平衡树

目录

1、二叉查找树（BST)

2、伸展树（splay)

3、Treap

前言

在OI题目中时常需要这样一种数据结构:
维护一个集合，要求支持:
插入或删除数字，查询数字排名，查询数字前驱、后继;

或维护一个序列，要求支持:
在序列中插入或删除元素，对一段区间修改或查询信息。

这时我们通常会使用平衡树。

二叉查找树（BST）

在二叉树的基础上，每个节点有一个权值，若每个节点满足：

1、其左子树所有权值小于等于自身权值

2、其右子树所有权值大于等于自身权值

我们将其称为二叉查找树。

二叉查找树满足：按其中序遍历输出权值，那么权值不降。

二叉查找树的操作

```
//遍历
void inorder(int x){
    if(!x) return;
    inorder(l[x]);
    visit(a[x]);
    inorder(r[x]);
}
//查找 v
int query(int x,int v){
    if(!x) return 0;
    if(v==a[x]) return x;
    else if(v<a[x]) return query(l[x],v);
    else return query(r[x],v);
}
//求最小值
int query_min(int x){
    if(!x) return 0;
    while(l[x]) x=l[x];
    return x;
}
```

```
//插入 v
void insert(int &x,int v){
    if(!x) x=++tot,a[tot]=v;
    else if(v<a[x]) insert(l[x],v);
    else if(v>a[x]) insert(r[x],v);
    //else cnt[x]++;
}
//删除最小节点
int delete_min(int &x){
    if(!l[x]){
        int ret=a[x];
        x=r[x];
        return ret;
    }
    else return delete_min(l[x]);
}
//删除 v
void del(int &x,int v){
    if(a[x]==v){
        //if(cnt[x]>1) cnt[x]--;
        if(l[x]&&r[x]) a[x]=delete_min(r[x]);
        else x=l[x]+r[x];
        return;
    }
    if(a[x]>v) del(l[x],v);
    else del(r[x],v);
}
```

平衡树

二叉查找树有一个明显的缺点：

特殊构造的数据能使树的深度达到O(n)级别，也就是每次查询或修改的最坏时间复杂度会达到O(n)。

在二叉查找树的基础上，设法控制其深度，便出现了平衡树。

其中常用的有三种：

Splay

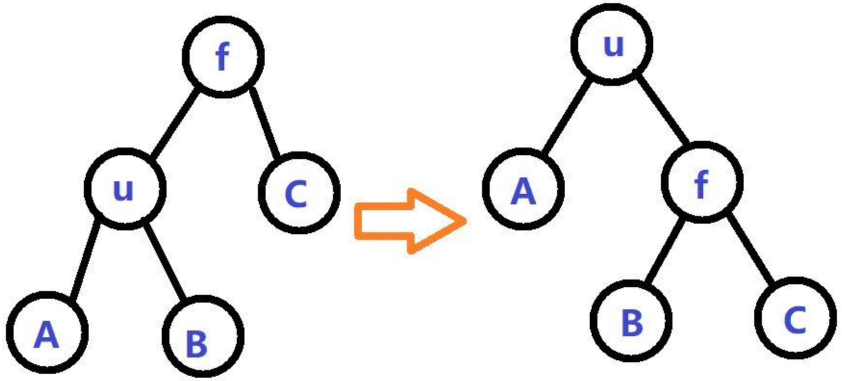
Treap

FHQ-Treap

splay树

rotate:

在二叉查找树的基础上，通过旋转等一系列操作，改变树的形态。分为左旋和右旋。




```
void update(int x){
    sum[x]=sum[ch[x][0]]+sum[ch[x][1]]+cnt[x];
}
int getwh(int x)
{
    return ch[fa[x]][0]==x?0:1;
}
void rotate(int x){
    int y=fa[x],z=fa[y],k=getwh(x);
    fa[x]=z;
    ch[z][getwh(y)]=x;
    fa[ch[x][k^1]]=y;
    ch[y][k]=ch[x][k^1];
    fa[y]=x;
    ch[x][k^1]=y;
    update(y);update(x);
}
```

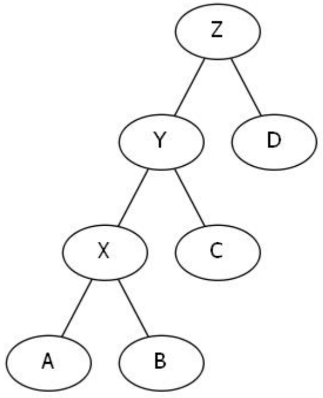
splay

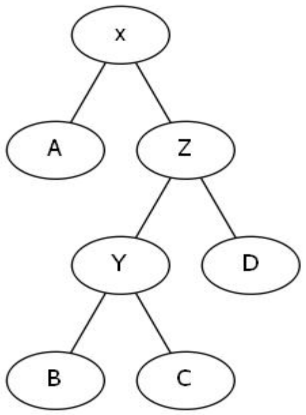
splay:

rotate(u)操作即为刚才所示，功能为将节点u旋转至其父节点f的位置。

splay还有另一个重要的操作称为伸展: splay(x, tar)，功能为将 x 不断 rotate 直至其成为 tar 的孩子（若tar = 0则 x 成为根节点）。

如果单纯地每次rotate(x)，我们称之为单旋，由其时间复杂度不能保证。通常使用双旋。





splay

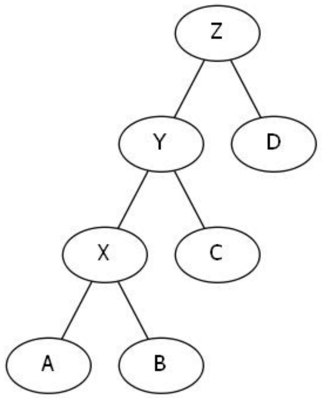
splay:

一个简单的描述:

若当前点的爷爷为tar，旋转自己，结束。

若当前点、父亲、爷爷在一条直线，则先旋转父亲，再旋转自己。

若当前点、父亲、爷爷不在一条直线，旋转两次自己。




```
void splay(int x,int tar){
    while(fa[x]!=tar){
        int y=fa[x],z=fa[y];
        if(z!=tar){
            if(getwh(x)==getwh(y)) rotate(y);
            else rotate(x);
        }
        rotate(x);
    }
    if(!tar) root=x;
}
```

splay

find v:

查找值 v 并旋转到根节点

```
void find(int v){
    int x=root;
    if(!x) return;
    while(ch[x][v>a[x]]&&v!=a[x]){
        x=ch[x][v>a[x]];
    }
    splay(x,0);
}
```

splay

insert v:

插入值 v 并旋转到根节点

```
void insert(int v){
    int x=root,y=0;
    while(x&&a[x]!=v){
        y=x;
        x=ch[x][v>a[x]];
    }
    if(x) cnt[x]++;
    else{
        x=++tot;
        if(y) ch[y][v>a[y]]=x;
        ch[x][0]=ch[x][1]=0;
        fa[x]=y,a[x]=v,cnt[x]=1,sum[x]=1;
    }
    splay(x,0);
}
```

splay

next v:

查找值 v 的前驱或后继（先将 v splay 为根，前驱则是左子树最大值，一直往右跳，后继相反）

```
int nextt(int v,int k){
    //k=0:前驱    k=1:后继
    find(v);
    int x=root;
    if(a[x]>v&&k==1) return x;
    if(a[x]<v&&k==0) return x;
    x=ch[x][k];
    while(ch[x][k^1]) x=ch[x][k^1];
    return x;
}
```

splay

del v:

删除值 v 的节点（先将 v 的前驱 splay 为根， v 的后继 splay 到 前驱的右儿子， 这时后继的左儿子就是 v)

```
void del(int v){
    int pre=nextt(v,0),next=nextt(v,1);
    splay(pre,0),splay(next,pre);
    int tmp=ch[next][0];
    if(cnt[tmp]>1) cnt[tmp]--,splay(tmp,0);
    else{
        ch[next][0]=0;
        fa[tmp]=0;
    }
}
```

splay

kth k:

查找第 k 大的值。

```
int kth(int k) {
    int x=root;
    if(sum[x]<k) return 0;
    while(1) {
        int lc=ch[x][0];
        if(k>sum[lc]+cnt[x]){
            k-=sum[lc]+cnt[x];
            x=ch[x][1];
        }else{
            if(sum[lc]>=k) x=lc;
            else return a[x];
        }
    }

}
```

splay

getrank v:

查找值v的排名。

```
int getrank(int v){
    insert(v);
    int res=sum[ch[root][0]];
    del(v);
    return res;
}
```

splay

merge(x,y)

x 当作 y 的左子树合并然后 splay x 为根.

也可以借此进行删除操作。

```
void merge(int x,int y) {
    if(x==0) {
        rt=y;
        return;
    }
    if(y==0) {
        rt=x;
        return;
    }
    while(ch[y][0]) {
        sum[y]+=sum[x];
        y=ch[y][0];
    }
    sum[y]+=sum[x];
    ch[y][0]=x,fa[x]=y;
    splay(x,0);
}
```

```
void del(int v) {
    find(v);
    int x=root;
    fa[ch[x][0]]=f[ch[x][1]]=0;
    merge(ch[x][0],ch[x][1]);
}
```

splay

若需要操作区间[l,r]，那么先后执行splay(l-1,0)，splay(r+1,l-1)，这样根节点的右孩子的左孩子为根的子树便对应区间[l,r]。
既可以对区间维护信息，又可以对区间修改、打标记。

为了方便提取区间，可以在序列两侧各添加一个辅助节点。

```
insert(INF);
insert(-INF);
```

文艺平衡树

维护一个集合，要求支持翻转一个区间，求最终的序列。

n<=100000

splay基本操作

对区间打翻转标记，下传标记即为交换左右孩子并对孩子打标记。

注意splay一个点之前，从上到下将根节点到该节点路径上的点依次pushdown

二逼平衡树（树套树）

维护一个序列，要求支持:

1、查询 x 在区间[l,r]中的排名

2、查询区间[l,r]中排名为 x 的数

3、修改某个位置的数值

4、查询 x 在区间[l,r]中的前驱

5、查询 x 在区间[l,r]中的后继

n<=50000

二逼平衡树（树套树）

线段树套平衡树模板

对序列建一棵线段树，线段树上的每个节点上，将这个区间内的数字按权值构建一棵平衡树。

二逼平衡树（树套树）

1、查询 x 在区间[l,r]中的排名

在线段树对应的O(log n)个区间上，在每个的平衡树上查询小于 x 的数的数量即可。

O(log^2 n)

2、查询区间[l,r]中排名为 x 的数

我们已经会了查询 x 在[l,r]中的排名，那么可以二分答案，每次查询其排名。

O(log^3 n)

二逼平衡树（树套树）

3、修改某个位置的数值

在线段树上从该位置到根节点的每个节点上，将原数值删除，插入新数值。

O(log^2 n)

4、查询 x 在区间[l,r]中的前驱

5、查询 x 在区间[l,r]中的后继

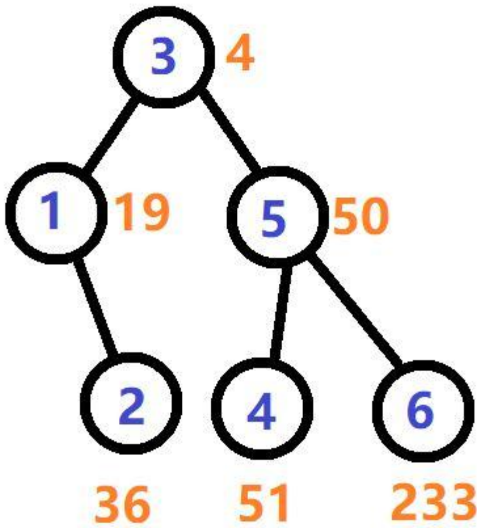
在线段树对应的O(log n)个区间上，在每个的平衡树上查询 x 的前驱/后继，最后取max/min即可。

O(log^2 n)

treap

Treap一词来源于Tree（树）与Heap（堆）的结合。

其原理是，对于每个节点，赋予一个随机权值，在构建平衡树时，使得原权值满足二叉查找树的性质，随机权值满足堆的性质，利用随机性，限制树的深度。



笛卡尔树

treap本质上是一种笛卡尔树。

笛卡尔树每一个结点由一个键值二元组 (k , v) 构成。要求 k 满足二叉搜索树的性质，而 v 满足堆的性质。

特别的，我们把 k 对应数组下标，v 对应数组元素值，我们可以用单调栈实现笛卡尔树的构建，每次插入的元素都在右链上。

```
int n,top;
int st[N],lc[N],rc[N],a[N]

for(int i=1;i<=n;i++){
    while(top && a[st[top]]>a[i]) lc[i]=st[top--];
    if(top) rc[st[top]]=i;
    st[++top]=i;
}
```

treap

rotate:

同splay的rotate。

```
void update(int x){
    sum[x]=sum[ch[x][0]]+sum[ch[x][1]]+cnt[x];
}
int getwh(int x){
    return ch[fa[x]][0]==x?0:1;
}
void rotate(int x){
    int y=fa[x],z=fa[y],k=getwh(x);
    fa[x]=z;
    ch[z][getwh(y)]=x;
    fa[ch[x]][k^1]=y;
    ch[y][k]=ch[x][k^1];
    fa[y]=x;
    ch[x][k^1]=y;
    update(y),update(x);
    if(!z) root=x;
}
```

treap

insert:

从根节点出发，按照二叉查找树的性质向下走，走到空节点时插入在该位置。

此时随机权值会不满足堆的性质，接下来，只要其随机权值小于父亲，就执行rotate操作，直到随机权值大于父亲。

```
void insert(int &k,int v) {
    if(!k){
        k=++tot,key[k]=rand(),a[k]=v;
        ch[k][0]=ch[k][1]=fa[k]=0,sum[k]=cnt[k]=1;
        return;
    }
    else sum[k]++;
    if(a[k]==v) cnt[k]++;
    else if(v<a[k]){
        insert(ch[k][0],v);
        if(key[ch[k][0]]<key[k]) rotate(ch[k][0]);
    }else{
        insert(ch[k][1],v);
        if(key[ch[k][1]]<key[k]) rotate(ch[k][1]);
    }
}
```

treap

del:

只要待删除节点不为叶节点，就选择其随机权值较小的孩子，对其执行rotate。

直至待删除节点为叶节点时，直接删掉即可。

```
void del(int v) {
    int x=root;
    while(x) {
        if(a[x]>v) {
            sum[x]--,x=ch[x][1];
        } else if(a[x]<v) {
            sum[x]--,x=ch[x][0];
        } else break;
    }
    if(cnt[x]>1) cnt[x]--,sum[x]--;
    else {
        int minwh=key[ch[x][0]]<key[ch[x][1]]?0:1;
        while(key[ch[x][minwh]]<INF) {
            rotate(ch[x][minwh]);
            minwh=key[ch[x][0]]<key[ch[x][1]]?0:1;
        }
        if(fa[x]==0) {
            root=0;
            return;
        }
        ch[fa[x]][getwh(x)]=0;
        fa[x]=0;
    }
}
```

treap

kth:

同splay kth.

```
int kth(int k) {
    int x=root;
    if(sum[x]<k) return 0;
    while(1) {
        int lc=ch[x][0];
        if(k>sum[lc]+cnt[x]){
            k-=sum[lc]+cnt[x];
            x=ch[x][1];
        }else{
            if(sum[lc]>=k) x=lc;
            else return a[x];
        }
    }

}
```

treap

getrank:

同splay getrank.

```
int getrank(int v) {
    int x=root,rank=0;
    while(x) {
        if(a[x]<v) {
            rank+=sum[ch[x][0]]+cnt[x];
            x=ch[x][1];
        } else x=ch[x][0];
    }
    return rank+1;
}
```

treap

同样，初始化时要现在平衡树中加入一个极大值和一个极小值，防止越界。

```
insert(root,-INF);
insert(root,INF);
update(root);
```

treap

OI中最常用的平衡树为Splay，其优点在于方便的区间操作，以及可以用来写Link-Cut Tree，缺点为常数大。

Treap的优点为常数小。

fhq-treap(非旋treap)

treap是静态的，不需要旋转，因此可以支持可持久化操作。

非旋 treap 通过两个核心操作：分裂split和合并merge 来改变树形态。

非旋 treap 编程复杂度较低。

fhq-treap(非旋treap)

merge:

根据附加权值 key 合并 x 和 y 两棵树。

```
int merge(int x,int y){
    if(!x||!y) return x+y;
    pushdown(x),pushdown(y);
    if(key[x]<key[y]){
        ch[x][1]=merge(ch[x][1],y);
        update(x);
        return x;
    }else{
        ch[y][0]=merge(x,ch[y][0]);
        update(y);
        return y;
    }
}
```

fhq-treap(非旋treap)

split:

将树 x 的前 k 个保留， 剩下的分裂成树 y。

```
void split(int now,int k,int &x,int &y){
    if(!now){
        x=y=0;return;
    }
    pushdown(now);
    if(sum[ch[now][0]]<k){
        x=now,split(ch[now][1],k-sum[ch[now][0]]-1,ch[x][1],y);
        update(x),update(y);
    }
    else{
        y=now,split(ch[now][0],k,x,ch[y][0]);
        update(x),update(y);
    }
}
```

fhq-treap(非旋treap)

split_v:

将树 x 的 <=v 的部分保留， 剩下的分裂成树 y。

```
void split_v(int now,int v,int &x,int &y){
	f(!now){
		x=y=0;return;
	}
	pushdown(now);
	if(a[now]<=v){
		x=now,split_v(ch[now][1],v,ch[x][1],y);
		update(x),update(y);
	}
	else{
		y=now,split_v(ch[now][0],v,x,ch[y][0]);
		update(x),update(y);
	}
}
```

fhq-treap(非旋treap)

getrank:

找值 v 的排名第 几 小/大。

```
int getrank(int v) {
    int o=root,rank=0;
    while(o) {
        if(v<=a[o]) o=ch[o][0];
        else rank+=sum[ch[o][0]]+1,o=ch[o][1];
    }
    return rank+1;
}
```

```
int getrerank(int v)
{   int o=root,rank=0;
    while(o)
    {   if(v>=a[o])o=ch[o][1];
        else rank+=sum[ch[o][1]]+1,o=ch[o][0];
    }
    return rank+1;
}
```

fhq-treap(非旋treap)

kth:

找第 k 小/大。

```
int kth(int k) {
    int o=root;
    while(o) {
        if(k==sum[ch[o][0]]+1)return a[o];
        if(k<=sum[ch[o][0]])o=ch[o][0];
        else k-=sum[ch[o][0]]+1,o=ch[o][1];
    }
}
```

```
int rekth(int k) {
    int o=root;
    while(o) {
        if(k==sum[ch[o][1]]+1) return a[o];
        if(k<=sum[ch[o][1]])o=ch[o][1];
        else k-=sum[ch[o][1]]+1,o=ch[o][0];
    }
}
```

fhq-treap(非旋treap)

insert:

插入 v。

```
void insert(int v) {
    a[++tot]=v,sum[tot]=1,key[tot]=rand();
    int k=getrank(v),x,y;
    split(root,k,x,y);
    root=merge(merge(x,tot),y);
}
```

fhq-treap(非旋treap)

del:

删除 v。

```
void del(int v) {
    int k=getrank(v),x,y,t;
    split(root,k,x,y);
    split(y,1,t,y);
    root=merge(x,y);
}
```

fhq-treap(非旋treap)

pre/next:

求 v 的前驱和后继。

```
int pre(int v) {
    return kth(getrank(v)-1);
}

int next(int ww) {
    return rekth(getrerank(v)-1);
}
```

总结

OI中平衡树并没有非常常用，而且在一些情况下线段树可替代之，但是平衡树依旧是一种功能强大的序列维护工具。

平衡树的题目也比较模板化，就是在线段树的基础上多了插入删除的功能。
想熟练掌握模板一定要多写，只能多写。