并查集

吉林大学计算机学院
谷方明
fmgu2002@sina.com

学习目标

- 了解等价类问题；
- 掌握并查集的定义、操作及实现
- 掌握按秩合并 和 路径压缩
- 了解并查集的效率分析

例题：亲戚

- 若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易。现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

- 规定：**x**和**y**是亲戚，**y**和**z**是亲戚，那么**x**和**z**也是亲戚。即如果**x,y**是亲戚，那么**x**的亲戚都是**y**的亲戚，**y**的亲戚也都是**x**的亲戚。
- 亲戚关系是一种等价关系

数据输入：

- 第一行：三个整数n,m,p，（n<=5000,m<=5000,p<=5000），分别表示有n个人，m个亲戚关系，询问p对亲戚关系。
- 以下m行：每行两个数Mi，Mj，1<=Mi，Mj<=N，表示Mi和Mj具有亲戚关系。
- 接下来p行：每行两个数Pi，Pj，询问Pi和Pj是否具有亲戚关系。

数据输出

- p行，每行一个’Yes’或’No’。表示第i个询问的答案为“具有”或“不具有”亲戚关系。

样例

- input.txt

6 4 3

1 2

1 3

5 4

5 3

1 4

2 3

5 6

- output.txt

Yes

Yes

No

等价类方法

- 亲戚关系是一个等价关系；会把人的集合划分成若干个等价类；

- 初始时，每个等价类都是一个人
- 每次遇到两个人有亲戚关系，就把两个人所在的等价类合并
- 查询两个人是否有亲戚关系时，就是查询两个人是否属于同一等价类

并查集

- 并查集用于维护一些不相交集合，
  $S = \{ S_1, S_2, \ldots, S_r \}$。

- 主要操作
  - ✓ UNION(x,y)：两个集合合并；
  - ✓ FIND (x)：查询某个元素所在的集合；

并查集的操作

- 集合代表元：每个集合S_i都有一个特殊元素rep[S_i];
- MAKE_SET(x): 初始化x为单元素集
- UNION(x, y): 把x和y所在的两个不同集合合并。相当于从S中删除S_x和S_y并加入 $S_x \cup S_y$
- FIND(x): 返回x所在集合S_x的代表rep[S_x]

并查集的实现——顺序存储

- 每个集合用一个长度为n的数组表示
  - 空间：O(n^2)
- 操作的实现及时间效率分析
  - 查找：O(n)
  - 合并：O(n)
- 标识数组？

并查集的实现——链式存储

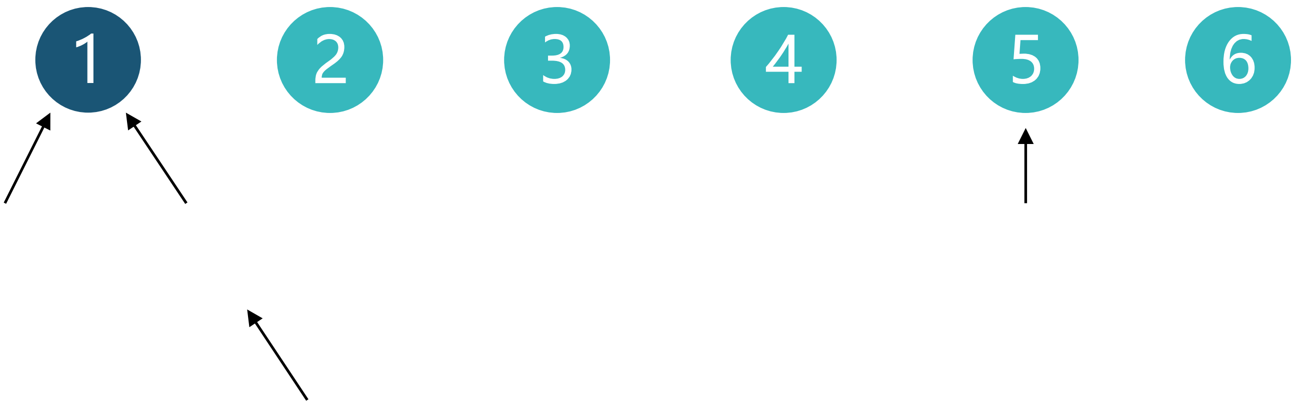
- 每个集合用一个链表表示
  - 空间：O(n)

- 操作的实现及时间效率分析
  - 查找：O(n)
  - 合并：O(n)
  - 启发式合并：O(nlogn)

并查集的实现——集合树

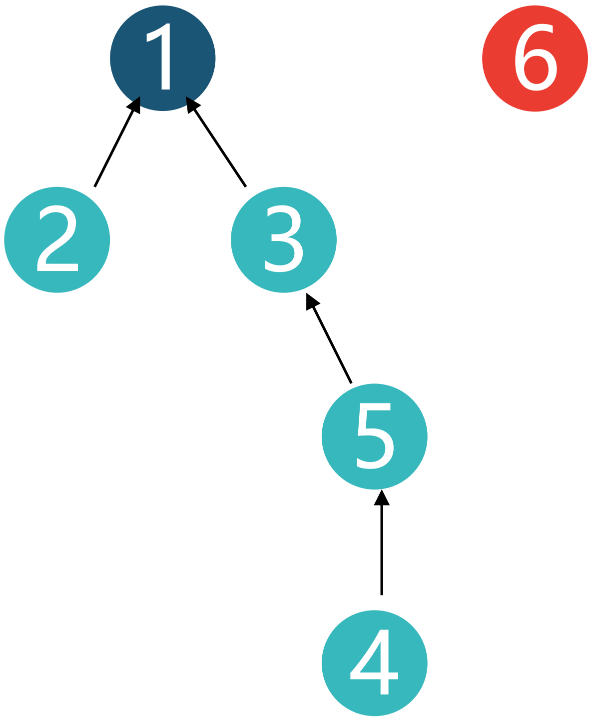
- 每个集合用一棵树表示
- 树的根节点为集合的代表元素

合并示例



- **合并1和2**
- **合并1和3**
- **合并5和4**
- **合并5和3**

查找示例



rep[1]=1

rep[2]=1

rep[3]=1

rep[4]=1

rep[5]=1

rep[6]=6

集合树的Father数组实现

- 存储结构：节点之间的关系用**father**数组维护

  int father[MAXN];

- **MAKE_SET：初始化时 father[v]=v**

/* 根据实际情况，father[v] 可以为 0或特殊值 */

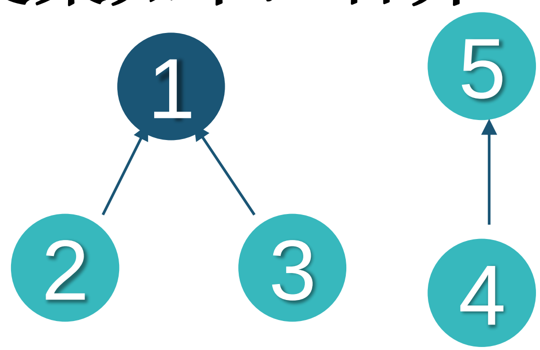
查找操作

1. 从结点v开始，沿father链向上，一直根结点。
2. 可用循环实现（课后练习）;也可用递归实现

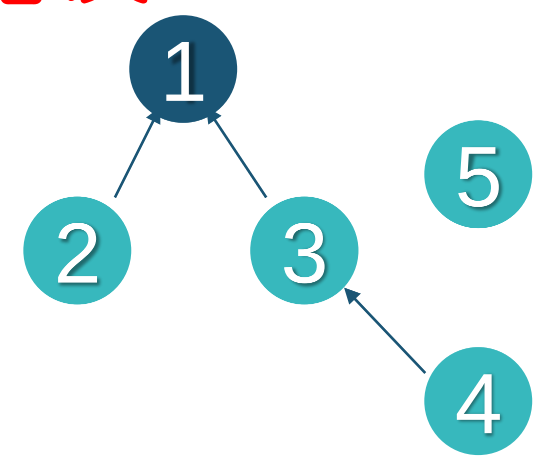
```
int FIND ( int v )
{
    if( father[v]==v ) return v;
    return FIND( father[v] );
}
```

合并操作的关键

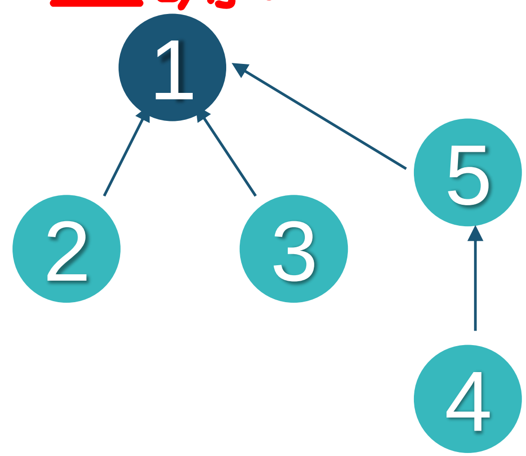
- **结点y(或x)所在树的根结点的父亲指向结点x(或y)所在树的根结点。**
- 例：两个不相交集如下，合并**3**和**4**



错误！



正确！

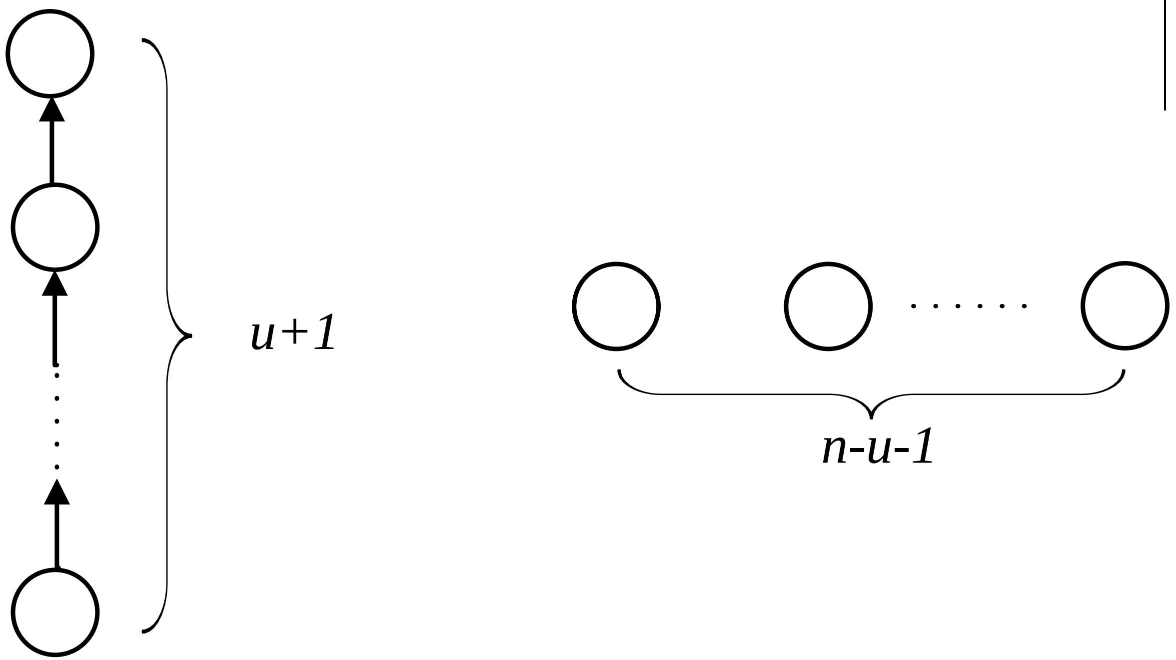


合并操作

```
void UNION( int x , int y) //安全
{
        int fx = FIND (x);
        int fy = FIND (y);
        if( fx != fy ) father[fy] = fx;
 }

void UNION( int x , int y) //f[v]=v
{
      father[FIND (y)] = FIND(x);
}
```

分析



- 查找的时间复杂度O(n)
- 合并O(n)

按秩合并

- 为了避免产生退化树，使用启发式的合并规则
- 按秩合并规则：对于每个结点，维护一个秩(rank)，表示以该结点为根的子树高度的一个上界。按秩合并策略让具有较小秩的根指向具有较大秩的根。
- 最直接的方法是选择以某结点为根的子树的高度作为该结点的秩。当然，也可以使用其它量，如以某结点为根的子树的结点个数(size).

按秩合并的实现

- **MAKE_SET时，每个结点的*rank*初始为0.**
- **UNION(*x*,*y*)操作时，设x和y所在树的根分别为*fx*和*fy*，**
  - ✓ **如果*rank*(*fx*) = *rank*(*fy*)，那么让*fy*指向*fx*，*rank*(*fx*)增1；**
  - ✓ **如果*rank*(*fx*)≠*rank*(*fy*)，那么让*rank*较小的根指向*rank*较大的根，秩不变。**
- **技巧：利用*father*域保存结点的*rank*。**
  - ✓ 如果x是根结点，Father[x]保存结点x的rank的相反数；
  - ✓ 如果x不是根结点，Father[x]保存结点x的父亲的地址。

按秩合并规则的实现

```
void MAKE_SET(x){

        father[x] = 0;

}


void UNION(x,y){
    int fx = FIND(x), fy = FIND(y);
    if( fx == fy ) return;
    if( father[fx] < father[fy]) father[fy]=fx;
    else{
        if(father[fx]==father[fy]) fahter[fy]--;
        father[fx]=fy;
    }
}
```

分析约定

- 设***n***表示**MAKE_SET**操作的次数，亦即并查集的元素总数；***u***表示**UNION**操作的次数；***f***表示**FIND**操作的次数；***m***表示**MAKE_SET**、**UNION**和**FIND**操作的总次数
  - ✓ $m = n + u + f$.
  - ✓ $u \le n$-1
  - ✓ $f \ge u$

定理5.4

- 设F是从初始并查集经过*u*次UNION操作形成的森林，UNION操作使用了按秩合并规则，则F中任一结点的秩最多为$\lfloor \log_2(u+1) \rfloor$.

证明

- u = 1时，定理成立。
- 假设对于所有的k < u 都成立。当 k = u 时，

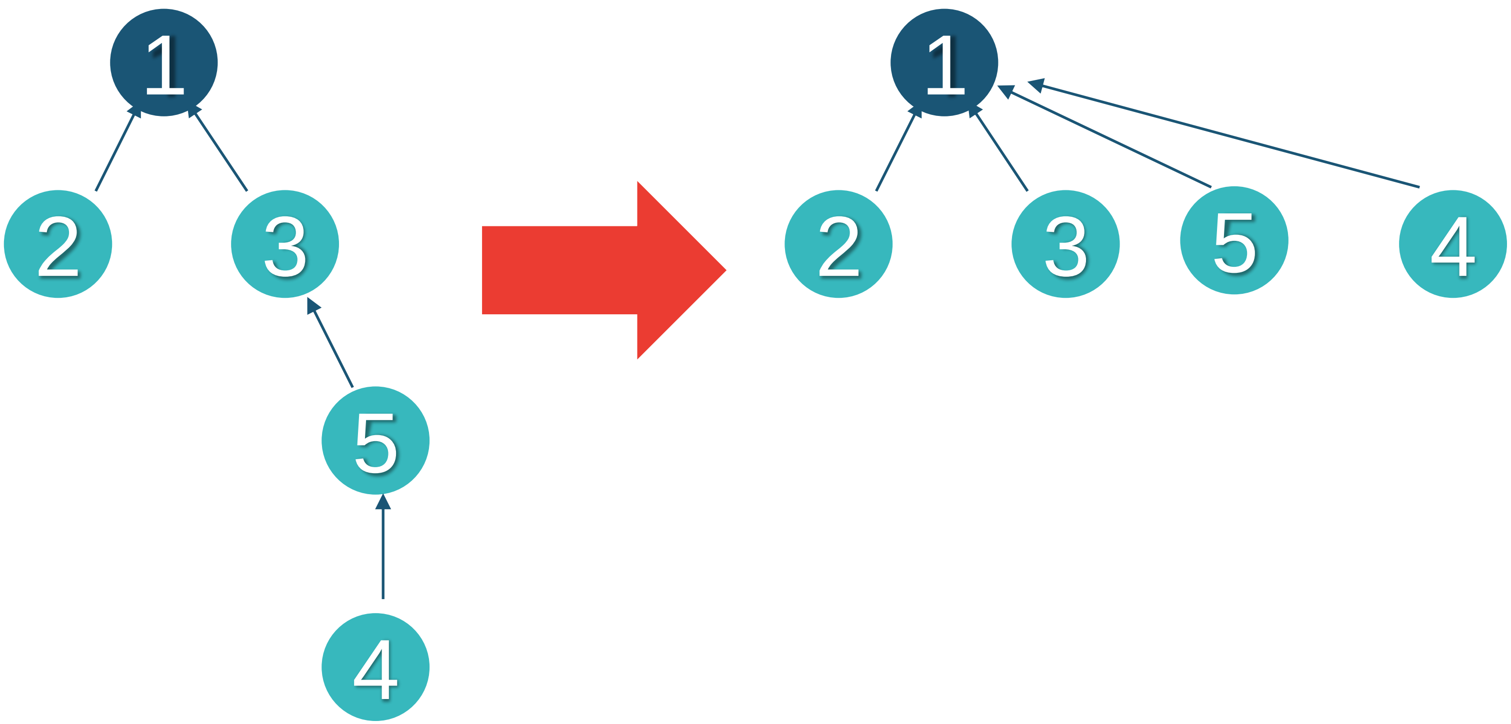
  考虑最后一次调用UNION操作的情况。设最后一次调用为UNION(x,y)，x所在树由p次UNION操作形成，根为fx，y所在树由q次UNION操作形成，根为fy. 显然，$p + q \le u - 1$.

  如果rank(fx)≠rank(fy)，那么fx和fy的秩都不变；

  如果rank(fx) = rank(fy)，秩要增1.

路径压缩

- 在FIND操作中，找到元素 x 所在树的根 fx 之后，将 x 到根 fx 路径上的所有结点的父亲都改成 fx . 这种策略称为路径压缩。
- 例如：Find(4)



带路径压缩的FIND操作

```
int FIND(int v)
{
   if( father[v]<=0 ) return v;
   return   father[v]= FIND (father[v]);
}
```

分析

- 路径压缩增加了一次**FIND**操作的时间，但可能导致树的高度变小，从而提高后续操作的效率
- 一组*m*个**MAKE_SET**、**UNION**和**FIND**操作的序列，其中*n*个是**MAKE_SET**操作，只使用路径压缩，最坏时间复杂度为$O(n+f(1+\log_{2+f/n}n))$.

- 一组*m*个MAKE_SET、UNION和FIND操作的序列，其中*n*个是MAKE_SET操作，在不相交集合森林上使用按秩合并与路径压缩，最坏时间时间复杂度为O(*m* α(*n*)).
- α(·)是Ackerman函数的反函数，增长得非常缓慢。只有对于非常大的n值，才会有α(*n*) > 4. 对于实际的应用，都有α(*n*)≤4.

更一般集合的表示

- 线性表
  - 查
  - 并
  - 交
  - ……

- 标志数组（集合中的元素范围较少）
- bitset (最大32位)