

并查集 (Union Find)

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

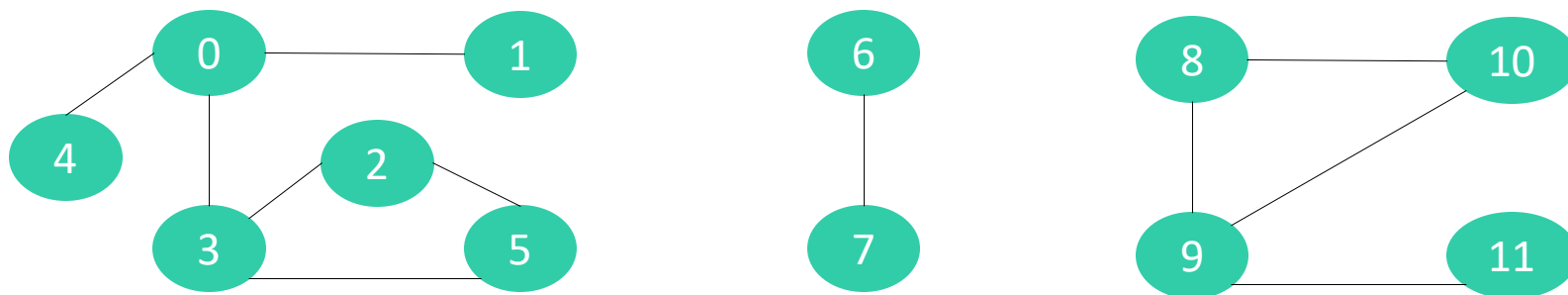
码拉松



实力IT教育 www.520it.com

需求分析

- 假设有 n 个村庄，有些村庄之间有连接的路，有些村庄之间并没有连接的路



- 设计一个数据结构，能够快速执行2个操作

- 查询2个村庄之间是否有连接的路

- 连接2个村庄

- 数组、链表、平衡二叉树、集合 (Set) ?

- 查询、连接的时间复杂度都是： $O(n)$

- 并查集能够办到查询、连接的均摊时间复杂度都是 $O(\alpha(n))$, $\alpha(n) < 5$

- 并查集非常适合解决这类“连接”相关的问题

并查集 (Union Find)

- 并查集也叫作不相交集合 (Disjoint Set)

- 并查集有2个核心操作

- 查找 (Find) : 查找元素所在的集合 (这里的集合并不是特指Set这种数据结构, 是指广义的数据集合)

- 合并 (Union) : 将两个元素所在的集合合并为一个集合

- 有2种常见的实现思路

- Quick Find

- ✓ 查找 (Find) 的时间复杂度: $O(1)$

- ✓ 合并 (Union) 的时间复杂度: $O(n)$

- Quick Union

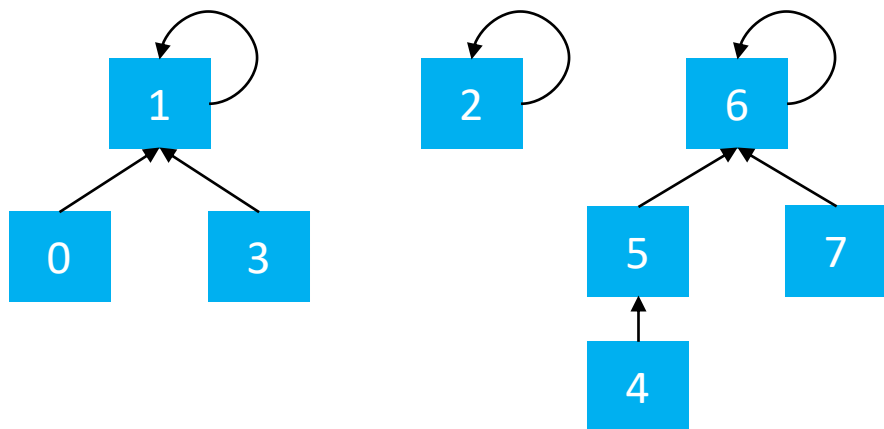
- ✓ 查找 (Find) 的时间复杂度: $O(\log n)$, 可以优化至 $O(\alpha(n))$, $\alpha(n) < 5$

- ✓ 合并 (Union) 的时间复杂度: $O(\log n)$, 可以优化至 $O(\alpha(n))$, $\alpha(n) < 5$

如何存储数据？

■ 假设并查集处理的数据都是整型，那么可以用整型数组来存储数据

0	1	2	3	4	5	6	7
1	1	2	1	5	6	6	6



■ 不难看出

□ 0、1、3 属于同一集合

□ 2 单独属于一个集合

□ 4、5、6、7 属于同一集合

■ 因此，并查集是可以用数组实现的树形结构（二叉堆、优先级队列也是可以用数组实现的树形结构）

```
/**
 * 查找v所属的集合（根节点）
 */
int find(int v);

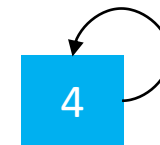
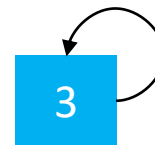
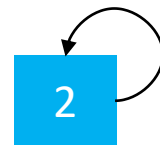
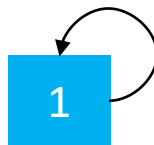
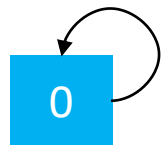
/**
 * 合并v1、v2所属的集合
 */
void union(int v1, int v2);

/**
 * 检查v1、v2是否属于同一个集合
 */
boolean isSame(int v1, int v2);
```

```
public boolean isSame(int v1, int v2) {
    return find(v1) == find(v2);
}
```

初始化

- 初始化时，每个元素各自属于一个单元素集合

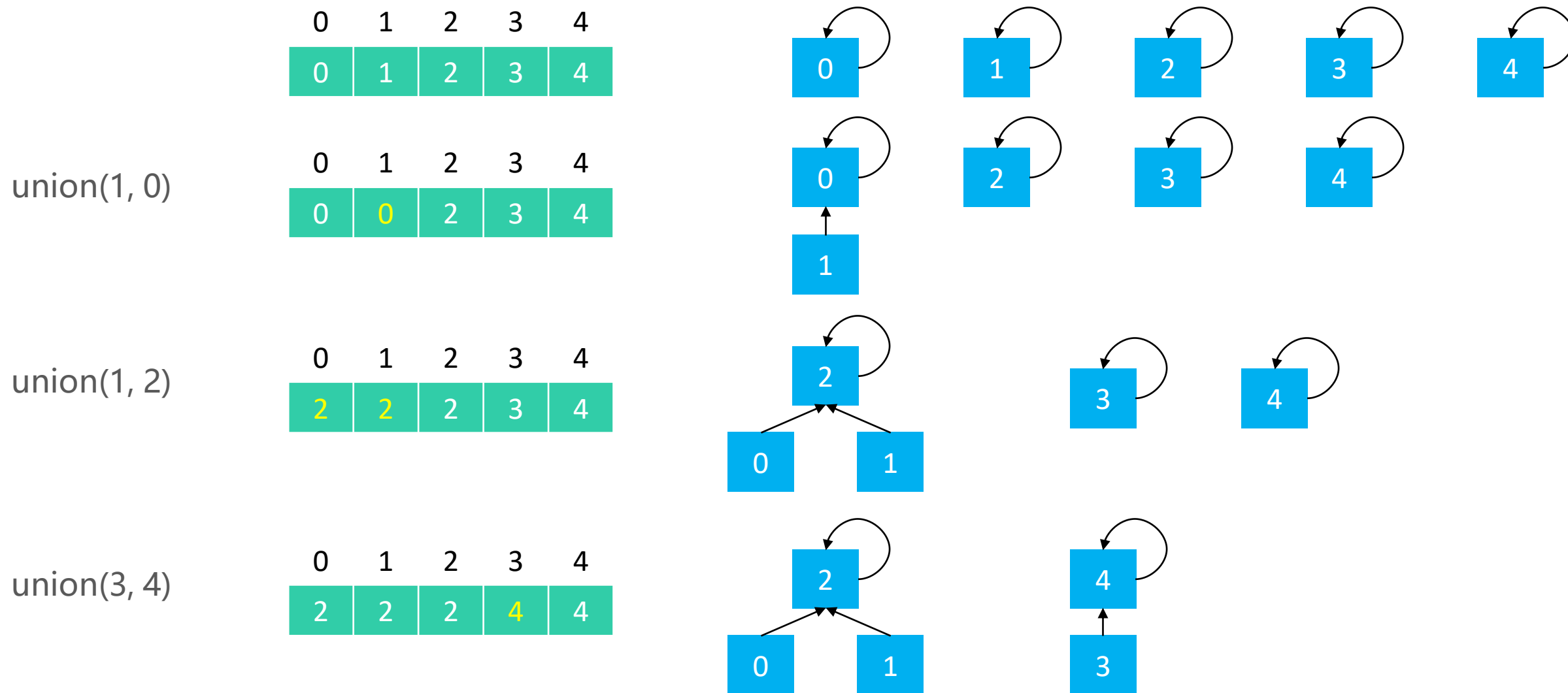


```
private int[] parents;
public UnionFind(int capacity) {
    if (capacity < 0) {
        throw new IllegalArgumentException("Capacity must >= 1.");
    }
    parents = new int[capacity];

    for (int i = 0; i < parents.length; i++) {
        parents[i] = i;
    }
}
```

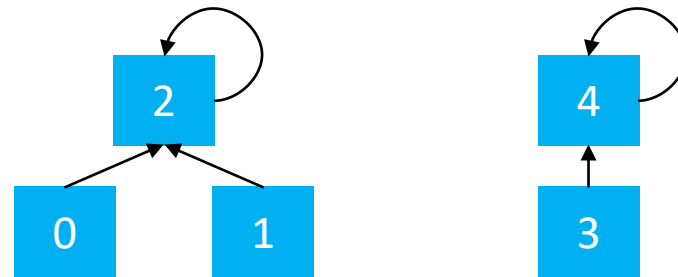
Quick Find – Union

■ Quick Find 的 $\text{union}(v1, v2)$: 让 $v1$ 所在集合的所有元素都指向 $v2$ 的根节点



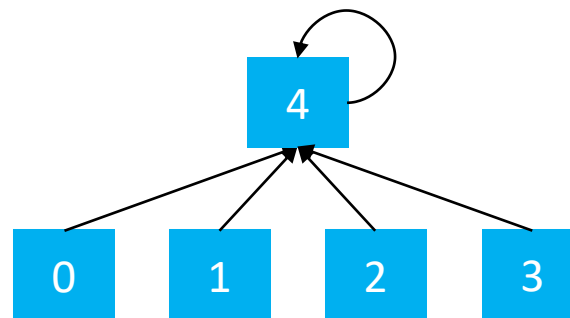
Quick Find – Union

0	1	2	3	4
2	2	2	4	4



union(0, 3)

0	1	2	3	4
4	4	4	4	4



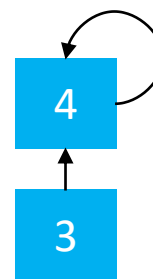
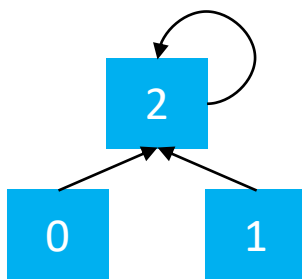
Quick Find – Union

```
public void union(int v1, int v2) {  
    int p1 = find(v1);  
    int p2 = find(v2);  
    if (p1 == p2) return;  
  
    for (int i = 0; i < parents.length; i++) {  
        if (parents[i] == p1) {  
            parents[i] = p2;  
        }  
    }  
}
```

■ 时间复杂度: $O(n)$

Quick Find – Find

0	1	2	3	4
2	2	2	4	4

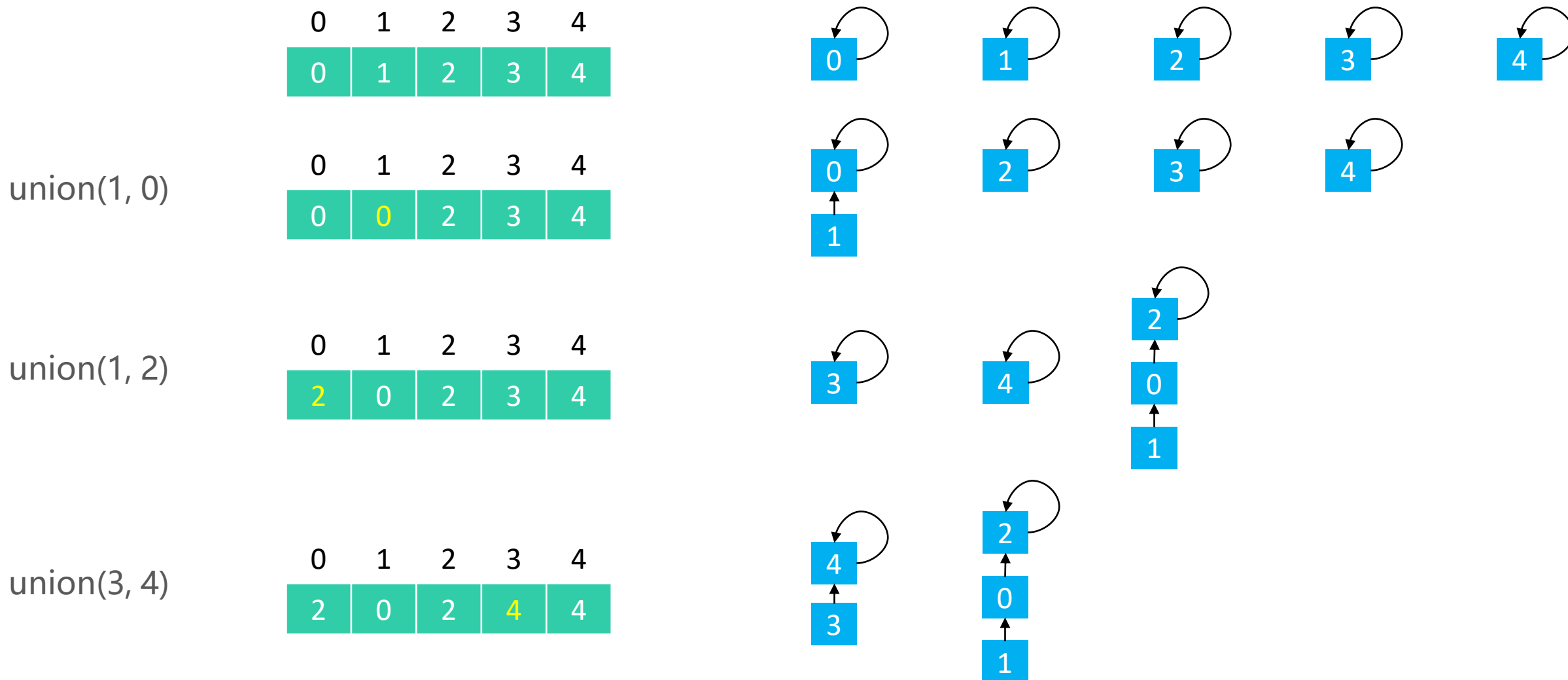


```
public int find(int v) {  
    rangeCheck(v);  
    return parents[v];  
}
```

- `find(0) == 2`
- `find(1) == 2`
- `find(3) == 4`
- `find(2) == 2`
- 时间复杂度: $O(1)$

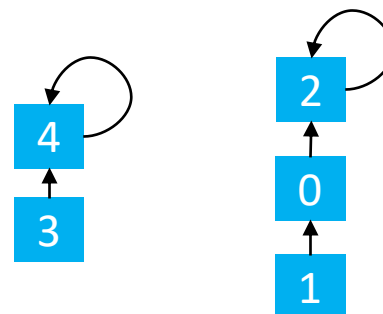
Quick Union – Union

■ Quick Union 的 $\text{union}(v1, v2)$: 让 $v1$ 的根节点指向 $v2$ 的根节点



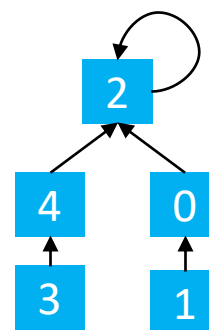
Quick Union – Union

0	1	2	3	4
2	0	2	4	4



union(3, 1)

0	1	2	3	4
2	0	2	4	2



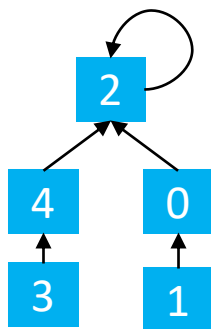
Quick Union – Union

```
public void union(int v1, int v2) {  
    int p1 = find(v1);  
    int p2 = find(v2);  
    if (p1 == p2) return;  
  
    parents[p1] = p2;  
}
```

■ 时间复杂度: $O(\log n)$

Quick Union – Find

0	1	2	3	4
2	0	2	4	2

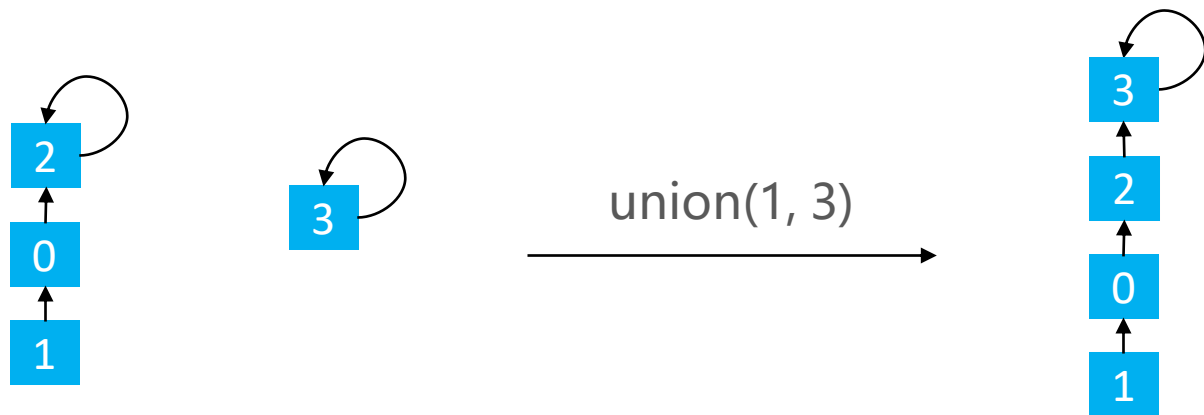


```
public int find(int v) {  
    rangeCheck(v);  
    while (v != parents[v]) {  
        v = parents[v];  
    }  
    return v;  
}
```

- `find(0) == 2`
- `find(1) == 2`
- `find(3) == 2`
- `find(2) == 2`
- 时间复杂度: $O(\log n)$

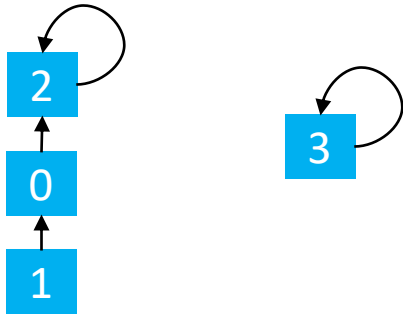
Quick Union – 优化

- 在Union的过程中，可能会出现树不平衡的情况，甚至退化成链表

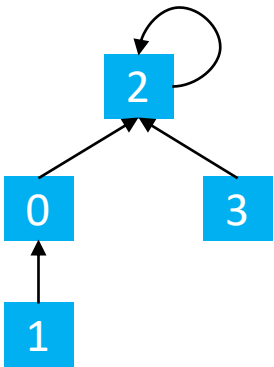


- 有2种常见的优化方案
 - 基于size的优化：元素少的树 嫁接到 元素多的树
 - 基于rank的优化：矮的树 嫁接到 高的树

Quick Union – 基于size的优化



↓ union(1, 3)



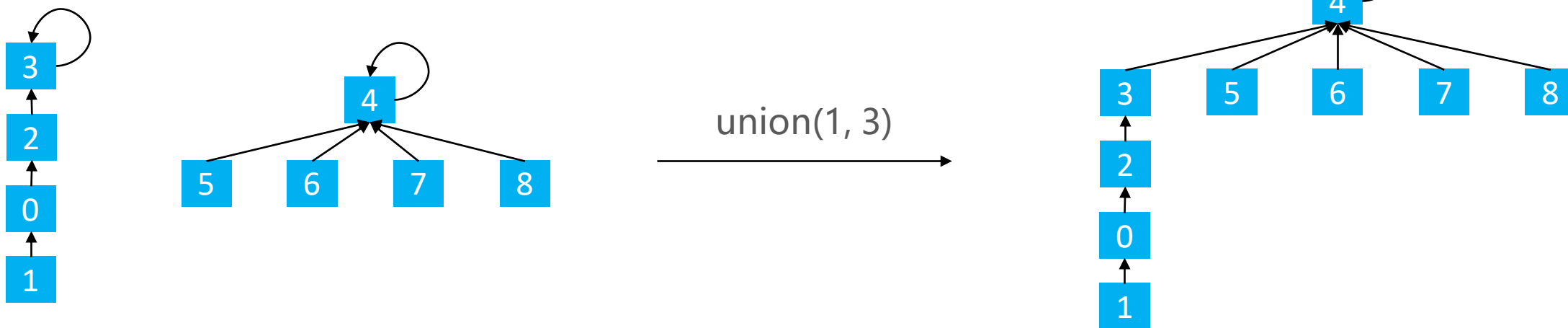
```
sizes = new int[capacity];
for (int i = 0; i < sizes.length; i++) {
    sizes[i] = 1;
}
```

```
private int[] sizes;
public void union(int v1, int v2) {
    int p1 = find(v1);
    int p2 = find(v2);
    if (p1 == p2) return;

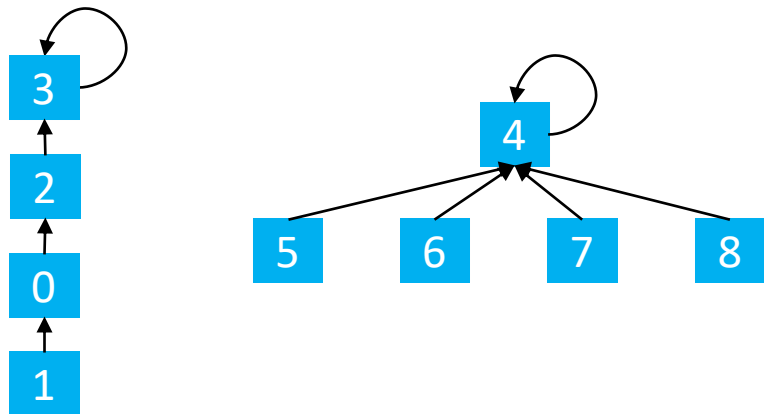
    if (sizes[p1] < sizes[p2]) {
        parents[p1] = p2;
        sizes[p2] += sizes[p1];
    } else {
        parents[p2] = p1;
        sizes[p1] += sizes[p2];
    }
}
```


Quick Union – 基于size的优化

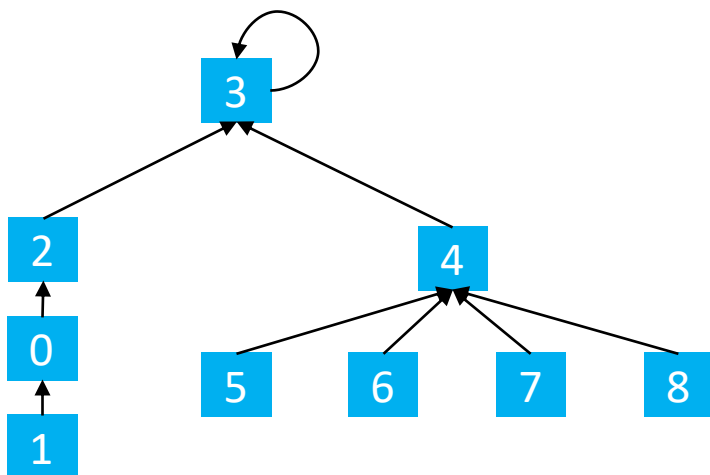
- 基于size的优化，也可能会存在树不平衡的问题



Quick Union – 基于rank的优化



union(1, 3)



```

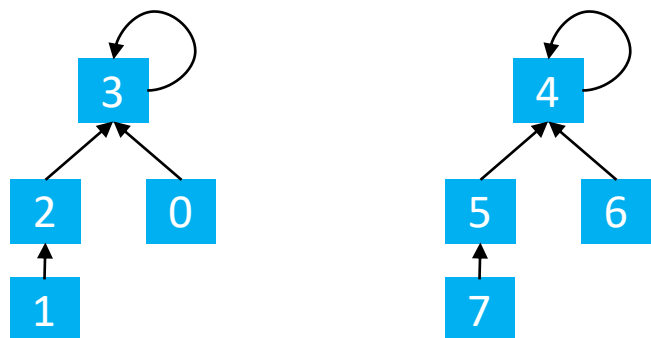
ranks = new int[capacity];
for (int i = 0; i < ranks.length; i++) {
    ranks[i] = 1;
}
  
```

```

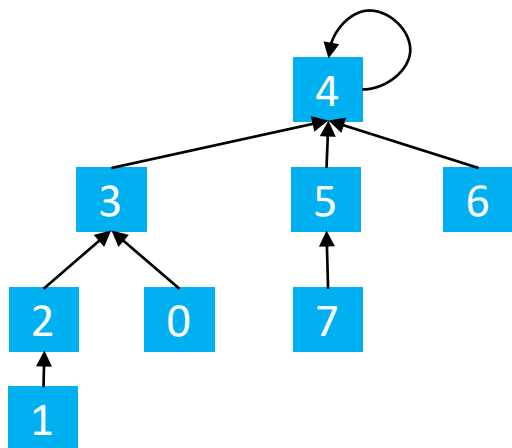
private int[] ranks;
public void union(int v1, int v2) {
    int p1 = find(v1);
    int p2 = find(v2);
    if (p1 == p2) return;

    if (ranks[p1] < ranks[p2]) {
        parents[p1] = p2;
    } else if (ranks[p2] < ranks[p1]) {
        parents[p2] = p1;
    } else {
        parents[p1] = p2;
        ranks[p2]++;
    }
}
  
```

路径压缩 (Path Compression)

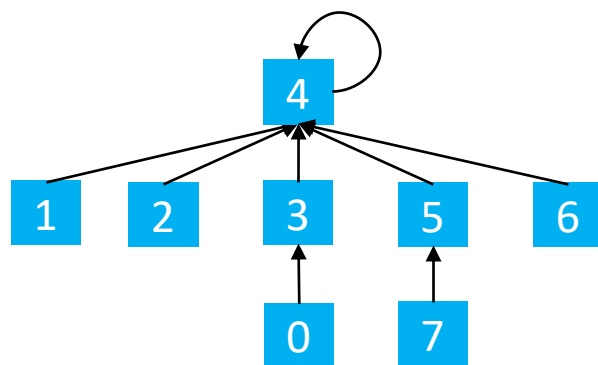


union(1, 5)

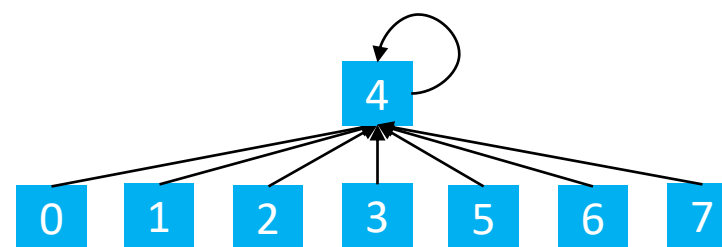


- 虽然有了基于rank的优化，树会相对平衡一点
- 但是随着Union次数的增多，树的高度依然会越来越高
- 导致find操作变慢，尤其是底层节点（因为find是不断向上找到根节点）
- 什么是路径压缩？
- 在find时使路径上的所有节点都指向根节点，从而降低树的高度

find(1)



find(0)、find(7)



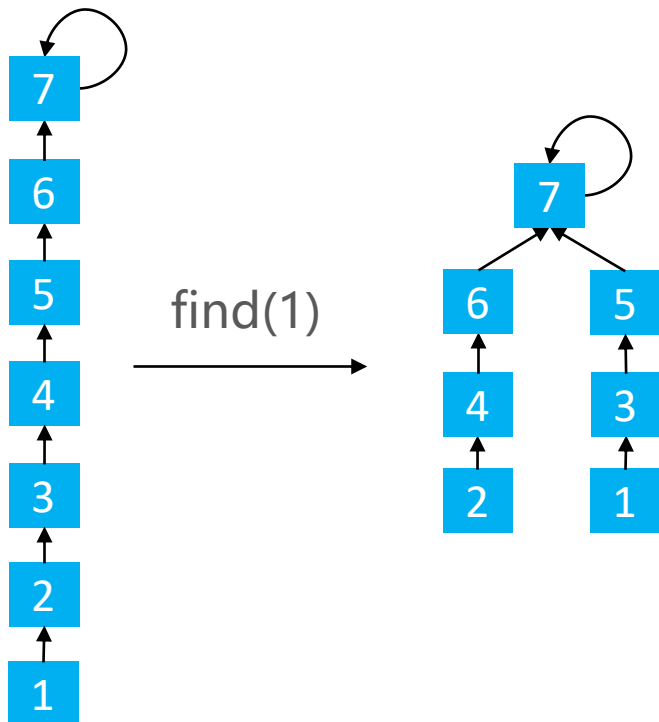
路径压缩 (Path Compression)

```
public int find(int v) {  
    rangeCheck(v);  
    if (parents[v] != v) {  
        parents[v] = find(parents[v]);  
    }  
    return parents[v];  
}
```

- 路径压缩使路径上的所有节点都指向根节点，所以实现成本稍高
- 还有2种更优的做法，不但能降低树高，实现成本也比路径压缩低
 - 路径分裂 (Path Splitting)
 - 路径减半 (Path Halving)
- 路径分裂、路径减半的效率差不多，但都比路径压缩要好

路径分裂 (Path Splitting)

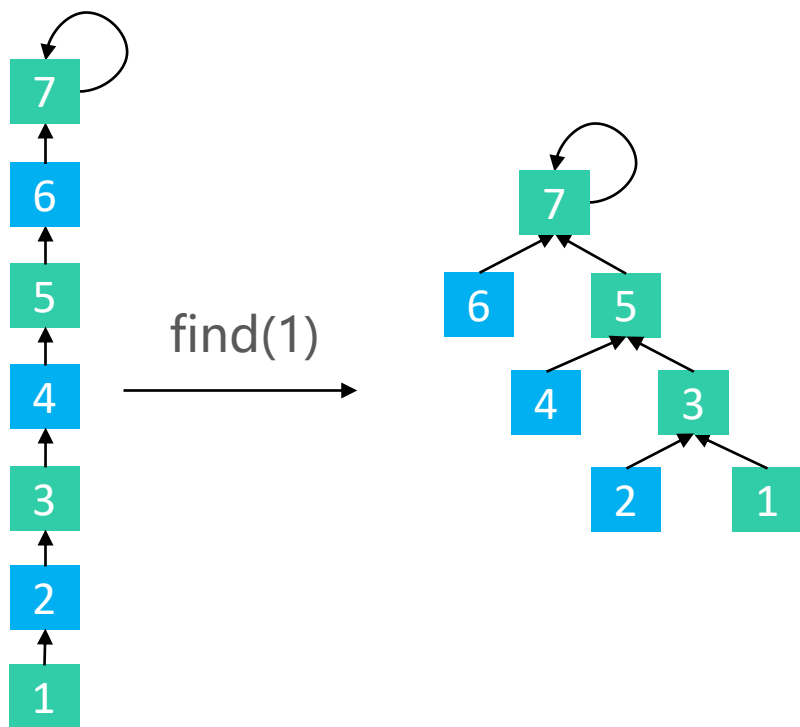
- 路径分裂：使路径上的每个节点都指向其祖父节点 (parent的parent)



```
public int find(int v) {  
    rangeCheck(v);  
    while (v != parents[v]) {  
        int parent = parents[v];  
        parents[v] = parents[parent];  
        v = parent;  
    }  
    return v;  
}
```

路径减半 (Path Halving)

- 路径减半：使路径上每隔一个节点就指向其祖父节点 (parent的parent)



```
public int find(int v) {  
    rangeCheck(v);  
    while (v != parents[v]) {  
        parents[v] = parents[parents[v]];  
        v = parents[v];  
    }  
    return v;  
}
```

- 摘自《维基百科》：https://en.wikipedia.org/wiki/Disjoint-set_data_structure#Time_complexity

Using both *path compression*, *splitting*, or *halving* and *union by rank* or *size* ensures that the amortized time per operation is only $O(\alpha(n))$,^{[4][5]} which is optimal,^[6] where $\alpha(n)$ is the inverse Ackermann function. This function has a value $\alpha(n) < 5$ for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.

- 大概意思是

- 使用路径压缩、分裂或减半 + 基于rank或者size的优化

- ✓ 可以确保每个操作的均摊时间复杂度为 $O(\alpha(n))$, $\alpha(n) < 5$

- 个人建议的搭配

- ✓ Quick Union

- ✓ 基于 rank 的优化

- ✓ Path Halving 或 Path Splitting

自定义类型

- 之前的使用都是基于整型数据，如果其他自定义类型也想使用并查集呢？
- 方案一：通过一些方法将自定义类型转为整型后使用并查集（比如生成哈希值）
- 方案二：使用链表+映射（Map）