2. 并查集

本节目标

- 并查集原理
- 并查集实现
- 并查集应用

1. 并查集原理

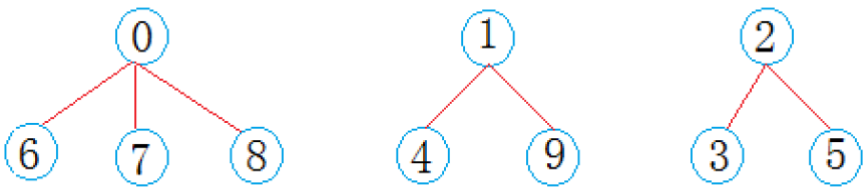
在一些应用问题中，需要**将n个不同的元素划分成一些不相交的集合。开始时，每个元素自成一个单元素集合，然后按一定的规律将归于同一组元素的集合合并。**在此过程中**要反复用到查询某一个元素归属于那个集合的运算。**适合于描述这类问题的抽象数据类型称为**并查集(union-find set)。**

比如：某公司今年校招全国总共招生10人，西安招4人，成都招3人，武汉招3人，10个人来自不同的学校，起先互不相识，每个学生都是一个独立的小团体，现给这些学生进行编号：{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; 给以下数组用来存储该小集体，数组中的数字代表：该小集体中具有成员的个数。(负号下文解释)

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

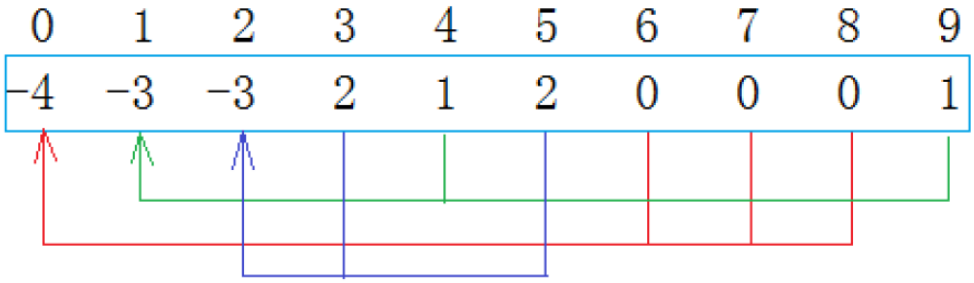
毕业后，学生们要去公司上班，每个地方的学生自发组织成小分队一起上路，于是：

西安学生小分队s1={0,6,7,8}，成都学生小分队s2={1,4,9}，武汉学生小分队s3={2,3,5}就相互认识了，10个人形成了三个小团体。假设右三个群主0,1,2担任队长，负责大家的出行。



集合的树形表示

一趟火车之旅后，每个小分队成员就互相熟悉，称为了一个朋友圈。



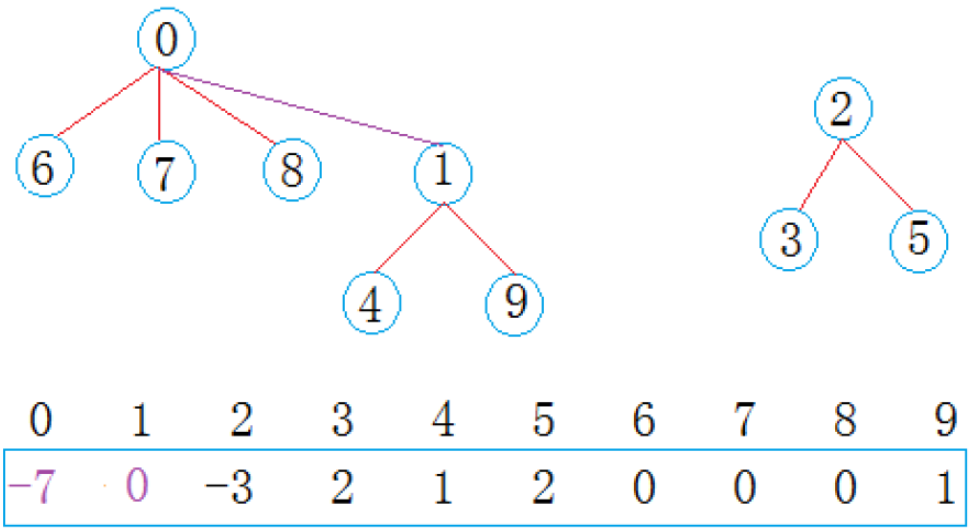
集合s1、s2、s3的森林父指针数组表示

从上图可以看出：编号6,7,8同学属于0号小分队，该小分队中有4人(包含队长0)；编号为4和9的同学属于1号小分队，该小分队有3人(包含队长1)，编号为3和5的同学属于2号小分队，该小分队有3个人(包含队长1)。

仔细观察数组中内融化，可以得出以下结论：

1. **数组的下标对应集合中元素的编号**
2. **数组中如果为负数，负号代表根，数字代表该集合中元素个数**
3. **数组中如果为非负数，代表该元素双亲在数组中的下标**

在公司工作一段时间后，西安小分队中8号同学与成都小分队1号同学奇迹般的走到了一起，两个小圈子的学生相互介绍，最后成为了一个小圈子：



现在0集合有7个人，2集合有3个人，总共两个朋友圈。

通过以上例子可知，并查集一般可以解决一下问题：

1. **查找元素属于哪个集合**

   沿着数组表示树形关系以上一直找到根(即：树中中元素为负数的位置)

2. **查看两个元素是否属于同一个集合**

   沿着数组表示的树形关系往上一直找到树的根，如果根相同表明在同一个集合，否则不在

3. **将两个集合归并成一个集合**
   - 将两个集合中的元素合并
   - 将一个集合名称改成另一个集合的名称
4. **集合的个数**

   遍历数组，数组中元素为负数的个数即为集合的个数。

2. 并查集实现

```
import java.util.List;
import java.util.ArrayList;

public class UnionFindSet {
    private List<Integer> ufs;

    public UnionFindSet(int size){
        ufs = new ArrayList<>(size);
    }

    // 给一个元素的编号，找到该元素所在集合的名称
    public int findRoot(int index)
    {
        // 如果数组中存储的是负数，找到，否则一直继续
        while(ufs.get(index) >= 0)
        {
            index = ufs.get(index);
        }

        return index;
    }

    boolean union(int x1, int x2)
    {
        int root1 = findRoot(x1);
        int root2 = findRoot(x2);

        // x1已经与x2在同一个集合
        if(root1 == root2)
            return false;

        // 将两个集合中元素合并
        ufs.set(root1, ufs.get(root2));

        // 将其中一个集合名称改变成另外一个
        ufs.set(root2, root1);
        return true;
    }

    // 数组中负数的个数，即为集合的个数
    int count()
    {
        int count = 0;
        for(int e : ufs)
        {
            if(e < 0)
                ++count;
        }

        return count;
    }
```

```
}
```

3. 并查集应用

朋友圈

```
class Solution {
    public int findCircleNum(int[][] M) {
        // 矩阵的行和列下标相当于人的编号，元素相当于两人是否为朋友关系
        UnionFindSet ufs = new UnionFindSet(M.length);
        for(int i = 0; i < M.length; ++i)
        {
            for(int j = 0; j < M[i].length; ++j)
            {
                // 自己和自己的关系除外
                if(i == j)
                    continue;

                // 如果i和j是朋友，将其添加到一个朋友圈
                if(1 == M[i][j])
                    ufs.union(i, j);
            }
        }

        return ufs.count();
    }
}
```

等式方程的可满足性

```
/*
解题思路:
  1. 将所有"=="两端的字符合并到一个集合中
  2. 检测"!=" 两端的字符是否在同一个结合中，如果在不满足，如果不在满足
*/

class Solution {
    public boolean equationsPossible(String[] equations) {
        UnionFindSet ufs = new UnionFindSet(26);
        for(int i = 0; i < equations.length; ++i)
        {
            // 将等号两端的字符合并到一个集合中
            if('=' == equations[i].charAt(1))
            {
                ufs.union(equations[i].charAt(0)-'a', equations[i].charAt(3)-'a');
            }
        }

        for(int i = 0; i < equations.length; ++i)
        {
            // 将等号两端的字符合并到一个集合中
            if('!' == equations[i].charAt(1))
```

```
                {
                    // 如果"!="两端的字符在同一个集合中，不满足
                    if(ufs.findRoot(equations[i].charAt(0)-'a') ==
ufs.findRoot(equations[i].charAt(3)-'a'))
                        return false;
                }
            }

            return true;
        }
}
```