

# ETT

## 算法简介

ETT 是一种利用括号序列维护动态树子树信息的算法。

## 单括号序列(dfs 序)

2021牛客暑期多校训练营6 E

### 题意

给定一棵树，初始时只有一个颜色为  $c$  的  $1$  号结点，接下来  $m$  个操作：

1. 对结点  $u$  添加一个颜色为  $c$  的叶子结点  $n+1$  (设当前共有  $n$  个结点)
2. 查询结点  $u$  子树颜色为  $c$  的结点数量

### 题解

维护每个结点  $u$  的最新儿子  $\text{hson}(u)$  以及每个结点的子树在单括号序列的终止结点  $\text{dfr}(u)$  (子树不包含该点)。

于是新插入叶子结点  $n+1$  时，如果  $u$  没有叶子结点，则有  $\text{dfr}(n+1) \leftarrow \text{dfr}(u)$  否则有  $\text{dfr}(n+1) \leftarrow \text{hson}(u)$

然后将  $\text{hson}(u)$  更新为  $n+1$  即可实现序列的维护。子树查询等价于查询序列  $[\text{pos}(u), \text{pos}(\text{dfr}(u))]$  的信息。

接下来为每种颜色开一个平衡树，每棵平衡树维护该颜色结点的所有位置。

则操作  $2$  等价于查询颜色为  $c$  的平衡树中位置位于  $[\text{pos}(u), \text{pos}(\text{dfr}(u))]$  的结点数。

发现  $\text{pos}(u)$  是动态更新的，但是  $\text{pos}(u), \text{pos}(v)$  的相对大小是不会改变的，因此考虑建立一棵平衡树维护所有  $\text{pos}(u)$

如果用  $\text{splay}$  维护  $\text{pos}(u)$  则每次查询  $\text{pos}(u)$  时间复杂度为  $O(\log n)$  在颜色为  $c$  的平衡树查询的结点数时间复杂度为  $O(\log^2 n)$

考虑  $O(1)$  查询  $\text{pos}(u)$  一种想法是将  $\text{pos}(u)$  映射到实数空间，每次插入叶子结点则将  $\text{pos}(n+1)$  赋值为  $\frac{\text{pos}(u) + \text{pos}(v)}{2}$

其中  $u, v$  表示  $n+1$  在单括号序列的左右相邻结点，但这样会导致精度误差。

考虑替罪羊树维护  $\text{long long}$  空间，替罪羊树定期重构重新赋值部分  $\text{pos}(u)$  保证了树的深度，不会导致精度误差。

算法总时间复杂度变为  $O(n \log n)$

```
const int MAXN=5e5+5;
const LL MAXV=1LL<<62;
LL val[MAXN];
namespace Tree1{
#define lch(k) node[node[k].lch]
#define rch(k) node[node[k].rch]
const double alpha=0.70;
struct Node{
    int lch,rch,sz;
}node[MAXN];
int root,a[MAXN],n;
bool isbad(int k){
    return alpha*node[k].sz<max(lch(k).sz,rch(k).sz);
}
void build(int &k,int lef,int rig,LL vl,LL vr){
    if(lef>rig) return k=0,void();
    int mid=lef+rig>>1;
    k=a[mid];
    val[k]=vl+vr>>1;
    if(lef==rig){
        node[k].lch=node[k].rch=0;
        node[k].sz=1;
        return;
    }
    build(node[k].lch,lef,mid-1,vl,val[k]-1);
    build(node[k].rch,mid+1,rig,val[k]+1,vr);
    node[k].sz=lch(k).sz+rch(k).sz+1;
}
void dfs(int k){
    if(!k) return;
    dfs(node[k].lch);
    a[++n]=k;
    dfs(node[k].rch);
}
void rebuild(int &k,LL vl,LL vr){
    n=0;
    dfs(k);
    build(k,1,n,vl,vr);
}
void check(int &k,int id,LL vl,LL vr){
    if(k){
        if(isbad(k))
            rebuild(k,vl,vr);
        else if(val[id]<val[k])
            check(node[k].lch,id,vl,val[k]-1);
        else
            check(node[k].rch,id,val[k]+1,vr);
    }
}
```

```

    }
}
void Insert(int &k,int id){
    if(!k){
        k=id;
        node[k].lch=node[k].rch=0;
        node[k].sz=1;
        return;
    }
    node[k].sz++;
    if(val[id]<val[k])
        Insert(node[k].lch,id);
    else
        Insert(node[k].rch,id);
}
void Insert(int id){
    Insert(root,id);
    check(root,id,0,MAXV);
}
#undef lch
#undef rch
}
namespace Tree2{
    struct Node{
        int r,sz,ch[2];
    }node[MAXN];
    #define lch(k) node[node[k].ch[0]]
    #define rch(k) node[node[k].ch[1]]
    void push_up(int k){
        node[k].sz=lch(k).sz+rch(k).sz+1;
    }
    void split(int k,int &k1,int &k2,LL v){
        if(!k) return k1=k2=0,void();
        if(val[k]<=v){
            k1=k;
            split(node[k].ch[1],node[k1].ch[1],k2,v);
            push_up(k1);
        }else{
            k2=k;
            split(node[k].ch[0],k1,node[k2].ch[0],v);
            push_up(k2);
        }
    }
    void merge(int &k,int k1,int k2){
        if(!k1||!k2)return k=k1|k2,void();
        if(node[k1].r>node[k2].r){
            k=k1;
            merge(node[k].ch[1],node[k1].ch[1],k2);
            push_up(k);
        }else{
            k=k2;

```

```
        merge(node[k].ch[0],k1,node[k2].ch[0]);
        push_up(k);
    }
}
void Insert(int &root,int id){
    node[id].r=rand();
    node[id].sz=1;
    node[id].ch[0]=node[id].ch[1]=0;
    int lef,rig;
    split(root,lef,rig,val[id]);
    merge(lef,lef,id);
    merge(root,lef,rig);
}
int rank(int root,LL v){
    int lef,rig,ans;
    split(root,lef,rig,v-1);
    ans=node[lef].sz+1;
    merge(root,lef,rig);
    return ans;
}
int query(int root,LL vl,LL vr){
    return rank(root,vr)-rank(root,vl);
}
#undef lch
#undef rch
};
int root[MAXN],col[MAXN],hson[MAXN],dfr[MAXN];
void solve(){
    int n=1;
    col[1]=read_int();
    val[1]=0;
    val[0]=MAXV;
    Tree1::Insert(1);
    Tree2::Insert(root[col[1]],1);
    int m=read_int(),lastans=0;
    while(m--){
        int t=read_int()^lastans,u=read_int()^lastans,c=read_int()^lastans;
        if(t==1){
            int v=hson[u]?hson[u]:dfr[u];
            hson[u]=++n;
            col[n]=c;
            dfr[n]=v;
            val[n]=val[u]+val[v]>>1;
            Tree1::Insert(n);
            Tree2::Insert(root[c],n);
        }
        else{
            lastans=Tree2::query(root[c],val[u],val[dfr[u]]);
            enter(lastans);
        }
    }
}
```

```
    }  
  }  
  Tree1::root=0;  
  _rep(i,1,n){  
    root[col[i]]=0;  
    hson[i]=dfr[i]=0;  
  }  
}  
int main()  
{  
  int T=read_int();  
  while(T--)  
    solve();  
  return 0;  
}
```

## 双括号序列(欧拉序)

咕咕咕

From:  
<https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link:  
[https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal\\_string:jxm2001:ett&rev=1628860580](https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:jxm2001:ett&rev=1628860580)

Last update: **2021/08/13 21:16**