并查集

口口木木
吃糯米的冰糖葫芦

网上关于并查集的介绍并不少，但是知识点都比较零碎，而且最近做题时发现很多朋友对并查集的知识不是很熟悉，有些细节的地方经常出错。于是整理了这篇文章，记录一下并查集的相关基础知识点以及需要注意的细节。

介绍

并查集被很多Oier认为是最简洁而优雅的数据结构之一，主要用于解决一些元素分组的问题。它管理一系列不相交的集合，并支持两种操作：

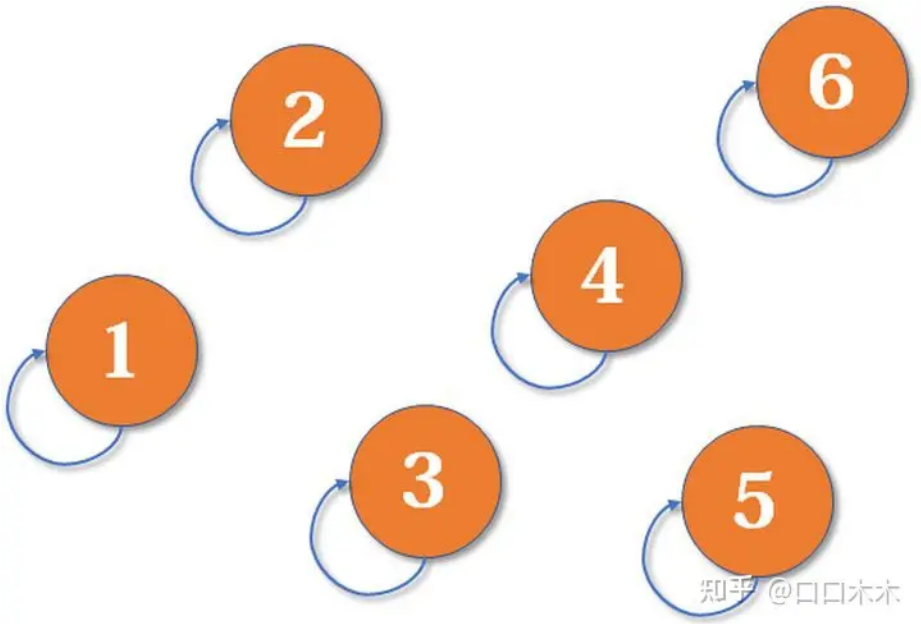
合并（Union）：把两个不相交的集合合并为一个集合。

查询（Find）：查询两个元素是否在同一个集合中。

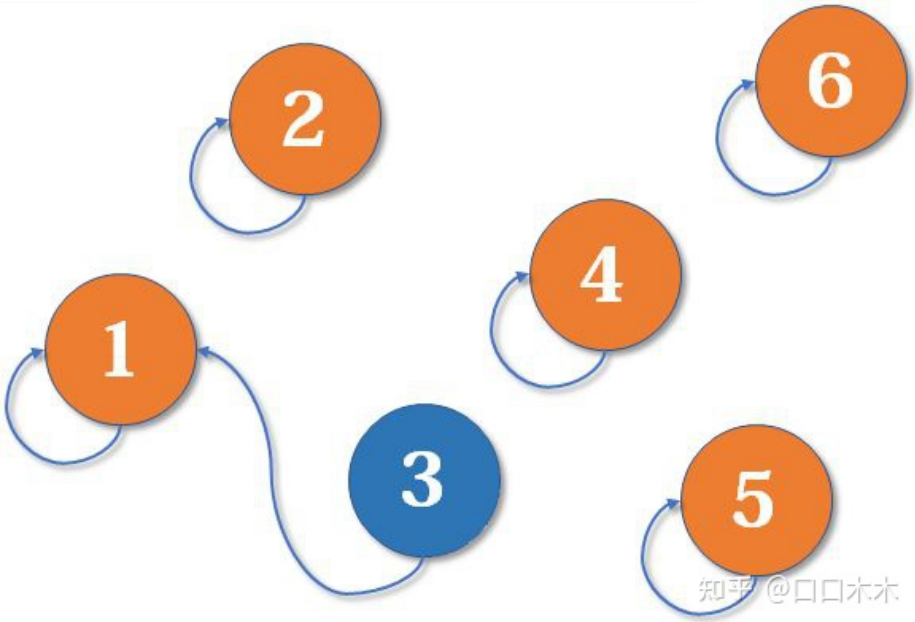
并查集的引入

并查集的重要思想在于，用集合中的一个元素代表集合。我曾看过一个有趣的比喻，把集合比喻成**帮派**，而代表元素则是**帮主**。接下来我们利用这个比喻，看看并查集是如何运作的。

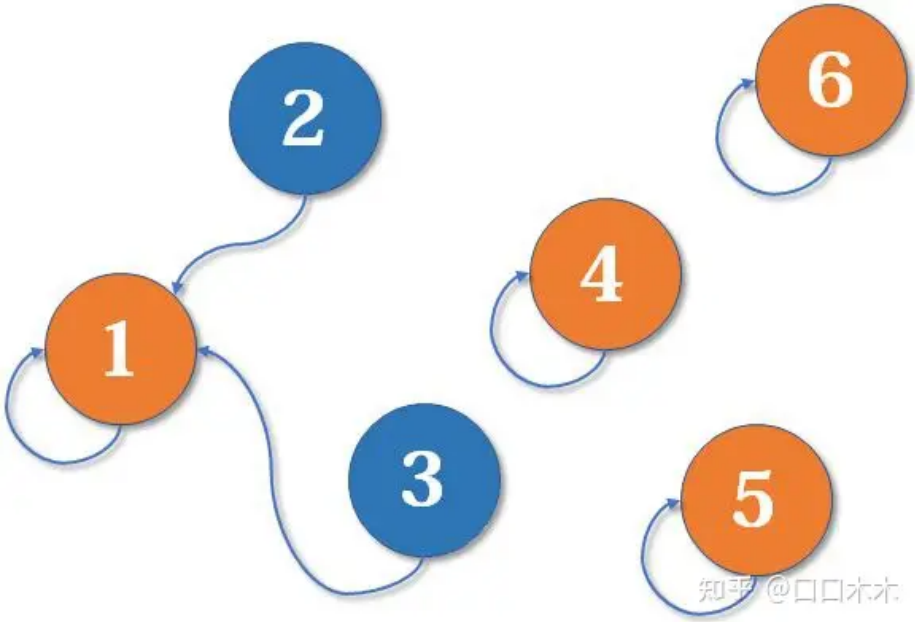
最开始，所有大侠各自为战。他们各自的帮主自然就是自己。（对于只有一个元素的集合，代表元素自然是唯一的那个元素）



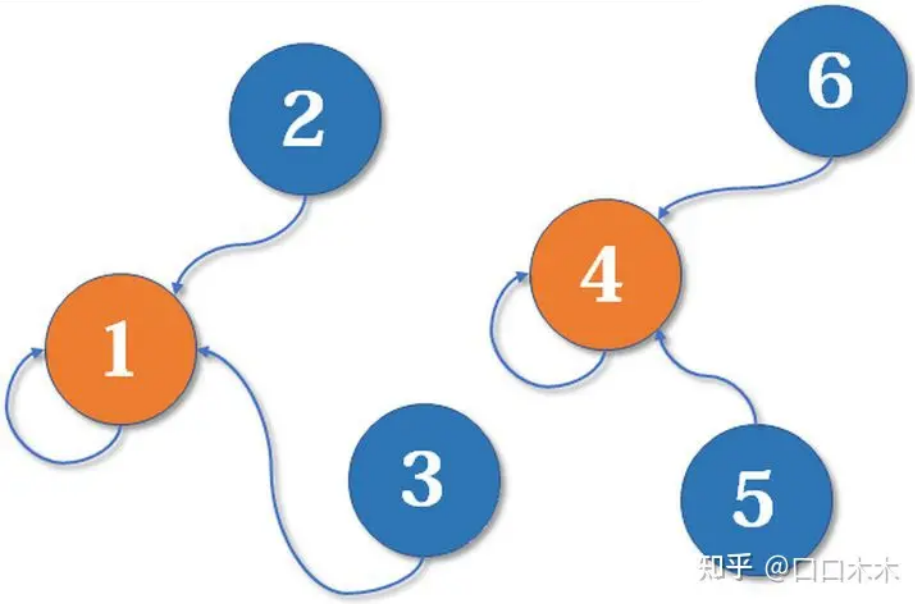
现在1号和3号比武，假设1号赢了（这里具体谁赢暂时不重要），那么3号就认1号作帮主（合并1号和3号所在的集合，1号为代表元素）。



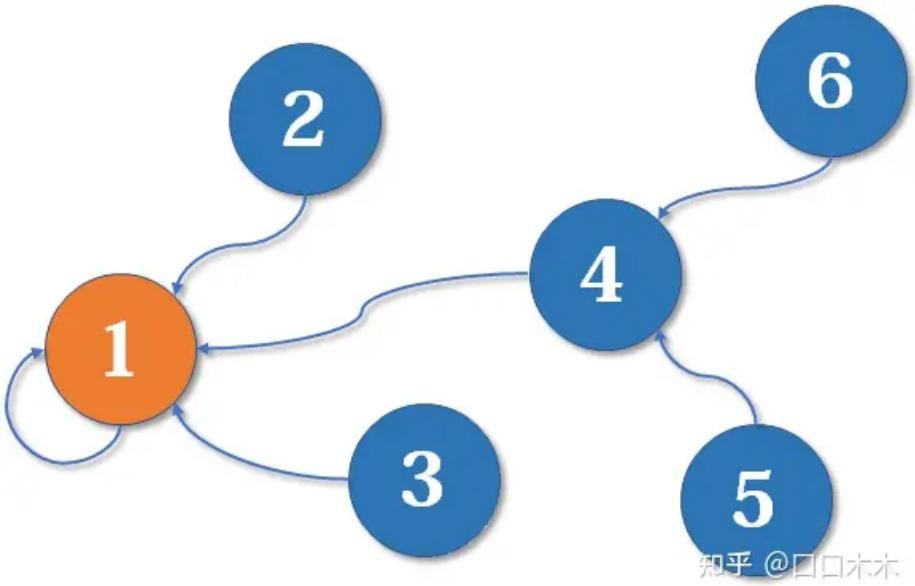
现在2号想和3号比武（合并3号和2号所在的集合），但3号表示，别跟我打，让我帮主来收拾你（合并代表元素）。不妨设这次又是1号赢了，那么2号也认1号做帮主。



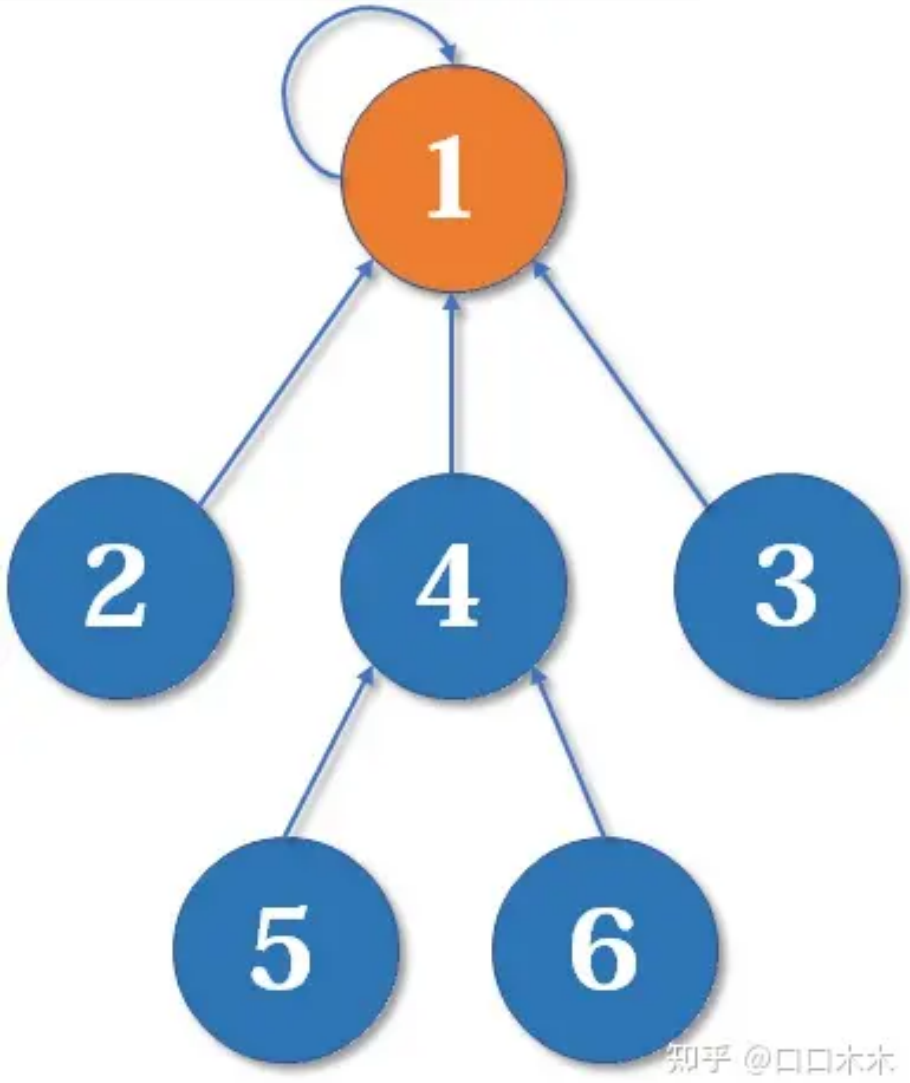
现在我们假设4、5、6号也进行了一番帮派合并，江湖局势变成下面这样：



现在假设2号想与6号比，跟刚刚说的一样，喊帮主1号和4号出来打一架（帮主真辛苦啊）。1号胜利后，4号认1号为帮主，当然他的手下也都是跟着投降了。



好了，比喻结束了。如果你有一点图论基础，相信你已经觉察到，这是一个树状的结构，要寻找集合的代表元素，只需要一层一层往上访问父节点（图中箭头所指的圆），直达树的根节点（图中橙色的圆）即可。根节点的父节点是它自己。我们可以直接把它画成一棵树：



用这种方法，我们可以写出最简单版本的并查集代码。

初始化

```
int[] fa = new int[n];
for (int i = 1; i <= n; ++i) {
    fa[i] = i;
}
```

假如有编号为1, 2, 3, ..., n的n个元素，我们用一个数组fa[]来存储每个元素的父节点（因为每个元素有且只有一个父节点，所以这是可行的）。一开始，我们先将它们的父节点设为自己。

查询

```
public int find(int x) {
    if (fa[x] == x) {
        return x;
    } else {
        return find(fa[x]);
    }
}
```

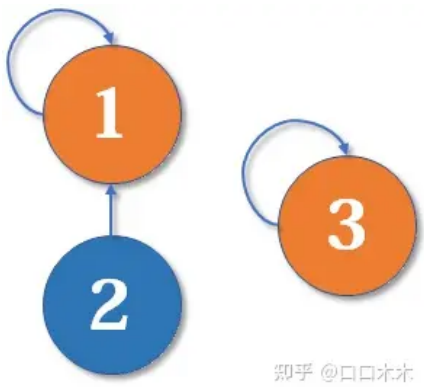
我们用递归的写法实现对代表元素的查询：一层一层访问父节点，直至根节点（根节点的标志就是父节点是本身）。要判断两个元素是否属于同一个集合，只需要看它们的根节点是否相同即可。

```
public void merge(int i, int j) {
    fa[find(i)] = find(j); // 修改的是根节点
}
```

合并操作也是很简单的，先找到两个集合的代表元素，然后将前者的父节点设为后者即可。当然也可以将后者的父节点设为前者，这里暂时不重要。本文末尾会给出一个更合理的比较方法。

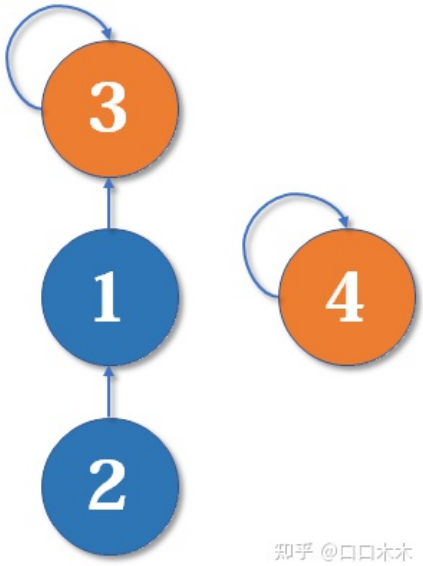
路径压缩

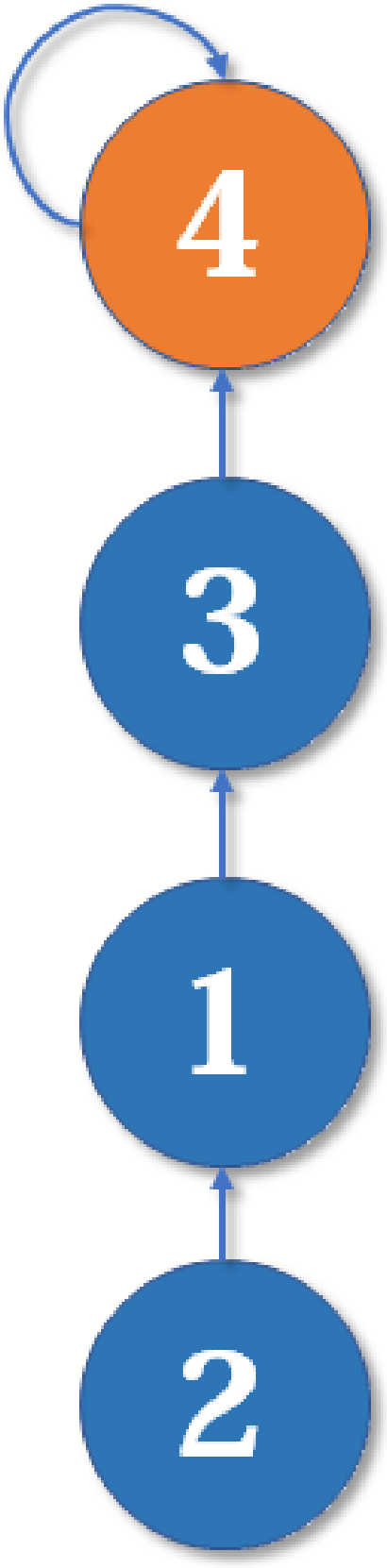
最简单的并查集效率是比较低的。例如，来看下面这个场景：假如现在2和1已经在一个集合内，3是单独一个集合：



现在我们要merge(2,3)，于是从2找到1，fa[1]=3，于是变成了这样：

然后我们又找来一个元素4，并需要执行merge(2,4)：





大家应该有感觉了，这样可能会形成一条长长的 **链**，随着链越来越长，我们想要从底部找到根节点会变得越来越难。

怎么解决呢？我们可以使用**路径压缩**的方法。既然我们只关心一个元素对应的**根节点**，那我们希望每个元素到根节点的路径尽可能短，最好只需要一步，像这样：

其实这说来也很好实现。只要我们在**查询**的过程中，把**沿途的每个节点的父节点都设为根节点**即可。下一次再查询时，我们就可以省很多事。这用递归的写法很容易实现：

查询（路径压缩）

```
public int find(int x) {
    if(x == fa[x]) [
        return x;
    } else {
        fa[x] = find(fa[x]);  //父节点设为根节点
        return fa[x];         //返回父节点
    }
}
```

以上代码常常简写为一行：

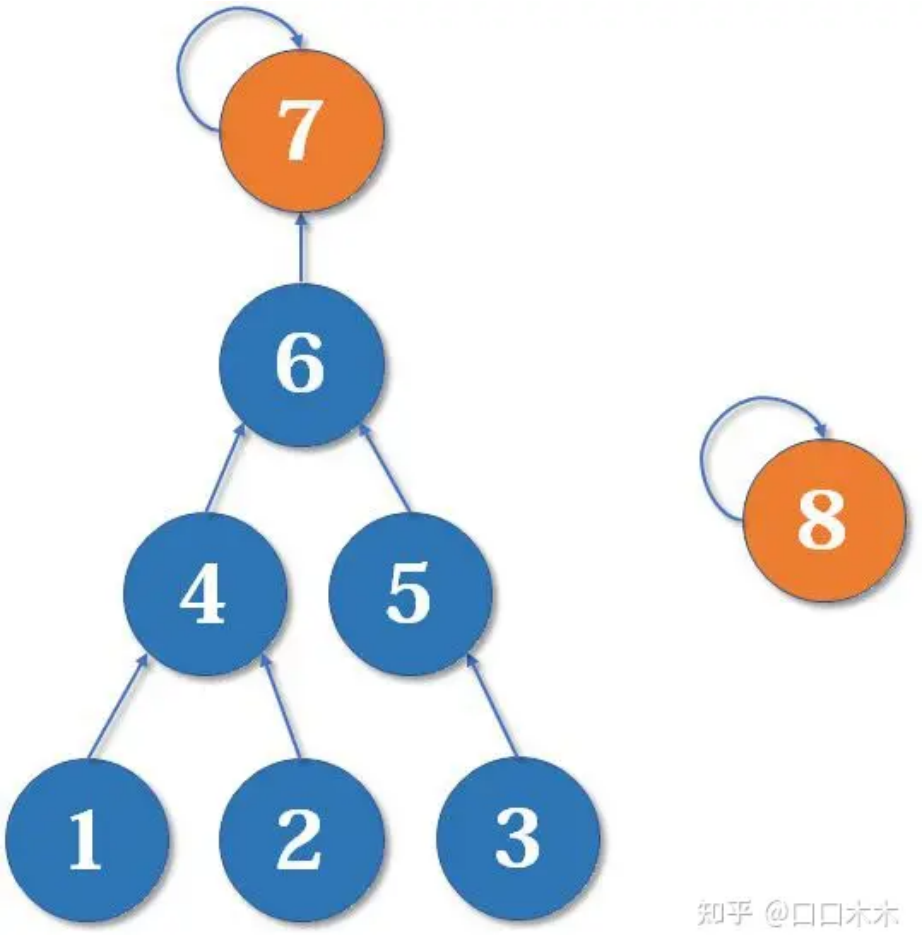
```
}
```

注意赋值运算符 = 的优先级没有三元运算符 ?: 高，这里要加括号。

路径压缩优化后，并查集的时间复杂度已经比较低了，绝大多数不相交集合的合并查询问题都能够解决。然而，对于某些时间卡得很紧的题目，我们还可以进一步优化。

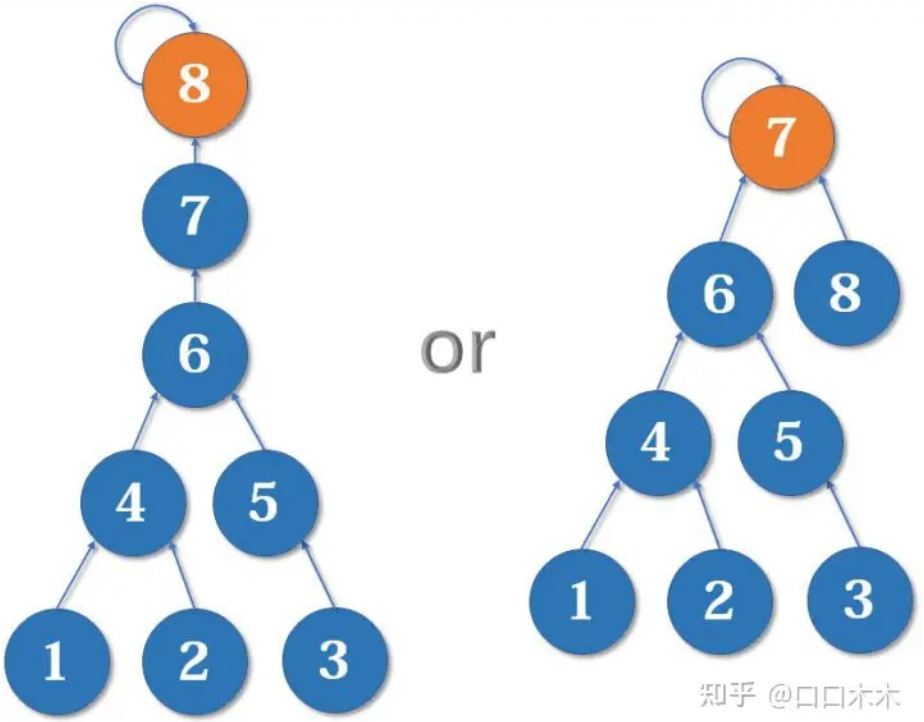
按秩合并

有些人可能有一个误解，以为路径压缩优化后，并查集始终都是一个 **菊花图** （只有两层的树的俗称）。但其实，由于路径压缩只在 **查询** 时进行，也只压缩 **一条路径**，所以并查集最终的结构仍然可能是比较复杂的。例如，现在我们有一棵较复杂的树需要与一个单元素的集合合并：



假如这时我们要merge(7,8)，如果我们可以选择的话，是把7的父节点设为8好，还是把8的父节点设为7好呢?

当然是后者。因为如果把7的父节点设为8，会使树的 **深度**（树中最长链的长度）加深，原来的树中每个元素到根节点的距离都变长了，之后我们寻找根节点的路径也就会相应变长。虽然我们有路径压缩，但路径压缩也是会消耗时间的。而把8的父节点设为7，则不会有这个问题，因为它没有影响到不相关的节点。



这启发我们：我们应该把简单的树往复杂的树上合并，而不是相反。因为这样合并后，到根节点距离变长的节点个数比较少。

我们用一个数组`rank[]`记录每个根节点对应的树的深度（如果不是根节点，其rank相当于以它作为根节点的**子树**的深度）。一开始，把所有元素的rank（**秩**）设为1。合并时比较两个根节点，把rank较小者往较大者上合并。**路径压缩和按秩合并如果一起使用，时间复杂度接近`O(n)`，但是很可能会破坏rank的准确性。**

值得注意的是，按秩合并会带来额外的**空间复杂度**，可能被一些卡空间的毒瘤题卡掉。

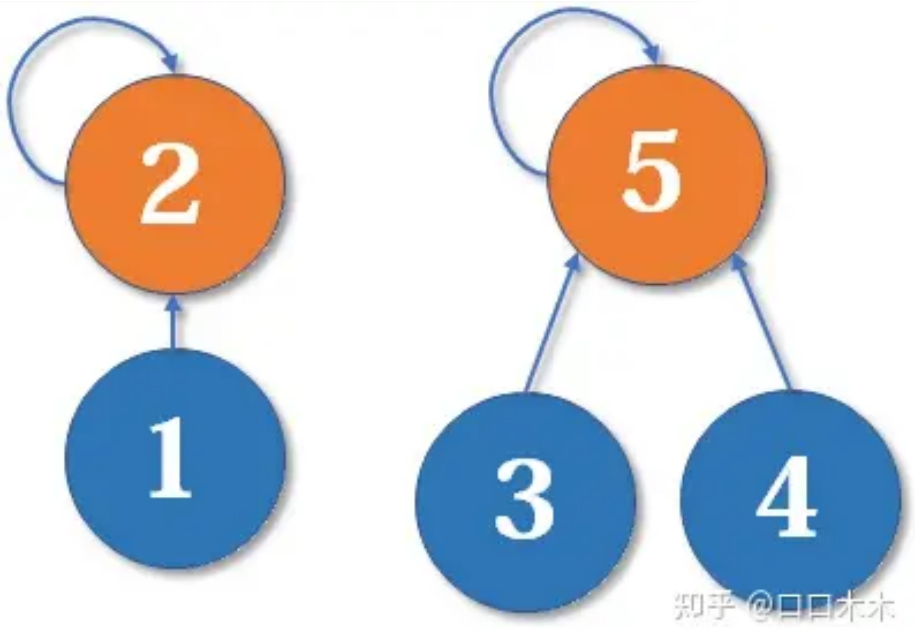
初始化（按秩合并）

```
public void init(int n) {
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        rank[i] = 1;
    }
}
```

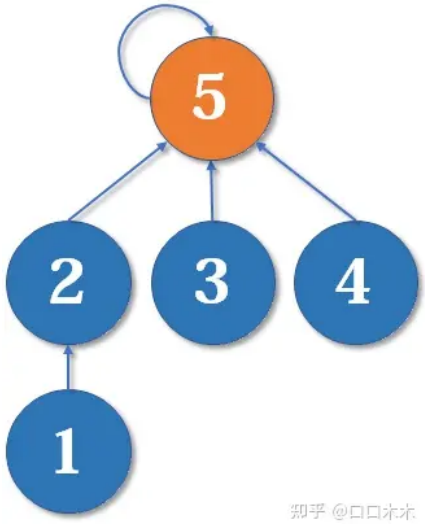
合并（按秩合并）

```
public void merge(int i, int j)
{
    int x = find(i), y = find(j);    //先找到两个根节点
    if (rank[x] <= rank[y]) {
        fa[x] = y; // 相同深度时，也指向y
    } else {
        fa[y] = x;
    }
    if (rank[x] == rank[y] && x != y) {
        rank[y]++; //如果深度相同(上述代码会将x指向y)且根节点不同，则新的根节点(上述代码会
    }
}
```

为什么深度相同，新的根节点深度要+1？如下图，我们有两个深度均为2的树，现在要merge(2,5)：



这里把2的父节点设为5，或者把5的父节点设为2，其实没有太大区别。我们选择前者，于是变成这样：



显然树的深度增加了1。另一种合并方式同样会让树的深度+1。

总结

1. 并查集主要解决的**分组管理**一类的问题，如果问题能抽象成**组与组之间**的问题，一般情况下可考虑并查集，并查集的常见思路：

> 是否在一个组；
> 在一个组的条件；
> 路径和组在图中都属于连通域，上述`组`均可替换为`路径`，问题不变；

2. 并查集的主要难点：

> **Union**时，有的问题已经告诉了分组的信息，有的问题则需要自行挖掘；
> 一般情况下，并查集底层为一个1d数组，有的问题需要对元素进行编号或者转化与之对应；
> 在不清楚并查集中到底会存放多少数据时，底层也可以**map**；

3. 并查集功能是**Union**也就是将两个组合并成一个组，对于拆分的情况，可以逆序思考问题，例如[leecode 803 打砖块](力扣)；

5. 最小生成树的相关问题，[leecode 1584 连接所有点的最小费用](leetcode-cn.com/problem...)

6. 二分+并查集，[leecode 1631 最小体力消耗路径](leetcode-cn.com/problem...), [leecode 778 水位上升的泳池中游泳](leetcode-cn.com/problem...)

例子

- [leecode 399 除法求值](力扣) (带权并查集)

- [leecode 547 省份的数量](力扣) （图中的连通分量数）

- [leecode 684 冗余连接](力扣) （无向图判环）

- [leecode 685 冗余连接II](力扣) （有向图判环）

- [leecode 721 账户合并](力扣)

- [Leecode 765 情侣牵手](力扣)

- [leecode 803 打砖块](力扣) （逆序补砖块）

- [leecode 947 移除最多的同行或同列石头](力扣) （横纵坐标合并）

- [leecode 990 等式方程的可满足性](leetcode-cn.com/problem...)

- [leecode 1202 交换字符串中的元素](力扣)

并查集的应用还有很多，例如最小生成树的Kruskal算法等。这里就不细讲了。总而言之，凡是涉及到元素的分组管理问题，都可以考虑使用并查集进行维护。

参考链接

Pecco：算法学习笔记(1) : 并查集

力扣

zh.wikipedia.org/zh-han...

并查集 - OI Wiki

编辑于 2021-11-16 17:30

搜索算法　图算法　并查集

发布一条带图评论吧

发布

还没有评论，发表第一个评论吧

推荐阅读

并查集详解

简介并查集是一种可以使用代表元来表示不相交集的数据结构，在一些只需要查询两个元素是否属于同一个集合的情况下它很有用。比如给定一个无向图，判断两个顶点是否属于同一个连通分量。 在...

Ander 发表于算法

并查集入门

鱼遇雨欲语与余