并查集

基础

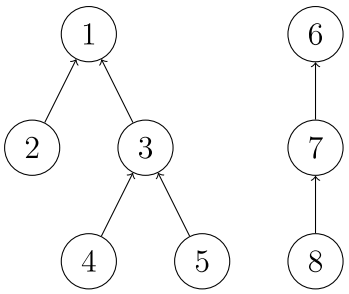
因为一个元素只可能属于一个集合，所以我们可以为每一个集合选取一个代表元。于是查询两个元素是否属于同一个集合实际上就是询问两个元素所在集合的代表元是否相同。这个询问的时间复杂度可以利用数组标记优化为$O(1)$

但是合并两个集合时需要改变其中一个集合中所有元素的代表元，时间复杂度仍然非常高，如何优化呢?

注意到，合并操作的时间复杂度远高于查询操作的复杂度，这启发我们通过一定的方式，提高查询操作的复杂度，降低合并操作的复杂度。

我们并不需要 $O(1)$的 知道每个元素所属集合的代表元，这启发我们用森林来维护代表元。用森林中的一棵树代表一个集合，树根为对应集合的代表元。

这样，对于每棵树上的元素，查询其代表元时，时间复杂度与树的高度成正比。这就是 并查集 。



并查集 首先是一种树形的数据结构，顾名思义， 并 就是 合并 ， 查 就是 查询 ，所以并查集一共支持两种操作.

- 查找(find)： 确定某个元素处于哪个子集。
- 合并(merge)： 将两个子集合并成一个集合。

定义以及初始化

定义:

```
int fa[N];       // fa[i]: i 所属的集合 (i 的祖先节点)
```

初始化：

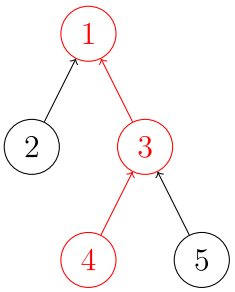
```
void init(int n) {
  for (int i = 1; i <= n; i++) fa[i] = i;  // i就在它本身的集合里
  return;
}
```

查找

通俗地讲一个故事：几个家族进行宴会，但是家族普遍长寿，所以人数众多。由于长时间的分离以及年龄的增长，这些人逐渐忘掉了自己的亲人，只记得自己的爸爸是谁了，而最长者（称为「祖先」）的父亲已经去世，他只知道自己是祖先。为了确定自己是哪个家族，他们想出了一个办法，只要问自己的爸爸是不是祖先，一层一层的向上问，直到问到祖先。如果要判断两人是否在同一家族，只要看两人的祖先是不是同一人就可以了。

```
int find(int x){
        if(fa[x] == x){ // 如果他的祖先是自己，那就直接返回
                return x;
        }
        return find(fa[x]);//否则就看看自己父亲的祖先是谁
}
```

显然这样最终会返回 x 的祖先。并且上述查询操作的代码的时间复杂度取决于每个集合对应的树的高度。



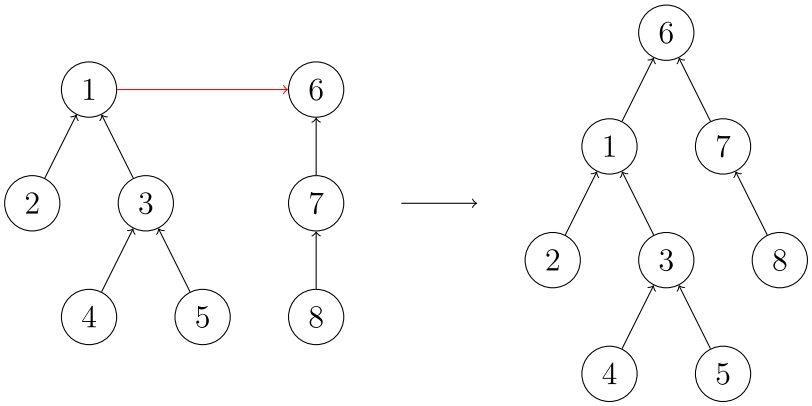
合并

宴会上，一个家族的祖先突然对另一个家族说：我们两个家族交情这么好，不如合成一家好了。另一个家族也欣然接受了。

我们之前说过，并不在意祖先究竟是谁，所以只要其中一个祖先变成另一个祖先的儿子就可以了。

```
void merge(int x, int y){
        int fx = find(x);
        int fy = find(y);
        if(fx != fy){ //只有他们祖先不是同一个才能合并，是同一个那肯定不需要任何操作
                fa[fx] = fy;
        }
}
```

可以看出，合并操作的时间复杂度取决于查找操作的时间复杂度，也就是每个集合对应的树的高度。



优化：

不幸的是，上面代码看似优秀，但实际上，在最坏情况下时间复杂度仍然很高——因为森林中树的深度可能比较大。如果我们不停地从一个深度比较大的点向上寻找代表元，时间复杂度就令人难以接受。

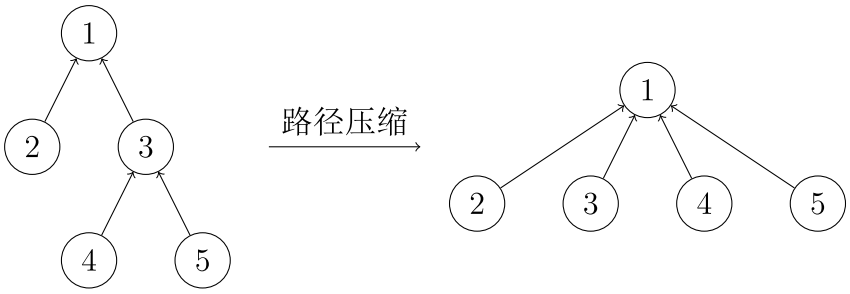
下面我们来讲讲上述操作的优化操作。

查找优化——路径压缩

这样的确可以达成目的，但是显然效率实在太低。为什么呢？因为我们使用了太多没用的信息，我的祖先是谁与我父亲是谁没什么关系，这样一层一层找太浪费时间，不如我直接当祖先的儿子，问一次就可以出结果了。甚至祖先是谁都无所谓，只要这个人可以代表我们家族就能得到想要的效果。

把在路径上的每个节点都直接连接到根上 ， 这就是 路径压缩 。

```
int find(int x){
        if(fa[x] == x){
                return x;
        }
        return fa[x] = find(fa[x]); //把自己的父亲指向父亲的父亲....一直找下去就会直接指向祖先。
}
```



合并优化——按秩合并

按秩合并有两种，任选其一就行，一个是将孩子 少 的合并合并到孩子 多 的里面，另外一个是将深度 小 的合并到深度 大 的中，两者任选其一，由于我比较喜欢存孩子数量，所以我选择前者。所以这里主要介绍前者。

一个祖先突然抖了个机灵：「你们家族人比较少，搬家到我们家族里比较方便，我们要是搬过去的话太费事了。」

由于需要我们支持的只有集合的合并、查询操作，当我们需要将两个集合合二为一时，无论将哪一个集合连接到另一个集合的下面，都能得到正确的结果。但不同的连接方法存在时间复杂度的差异。具体来说，如果我们将一棵点数与深度都较小的集合树连接到一棵更大的集合树下，显然相比于另一种连接方案，接下来执行查找操作的用时更小（也会带来更优的最坏时间复杂度）。

当然，我们不总能遇到恰好如上所述的集合————点数与深度都更小。鉴于点数与深度这两个特征都很容易维护，我们常常从中择一，作为估价函数。而无论选择哪一个，时间复杂度都为 $O(m\alpha(m,n))$

由于路径压缩单次合并可能造成大量修改，有时路径压缩并不适合使用。例如，在可持久化并查集、线段树分治 + 并查集中，一般使用只启发式合并的并查集。

其中 `sz` 表示第 `i` 个节点有多少个子节点。

```
void merge(int x, int y){
        int fx = find(x);
        int fy = find(y);
    if(fx == fy){
                return ;
        }
        if(sz[fx] > sz[fy]){ //将小的合并到大的里
                swap(fx, fy);
        }
        fa[fx] = fy;
        sz[fy] += sz[fx];
}
```