可持久化并查集

n 个集合 m 个操作
操作：
1 a b 合并 a,b 所在集合
2 k 回到第 k 次操作之后的状态(查询算作操作)
3 a b 询问 a,b 是否属于同一集合，是则输出 1 否则输出 0
输入
```
5 6
1 1 2
3 1 2
2 0
3 1 2
2 1
3 1 2
```
输出
```
1 0 1
```

首先明确一点，本题考得不是并查集，而是可持久化 跟并查集没啥关系。
而且在这道题中**用不带路径压缩的并查集**（**欢迎推翻这个 flag**）
然后我们看到'历史版本'，自然而然想到可持久化数据结构--主席树。
那我们用主席树干什么呢？
维护每一个版本中每一个点的**父亲**，也就是我们熟悉的并查集中的 fa[x]
再然后，因为我们用**用不带路径压缩的并查集**
所以对于每一次合并只会改一个点的父亲
所以一个版本的相对于上一个版本只会改一个点
所以有很多地方可以共用
这也证明了为什么要用可持久化数据结构 （可持久化数据结构就是共用一些点来达到节省空间的效果）
明白了主席树的作用之后，并查集中 find_fa 的思路大体上就出来了：
1.在主席树上，查询某一个版本中一个点的父亲
2.它成为它的父亲
3.重复步骤 1,直到找到 root
但是我们发现，如果并查集退化成一条链，find_fa 复杂度会很高（虽然这题很水，暴力都可以过去）
又不能路径压缩（不然就不能一次修改一个点，就不好搞可持久化了）（**欢迎推翻这个 flag**）
于是我们可以在 Union 上下一点功夫使得它不会变成一条链
想到合并方法--启发式合并
怎么启发？
把**最大深度小**的**往最大深度大**的上并
于是**最大深度大**的**深度不会增加**
而是**最大深度小**的**增加深度** 这样不久巧妙地保证了深度均衡吗？
好了，基本上是讲完了 实在不懂可以看着代码理解 代码在下面，有注释

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <vector>
#include <queue>
#define rg register int
#define ll long long
#define RG register
#define il inline
using namespace std;

il int gi()
{
    rg x=0,o=0;RG char ch=getchar();
    while(ch!='-'&&(ch<'0'||'9'<ch)) ch=getchar();
    if(ch=='-') o=1,ch=getchar();
    while('0'<=ch&&ch<='9') x=(x<<1)+(x<<3)+ch-'0',ch=getchar();
    return o?-x:x;
}

#define SZ 200010
int n,m,fa[SZ*30],deep[SZ*30];
// deep 存最大深度
// fa 存 一个点在某个版本的父亲

struct Tree {
  int l,r;
}tr[SZ*30];
#define lson tr[rt].l
#define rson tr[rt].r
int Ed[SZ],tot;
// Ed 是版本号 ， tot 是节点总数（这些就是主席树啦）

void build(int &rt,rg l,rg r)
{
    rt=++tot;
    if(l==r)
    {
        fa[rt]=l;  // 初始版本 ： 父亲是自己
        // 就像并查集初始化每个点的父亲是它自己
        return;
    }
    rg mid=l+r>>1;
    build(lson,l,mid);
    build(rson,mid+1,r);
}

// 主席树维护的是 每一个版本 每一个点的父亲是谁
void update(int &rt,rg last,rg l,rg r,rg pos,rg ff) //把 pos 的父亲改成 ff
```

```
{
    rt=++tot;
    if(l==r)
    {
        fa[rt]=ff;
        deep[rt]=deep[last];  // deep 用于 启发式合并
        return;
    }
    lson=tr[last].l;rson=tr[last].r;
    rg mid=l+r>>1;
    if(pos<=mid) update(lson,tr[last].l,l,mid,pos,ff);
    else update(rson,tr[last].r,mid+1,r,pos,ff);
}

int query(rg rt,rg l,rg r,rg pos) // 询问某一个版本的一个点的父亲
{
    if(l==r) return rt;
    rg mid=l+r>>1;
    if(pos<=mid) return query(lson,l,mid,pos);
    else return query(rson,mid+1,r,pos);
}

void add(rg rt,rg l,rg r,rg pos) // 把某一个并查集联通块中每一个点的深度加一
{
    if(l==r)
    {
        ++deep[rt];
        return;
    }
    rg mid=l+r>>1;
    if(pos<=mid) add(lson,l,mid,pos);
    else add(rson,mid+1,r,pos);
}

int find_fa(rg ed,rg x) // ed 版本编号
{
    rg ff=query(ed,1,n,x);  // 查询在这一版本里 一个点的父亲
    if(x==fa[ff]) return ff;
    return find_fa(ed,fa[ff]);  // 不带路径压缩的并查集
}

int main()
{
    n=gi(),m=gi();
    build(Ed[0],1,n);
//init
    for(rg opt,k,a,b,i=1;i<=m;++i)
    {
        opt=gi();
        if(opt==1)
        {
            Ed[i]=Ed[i-1];
            a=gi(),b=gi();
            rg f1=find_fa(Ed[i],a);
            rg f2=find_fa(Ed[i],b);
            if(fa[f1]==fa[f2]) continue;
            if(deep[f1]>deep[f2]) swap(f1,f2);
            // 把大的往小的 并，保证 f1 儿子节点数一定是小于等于 f2
            update(Ed[i],Ed[i-1],1,n,fa[f1],fa[f2]);   // 把 f1
            if(deep[f1]==deep[f2]) add(Ed[i],1,n,fa[f2]);
          // 因为 f2 并到了 f1 所以 f1 的深度要加 1
          //我们用 启发式合并 来保证 病查集合并的复杂度
        }
        if(opt==2)
        {
            k=gi();
            Ed[i]=Ed[k];
        }
        if(opt==3)
        {
            Ed[i]=Ed[i-1];
            a=gi(),b=gi();
            rg f1=find_fa(Ed[i],a);
            rg f2=find_fa(Ed[i],b);
            if(fa[f1]==fa[f2]) puts("1");
            else puts("0");
        }
    }
  return 0;
}
```