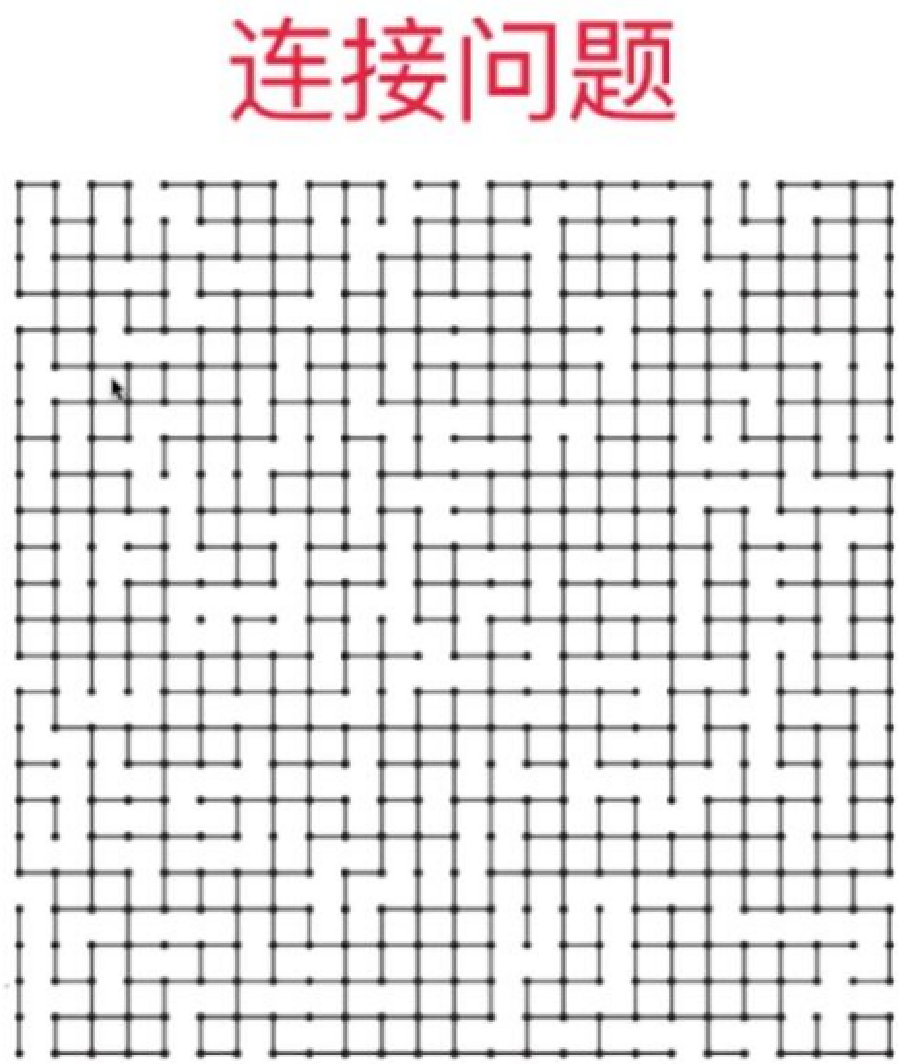
并查集

- 并查集可用于解决两点间是否连通的问题



  对于上面的图，如果给定两个点，如果两点很近，或许我们可以一眼看出两点之间是否连通，但是对于相隔很远的两个点，则比较难以判断了，而并查集则可以很好的处理这一类问题。

  并查集用于解决网络中两点的连通问题，需要注意的是这里的网络不仅仅指计算机网络，我们还可以抽象成社交网络，城市网络等。

  还有一点需要注意的是，上图似乎也是一个图论问题，我们也可以用图相关的算法判断两个点之间是否连通，那为什么还需要使用并查集呢？图论算法可以获取结果，但是需要注意的是图论一般获取的是两点之间的路径，这会带来额外的性能损耗，我们只需要判断两点是否连通，并不需要两点之间的路径。

- 并查集支持的操作

  由并查集这个名字我们也应该能判断出其支持的操作主要有

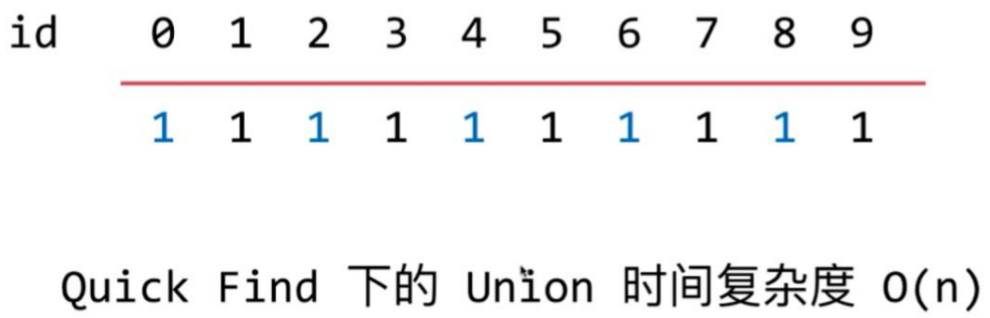
  - union(p, q)：合并操作
  - isConnected(p, q)：判断两点之间是否连通

- 并查集实现

  - 使用数组实现

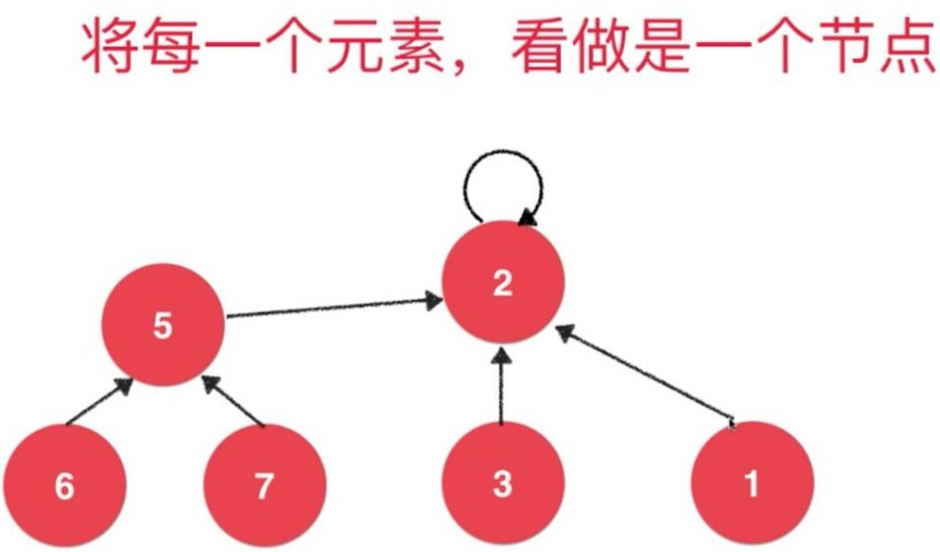
    一个比较常见的想法是，维护一个数组，数组中记录这对应元素属于哪一个集合，当我们需要进行相关操作时，直接查询数组即可

对于此种实现方式，我们可以快速判断两个元素是否属于同一集合之中，但是对于合并操作会存在一定的问题。当我们合并索引为`1`和`2`的两个节点时，实际上`1`和`2`所属的集合也就连通了，此时我们需要遍历数组，更改相应的数组值。



- 使用树形式实现：数组作为底层

  实际上，我们可以借助堆的表现形式，将数组中存放的元素构成一棵树



当我们初始化这个数组时，每个节点都是指向自身的一棵树，数组中存放着该元素指向的元素的索引。当我们合并两个元素时，我们将某个节点指向另一个节点即可。

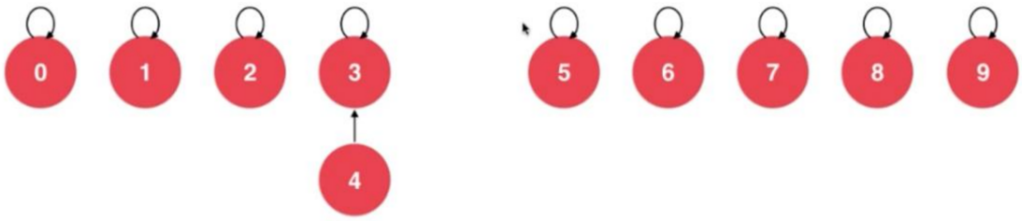
Quick Union 下的数据表示

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

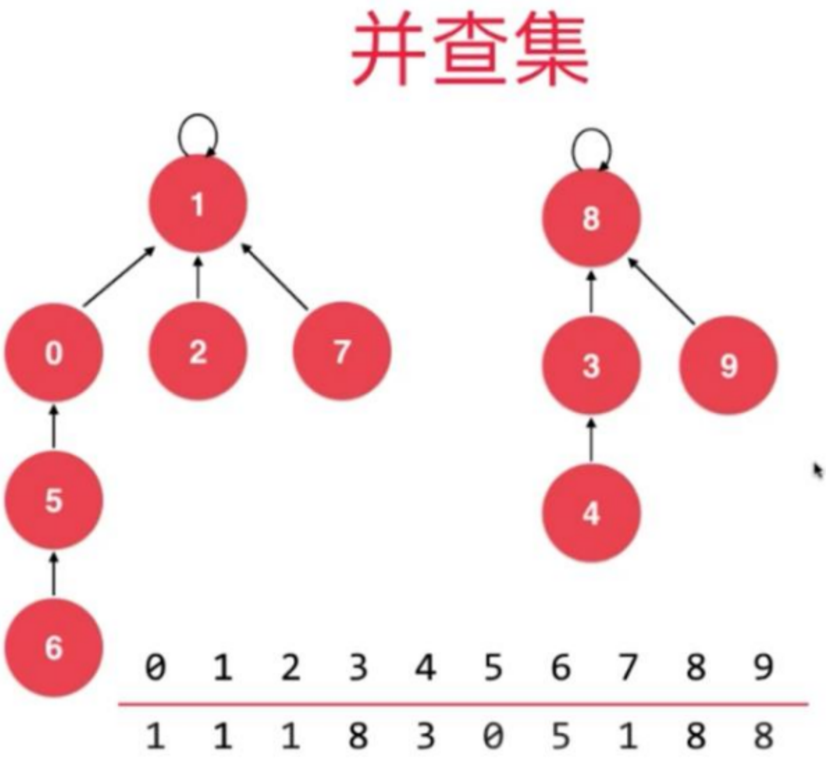
union 4 , 3

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

union 4 , 3



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



- 代码实现
  - 数组实现

```
public class UnionFind{
     private int[] id;    // 我们的第一版Union-Find本质就是一个数组

    public UnionFind1(int size) {

        id = new int[size];

        // 初始化, 每一个id[i]指向自己, 没有合并的元素
        for (int i = 0; i < size; i++)
            id[i] = i;
    }

    @Override
    public int getSize(){
        return id.length;
    }

    // 查找元素p所对应的集合编号
    // O(1)复杂度
    private int find(int p) {
        if(p < 0 || p >= id.length)
            throw new IllegalArgumentException("p is out of bound.");

        return id[p];
    }

    // 查看元素p和元素q是否所属一个集合
    // O(1)复杂度
    @Override
    public boolean isConnected(int p, int q) {
```

```
        return find(p) == find(q);
    }

    // 合并元素p和元素q所属的集合
    // O(n) 复杂度
    @Override
    public void unionElements(int p, int q) {

        int pID = find(p);
        int qID = find(q);

        if (pID == qID)
            return;

        // 合并过程需要遍历一遍所有元素, 将两个元素的所属集合编号合并
        for (int i = 0; i < id.length; i++)
            if (id[i] == pID)
                id[i] = qID;
    }
}
```

- 树形式实现，本质上还是数组作为底层

```
public class UnionFind{
    private int[] parent; //维护一个数组，记录当前节点指向的父节点

    public UnionFind(int size){
        parent = new int[size];
        for(int i=0;i<parent.length;i++){
            parent[i] = i;
        }
    }
    // 查找过程, 查找元素p所对应的集合编号
    // O(h)复杂度, h为树的高度
    public int find(int p){
        if(p<0 || p>parent.length)
            throw new IllegalArguement("index out of range!");
        // 不断去查询自己的父亲节点, 直到到达根节点
        // 根节点的特点: parent[p] == p
        while(p != parent[p])
            p = parent[p];
        return p;
    }
    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度, h为树的高度
    public boolean isConnected(int p, int q){
        return find(p) == find(q);
    }
    // 合并元素p和元素q所属的集合
    // O(h)复杂度, h为树的高度
    public void union(int p, int q){
        int pRoot = find(p);
        int qRoot = find(q);
        if(pRoot == qRoot)
            return;
        parent[pRoot] = qRoot;
    }
```

```
}
```

此种实现方式已经在一定程度上提高了并查集的性能，但是在某些情况下还可能变得很糟糕。比如当我们顺序连接节点时，此时的并查集就会退化成为一个链表的形式，这是不可以忍受的，因此我们需要对代码进行一定的优化

- 基于`size`的优化

  一个比较常见的想法是，当我们将两个集合合并时，我们可以将节点数较少的集合合并到节点数较大的集合中，一般而言，节点数较少时其树的高度也会较小

```java
public class UnionFind{
    private int[] parent;
    private int[] sz;

    public UnionFind(int size){
        parent = new int[size];
        sz = new int[size];
        for(int i=0; i<parent.length;i++){
            parent[i] = i;
            sz[i] = 1;//每个节点的size值为1
        }
    }

    public int find(int p){
        if(p<0 || p>parent.length)
            throw new IllegalArguement("index out of range!");
        while(p != parent[p])
            p = parent[p];
        return p;
    }

    public boolean isConnected(int p, int q){
        return find(p) == find(q);
    }

    public void union(int p, int q){
        int pRoot = find(p);
        int qRoot = find(q);

        if(pRoot == qRoot)
            return;
        if(sz[pRoot] > sz[qRoot]){
            parent[qRoot] = pRoot;
            sz[pRoot] += sz[qRoot];
        }else{
            parent[pRoot] = qRoot;
            sz[qRoot] += pRoot;
        }
    }

}
```

  此种实现还存在一定的问题，当一个集合拥有较多元素但是都处于第二层，而另一个集合虽然元素较少，但是形成链表时，利用size来判断就会出现问题

- 基于rank的优化，rank不完全表示树的层数，我们可以理解为层数，但是更多的表示的是一种优先级的思想

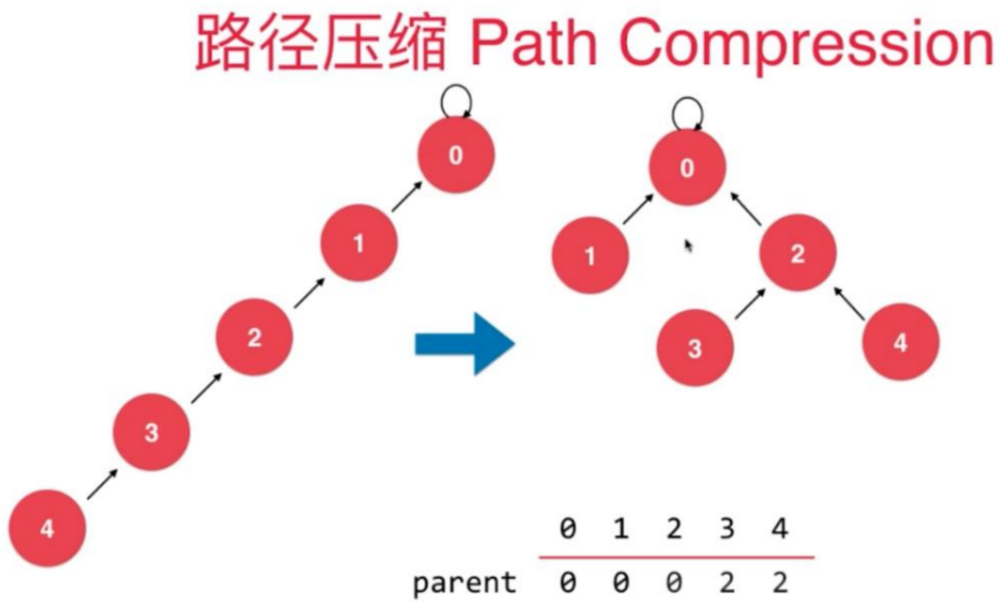
```java
public class UnionFind{
     private int[] rank;   // rank[i]表示以i为根的集合所表示的树的层数
    private int[] parent; // parent[i]表示第i个元素所指向的父节点

    // 构造函数
    public UnionFind(int size){

        rank = new int[size];
        parent = new int[size];

        // 初始化, 每一个parent[i]指向自己, 表示每一个元素自己自成一个集合
        for( int i = 0 ; i < size ; i ++ ){
            parent[i] = i;
            rank[i] = 1;
        }
    }

    // 查找过程, 查找元素p所对应的集合编号
    // O(h)复杂度, h为树的高度
    private int find(int p){
        if(p < 0 || p >= parent.length)
            throw new IllegalArgumentException("p is out of bound.");

        // 不断去查询自己的父亲节点, 直到到达根节点
        // 根节点的特点: parent[p] == p
        while(p != parent[p])
            p = parent[p];
        return p;
    }

    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度, h为树的高度
    public boolean isConnected( int p , int q ){
        return find(p) == find(q);
    }

    public void union(int p, int q){
        int pRoot = find(p);
        int qRoot = find(q);
        if(pRoot == qRoot)
            return;
        if(rank[pRoot] > rank[qRoot]){
            parent[qRoot] = pRoot;
            //pRoot的层数小于qRoot，此时qRoot的层数是不会改变的，因为直接指向根节点了
        }else if(rank[pRoot] < rank[qRoot]){
            parent[pRoot] = qRoot;
        }else{
            //此时，树的层数会发生变化
            parent[pRoot] = qRoot;
            rank[pRoot]++;
        }

    }

}
```

此时这种实现已经有较好的性能了，但是我们依旧可以继续优化

- 路径压缩

  虽然，采用上面的方法已经可以建立一个比较矮的树，但是我们依旧可以继续压缩树的高度

  parent[p] = parent[parent[p]]，利用该式，我们可以在find的过程中降低树的层数（将节点指向父节点的父节点）




```
public int find(int p){
    if(p < 0 || p >= parent.length)
            throw new IllegalArgumentException("p is out of bound.");
    //当前节点不是根节点
    while(p != parent[p]){
        //让当前节点指向父节点的父节点，则可以去掉一重路径
        parent[p] = parent[parent[p]];
        //跳过父节点，直接走向父节点的父节点
        p = parent[p];
    }
    return p;
}
//此方法可以将树转换成只有两层的形式，但是需要多次调用find
```

此时，我们并没有维护rank数组的值，rank[i]表示以i为根的集合所表示的树的层数，在后续的代码中, 我们并不会维护rank的语意, 也就是rank的值在路径压缩的过程中, 有可能不在是树的层数，这也是我们的rank不叫height或者depth的原因, 他只是作为比较的一个标准

```
// 查找过程, 查找元素p所对应的集合编号
// O(h)复杂度, h为树的高度
private int find(int p){
    if(p < 0 || p >= parent.length)
        throw new IllegalArgumentException("p is out of bound.");

    // path compression 2, 递归算法
    if(p != parent[p])
        parent[p] = find(parent[p]);//将值直接改为节点的父节点
    return parent[p];
}
//可以一次将树转换为只有两层的形式，但是递归调用会有额外的开销，性能并不比上面的方法好
```