Code:SBT/C

**来自"NOCOW"**

(跳转自[Code:SBT C](http://www.nocow.cn/index.php?title=Code:SBT_C&redirect=no))

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

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* SBT.cc*

*\**

*\* Mon Jun 14 16:57:48 2007*

*\* Copyright 2007 巨菜逆铭*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*//————BST的储存结构————*

typedef struct SBTNode{

SBTNode \*left,\*right;

long key; *//这里省略了卫星数据域*

unsigned long size;

SBTNode(long \_key){ *//构造函数：未考虑卫星数据*

left=right=**NULL**;

size=1;

key=\_key;

}

}SBTNode, \*SBTree;

*//————SBT基本操作函数原型说明————*

SBTNode \*SBT\_Search(SBTree T,long key);

*//在T中中寻找关键字为key的结点*

*//若能找到则返回指向它的指针，否则返回NULL*

void SBT\_Insert(SBTree &T, SBTNode \*x);

*//将节点x插入树中*

*//初始要求x的left和right域为NULL，size域为1*

SBTNode \*SBT\_Delete(SBTree &T, long key);

*//从以T为根的SBT中删除一个关键字为v的结点并返回其指针*

*//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并返回其指针*

SBTNode \*SBT\_Pred(SBTree T, long key);

*//返回指向关键字为key的节点在T的中序遍历中的直接前趋的指针*

*//要求T中必须有关键字为key的节点*

SBTNode \*SBT\_Succ(SBTree T,long key);

*//返回指向关键字为key的节点在T的中序遍历中的直接后继的指针*

*//要求T中必须有关键字为key的节点*

SBTNode \*SBT\_Select(SBTree T, unsigned long i);

*//从树T中找到关键字第i小的结点并返回其指针*

unsigned long SBT\_Rank(SBTree T, long key);

*//返回关键字为key的节点在树T中的秩*

*//若不存在此节点则返回0*

*//————SBT的修复操作————*

**inline** void SBT\_LeftRotate(SBTree &x){

*//左旋*

SBTNode \*y=x->right;

assert(y!=**NULL**);

x->right=y->left;

y->left=x;

y->size=x->size;

x->size=(!x->left?0:x->left->size)

+(!x->right?0:x->right->size)+1;

x=y;

}

**inline** void SBT\_RightRotate(SBTree &x){

*//右旋*

SBTNode \*y = x->left;

assert(y!=**NULL**);

x->left=y->right;

y->right=x;

y->size=x->size;

x->size=(!x->left?0:x->left->size)

+(!x->right?0:x->right->size)+1;

x=y;

}

void SBT\_Maintain(SBTree &T,bool flag){

*//维护操作的核心：保持*

if(!T) return; *//空树无需Maintain*

if(!flag){

if( T->left&&T->left->left

&&(!T->right||T->left->left->size > T->right->size) ) *//情况1*

SBT\_RightRotate(T);

else if( T->left&&T->left->right

&&(!T->right||T->left->right->size > T->right->size) ){ *//情况2*

SBT\_LeftRotate(T->left);

SBT\_RightRotate(T);

}

else return; *//无需修复*

}

else{

if( T->right&&T->right->right

&&(!T->left||T->right->right->size > T->left->size) ) *//情况1'*

SBT\_LeftRotate(T);

else if( T->right && T->right->left

&&(!T->left||T->right->left->size > T->left->size) ){ *//情况2'*

SBT\_RightRotate(T->right);

SBT\_LeftRotate(T);

}

else return;*//无需修复*

}

SBT\_Maintain(T->left,0); *//修复左子树*

SBT\_Maintain(T->right,1); *//修复右子树*

SBT\_Maintain(T,0); *//修复整棵树*

SBT\_Maintain(T,1);

}

*//————SBT基本操作的算法描述————*

SBTNode \*SBT\_Search(SBTree T,long key){

*//在T中中寻找关键字为key的结点*

*//若能找到则返回指向它的指针，否则返回NULL*

return !T||T->key==key?T:SBT\_Search(key<T->key?T->left:T->right,key);

}

void SBT\_Insert(SBTree &T, SBTNode \*x){

*//将节点x插入树中*

if(!T) T=x;

else{

T->size++;

SBT\_Insert(x->key<=T->key?T->left:T->right,x);

SBT\_Maintain(T,x->key>T->key);

}

}

SBTNode \*SBT\_Delete(SBTree &T, long key){

*//从以T为根的SBT中删除一个关键字为key的结点并返回“实际”被删除结点的指针*

*//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并返回其指针*

if(!T) return **NULL**;

T->size--;

if(key==T->key||key<T->key&&!T->left||key>T->key&&!T->right)

{

SBTNode \*del;

if(!T->left||!T->right){

del=T;

T=(T->left?T->left:T->right);

}

else{

del=SBT\_Delete(T->right,key-1);

T->key=del->key; *//若有卫星数据也需复制*

}

return del;

}

else return SBT\_Delete(key<T->key?T->left:T->right,key);

}

SBTNode \*SBT\_Pred(SBTree T, long key){

*//返回指向拥有比key小的最大关键字的节点的指针*

if(!T) return **NULL**;

if(key<=T->key) return SBT\_Pred(T->left,key);

else{

SBTNode \*pred=SBT\_Pred(T->right,key);

return (!pred?T:pred);

}

}

SBTNode \*SBT\_Succ(SBTree T,long key){

*//返回指向拥有比key大的最小关键字的节点的指针*

if(!T) return **NULL**;

if(key>=T->key) return SBT\_Succ(T->right,key);

else{

SBTNode \*succ= SBT\_Succ(T->left,key);

return(!succ?T:succ);

}

}

SBTNode \*SBT\_Select(SBTree T, unsigned long i){

*//从树T中找到关键字第i小的结点并返回其指针*

if(!T||i>T->size) return **NULL**;

unsigned long r = (!T->left?0:T->left->size)+1;

if(i==r) return T;

else if(i<r) return SBT\_Select(T->left,i);

else return SBT\_Select(T->right,i-r);

}

unsigned long SBT\_Rank(SBTree T, long key){

*//返回关键字为key的节点在树T中的秩*

*//若不存在此节点则返回0*

if(!T) return 0;

if(T->key==key) return (!T->left?0:T->left->size)+1;

else if(key<T->key) return SBT\_Rank(T->left,key);

else{

unsigned long r=SBT\_Rank(T->right,key);

return r==0?0:r+(!T->left?0:T->left->size)+1;

}

}

下面是一个根据上面的版本修改得来的更简练的版本，代码实际长度约90行：

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* SBT.cc*

*\**

*\* Mon Jun 18 12:17:48 2007*

*\* Copyright 2007 巨菜逆铭*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*//————SBT的储存结构————*

struct SBTNode{

SBTNode \*ch[2],\*p; *//ch[0]、ch[1]分别为左右孩子，p为双亲*

long key; *//这里省略了卫星数据域*

unsigned long size;

SBTNode(long \_key,unsigned long \_size);

}NIL=SBTNode(0,0);

typedef SBTNode \*SBTree;

SBTNode::SBTNode(long \_key,unsigned long \_size=1){ *//构造函数：未考虑卫星数据*

ch[0]=ch[1]=p=&NIL;

size=\_size;

key=\_key;

}

*//————SBT基本操作函数原型说明————*

SBTNode \*SBT\_Search(SBTree T,long key);

*//在T中中寻找关键字为key的结点*

*//若能找到则返回指向它的指针，否则返回NULL*

void SBT\_Insert(SBTree &T, SBTNode\* x);

*//将节点x插入树中*

SBTNode \*SBT\_Delete(SBTree &T, long key);

*//从以T为根的SBT中删除一个关键字为key的结点并返回“实际”被删除结点的指针*

*//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并返回其指针*

SBTNode \*SBT\_Pred(SBTree T, long key);

*//返回指向关键字为key的节点在T的中序遍历中的直接前趋的指针*

*//要求T中必须有关键字为key的节点*

SBTNode \*SBT\_Succ(SBTree T,long key);

*//返回指向关键字为key的节点在T的中序遍历中的直接后继的指针*

*//要求T中必须有关键字为key的节点*

SBTNode \*SBT\_Select(SBTree T, unsigned long i);

*//从树T中找到关键字第i小的结点并返回其指针*

unsigned long SBT\_Rank(SBTree T, long key);

*//返回关键字为key的节点在树T中的秩*

*//若不存在此节点则返回0*

*//————SBT的修复操作————*

**inline** void SBT\_Rotate(SBTree &x, bool flag){

*//旋转：flag==0为左旋，否则为右旋*

SBTNode \*y=x->ch[!flag];

x->ch[!flag]=y->ch[flag];

if(y->ch[flag]!=&NIL) y->ch[flag]->p=x; *//维护p*

y->ch[flag]=x;

y->p=x->p; *//维护p*

x->p=y; *//维护p*

y->size=x->size;

x->size=x->ch[0]->size+x->ch[1]->size+1;

x=y;

}

void SBT\_Maintain(SBTree &T,bool flag){

*//维护操作的核心：保持*

if(T->ch[flag]->ch[flag]->size>T->ch[!flag]->size) *//情况1*

SBT\_Rotate(T,!flag);

else if(T->ch[flag]->ch[!flag]->size>T->ch[!flag]->size){ *//情况2*

SBT\_Rotate(T->ch[flag],flag);

SBT\_Rotate(T,!flag);

}

else return; *//无需修复*

SBT\_Maintain(T->ch[0],0); *//修复左子树*

SBT\_Maintain(T->ch[1],1); *//修复右子树*

SBT\_Maintain(T,0); *//修复整棵树*

SBT\_Maintain(T,1);

}

*//————SBT基本操作的算法描述————*

SBTNode \*SBT\_Search(SBTree T,long key){

*//在T中中寻找关键字为key的结点*

*//若能找到则返回指向它的指针，否则返回NULL*

return T==&NIL||T->key==key?T:SBT\_Search(T->ch[key>T->key],key);

}

void SBT\_Insert(SBTree &T, SBTNode\* x){

*//将节点x插入树中*

if(T==&NIL) T=x;

else{

T->size++;

x->p=T; *//维护p*

SBT\_Insert(T->ch[x->key>T->key],x);

SBT\_Maintain(T,x->key>T->key);

}

}

SBTNode \*SBT\_Delete(SBTree &T, long key){

*//从以T为根的SBT中删除一个关键字为key的结点并返回“实际”被删除结点的指针*

*//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并返回其指针*

if(T==&NIL) return &NIL;

T->size--;

if(key==T->key||T->ch[T->key<key]==&NIL){

SBTNode \*toDel;

if(T->ch[0]==&NIL||T->ch[1]==&NIL){

toDel=T;

SBTNode \*ch = T->ch[T->ch[1]!=&NIL];

if(ch!=&NIL) ch->p=T->p; *//维护p*

T=ch;

}

else{

toDel=SBT\_Delete(T->ch[1],key-1);

T->key=toDel->key; *//若有卫星数据也需复制*

}

return toDel;

}

else return SBT\_Delete(T->ch[key>T->key],key);

}

SBTNode \*SBT\_Pred(SBTree T, long key){

*//返回指向拥有比key小的最大关键字的节点的指针*

if(T==&NIL) return &NIL;

if(key<=T->key) return SBT\_Pred(T->ch[0],key);

else{

SBTNode \*pred=SBT\_Pred(T->ch[1],key);

return (pred!=&NIL?pred:T);

}

}

SBTNode \*SBT\_Succ(SBTree T,long key){

*//返回指向拥有比key大的最小关键字的节点的指针*

if(T==&NIL) return &NIL;

if(key>=T->key) return SBT\_Succ(T->ch[1],key);

else{

SBTNode \*succ= SBT\_Succ(T->ch[0],key);

return(succ!=&NIL?succ:T);

}

}

SBTNode \*SBT\_Select(SBTree T, unsigned long i){

*//从树T中找到关键字第i小的结点并返回其指针*

if(i>T->size) return &NIL;

unsigned long r = T->ch[0]->size+1;

if(i==r) return T;

else return SBT\_Select(T->ch[i>r],i>r?i-r:i);

}

unsigned long SBT\_Rank(SBTree T, long key){

*//返回关键字为key的节点在树T中的秩*

*//若不存在此节点则返回0*

if(T==&NIL) return 0;

if(T->key==key) return T->ch[0]->size+1;

else if(key<T->key) return SBT\_Rank(T->ch[0],key);

else{

unsigned long r=SBT\_Rank(T->ch[1],key);

return r==0?0:r+T->ch[0]->size+1;

}

}

下面是一个根据上面的版本修改得来的版本，代码实际长度未统计：

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* SBT.cpp*

*\**

*\* 2008-11-17 17:06:02 Fixed a bug in function SBT\_Delete, it's now be test.*

*\* 2008-11-15 16:34:19 Give a new versin SBT\_Delete that we don't need to using key-1.*

*\* Any problem contact yonggangluo@hotmail.com*

*\* Copyright 2008 Yonggang Luo base on 巨菜逆铭*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <string.h>

#include <iostream>

typedef int32\_t key\_t;

struct SBTNode {

SBTNode \*p,\*ch[2]; *///ch[0]、ch[1]分别为左右孩子，p为双亲*

size\_t size;

key\_t key;*///这里省略了卫星数据域*

SBTNode(key\_t \_key,size\_t \_size);*///构造函数：未考虑卫星数据*

}NULL\_NODE(0,0);

typedef SBTNode \*SBTree;

SBTree NULL\_TREE = &NULL\_NODE;

SBTNode::SBTNode(key\_t \_key, size\_t \_size=1) {

p=ch[0]=ch[1]= NULL\_TREE;

key=\_key, size=\_size;

}

bool cmp(key\_t x, key\_t y);

*//————SBT的旋转操作————*

**inline** void SBT\_Rotate(SBTree &x, bool flag) {

*//flag: 0为将左子节点移到根, 1为将右子节点移到根*

SBTree sub=x->ch[flag], &leaf = sub->ch[!flag];

x->ch[flag] = leaf;

if (leaf!=NULL\_TREE) leaf->p = x; */\*设置双亲p\*/*

sub->p = x->p; */\*设置双亲p\*/*

x->p = sub; */\*设置双亲p\*/*

leaf = x;

sub->size = x->size;

x->size = x->ch[0]->size + x->ch[1]->size +1;

x = sub;

}

*//————SBT的维护操作————*

void SBT\_Maintain(SBTree &T,bool flag) {

if (T->ch[flag]->ch[!flag]->size > T->ch[!flag]->size) SBT\_Rotate(T->ch[flag], !flag);

else if (T->ch[flag]->ch[flag]->size <= T->ch[!flag]->size) return;

SBT\_Rotate(T, flag);

SBT\_Maintain(T->ch[0],0); *//修复左子树*

SBT\_Maintain(T->ch[1],1); *//修复右子树*

SBT\_Maintain(T,0); *//修复整棵树*

SBT\_Maintain(T,1);

}

*//————SBT的插入操作————*

void SBT\_Insert(SBTree &T, SBTNode\* x) {

*//将节点x插入树中*

if (T!=NULL\_TREE) {

++T->size;

x->p =T; */\*设置双亲p\*/*

SBT\_Insert(T->ch[cmp(T->key, x->key)], x);

SBT\_Maintain(T, cmp(T->key, x->key)); */\* here must be verify\*/*

} else T=x;

}

*//————SBT的查找————*

SBTree &SBT\_Search(SBTree \*T, key\_t key, int resize=0) {

*//在T中中寻找关键字为key的结点*

*//若能找到则返回指向它的指针，否则返回NULL\_TREE*

while (\*T!=NULL\_TREE && (\*T)->key!=key) {

(\*T)->size += resize;

T=&((\*T)->ch[cmp((\*T)->key, key)]);

}

return \*T;

}

*//————SBT的极值————*

SBTree &SBT\_Extremum(SBTree \*T, bool maximum = **true**, int resize=0) { */\*返回一棵树的极值 \*/*

while ((\*T)->ch[maximum]!=NULL\_TREE) {

(\*T)->size += resize;

T=&(\*T)->ch[maximum];

}

return \*T;

}

*/\*删除一棵树的根节点，并返回被删节点\*/*

SBTNode \*SBT\_Delete(SBTree &T){

--T->size; *///维护size*

int flag = T->ch[0]==NULL\_TREE || T->ch[1]==NULL\_TREE;

SBTree \*P = flag ? &T:&SBT\_Extremum(&T->ch[1] , 0, -1); *///如果T有两棵子树，那么我们就先删T的后继*

SBTree D=\*P;*///D用来保存 真正 被删的节点*

\*P=D->ch[D->ch[0]==NULL\_TREE]; *///把被删节点设为此节点的其中一个儿子*

if (\*P!=NULL\_TREE) (\*P)->p = D->p; *//设置双亲p*

if (flag) return D; *///如果 flag，则T是真正应当被删的节点，故直接返回D，否则继续处理D*

T->key = D->key;*///简单替换*

*/\*std::swap(T, D);///本质替换*

*memcpy(T, D, sizeof(\*T)-sizeof(T->key)); //把关系复制过去*

*if (T->ch[0]!=NULL\_TREE) T->ch[0]->p = T; //设置双亲p*

*if (T->ch[1]!=NULL\_TREE) T->ch[1]->p = T; //设置双亲p*

*\*/*

return D;

}

*//————SBT的删除————*

SBTNode \*SBT\_Delete(SBTree &T, key\_t key) {

*//如果树中没有一个这样的结点，则返回 NULL\_TREE*

if (SBT\_Search(&T, key) == NULL\_TREE) return NULL\_TREE; */\*表示key不在树中\*\*/*

return SBT\_Delete(SBT\_Search(&T, key, -1)); */\*RP 就是指向待删子树的指针\*/*

}

SBTree &SBT\_Near(SBTree &T, key\_t key, bool flag) {

*///flag: 0 前驱 ; 1 后继*

if (T==NULL\_TREE) return T;

if (key==T->key || cmp(T->key, key)==flag ) return SBT\_Near(T->ch[flag],key, flag);

SBTree &near=SBT\_Near(T->ch[!flag],key, flag);

return near!=NULL\_TREE ? near:T;

}

*//————SBT的选取第i个元素————*

SBTree SBT\_Select(SBTree T, size\_t i) {

*///从树T中找到关键字第i的结点并返回其指针,从1开始计数，如果没有就返回NULL\_TREE*

if (i > T->size) return NULL\_TREE;

size\_t r = T->ch[0]->size+1;

return i==r ? T:SBT\_Select(T->ch[i>r], i>r?i-r:i);

}

*//————SBT的给定节点是第几个元素————*

size\_t SBT\_Rank(SBTree T, key\_t key) {

*///此功能不完善，如果同一个key有两个不同的rank,那么返回值是其中的任意一个的rank。*

*///可以考虑用二分法加 SBT\_Select实现*

*///若不存在此节点则返回0*

if (T==NULL\_TREE) return 0;

if (key==T->key) return T->ch[0]->size+1;

if (cmp(key, T->key)) return SBT\_Rank(T->ch[0],key);

size\_t r=SBT\_Rank(T->ch[1],key);

return r==0?0:r+T->ch[0]->size+1;

}

**inline** void SBT\_Free(SBTNode \*t) {

if (t!=NULL\_TREE) delete t;

}

*/\*\*下面是一些测试代码\*\*/*

bool cmp(key\_t x, key\_t y)

{

return x<y;

}

#include <stdlib.h>

#include <time.h>

int deep;

size\_t count;

SBTree root=NULL\_TREE;

void SBT\_Print(SBTree T, int d) {

if (T==NULL\_TREE) return;

SBT\_Print(T->ch[0], d+1);

static size\_t r;

key\_t value;

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Rank = %d**\n**", r=SBT\_Rank(root, T->key));

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("value = %d**\n**", value = SBT\_Select(root, r)->key);

if (value!=T->key)

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error!");

exit(1);

}

for (int i=0; i<d; ++i) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)(" ");

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%d**\n**", T->key);

if ( (T->ch[0]->p!=NULL\_TREE && T->ch[0]->p!=T) || (T->ch[1]->p!=NULL\_TREE && T->ch[1]->p!=T) ) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error parent!**\n**");

exit(0);

}

SBT\_Print(T->ch[1], d+1);

}

void show() {

if (root->p!=NULL\_TREE) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error!**\n**");

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("The size of the tree is %d**\n**", root->size);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("The tree is**\n**");

count = 0;

SBT\_Print(root,0);

if (root->size != count) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("root->size = %u**\n** count = %u **\n** Error size!**\n**", root->size, count);

}

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("End of the tree!**\n\n**");

}

int main() {

srand(time(**NULL**));

size\_t n=3000000;

key\_t x;

freopen("output.txt","w",stdout);

*//#define READFILE*

#ifdef READFILE

char order;

freopen("input.txt","r",stdin);

scanf("%d**\n**",&n);

for (size\_t i=0; i<n; ++i) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("i=%u**\n**", i);

scanf("%c %d**\n**", &order, &x);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x=%d**\n**", x);

if (order=='I')

SBT\_Insert(root, new SBTNode(x));

else

SBT\_Free(SBT\_Delete(root, x));

show();

if (NULL\_TREE->key !=0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error NULL");

}

#else

for (size\_t i=0; i<n; ++i) {

if (i%3==0)

*//printf("D %ld\n", x), SBT\_Free(SBT\_Delete(&root, x));*

SBT\_Free(SBT\_Delete(root, x));

else

*//printf("I %ld\n",x = rand()),SBT\_Insert(root, new SBTNode(x));*

SBT\_Insert(root, new SBTNode(x=rand()));

*//show();*

if (NULL\_TREE->key !=0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error NULL");

}

*// show();*

SBTNode \*pre, \*suc;

FILE\* compare=fopen("compare.txt","w");

for (int i=0; i<32767; ++i)

{

pre = SBT\_Near(root, i, 0);

suc = SBT\_Near(root, i, 1);

*// fprintf(compare,"pre=%d i=%d suc=%d\n",pre->key, i, suc->key);*

}

#endif

return 0;

}