动态树 LCT （Link-Cut-Tree）

【学习建议】

1. 理清思路
2. 背模版（反正比英文课文好背 lol）
3. 题目列表：https://www.fzoi.top/exercise/853

什么是动态树?

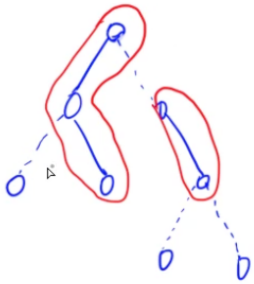
动态维护一个森林，动态的支持2个操作——添加边、删除边的数据结构。

> 它还可以维护树上路径的一些信息。
>
> 时间复杂度单独操作：logn
>
> 虽然树连剖分的复杂度比动态树大，但是常数比动态树小。

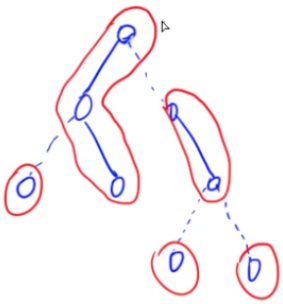
回忆一下，树连剖分是维护一些重链来维护一些树，动态树也是类似的。

树链剖分可以将边分为重边和虚边，动态树可以人为的去维护实边和虚边。具体这样表述

任意一个点最多只有1个实边。我们可以定义一条与实边对应的路径。如下图所示



孤立点我们也单独当作实边处理



Spaly维护的所有实边路径。

红圈圈出来的就是一个splay

Q： 具体怎么维护?

A： splay中序遍历就是这个路径从上到下的遍历

> splay维护的是当前联通的所有实边的极大路径。
>
> splay通过其中的后继与前驱关系，来维护原树中的父子关系。

Q： 树跟树之间的关系如何维护?

A：splay中，root的fa信息还没利用上，所以用 Splay中root的结点来维护

> 提示，x的父节点不一定是y，因为x不一定是spaly中的root

> 举个例子，假设x的发r是整个splay的root，那么tr[r].p = y

>

Q：如何判断实边虚边?

因为每个结点最多只有一个实边，如果是虚边，在链接splay的时候只有只有这个splay的root的fa（点A）指向了某个点（点B），但是这个点B的儿子却不是点A。

虚边：子结点知道父结点是谁，但是父结点不知道子结点是谁。

实边：父子之间相互知道。

> 基于这种情况，我们只需要修改一次父子关系，我们可以很轻松的完成实边和虚边的转换
>
> 即：只需要确定父亲的儿子是谁就行。
>
> 具体如何请见下面的基本操作：

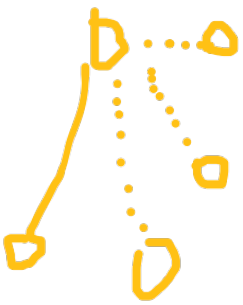
LCT基本操作

一共有7个操作，建议大家理解中文（或者看了几遍不知道这个竟然是中文的情况下）结合代码进行理解。

换边操作

将结点fa的后继改为想要变成实边的那个点x。

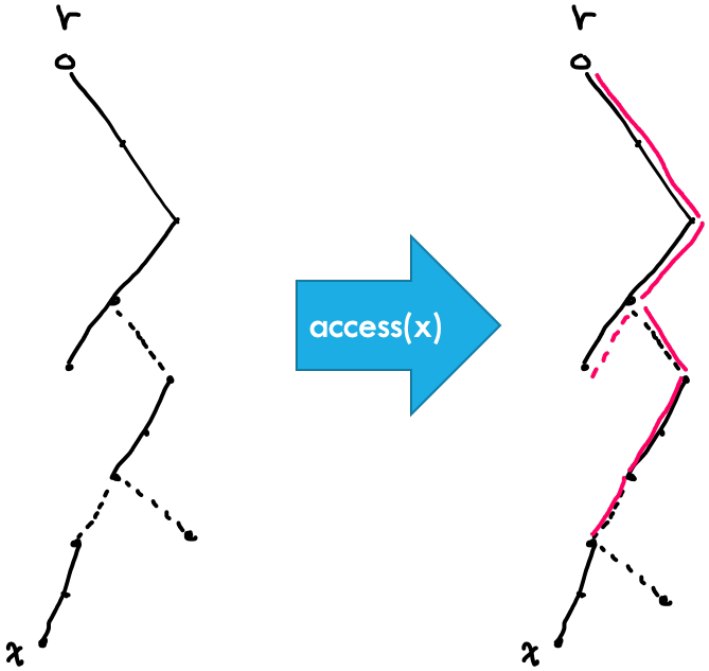
因为虚边的父亲信息已经存了。其余结点不需要任何操作



注意，得把fa转到根节点，此时fa的右子树是空的（因为fa的序号最大）。然后把x点接到fa的右子树即可。

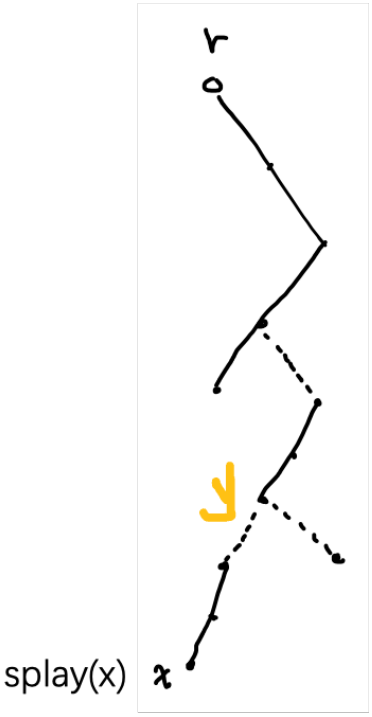
access(x)

核心操作：将root到x的路径全部变成实边。(建立一条root到x的实边路径) 注意这个实边路径只能包含root到x的实边路径。



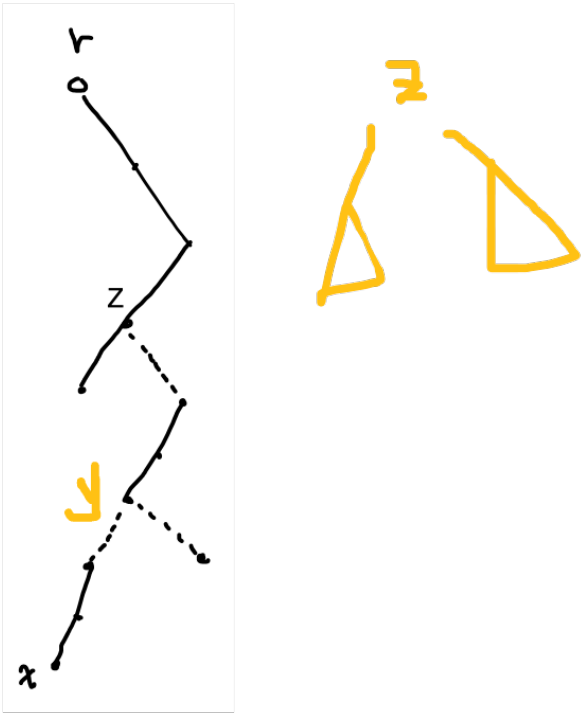
这个过程是从下往上做的。

首先需要将x旋到当前实边的root上，接下来希望将x与y的连边边成实边，因为y是最大的点，那么直接将y旋到root上，此时y没有右子树，直接将x接到y的后继节点上。那么直接将x插到y的右节点即可。



此时，我们以，以y为root的splay维护了整个路径。

同样的道理对于z来讲，先把z旋转到root，此时z有左右子树（因为原树的有子树），解决方案很简单，基于之前的换边理论，我们直接将y挂到z的右子树上。此时z就是这个splay的root。z代表了整个splay。



总结一下：节点直接旋上去。挂到右子树。递归做即可。

make_root(x)

将x变为根节点。

利用第一个操作access(x) 建立一条root到x的实边路径。

对于一个无根树而言，我们将路径翻转是不会影响这个树的拓扑结构的。所以直接将x转到root，然后将整个路径翻转即可。

> 拓扑结构不变：翻转前后 任意两点x到y经过的路径点不会发生变化。

路径翻转是splay的一个经典操作，直接将某个区间翻转即可。利用lazytag实现。

> 提示：此处2个关系别搞混了，一个是从原树的父子关系，一个是splay的父子关系。必须要区分。
>
> 一个疑惑：区间翻转后为什么不会破坏其他边的偏序关系。
>
> 因为父子关系不会发生改变。

find_root(x)

找到x所在的树的根节点。

1. 建立x到root的实边路径（调用access）
2. 将x旋转到root
3. 整个路径深度最小的点就是根节点，只需要找到整个树的最左的节点即可。

进阶操作：判断x和y是否在同一个splay之中。做两遍find_root即可。

split(x,y)

将从x到y的路径变为一条实边路径。

首先通过makeroot将x变为root，再access(y) 这样就实现了split

link(x,y)

如果x y不连通，则加xy这一条边。

具体操作：

1. 判断连通，make_root(x) 将x变为root，再去find_root(y)是否为x，如果find_root(y)!=x，则说明他们不连通。
2. 因为在判断连通时，x已经是root了，所以若需要加边，只需要将x的fa记为y即可。

cut(x,y)

若x y有边，则删掉该边。

操作：

1. 将x变为root
2. find_root(y) 找到y所在的根节点。
3. 考虑第二个操作的副作用：当find_root(y)后，除了找到y的根节点外的同时，会将y所在树的root旋转到y所在实边splay的root上（在find_root后，会从左一直走找到root，然后splay操作为了保证时间复杂度，最后还会splay一次把root旋上去。）y所在实边splay的根节点就应该是整个树的根节点，也就是x。
4. 如果x y边，则意味着y应是x的后继。所以只需要判断y是否为x的后继即可（y是否为x的后继的判断标准：y是不是x的右子树，同时y的左子树是否为空。

这样就ok了

isroot(x)

判断x是否为所在splay的root（注意不是原树的root）

如果x不是root，则x必然存在父节点。则x必然是其fa的左儿子或右儿子。

因此，若x既不是其fa的左儿子，也不是其fa的右儿子。那么x一定是splay的root。

信息维护

信息维护到splay中。具体见代码的操作。

翻转操作；维护rev表示是否翻转。

求xor和：求x到y所有点权的xor和，再额外维护一个sum即可。具体来说，直接用split函数直接将x y建立成一条实边路径。然后会有一个splay对应维护

维护信息时本质就是一个路径上信息，直接用一个一维splay维护即可。

点权修改：x splay到root，再改x的值，再pushup pushdown即可。

LCT模版代码

```
//供参考
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 100010;

int n, m;
struct Node
{
    int s[2], p, v;
    int sum, rev;
}tr[N];
int stk[N];

void pushrev(int x)
{
    swap(tr[x].s[0], tr[x].s[1]);
    tr[x].rev ^= 1;
}

void pushup(int x)
{
    tr[x].sum = tr[tr[x].s[0]].sum ^ tr[x].v ^ tr[tr[x].s[1]].sum;
}

void pushdown(int x)
{
    if (tr[x].rev)
    {
        pushrev(tr[x].s[0]), pushrev(tr[x].s[1]);
        tr[x].rev = 0;
    }
```

```
}

bool isroot(int x)
{
    return tr[tr[x].p].s[0] != x && tr[tr[x].p].s[1] != x;
}

void rotate(int x)
{
    int y = tr[x].p, z = tr[y].p;
    int k = tr[y].s[1] == x;
    if (!isroot(y)) tr[z].s[tr[z].s[1] == y] = x;
    tr[x].p = z;
    tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
    tr[x].s[k ^ 1] = y, tr[y].p = x;
    pushup(y), pushup(x);
}

void splay(int x)
{
    int top = 0, r = x;
    stk[ ++ top] = r;
    while (!isroot(r)) stk[ ++ top] = r = tr[r].p;
    while (top) pushdown(stk[top -- ]);
    while (!isroot(x))
    {
        int y = tr[x].p, z = tr[y].p;
        if (!isroot(y))
            if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
            else rotate(y);
        rotate(x);
    }
}

void access(int x)  // 建立一条从根到x的路径，同时将x变成splay的根节点
{
    int z = x;
    for (int y = 0; x; y = x, x = tr[x].p)
    {
        splay(x);
        tr[x].s[1] = y, pushup(x);
    }
    splay(z);
```

```cpp
}

void makeroot(int x)  // 将x变成原树的根节点
{
    access(x);
    pushrev(x);
}

int findroot(int x)  // 找到x所在原树的根节点，再将原树的根节点旋转到splay的根节点
{
    access(x);
    while (tr[x].s[0]) pushdown(x), x = tr[x].s[0];
    splay(x);
    return x;
}

void split(int x, int y)  // 给x和y之间的路径建立一个splay，其根节点是y
{
    makeroot(x);
    access(y);
}

void link(int x, int y)  // 如果x和y不连通，则加入一条x和y之间的边
{
    makeroot(x);
    if (findroot(y) != x) tr[x].p = y;
}

void cut(int x, int y)  // 如果x和y之间存在边，则删除该边
{
    makeroot(x);
    if (findroot(y) == x && tr[y].p == x && !tr[y].s[0])
    {
        tr[x].s[1] = tr[y].p = 0;
        pushup(x);
    }
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &tr[i].v);
```

```
121     while (m -- )
122     {
123         int t, x, y;
124         scanf("%d%d%d", &t, &x, &y);
125         if (t == 0)
126         {
127             split(x, y);
128             printf("%d\n", tr[y].sum);
129         }
130         else if (t == 1) link(x, y);
131         else if (t == 2) cut(x, y);
132         else
133         {
134             splay(x);
135             tr[x].v = y;
136             pushup(x);
137         }
138     }
139 
140     return 0;
141 }
142 
```