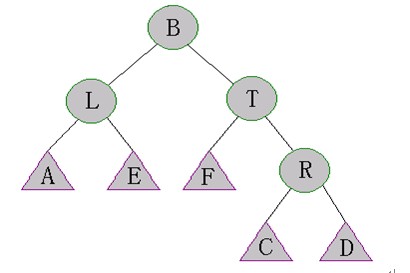


需要执行一下步骤来修复SBT：

执行Left-Rotate(L)。如下图所示



执行Right-Rotate(T)。如下图所示



当执行完(1)(2)后，树的结构变得不可预测了。但是幸运的是，在上图中，A,E,F,R子树仍然是SBT。因此我们可以执行Maintain(L)和Maintain(T)来修复B的子树。

**Case 3**：

这种情况和case 1是对称的

**Case 4**：

这种情况和case 2是对称的

Maintain操作的伪代码：

在Maintain过程中，用一个变量flag来避免额外的检查。当flag为false时，代表case 1和case 2需要被检查,否则case 3和case 4需要被检查。

Maintain (t,flag)

**If** flag=false **then**

**If** s[left[left[t]]>s[right[t]] **then**

                    Right-Rotate(t)

**Elseif** s[right[left[t]]>s[right[t]] **then**

                    Left-Rotate(left[t])

                    Right-Rotate(t)

**Else** exit

**Elseif** s[right[right[t]]>s[left[t]] **then**

             Left-Rotate(t)

**Elseif** s[left[right[t]]>s[left[t]] **then**

             Right-Rotate(right[t])

             Left-Rotate(t)

**Else** exit

      Maintain(left[t],false)

Maintain(right[t],true)

Maintain(t,false)

      Maintain(t,true)

**四．常用操作**

**插入操作**

SBT和插入操作和BST的基本相同，只是在插入之后需要执行下Maintain操作。

Insert (t,v)

**If** t=0 **then**

t←NEW-NODE(v)

**Else**

s[t] ←s[t]+1

**If** v<key[t] **then**

Simple-Insert(left[t],v)

**Else**

Simple-Insert(right[t],v)

Maintain(t,v≥key[t])

**删除操作**

如果没有找到要删除的结点，那么就删除最后一个访问的结点并记录。

Delete (t,v)

**If** s[t]≤2 **then**

record←key[t]

t←left[t]+right[t]

Exit

s[t] ←s[t]－1

**If** v=key[t] **then**

Delete(left[t],v[t]+1)

Key[t] ←record

Maintain(t,true)

**Else**

**If** v<key[t] **then**

Delete(left[t],v)

**Else**

Delete(right[t],v)

Maintain(t,v<key[t])

另外，由于SBT的平衡性质是靠size域来维护的，而size域本身(子树所含节点个数)对于很多查询算法都特别有用，这样使得查询集合里面的譬如第n小的元素，以及一个元素在集合中的排名等操作都异常简单，并且时间复杂度都稳定在O(log n)。下面仅介绍下上表提到的select(t,k)操作和rank(t,v)操作。

       由于SBT的性质（结点t的关键字比其左子树中所有结点的关键字都大，比其左子树中所有的关键字都小），理解下面的算法非常容易。

**3．Select操作**

Select(t,k)

**If** k=s[left[t]]+1 **then**

              return key[t]

**If** k<=s[left[t]] **then**

              return Select(left[t],k)

**Else**

              return Select(right[t],k-1-s[left[t]])

**4．Rank操作**

Rank(t,v)

**If** t=0 **then**

              return 1

**If** v<=key[t] **then**

              return rank(left[t],v)

**Else**

              return s[left[t]]+1+rank(right[t],v)

同样,求前驱结点的操作Pred和后继结点的操作都很容易通过size域来实现。

**五．相关证明分析**

显然Maintain操作是一个递归过程，可能你会怀疑它是否会结束。下面我们可以证明Maintain操作的平摊时间复杂度为O(1)。

**1．关于树的高度的分析**

设f[h]表示高度为h的SBT中结点数目的最小值，则有

                             1                                    (h=0)

f[h]=       2                                    (h=1)

           f[h-1]+f[h-2]+1                  (h>1)

a．证明：

（1）       很明显f[0]=1,f[1]=2。

（2）       首先，对于任意h>1,我们假设t是一颗高度为h的SBT的根结点，则这颗SBT包含一颗高度为h-1的子树。不妨假设t的左子树的高度为h-1，根据f[h]的定义，有

s[left[t] ]≥f[h-1]，同样的，左子树中有一颗高度为h-2的子树，换句话说，左子树中含有一颗结点数至少为f[h-2]的子树。由SBT的性质(b)，可知s[right[t]] ≥f[h-2]。因此我们有s[t]=s[left[t]]+s[right[t]]+1≥f[h-1]+f[h-2]+1。

另外一方面，我们可以构造一颗高度为h，并且结点数正好为f[h]的SBT,称这样的SBT为tree[t]。可以这样来构造tree[h]：

                             含有一个结点的SBT                                    (h=0)

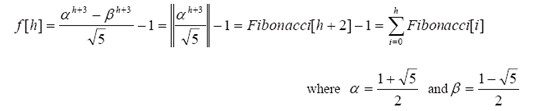
tree[h]=     含有2个结点的任意SBT                             (h=1)

         左子树为tree[h-1],右子树为tree[h-2]的SBT    (h>1)

由f[h]的定义可知f[h] ≤f[h-1]+f[h-2]+1(h>1)。因此f[h]的上下界都为f[h-1]+f[h-2]+1,因此有f[h]=f[h-1]+f[h-2]+1。

b．最坏情况下的高度

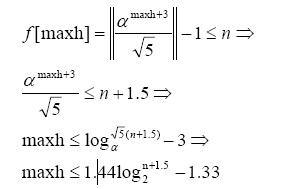
事实上f[h]是一个指数函数，通过f[h]的递推可以计算出通项公式。



定理：

含有n个结点的SBT在最坏情况下的高度是满足f[h] ≤n的最大的h值。

假设maxh为含有n个结点的SBT的最坏情况下的高度。由上面的定理，有



于是很明显SBT的高度为O(logn)，是一颗高度平衡的BST！

**2．对Maintain操作的分析**

通过前面的计算分析我们能够很容易分析出Maintain操作是非常高效的。

首先，有一个非常重要的值来评价一颗BST的好坏：所有结点的平均深度。它是通过所有结点的深度之和SD除以结点个数n计算出来的。一般来说，这个值越小，这颗BST就越好。由于对于一颗BST来说，结点数n是一个常数，因此我们期望SD值越小越好。

现在我们集中来看SBT的SD值，它的重要性在于能够制约Maintain操作的执行时间。回顾先前提到的BST中的旋转操作，有个重要的性质就是：每次执行旋转操作后，SD值总是递减的！

由于SBT树的高度总是O(log n)，因此SD值也总是保持在O（log n）。并且SD仅在插入一个结点到SBT后才增加，因此(T是Maintain操作中执行旋转的次数)

http://images.cnblogs.com/cnblogs_com/woodfish1988/c9.jpg

Maintain操作的次数等于T加上不需要旋转操作的Maintain操作的次数。由于后者为O(nlogn)+O(T),因此Maintain的平摊分析时间复杂度为：

http://images.cnblogs.com/cnblogs_com/woodfish1988/c10.jpg

**对各个操作时间复杂度的分析**

现在我们知道了SBT的高度为O(log n)，并且 Maintain操作的平摊分析时间复杂度为O(1)，因此对于所有的常用操作，时间复杂度都稳定在O(log n)！

附自己整理的SBT模板:

struct SBTNode{  
    SBTNode \*lc,\*rc;  
    int key,sz;  
    SBTNode(int \_key){  
        key=\_key;  
        sz=1;  
        lc=NULL;  
        rc=NULL;  
    }  
};  
  
///////////////////////////////  
  
SBTNode\* SBTSearch(SBTNode\* T,int key);  
  
void SBTInsert(SBTNode\* &T,SBTNode\* x);//修改了树的结构，故参数需引用  
  
SBTNode\* SBTDel(SBTNode\* &T,int key);//  
  
SBTNode\* SBTPred(SBTNode\* T,int key);  
  
SBTNode\* SBTSucc(SBTNode\* T,int key);  
  
SBTNode\* SBTSelect(SBTNode\* T,int k);  
  
int SBTRank(SBTNode\* T,int key);  
  
//////////////////////////////  
  
  
void RightRotate(SBTNode\* &x){  
    SBTNode\* y=x->lc;  
    x->lc=y->rc;  
    y->rc=x;  
    y->sz=x->sz;  
    x->sz=x->lc->sz+x->rc->sz+1;  
    x=y;  
}  
  
void LeftRotate(SBTNode\* &x){  
    SBTNode\* y=x->rc;  
    x->rc=y->lc;  
    y->lc=x;  
    y->sz=x->sz;  
    x->sz=x->lc->sz+x->rc->sz+1;  
    x=y;  
}  
  
SBTNode\* SBTSearch(SBTNode\* T,int key){  
    if(!T || key==T->key)  
        return T;  
    if(key<T->key)  
        return SBTSearch(T->lc,key);  
    else  
        return SBTSearch(T->rc,key);  
}  
  
void Maintain(SBTNode\* &T,bool RightDeeper){  
    if(!RightDeeper){  
        if(T->lc==NULL)  
            return;  
        if(T->lc->lc->sz>T->rc->sz)  
            RightRotate(T);  
        else if(T->lc->rc->sz>T->rc->sz){  
            LeftRotate(T->lc);  
            RightRotate(T);  
        }  
        else  
            return;  
    }  
    else{  
        if(T->rc==NULL)  
            return;  
        if(T->rc->rc->sz>T->lc->sz)  
            LeftRotate(T);  
        else if(T->rc->lc->sz>T->lc->sz){  
            RightRotate(T->rc);  
            LeftRotate(T);  
        }  
        else  
            return;  
    }  
    Maintain(T->lc,false);  
    Maintain(T->rc,true);  
    Maintain(T,false);  
    Maintain(T,true);  
}  
  
void SBTInsert(SBTNode\* &T,SBTNode\* x){  
    if(T==NULL){  
        T=x;  
        return;  
    }  
    T->sz++;  
    if(x->key<T->key)  
        SBTInsert(T->lc,x);  
    else  
        SBTInsert(T->rc,x);  
    Maintain(T,x->key>=T->key);  
}  
  
SBTNode\* SBTDel(SBTNode\* &T,int key){  
    SBTNode\* record;  
    if(T->sz<=2){  
        record=T;  
        //T=T->lc+T->rc;  
        T=T->lc?T->lc:T->rc;  
        return record;  
    }  
    T->sz--;  
    if(key==T->key){  
        record=SBTDel(T->lc,key+1);  
        T->key=record->key;  
        T->sz=record->sz;  
        Maintain(T,true);  
    }  
    else if(key<T->key)  
        record=SBTDel(T->lc,key);  
    else  
        record=SBTDel(T->rc,key);  
    Maintain(T,key<T->key);  
    return record;  
}  
  
SBTNode\* SBTPred(SBTNode\* T,int key){  
    if(!T)  
        return NULL;  
    if(key<=T->key)  
        return SBTPred(T->rc,key);  
    else{  
        SBTNode\* pred=SBTPred(T->rc,key);  
        return (pred?pred:T);  
    }  
}  
  
SBTNode\* SBTSucc(SBTNode\* T,int key){  
    if(!T)  
        return NULL;  
    if(key>=T->key)  
        return SBTSucc(T->rc,key);  
    else{  
        SBTNode\* succ=SBTSucc(T->lc,key);  
        return (succ?succ:T);  
    }  
}  
   
SBTNode\* SBTSelect(SBTNode\* T,int k){  
    if(k==T->lc->sz+1)  
        return T;  
    if(k<=T->lc->sz)  
        return SBTSelect(T->lc,k);  
    else  
        return SBTSelect(T->rc,k-1-T->lc->sz);  
}  
  
  
int SBTRank(SBTNode\* T,int key){  
    if(T==NULL)  
        return 1;  
    if(key<=T->key)  
        return SBTRank(T->lc,key);  
    else  
        return T->lc->sz+1+SBTRank(T->rc,key);  
}