Link Cut Tree, LCT

可以 **在线** 维护 **树** (或是森林) 的形态和一些信息, LCT 是通用性很强的数据结构.

定义

link cut tree 使用 **splay** 维护树上的链.

偏爱子节点 (preferred child) : 对于节点 **x** , 若其子节点 **y** 与 x 在同一条路径上, 那么 y 为 x 的偏爱子节点. 显然, 一个点的偏爱子节点最多只有一个.

偏爱边 (preferred edge) : 对于节点 **x** , 若其有偏爱子节点, 它连向偏爱子节点的边为偏爱边.

偏爱路径 (preferred path) : 由偏爱边组成的极长路径.

实现

首先需要将树拆分成若干条偏爱路径, 每一条偏爱路径都用 **splay** 维护, 不同的偏爱路径用虚边相连.

一条路径按照深度建成 splay, 不同的路径对应的 splay 按照路径之间的祖先关系连边. 我们用 2 个数组来描述 splay 上的节点: $fa[x]$ 和 $ch[x][2]$. 其中 $ch[x]$ 表示的就是 splay 中的儿子, 左儿子表示深度比 x 小的部分, 右儿子表示深度比 x 大的部分. $fa[x]$ 有两种情况, 若 x 不是某一个 Splay 的根节点, 那么 $fa[x]$ 表示的是它在 Splay 二叉树上的父亲, 否则其指向的是这颗 Splay 树表示的路径的父亲节点 (虚边).

Access

Access 是 LCT 中的基本操作, 它将点 x 到当前树根的路径变为偏爱路径.

完成这个操作需要两步:

1. 将 x 到其所有直接儿子的边变为非偏爱边.

2. 将 x 到当前树根的路径上的所有边变为偏爱边. 相当于将路径上所有点 p 原先的偏爱子节点改变成 x 到根路径上的节点.

考虑我们是如何判断一条边是否为偏爱边. 对于点 p, 其父亲若和它在同一个 Splay 树中, 那么 p 的父亲到 p 的边就是偏爱边, 否则不是.

那么转化偏爱边 (或者说偏爱子节点) 就可以非常容易, 对于点 x, 将其旋转至 Splay 树的根节点, 那么 $ch[x][1]$ 对应的 Splay 子树对应的就是 x 所在偏爱路径中深度大于 x 的子路径. 此时修改 $ch[x][1]$ 就可以修改 x 的偏爱子节点.

可以统一前面的两个操作来简化代码: 自下向上的访问一条路径, 对于当前点 u, 将他的偏爱子节点修改为 v (v 是在访问 u 之前访问的节点). 对于 $u = x$, $v = 0$.

考虑如何自下向上访问一条路径. 起始点 x 就是最底端的节点, 所以只要考虑每次如何得到路径上的某点 p 的父节点就可以了. 如果 p 是原先的某条偏爱路径的端点 (某条偏爱路径中深度最小的点), 只需将其旋转至其所在 Splay 的根节点, 此时 $fa[p]$ 指向的就是 p 在树中的父节点. 若 p 不是偏爱路径的端点, 那么此时 $fa[p]$ 指向的仍是是端点在树中的父节点 (由 fa 数组的定义可知) . 所以访问路径端点的父亲容易, 而访问任意点的父亲难. 但是 Access 操作中, 我们是要将 x 到根的路径转化为偏爱路径, 所以可以避免这个问题. 对于 x 所在的偏爱路径, 只需要不断地把它的端点 p 变为它父亲的偏爱子节点就可以连接偏爱路径了.

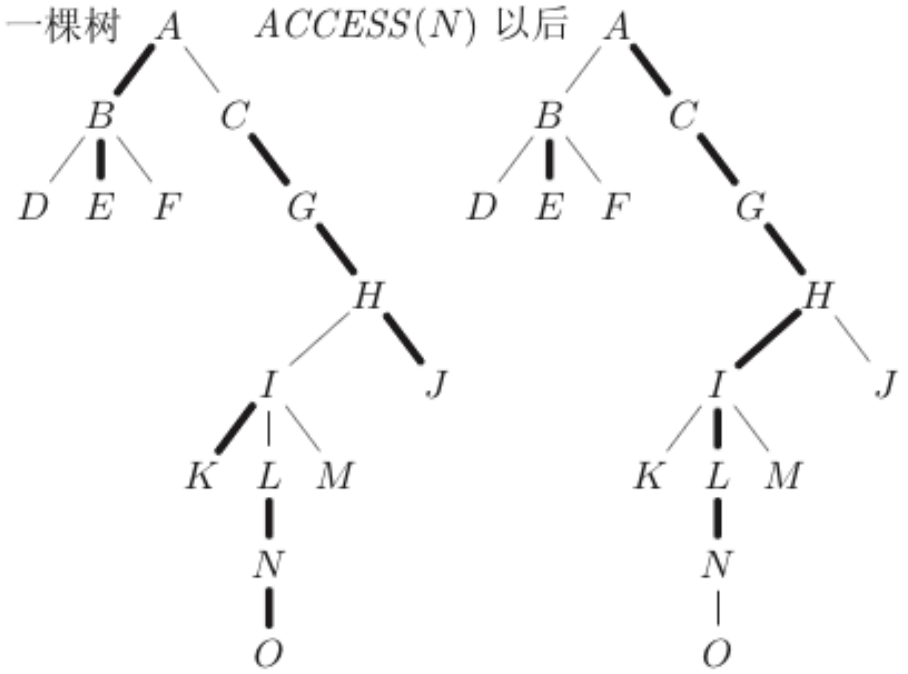

Figure 1:

```
void Access(int x) {
    int p = 0;
    while (x) {
        Splay(x);//旋转至根
        ch[x][1] = p;
        p = x;
        x = fa[x];
    }
}
```

代码是非常简洁的.

makeRoot

makeRoot, 就是将 x 所在树中的根换为 x.

考虑是如何确定树根的. 由于 Splay 是按照深度建树的, 所以肯定也要从深度考虑. 树根, 就是一棵树中深度最小的节点:-) 或者说, 深度最小的偏爱路径中深度最小的节点 (路径的深度可以定义为路径中深度最小的点的深度) .

所以可以这么做: 先 Access(x), 此时 x 位于深度最小的偏爱路径中的最大深度, 然后翻转 x 所在的偏爱路径 (翻转 x 所在的 Splay , 就是翻转一个序列) , 那么 x 就是当前树中深度最小的节点了. 那么如何处理其他 Splay 呢? 不需要处理, 因为它们使用虚边连接的, 只有父子关系而已:-)

翻转序列可以用延迟标记做到.

```
void makeRoot(int x) {
    Access(x);
    Splay(x);
    Flip(x);
}
```

涉及到延迟标记的 Splay:

```
bool isRoot(int x) {
    int p = fa[x];
    return ch[p][0] != x && ch[p][1] != x;
}
void Flip(int x) {
    fp[x] ^= 1;
    swap(ch[x][0], ch[x][1]);
}
void down(int x) {
    int l = ch[x][0], r = ch[x][1];
    if (fp[x]) {
        if (l) Flip(l);
        if (r) Flip(r);
        fp[x] = false;
    }
}
void Splay(int x) {
    static int stk[MAXN];
    int tp = 0, p = x;
    stk[tp++] = p;
    while (!isRoot(p)) stk[tp++] = p = fa[p];
    per (i, 0, tp - 1) down(stk[i]);
    while (!isRoot(x)) {
        int y = fa[x];
        if (isRoot(y)) Rotate(x);//Rotate 就是旋转一个点
        else {
            int z = fa[y];
            if ((ch[z][0] == y) ^ (ch[y][0] == x)) Rotate(x);
            else Rotate(y);
            Rotate(x);
        }
    }
}
```

link / cut

都叫 link cut tree 了, 肯定少不了动态加边和删边了.

link x y splay 之间连的是虚边, 在 link 时提供了一些方便. 虚边只维护了路径与路径的父子关系, 所以只需要维护 x 所在树的根节点所在的偏爱路径 (x 也要在上面) 和 y 所在的偏爱路径的父子关系就可以了.

cut x y 删边可在很多情况下删, 可以把他们转化为: 以 y 为根, 删去 x 到它父亲的边. 所以必须判断 x 和 y 是否在同一颗子树中. 删去父亲边是很容易的, 先 Access(x), 此时 x 为偏爱路径中深度最大的点, 然后在 Splay 树中切断 x 和其左儿子的关系即可.

```
void cut(int x, int y) {
    makeRoot(y);
    Access(x);
    Splay(x);
    fa[ch[x][0]] = 0;
    ch[x][0] = 0;
}
void link(int x, int y) {
    Access(x);
    Splay(x);
    fa[x] = y;
}
```

(这里都没有判断特殊情况)

用 LCT 维护一些信息

LCT 可以直接维护树的形态. 除此之外还可以维护一些权值的信息.

维护路径上的权值

路径权值是比较好维护的, 因为 LCT 本来就是把树拆成若干路径.

点权

通常会维护路径点权的和, 乘积, 最值等等. 只需要将路径的一个端点定为根, Access 另一个端点, 就切出了对应路径的 Splay 树, 然后就转化为用 Splay 维护信息的问题了.

在 Splay 中合并信息时可能会有顺序要求, 比如按照深度顺序合并. 此时要利用 LCT 的 Splay 是按照深度建树的性质, 所以左子树的深度小, 右子树的深度大.

边权

LCT 里只存了点, 没有存边. 所以要维护边权要把边当成点, 边和点之间再用一条边连接. 其实和维护点权是一样的.

要注意开两倍的内存.

维护子树权值和

由于 LCT 将树拆成了路径在 Splay 里维护, 所以无法在 Splay 里得到子树信息. 仅仅靠 Splay 是无法维护子树的.

LCT 中的边分为两种:

1. 连向子树的实边
2. 连向父亲的虚边

LCT 的操作, 无论是 Access, link, cut 最多改变一条实边和一条虚边. 我们可以额外维护虚子树信息, 然后利用 Splay 合并, 得到整颗子树的信息.

例如 Access 时, 对于路径上某点 x, 我们会改变他的偏爱子节点. 对于原来的偏爱子节点, 将其连接方式从实边变为了虚边, 所以要加入它的信息到 x 的虚子树中. 对于后来的偏爱子节点, 其连接方式从虚边变成了实边, 所以在 x 的虚子树中删去它的信息.

维护了虚子树信息, 然后在 Splay 中合并, 就可以在 Splay 的根上查询到 Splay 对应路径的端点的子树信息.

```
void Access(int x) {
    int p = 0;
    while (x) {
        Splay(x);
        int &t = ch[x][1];
        if (t) {
            sumi[x] += sum[t];//将 t 的信息加入 x 的虚子树
        }
        t = p;
        if (t) {
            sumi[x] -= sum[t];//将 t 的信息从 x 虚子树中删除
        }
        up(x);
        p = x;
        x = fa[x];
    }
}
void up(int x) {//合并函数长这样
    int l = ch[x][0], r = ch[x][1];
    sum[x] = sumi[x] + val[x];//虚子树权值和 加上 自己的权值
    if (l) sum[x] += sum[l];
    if (r) sum[x] += sum[r];
}
```

调试

调个毛线.....

检查树的形态

可以把所有点的父亲都输出来, 如果树的形态没错可以少检查很多东西.

如果有过 **makeRoot**, 可以倒序再 **makeRoot** 一遍, 保证相同的树输出的结果是相同的.

(注意向下查询时要不断 **pushDown**)

```
int findfa(int x) {
        Access(x);
        Splay(x);
        x = ch[x][0];
        while (ch[x][1]) {
                x = ch[x][1];
                down(x);
        }
        return x;
}
void print(int n) {
        per (i, 1, n) makeRoot(i);
        rep (i, 1, n) cerr << findfa(i) << " ";
        cerr << endl;
}
```

检查各种标记

pushDown 不能写太骚了, 否则哪里少了一个都不知道, 而且很难调试.

比如从上向下访问时, 除了访问根节点 (一直走 $ch[x][0]$) , 否则是需要 **pushDown** 下传翻转的标记的, 然而不下传也不容易挂, 所以就发现不了错误.

2018 年 1 月 8 日