动态树

Link-Cut Tree

Idea：对树进行剖分，不过每条实链用一个 Splay 按照原树中的**深度**顺序进行维护，Splay 与 Splay 之间用虚边连接，虚边连接的子节点的 `fa` 置为父节点，但父节点的 `son` 不置为子节点。

OPT：

- `access(x)`：将 LCT **的根**到 x 的路径设为一条实链（`access` 之后，由于该链最深点为 x，所以 x 是该链的 Splay 的最后一个元素）；
- `makeRoot(x)`：将 x 设置为**原树的根**；
- `findRoot(x)`：查找 x 所在**原树的根**；
- `link(x, y)`：连接 x 和 y，即将 x 所在的 LCT 与 y 所在的 LCT 连接起来；
- `cut(x, y)`：断开 x 和 y，即将 x 和 y 分到两个不同的 LCT 去；
- `queryPath/modifyPath(x, y)`：询问或更改 x 到 y 的路径上的信息；
- `queryNode/modifyNode(x)`：询问或更改点 x 的信息。

Complexity：$O(n \lg n)$

ATT：注意区分三种“根”：**原树**的根；LCT 的根（其实是没有虚边的那个 Splay 的根）；每个 Splay 的根。

Code：

```
struct LinkCutTree{
    int sta[N], staTop;
    struct Splay{
        int son[2], fa;
        int val, XOR; // information needed to be maintained
        bool rev;
    }tr[N];
    #define which(x, y) (tr[y].son[1] == x)
    inline void pushup(int x){
        if(x){
            tr[x].XOR = tr[x].val;
            if(tr[x].son[0])    tr[x].XOR ^= tr[tr[x].son[0]].XOR;
            if(tr[x].son[1])    tr[x].XOR ^= tr[tr[x].son[1]].XOR;
        }
    }
```

```
    inline void pushdown(int x){
        if(tr[x].rev){
            if(tr[x].son[0]){
                tr[tr[x].son[0]].rev ^= 1;
                swap(tr[tr[x].son[0]].son[0], tr[tr[x].son[0]].son[1]);
            }
            if(tr[x].son[1]){
                tr[tr[x].son[1]].rev ^= 1;
                swap(tr[tr[x].son[1]].son[0], tr[tr[x].son[1]].son[1]);
            }
            tr[x].rev ^= 1;
        }
    }
    inline bool isRoot(int x){ return tr[tr[x].fa].son[0] != x && tr[tr[x].fa].son[1] != x; }
    inline void rotate(int x, int dir){ // dir == 0: left; dir == 1: right
        int y = tr[x].fa, z = tr[y].fa, B = tr[x].son[dir];
        if(!isRoot(y))  tr[z].son[which(y,z)] = x;
        tr[x].son[dir] = y; tr[y].son[dir^1] = B;
        tr[x].fa = z; tr[y].fa = x; tr[B].fa = y;
        pushup(y); pushup(x);
    }
    inline void splay(int x){ // rotate x to the root of its splay tree
        sta[staTop = 1] = x;
        for(int i = x; !isRoot(i); i = tr[i].fa)    sta[++staTop] = tr[i].fa;
        while(staTop)   pushdown(sta[staTop--]); // pushdown the tag
        while(!isRoot(x)){
            int y = tr[x].fa, z = tr[y].fa, dir1 = which(x,y)^1, dir2 = which(y,z)^1;
            if(isRoot(y))   rotate(x, dir1);
            else{
                if(dir1 == dir2)    rotate(y, dir2);
                else    rotate(x, dir1);
                rotate(x, dir2);
            }
        }
    }
    inline void access(int x){ // connect x with the root of LCT
        for(int y = 0; x; y = x, x = tr[x].fa){
            splay(x); tr[x].son[1] = y; pushup(x);
        }
    }
```

```
    inline void makeRoot(int x){ // make x the root of original tree
        access(x); splay(x);
        tr[x].rev ^= 1; swap(tr[x].son[0], tr[x].son[1]);
//splay::reverse an interval
        pushup(x);
    }
    inline int findRoot(int x){ // find the root of original tree
        access(x); splay(x);
        while(tr[x].son[0]) x = tr[x].son[0];
        return x;
    }
    inline void link(int x, int y){
        makeRoot(x); access(y); splay(y);
        if(findRoot(y) != x)    tr[x].fa = y;
    }
    inline void cut(int x, int y){
        makeRoot(x); access(y); splay(y);
        if(tr[y].son[0] != x)   return; // not connected
        tr[y].son[0] = tr[x].fa = 0;
        pushup(y);
    }

    inline int queryXor(int x, int y){ // query a path
        makeRoot(x); access(y); splay(y);
        // the splay tree now contains and only contains all the
node on the path from x to y
        return tr[y].XOR;
    }
    inline void modify(int x, int val){ // modify a node
        splay(x);
        tr[x].val = val;
        pushup(x);
    }
}LCT;
```