

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:

```
1 struct LinkCutTree{
2     int sta[N], staTop;
3     struct Splay{
4         int son[2], fa;
5         int val, XOR; // information needed to be maintained
6         bool rev;
7     }tr[N];
8     #define which(x, y) (tr[y].son[1] == x)
9     inline void pushup(int x){
10         if(x){
11             tr[x].XOR = tr[x].val;
12             if(tr[x].son[0]) tr[x].XOR ^= tr[tr[x].son[0]].XOR;
13             if(tr[x].son[1]) tr[x].XOR ^= tr[tr[x].son[1]].XOR;
14         }
15     }
16     inline void pushdown(int x){
17         if(tr[x].rev){
18             if(tr[x].son[0]){
19                 tr[tr[x].son[0]].rev ^= 1;
20                 swap(tr[tr[x].son[0]].son[0], tr[tr[x].son[0]].son[1]);
21             }
22             if(tr[x].son[1]){
23                 tr[tr[x].son[1]].rev ^= 1;
24                 swap(tr[tr[x].son[1]].son[0], tr[tr[x].son[1]].son[1]);
25             }
26             tr[x].rev ^= 1;
27         }
28     }
29     inline bool isRoot(int x){ return tr[tr[x].fa].son[0] != x && tr[tr[x].fa].son[1] != x; }
30     inline void rotate(int x, int dir){ // dir == 0: left; dir == 1: right
31         int y = tr[x].fa, z = tr[y].fa, B = tr[x].son[dir];
32         if(!isRoot(y)) tr[z].son[which(y,z)] = x;
33         tr[x].son[dir] = y; tr[y].son[dir^1] = B;
34         tr[x].fa = z; tr[y].fa = x; tr[B].fa = y;
35         pushup(y); pushup(x);
36     }
```

```

37     inline void splay(int x){ // rotate x to the root of its splay tree
38         sta[staTop = 1] = x;
39         for(int i = x; !isRoot(i); i = tr[i].fa)    sta[++staTop] = tr[i].fa;
40         while(staTop)    pushdown(sta[staTop--]); // pushdown the tag
41         while(!isRoot(x)){
42             int y = tr[x].fa, z = tr[y].fa, dir1 = which(x,y)^1, dir2 = which(y,z)^1;
43             if(isRoot(y))    rotate(x, dir1);
44             else{
45                 if(dir1 == dir2)    rotate(y, dir2);
46                 else    rotate(x, dir1);
47                 rotate(x, dir2);
48             }
49         }
50     }
51     inline void access(int x){ // connect x with the root of LCT
52         for(int y = 0; x; y = x, x = tr[x].fa){
53             splay(x); tr[x].son[1] = y; pushup(x);
54         }
55     }
56     inline void makeRoot(int x){ // make x the root of original tree
57         access(x); splay(x);
58         tr[x].rev ^= 1; swap(tr[x].son[0], tr[x].son[1]); //splay::reverse an interval
59         pushup(x);
60     }
61     inline int findRoot(int x){ // find the root of original tree
62         access(x); splay(x);
63         while(tr[x].son[0]) x = tr[x].son[0];
64         return x;
65     }
66     inline void link(int x, int y){
67         makeRoot(x); access(y); splay(y);
68         if(findRoot(y) != x)    tr[x].fa = y;
69     }
70     inline void cut(int x, int y){
71         makeRoot(x); access(y); splay(y);
72         if(tr[y].son[0] != x)    return; // not connected
73         tr[y].son[0] = tr[x].fa = 0;
74         pushup(y);
75     }
76
77     inline int queryXor(int x, int y){ // query a path
78         makeRoot(x); access(y); splay(y);
79         // the splay tree now contains and only contains all the node on the path from x to y
80         return tr[y].XOR;
81     }
82     inline void modify(int x, int val){ // modify a node
83         splay(x);
84         tr[x].val = val;
85         pushup(x);
86     }
87 }LCT;

```