基础

思路

并查集需要高效的处理集合的合并和查询。用线性的数据结构比如数组或者链表存储会导致时间复杂度很高，所以用图或树来存储。

考虑把所有的元素都存储在一棵树里面，合并集合只用把两棵树合并在一起，接下来考虑如何查询。

树上每个元素都可以一直向上遍历直到根节点，所以定义一个集合的代表元是这个集合所在的树的根节点，查询两个数是否在一个集合的时候直接比较代表元即可，因为只涉及到找父亲节点的操作，只用记录每个节点的直接父亲。

封装形式：

```
#define MAXN 100000
class node{
    public:
        void Union(int x,int y){
            int fx=getfa(x);
            int fy=getfa(y);
            fa[fx]=fy;
            return ;
        }
        bool query(int x,int y){
            return getfa(x)==getfa(y);
        }
        node(int n){
            for(int i=1;i<=n;i++){
                fa[i]=i;
            }
        }
    private:
        int getfa(int x){
            return x==fa[x]?x:getfa(fa[x]);
        }
        int fa[MAXN+5];
};
```

上述算法时间复杂度依赖于树高，最坏时间复杂度可能达到 $\mathcal{O}(n)$，所以要优化。

优化

启发式合并

其实就是把较小的集合的子树合并到较大的集合的子树中。

路径压缩

每次求父亲回溯的时候，把他的直接父亲赋值为代表元，减小树高。

优化后封装代码：

```
#define MAXN 100000
class node{
    public:
        void Union(int x,int y){
            int fx=getfa(x);
            int fy=getfa(y);
            if(siz[fx]>siz[fy]){
                std::swap(fx,fy);
            }
            fa[fx]=fy;
            return ;
        }
        bool query(int x,int y){
            return getfa(x)==getfa(y);
        }
        node(int n){
            for(int i=1;i<=n;i++){
                fa[i]=i;
                siz[i]=1;
            }
        }
    private:
        int getfa(int x){
            return x==fa[x]?x:fa[x]=getfa(fa[x]);
        }
        int fa[MAXN+5];
        int siz[MAXN+5];
};
```

扩展

敌对并查集

并查集所维护的是朋友的朋友是朋友的一种传递关系。

也可以维护敌人的敌人是朋友的关系。

对于每个要维护的值 n，假设两种元素 $A(n)$ ，$B(n)$， $A(n)$用来表示元素本身， $B(n)$则用来表示这个元素的敌人。

n 和 m 是敌人的话，直接合并 $A(n)$和 $B(m)$ 以及 $A(m)$和 $B(n)$。

n 和 m 是朋友的话，直接合并 $A(n)$和 $A(m)$ 以及 $B(m)$和 $B(n)$。

同样的，也可以维护有三种阵营的情况，每个值三种元素即可，详细略。

树边维护信息

并查集的结构是森林，所以树边可以带权值，路径压缩的时候直接根据定义合并边就行了。