**平衡树（splay树）讲解**

变量声明：f[i]表示i的父结点，ch[i][0]表示i的左儿子，ch[i][1]表示i的右儿子，key[i]表示i的关键字（即结点i代表的那个数字），cnt[i]表示i结点的关键字出现的次数（相当于权值），size[i]表示包括i的这个子树的大小；sz为整棵树的大小，root为整棵树的根。

再介绍几个基本操作：

【clear操作】：将当前点的各项值都清0（用于删除之后）

**inline** **void** clear(**int** x){

     ch[x][0]=ch[x][1]=f[x]=cnt[x]=key[x]=size[x]=0;

}

【get操作】：判断当前点是它父结点的左儿子还是右儿子

**inline** **int** get(**int** x){

**return** ch[f[x]][1]==x;

}

【update操作】：更新当前点的size值（用于发生修改之后）

**inline** **void** update(**int** x){

**if** (x){

          size[x]=cnt[x];

**if** (ch[x][0]) size[x]+=size[ch[x][0]];

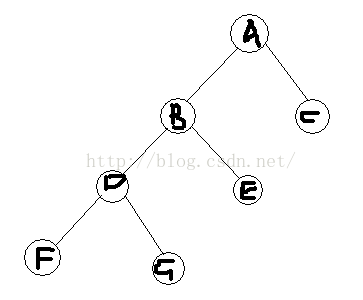
**if** (ch[x][1]) size[x]+=size[ch[x][1]];

     }

}

下面boss来了：

【rotate操作图文详解】



这是原来的树，假设我们现在要将D结点rotate到它的父亲的位置。

step 1：

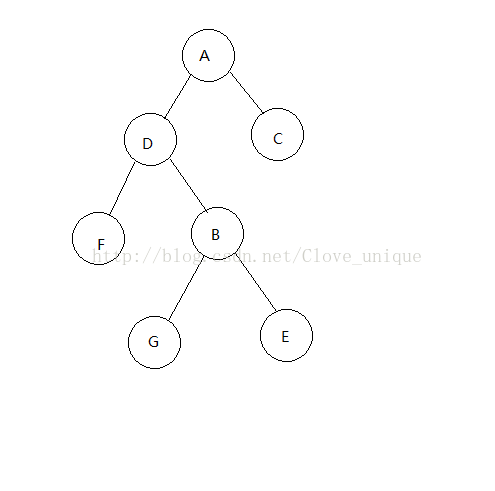
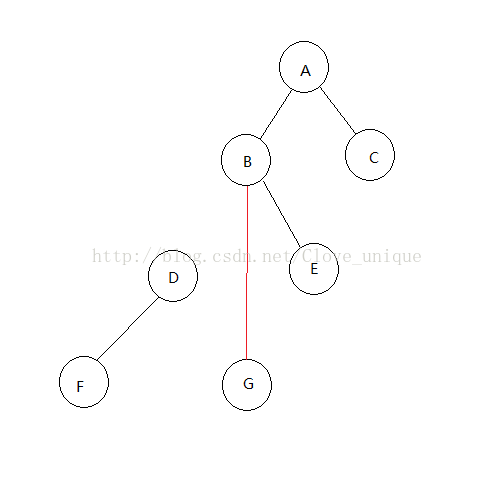
找出D的父亲结点（B）以及父亲的父亲（A）并记录。判断D是B的左结点还是右结点。

step 2：

我们知道要将Drotate到B的位置，二叉树的大小关系不变的话，B就要成为D的右结点了没错吧？

咦？可是D已经有右结点了，这样不就冲突了吗？怎么解决这个冲突呢？

我们知道，D原来是B的左结点，那么rotate过后B就一定没有左结点了对吧，那么正好，我们把G接到B的左结点去，并且这样大小关系依然是不变的，就完美的解决了这个冲突。



这样我们就完成了一次rotate，如果是右儿子的话同理。step 2的具体操作：

我们已经判断了D是B的左儿子还是右儿子，设这个关系为K；将D与K关系相反的儿子的父亲记为B与K关系相同的儿子（这里即为D的右儿子的父亲记为B的左儿子）；将D与K关系相反的儿子的父亲即为B（这里即为把G的父亲记为B）；将B的父亲即为D；将D与K关系相反的儿子记为B（这里即为把D的右儿子记为B）；将D的父亲记为A。

最后要判断，如果A存在（即rotate到的位置不是根的话），要把A的儿子即为D。

显而易见，rotate之后所有牵涉到变化的父子关系都要改变。以上的树需要改变四对父子关系，BG DG BD AB，需要三个操作（BG BD AB）。

step 3：update一下当前点和各个父结点的各个值

【代码】

**inline** **void** rotate(**int** x){

**int** old=f[x],oldf=f[old],which=get(x);

     ch[old][which]=ch[x][which^1];f[ch[old][which]]=old;

     f[old]=x;ch[x][which^1]=old;

     f[x]=oldf;

**if** (oldf)

          ch[oldf][ch[oldf][1]==old]=x;

     update(old);update(x);

}

【splay操作】

其实splay只是rotate的发展。伸展操作只是在不停的rotate，一直到达到目标状态。如果有一个确定的目标状态，也可以传两个参。此代码直接splay到根。

splay的过程中需要分类讨论，如果是三点一线的话（x，x的父亲，x的祖父）需要先rotate x的父亲，否则需要先rotate x本身（否则会形成单旋使平衡树失衡）

**inline** **void** splay(**int** x){

**for** (**int** fa;(fa=f[x]);rotate(x))

**if** (f[fa])

               rotate((get(x)==get(fa)?fa:x));

     root=x;

}

【insert操作】

其实插入操作是比较简单的，和普通的二叉查找树基本一样。

step 1：如果root=0，即树为空的话，做一些特殊的处理，直接返回即可。

step 2：按照二叉查找树的方法一直向下找，其中：

如果遇到一个结点的关键字等于当前要插入的点的话，我们就等于把这个结点加了一个权值。因为在二叉搜索树中是不可能出现两个相同的点的。并且要将当前点和它父亲结点的各项值更新一下。做一下splay。

如果已经到了最底下了，那么就可以直接插入。整个树的大小要+1，新结点的左儿子右儿子（虽然是空）父亲还有各项值要一一对应。并且最后要做一下他父亲的update（做他自己的没有必要）。做一下splay。

**inline** **void** insert(**int** v){

**if** (root==0) {sz++;ch[sz][0]=ch[sz][1]=f[sz]=0;key[sz]=v;cnt[sz]=1;size[sz]=1;root=sz;**return**;}

**int** now=root,fa=0;

**while** (1){

**if** (key[now]==v){

               cnt[now]++;update(now);update(fa);splay(now);**break**;

          }

          fa=now;

          now=ch[now][key[now]<v];

**if** (now==0){

               sz++;

               ch[sz][0]=ch[sz][1]=0;key[sz]=v;size[sz]=1;

               cnt[sz]=1;f[sz]=fa;ch[fa][key[fa]<v]=sz;

               update(fa);

               splay(sz);

**break**;

          }

     }

}

【find操作】查询x的排名

初始化：ans=0，当前点=root

和其它二叉搜索树的操作基本一样。但是区别是：

如果x比当前结点小，即应该向左子树寻找，ans不用改变（设想一下，走到整棵树的最左端最底端排名不就是1吗）。

如果x比当前结点大，即应该向右子树寻找，ans需要加上左子树的大小以及根的大小（这里的大小指的是权值）。

不要忘记了再splay一下

**inline** **int** find(**int** v){

**int** ans=0,now=root;

**while** (1){

**if** (v<key[now])

               now=ch[now][0];

**else**{

               ans+=(ch[now][0]?size[ch[now][0]]:0);

**if** (v==key[now]) {splay(now);**return** ans+1;}

               ans+=cnt[now];

               now=ch[now][1];

          }

     }

}

【findx操作】找到排名为x的点

初始化：当前点=root

和上面的思路基本相同：

如果当前点有左子树，并且x比左子树的大小小的话，即向左子树寻找；

否则，向右子树寻找：先判断是否有右子树，然后记录右子树的大小以及当前点的大小（都为权值），用于判断是否需要继续向右子树寻找。

**inline** **int** findx(**int** x){

**int** now=root;

**while** (1){

**if** (ch[now][0]&&x<=size[ch[now][0]])

               now=ch[now][0];

**else**{

**int** temp=(ch[now][0]?size[ch[now][0]]:0)+cnt[now];

**if** (x<=temp)

**return** key[now];

               x-=temp;now=ch[now][1];

          }

     }

}

【求x的前驱（后继），前驱（后继）定义为小于（大于）x，且最大（最小）的数】

这类问题可以转化为将x插入，求出树上的前驱（后继），再将x删除的问题。

其中insert操作上文已经提到。

【pre/next操作】

这个操作十分的简单，只需要理解一点：在我们做insert操作之后做了一遍splay。这就意味着我们把x已经splay到根了。求x的前驱其实就是求x的左子树的最右边的一个结点，后继是求x的右子树的左边一个结点（想一想为什么？）

**inline** **int** pre(){

、     **int** now=ch[root][0];

**while** (ch[now][1]) now=ch[now][1];

**return** now;

}

**inline** **int** next(){

**int** now=ch[root][1];

**while** (ch[now][0]) now=ch[now][0];

**return** now;

}

【del操作】

删除操作是最后一个稍微有点麻烦的操作。

step 1：随便find一下x。目的是：将x旋转到根。

step 2：那么现在x就是根了。如果cnt[root]>1，即不只有一个x的话，直接-1返回。

step 3：如果root并没有孩子，就说名树上只有一个x而已，直接clear返回。

step 4：如果root只有左儿子或者右儿子，那么直接clear root，然后把唯一的儿子当作根就可以了（f赋0，root赋为唯一的儿子）

剩下的就是它有两个儿子的情况。

step 5：我们找到新根，也就是x的前驱（x左子树最大的一个点），将它旋转到根。然后将原来x的右子树接到新根的右子树上（注意这个操作需要改变父子关系）。这实际上就把x删除了。不要忘了update新根。

**inline** **void** del(**int** x){

**int** whatever=find(x);

**if** (cnt[root]>1) {cnt[root]--;**return**;}

     //Only One Point

**if** (!ch[root][0]&&!ch[root][1]) {clear(root);root=0;**return**;}

     //Only One Child

**if** (!ch[root][0]){

**int** oldroot=root;root=ch[root][1];f[root]=0;clear(oldroot);**return**;

     }

**else** **if** (!ch[root][1]){

**int** oldroot=root;root=ch[root][0];f[root]=0;clear(oldroot);**return**;

     }

     //Two Children

**int** leftbig=pre(),oldroot=root;

     splay(leftbig);

     f[ch[oldroot][1]]=root;

     ch[root][1]=ch[oldroot][1];

     clear(oldroot);

     update(root);

**return**;

}

【总结】

平衡树的本质其实是二叉搜索树，所以很多操作是基于二叉搜索树的操作。

splay的本质是rotate，旋转其实只是为了保证二叉搜索树的平衡性。

所有的操作一定都满足二叉搜索树的性质，所有改变父子关系的操作一定要update。

关键是理解rotate，splay的原理以及每一个操作的原理。

【bzoj3224】普通平衡树

Description

您需要写一种数据结构（可参考题目标题），来维护一些数，其中需要提供以下操作：

1. 插入x数

2. 删除x数(若有多个相同的数，因只删除一个)

3. 查询x数的排名(若有多个相同的数，因输出最小的排名)

4. 查询排名为x的数

5. 求x的前驱(前驱定义为小于x，且最大的数)

6. 求x的后继(后继定义为大于x，且最小的数)

Input

第一行为n，表示操作的个数,下面n行每行有两个数opt和x，opt表示操作的序号(1<=opt<=6)

Output

对于操作3,4,5,6每行输出一个数，表示对应答案

Sample Input

10

1 106465

4 1

1 317721

1 460929

1 644985

1 84185

1 89851

6 81968

1 492737

5 493598

Sample Output

106465

84185

492737

HINT

1.n的数据范围：n<=100000

2.每个数的数据范围：[-1e7,1e7]

#include<iostream>

#include<cstring>

#include<cstdio>

using namespace std;

#define MAXN 1000000

int ch[MAXN][2],f[MAXN],size[MAXN],cnt[MAXN],key[MAXN];

int sz,root;

inline void clear(int x){

ch[x][0]=ch[x][1]=f[x]=size[x]=cnt[x]=key[x]=0;

}

inline bool get(int x){

return ch[f[x]][1]==x;

}

inline void update(int x){

if (x){

size[x]=cnt[x];

if (ch[x][0]) size[x]+=size[ch[x][0]];

if (ch[x][1]) size[x]+=size[ch[x][1]];

}

}

inline void rotate(int x){

int old=f[x],oldf=f[old],whichx=get(x);

ch[old][whichx]=ch[x][whichx^1]; f[ch[old][whichx]]=old;

ch[x][whichx^1]=old; f[old]=x;

f[x]=oldf;

if (oldf)

ch[oldf][ch[oldf][1]==old]=x;

update(old); update(x);

}

inline void splay(int x){

for (int fa;fa=f[x];rotate(x))

if (f[fa])

rotate((get(x)==get(fa))?fa:x);

root=x;

}

inline void insert(int x){

if (root==0){sz++; ch[sz][0]=ch[sz][1]=f[sz]=0; root=sz; size[sz]=cnt[sz]=1; key[sz]=x; return;}

int now=root,fa=0;

while(1){

if (x==key[now]){

cnt[now]++; update(now); update(fa); **splay(now)**; break;

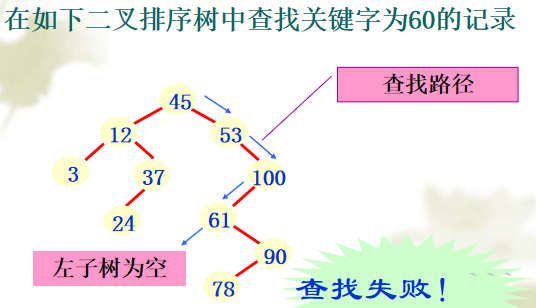
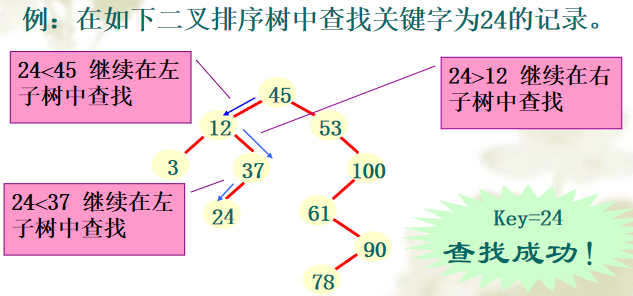
}

fa=now;

now=ch[now][key[now]<x];

if (now==0){

sz++;



ch[sz][0]=ch[sz][1]=0;

f[sz]=fa;

size[sz]=cnt[sz]=1;

ch[fa][key[fa]<x]=sz;

key[sz]=x;

update(fa);

**splay(sz)**;

break;

}

}

}

inline int find(int x){//值x排第几小

int now=root,ans=0;

while(1){

if (x<key[now])

now=ch[now][0];

else{

ans+=(ch[now][0]?size[ch[now][0]]:0);

if (x==key[now]){

**splay(now);** return ans+1;

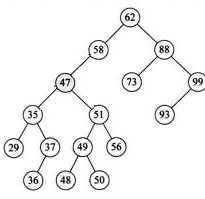
}

ans+=cnt[now];

now=ch[now][1];

}

}

}

inline int findx(int x){//找第x小的值。

int now=root;

while(1){

if (ch[now][0]&&x<=size[ch[now][0]])

now=ch[now][0];

else{

int temp=(ch[now][0]?size[ch[now][0]]:0)+cnt[now];

if (x<=temp) return key[now];

x-=temp; now=ch[now][1];

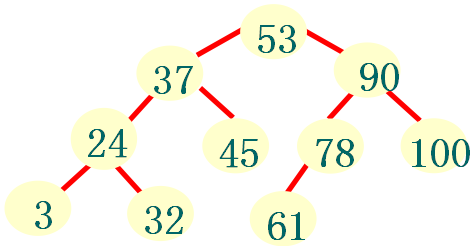
}

}

}

inline int pre(){//找root的前驱

int now=ch[root][0];

 while (ch[now][1]) now=ch[now][1];

return now;

}

inline int next(){//找root的后继

int now=ch[root][1];

while (ch[now][0]) now=ch[now][0];

return now;

}

inline void del(int x){

int whatever=find(x);

if (cnt[root]>1){cnt[root]--; update(root); return;}//被删的数有多个，不用删除节点

if (!ch[root][0]&&!ch[root][1]) {clear(root); root=0; return;}//只有根节点的树

if (!ch[root][0]){//没有左子树

int oldroot=root; root=ch[root][1]; f[root]=0; clear(oldroot); return;

}

else if (!ch[root][1]){//没有右子树

int oldroot=root; root=ch[root][0]; f[root]=0; clear(oldroot); return;

}

//有左、右子树

int leftbig=pre(),oldroot=root;//找比x的前驱点leftbig，保存删除节点oldroot。

**splay(leftbig);**//将leftbig旋转为根

//删除数x所在节点

ch[root][1]=ch[oldroot][1];

f[ch[oldroot][1]]=root;

clear(oldroot);

update(root);

}

int main(){

int n,opt,x;

scanf("%d",&n);

for (int i=1;i<=n;++i){

scanf("%d%d",&opt,&x);

switch(opt){

case 1: insert(x); break;

case 2: del(x); break;

case 3: printf("%d\n",find(x)); break;

case 4: printf("%d\n",findx(x)); break;

//case 5: insert(x); printf("%d\n",key[pre()]); del(x); break;

//case 6: insert(x); printf("%d\n",key[next()]); del(x); break;

case 5: find(x); printf("%d\n",key[pre()]);break;

case 6: find(x); printf("%d\n",key[next()]); break;

}

}

}