# 简介

二叉查找树满足任意一个节点，它的左儿子的权值自己的权值右儿子的权值。平衡树在满足二叉查找树条件的情况下各点尽可能分布均匀，使时间常数最小化。

splay平衡树是一种按值排序时可以实现所有普通平衡树的操作，按位置排序时可以实现区间翻转和平移的多功能平衡树，下面我们来介绍按值排序的splay平衡树。

# 变量声明

f[i]//i的父节点  
ch[i][0]//i的左儿子  
ch[i][1]//i的右儿子  
key[i]//i的关键字（一般为i代表的那个数字）  
cnt[i]//i节点关键字出现的次数  
siz[i]//i这个子树的大小（下面有多少个结点）  
sz//整棵树的大小（可以参考链向前式星那个表示总边数的变量）  
root//整棵树的根

# 基础构造

下面是几个简单的基本操作：

## clear操作

作用：将当前点的各项值都清零（删除之后清理数据）。代码略

## get操作

作用：判断当前的点是它父亲结点的左儿子还是右儿子。

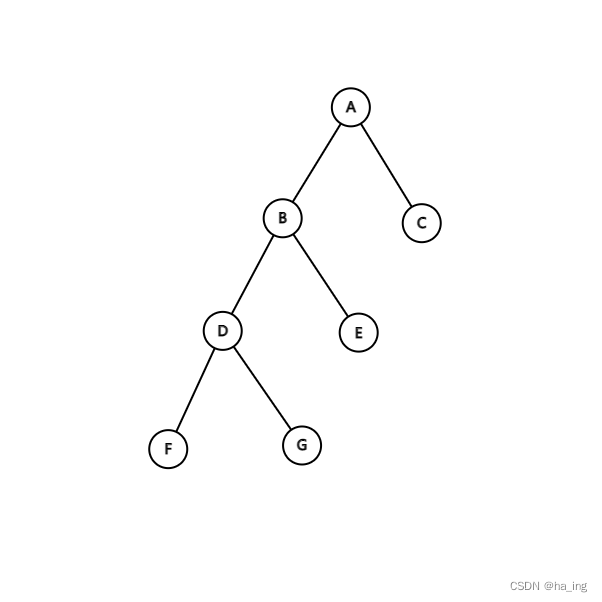
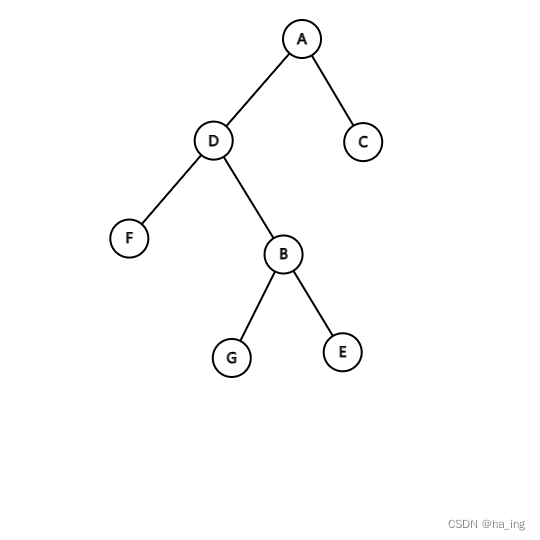
inline int get( int x ) {  
 return ch[f[x]][1] == x;//0左1右   
}

## update操作

作用：把更新的siz值从当前节点的左右儿子传到当前节点，更新当前结点的siz值（发生修改后用）。代码略

## rotate操作

作用：一个利用splay左右儿子标示的特殊性而把左旋右旋结合在一起的操作，使得平衡树更加平衡。这是splay平衡树最重要的操作之一。

分析： 我们假设这是原来的树，我们现在要从点D开始rotate，让D和它的父结点B调换位置，我们在此列出了B，D以及所有可能被影响到的其它点（无论B，D为自己父节点的左儿子还是右儿子，这个操作都不失一般性）。  这是我们希望的交换后的效果图。  我们先设变量which等于get( x )，其中x代表点D，用which表示D是左儿子还是右儿子。

我们通过观察，不难发现图中B的which儿子，D的which ^ 1儿子，以及B的父结点A（如果有的话）对应的左或右（当场判断）儿子也要换。

不难发现二叉查找树的性质左儿子的权值自己的权值右儿子的权值没变。

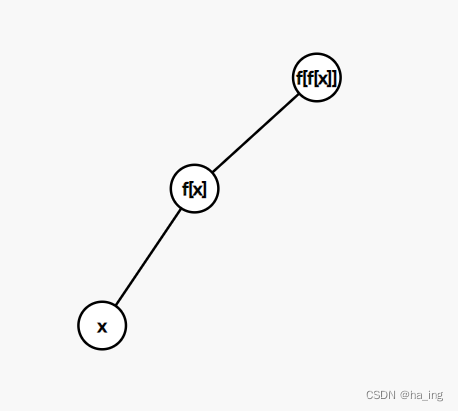
总而言之，我们先连BG，再连DB，最后连AD即可。并注意按顺序更新B，D的siz值。

inline void rotate( int x ) {  
 int old = f[x] , oldf = f[old] , which = get( x );  
 ch[old][which] = ch[x][which ^ 1];//连BG，断BD  
 f[ch[old][which]] = old;//连BG，断DG  
 ch[x][which ^ 1] = old;//连BD，断DB  
 f[old] = x;//连BD，断DB  
 f[x] = oldf;//连AD，断AB  
 if( oldf ) {  
 ch[oldf][ch[oldf][1] == old] = x;//连AD，断AB  
 }  
 update( old );//更新siz  
 update( x );//更新siz  
 return;   
}

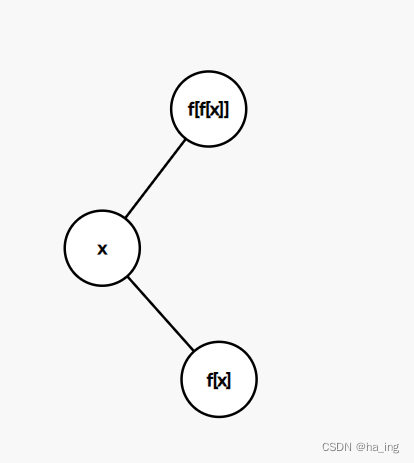
## splay操作

作用：rotate的发展，本质是不停地rotate，一直splay到根（或其它你喜欢的点），这也是splay中一个非常重要的操作。

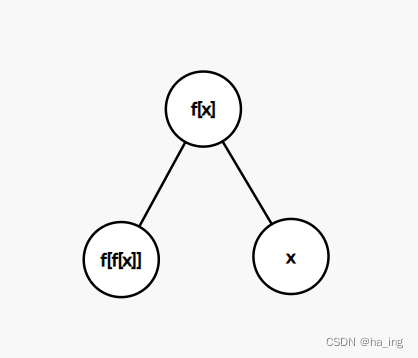
splay的过程中我们要分类讨论：

情况一：三点一线（x，x的父结点，x的祖父结点在一条），先rotatex的父结点，再rotatex本身。否则会形成单旋使平衡树失衡； 

直接rotate x而失衡



先rotate f[x]，再rotate x而没有失衡



情况二：没有三点一线，rotate x即可。

inline void splay( int x ， int goal ) {  
 for ( int fa ; ( fa = f[x] ) != goal ; rotate(x) ) {  
 if( f[fa] != goal ) {//如果父节点就是目标，就不能目标了  
 rotate( ( get( x ) == get( fa ) ? fa : x ) );//两个get判断是否有三点一线  
 }  
 }  
 if( !goal ) {  
 root = x;  
 }  
 return;  
}

以上我们完成了对splay平衡树的结构上的基本构造操作，接下来我们就讨论一下splay该怎么使用了。

# 基础操作

## insert操作

这个操作是插入一个值为k的数据到splay中，同时不改变splay的结构。 这里我们要分类讨论：

**情况一**，如果，即树为空，我们处理几个变量后直接返回；

**情况二**，，我们按照二叉查找树的性质一直往下找，若当前结点的关键字和要插入的点一样的话，把这个点加一个权值，更新一下当前点和父结点的siz和cnt，再splay上去；

**情况三**，，我们按照二叉查找树的性质一直往下找，若到了最底下，直接插入，整棵树大小sz加一，新结点的各项值更新一下（父，左右儿子，权值，大小），更新一下当前点的父结点的siz，再splay上去。

inline void insert( int v ) {  
 if( root == 0 ) {//情况一  
 ++sz;  
 root = sz;  
 ch[root][0] = ch[root][1] = f[root] = 0;  
 key[root] = v;  
 cnt[root] = siz[root] = 1;  
 return;  
 }  
 int cur = root , fa = 0;  
 while ( true ) {  
 if( key[cur] == v ) {//情况二  
 ++cnt[cur];  
 update( cur );  
 update( fa );  
 splay( cur );//一边rotate，一边往上传值（splay内含传值功能）  
 break;   
 }  
 fa = cur;  
 cur = ch[cur][key[cur] < v];  
 if( cur == 0 ) {//情况三  
 ++sz;  
 ch[sz][0] = ch[sz][1] = 0;  
 key[sz] = v;  
 siz[sz] = 1;  
 cnt[sz] = 1;  
 f[sz] = fa;  
 ch[fa][key[fa] < v] = sz;  
 update( fa );  
 splay( sz );  
 break;  
 }   
 }  
}

## find操作

这个操作目的是查询关键字（值）为v时的排名。一开始ans为零，当前点为root。

* 如果v比当前结点的关键字小，则应该向左子树寻找，ans不变；
* 如果v比当前结点的关键字大，则应该向右子树寻找，ans加上左子树的siz和当前点的cnt。

找到之后ans加一。最后要splay，别的操作有用。

inline int find( int v ) {  
 int ans = 0 , cur = root;  
 while ( true ) {  
 if( v < key[cur] ) {//向左子树寻找，ans不变  
 cur = ch[cur][0];  
 } else {//向右子树寻找，ans加上左子树的siz和当前点的cnt  
 ans += ( ch[cur][0] ? siz[ch[cur][0]] : 0 );  
 if( v == key[cur] ) {  
 splay( cur );  
 return ans + 1;//找到之后ans加一，前面本质是找比它小的有多少个  
 }  
 ans += cnt[cur];  
 cur = ch[cur][1];  
 }  
 }  
}

## findth操作

这个操作目的是查询排名为k的点。一开始当前点为root。

**情况一**，如果当前点有左子树，并且k小于左子树大小时，可以向左子树寻找；

**情况二**，先用tem表示左子树的siz（没有则为零）和当前点的cnt，看看排名为k的点是否为当前点（小于tem），然后k去减tem，从右子树开始找。

inline int findth( int k ) {  
 int cur = root;  
 while ( true ) {  
 if( ch[cur][0] && k <= siz[ch[cur][0]] ) {//情况一  
 cur = ch[cur][0];  
 } else {//情况二  
 int tem = ( ch[cur][0] ? siz[ch[cur][0]] : 0 ) + cnt[cur];  
 if( cur <= tem ) {//判断排名为k的点是否为当前点  
 return key[cur];  
 }  
 k -= tem;  
 cur = ch[cur][1];  
 }  
 }  
}

## pre/nxt操作

作用：pre找前驱，nxt找后继。

splay平衡树找前驱后继的思路是先插入被查找数（已被splay到根结点），前驱就是根节点左子树最右边的结点（最大的小于），后继就是根节点右子树最左边的结点（最小的大于），查找完后再删除被查找数。

inline int pre() {  
 int cur = ch[root][0];//比根小  
 while ( ch[cur][1] ) {//尽可能大  
 cur = ch[cur][1];  
 }  
 return cur;  
}  
inline int nxt() {  
 int cur = ch[root][1];//比根大  
 while ( ch[cur][0] ) {//尽可能小  
 cur = ch[cur][0];  
 }  
 return cur;  
}

为了便于大家理解，我展示一下如何在main函数中找到值为x的数的前驱，后继。 del()操作下面会讲。

```cpp  
int main() {  
 insert( x );  
 pre();  
 del( x );  
}

int main() {  
 insert( x );  
 nxt();  
 del( x );  
}

## del操作

这个操作作用是删除关键字（值）为v的点。

这个操作比较麻烦，我们先find一下v，把它旋到根。接下来我们要分多种情况讨论：

**情况一**：cnt[root]>1，不止有一个，直接减一；

**情况二**：splay只有一个点，直接clear；

**情况三**：如果root只有左儿子或只有右儿子，直接删了root，唯一的儿子做root；

**情况四**：root有两个儿子，我们要拿root的前驱作新根，将原先root的右子树接到新root的右子树上（由于选的是前驱，原先root的前驱一定没有右子树）。 删完后不忘update。

inline void del( int v ) {  
 find( v );  
 if( cnt[root] > 1 ) {//情况一  
 --cnt[root];  
 update( root );  
 return;  
 }  
 if( !ch[root][0] && !ch[root][1] ) {//情况二  
 clear( root );  
 root = 0;  
 sz = 0;  
 return;  
 }  
 if( !ch[root][0] ) {//情况三  
 int oldroot = root;  
 root = ch[root][1];  
 f[root] = 0;  
 clear( oldroot );  
 --sz;  
 return;  
 } else if( !ch[root][1] ) {//情况三  
 int oldroot = root;  
 root = ch[root][0];  
 f[root] = 0;  
 clear( oldroot );  
 --sz;  
 return;  
 }  
 //情况四  
 int lpre = pre() , oldroot = root;  
 splay( lpre );//拿root的前驱作新根  
 f[ch[oldroot][1]] = root;//将原先root的右子树接到新root的右子树上  
 ch[root][1] = ch[oldroot][1];//将原先root的右子树接到新root的右子树上  
 clear( oldroot );  
 update( root );  
 return;  
}

至此我们终于学会了splay的基本构造和基本操作了\ ^ o ^ /。 以上就是splay平衡树的基本构造和基本操作了，感谢大家阅读。