并查集笔记

使用适合条件

- 并查集是一种树型的数据结构，用于处理一些不交集（Disjoint Sets）的合并及查询问题。有一个联合-查找算法（union-find algorithm）定义了两个用于此数据结构的操作：
    - Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
    - Union：将两个子集合并成同一个集合。
- 常用实现思路

1.QuickFind

```
/// 返回根节点
    /// - Parameter e: <#e description#>
    /// - Returns: <#description#>
    override func  find(e:Int) -> Int {
        checkRange(e: e)

        return parents[e]
    }

    override func union(e1:Int,e2:Int) {
        checkRange(e: e1)
        checkRange(e: e2)

        let p1 = find(e: e1)
        let p2 = find(e: e2)
        if p1 == p2 {
            return
        }

        for i in 0..<parents.count {
            if parents[i] == p1 {
                parents[i] = p2
            }
        }
    }
```

- 过程
    - find: 查找当前节点的父节点，即为根结点
    - union:遍历所有节点找出与当前节点父节点相同的节点，将所有符合条件的节点父节点指向目标节点
- 时间复杂度:
    - find:时间复杂度O(1)
    - Union:时间复杂度O(n) 优缺点:查找快速，合并时需要遍历数组将目标节点指向目标父节点

2.QuickUnion

```
override func find(e: Int) -> Int {
        checkRange(e: e)

        var v = e
        while parents[v] != v {
            v = parents[v]
        }
        return v
    }

    override func union(e1: Int, e2: Int) {
        checkRange(e: e1)
        checkRange(e: e2)

        let p1 = find(e: e1)
        let p2 = find(e: e2)
```

```
        if p1 == p2 {
            return
        }
        parents[p1] = p2
    }
}
```

- 过程
  - find: 类似于二叉树查找根节点，找到父节点为自己的节点即为根节点
  - union:找到当前节点的根节点，将该节点的根节点指向目标节点的根节点，中间节点不变。
- 时间复杂度:
  - find:时间复杂度O(logn)
  - Union:时间复杂度O(logn) 优缺点:树形结构，查找和联合时间复杂度全部时O(logn)，当数据出现极度不平衡的情况下可能退化成链表

3.QuickUnion根据Size进行优化 - 将需要联合的节点的树的数量进行统计，将节点数量少的树嫁接到数量多的树上

```
    /// 记录节点数量
    var sizes = [Int]()

    override init(count: Int) {
        super.init(count: count)
        for _ in 0..<count {
            sizes.append(1)
        }
    }

override func union(e1: Int, e2: Int) {
        checkRange(e: e1)
        checkRange(e: e2)

        let p1 = find(e: e1)
        let p2 = find(e: e2)

        if p1 == p2 {
            return
        }

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

- 过程
  - find: 同QuickUnion
  - union:判断需要联合的两个节点的树的节点数量，将数量少的按照QuickUnion嫁接到数量多的树上
- 时间复杂度:
  - find:时间复杂度O(logn)
  - Union:时间复杂度O(logn) 优缺点:树形结构，查找和联合时间复杂度全部时O(logn)，当数据出现极度不平衡的情况下树的高度仍不会平衡。

4.QuickUnion根据Rank进行优化 - 将需要联合的节点的树的高度进行统计，将节点数量少的树嫁接到数量多的树上

```
    /// 记录树的高度
    var ranks = [Int]()

    override init(count: Int) {
        super.init(count: count)
        for _ in 0..<count {
            ranks.append(1)
        }
    }
```

```
override func union(e1: Int, e2: Int) {
        checkRange(e: e1)
        checkRange(e: e2)

        let p1 = find(e: e1)
        let p2 = find(e: e2)
        if p1 == p2 {
            return
        }

        if ranks[p1] < ranks[p2] {
            parents[p1] = p2
        }else if ranks[p1] > ranks[p2]{
            parents[p2] = p1
        }else{
            parents[p1] = p2
            ranks[p2] += 1
        }
    }
```

- 过程
  - find: 同QuickUnion
  - union:判断需要联合的两个节点的树的节点高度，将高度低的按照QuickUnion嫁接到高度高的树上
- 时间复杂度:
  - find:时间复杂度O(logn)
  - Union:时间复杂度O(logn) 优缺点:树形结构，查找和联合时间复杂度全部时O(logn)，基于高度进行优化，可使树的高度尽可能的平衡

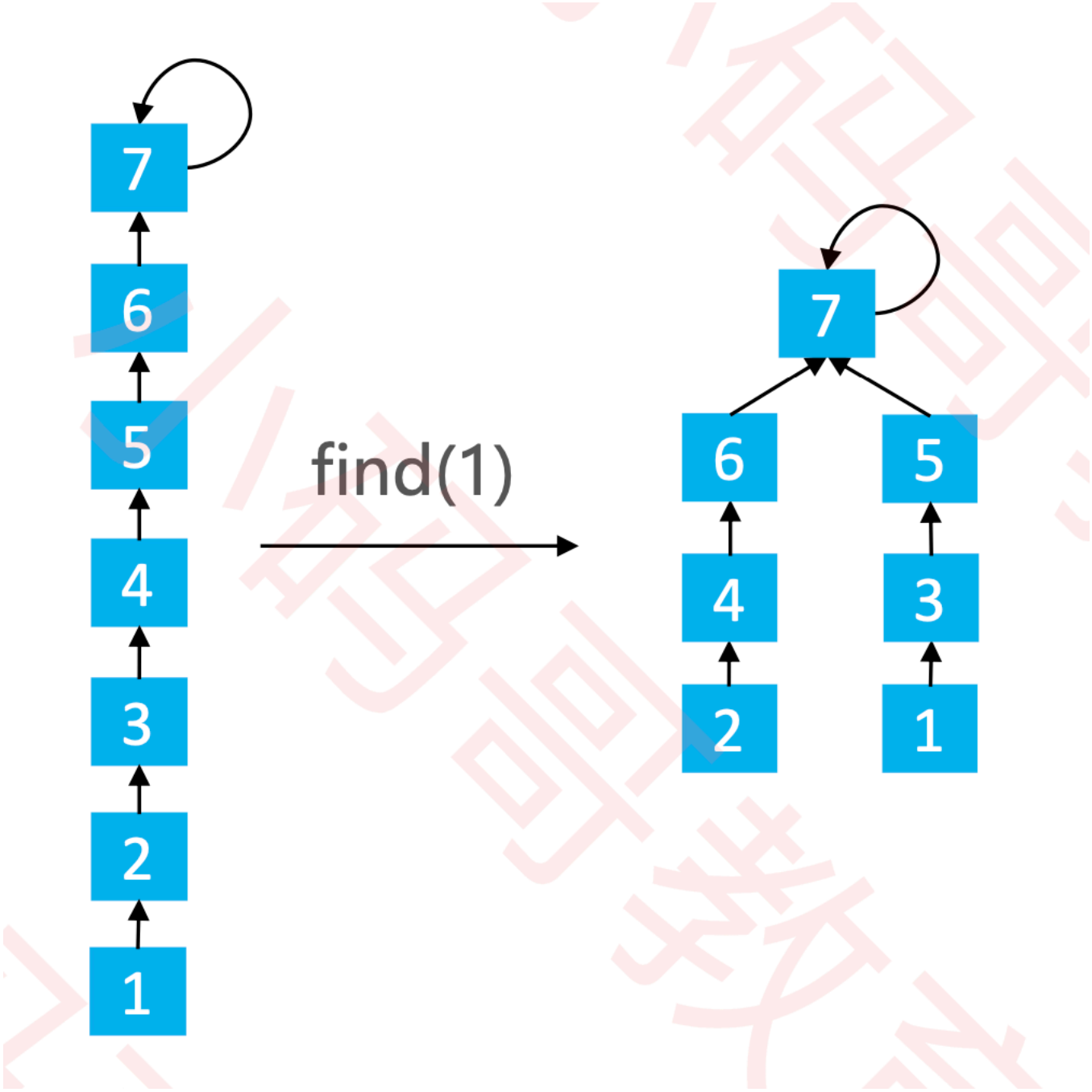
4.QuickUnion根据Rank进行优化且使用路径压缩 - 在树变高之后，会使find操作变慢，该操作将find所有节点都指向根节点，从而降低树高

```
override func find(e: Int) -> Int {
        checkRange(e: e)

        if parents[e] != e {
            parents[e] = find(e: parents[e])
        }

        return parents[e]
    }
```

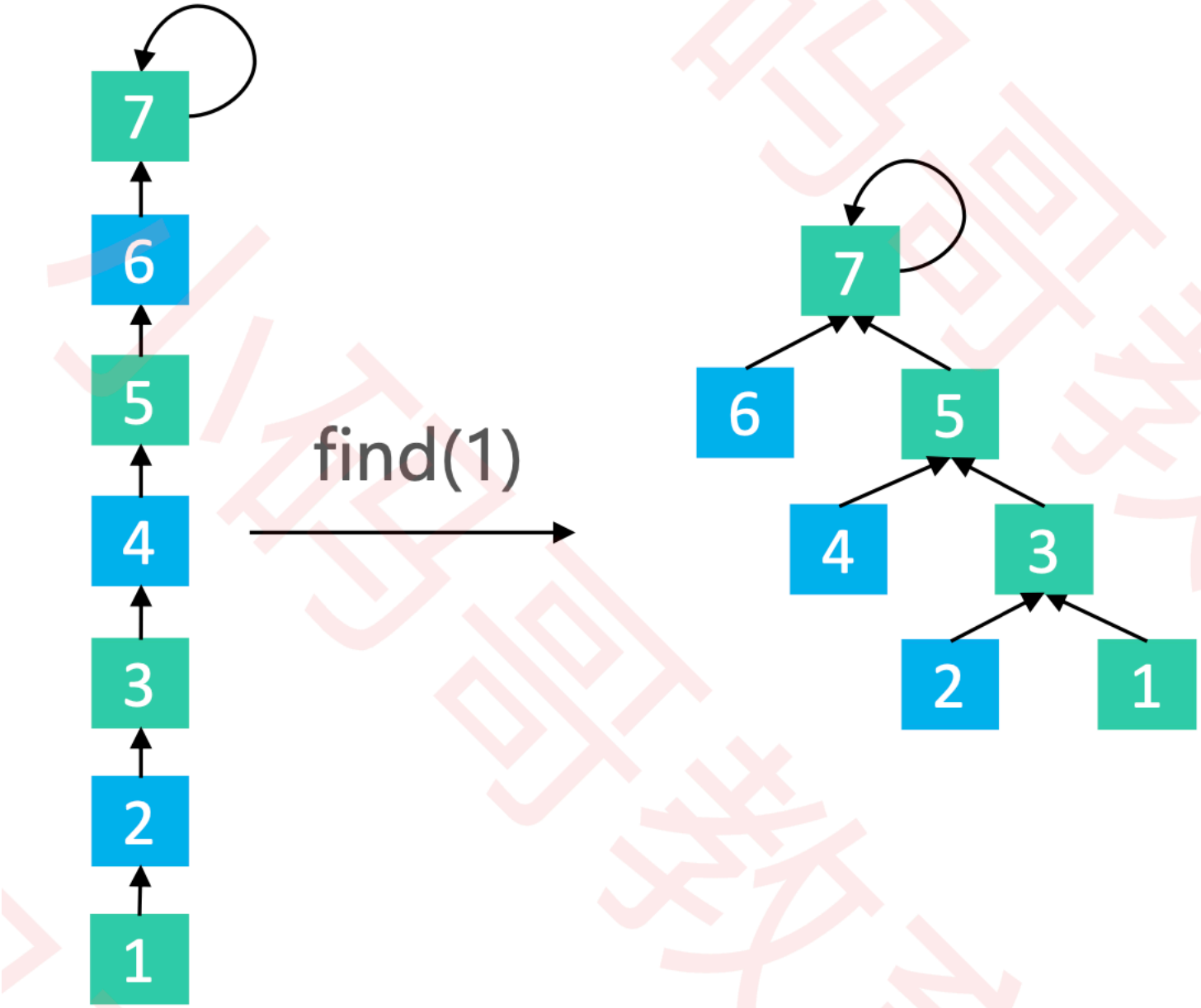
5.QuickUnion根据Rank进行优化且使用路径分裂 - 使用4的操作会递归调用find，实现成本较高，路径分裂会使每个节点指向其祖父节点




```
override func find(e: Int) -> Int {
        checkRange(e: e)
        var v = e

        while parents[v] != v {
            let f = parents[v]
            parents[v] = parents[f]
            v = f
        }
        return v
    }
```

5.QuickUnion根据Rank进行优化且使用路径减半 - 使用4的操作会递归调用find，实现成本较高，路径减半会每隔一个节点指向其祖父节点




```
override func find(e: Int) -> Int {
        checkRange(e: e)

        var v = e

        while parents[v] != v {
            parents[v] = parents[parents[v]]
            v = parents[parents[v]]
        }

        return v
    }
```

在使用QuickUnionRank基于路径减半或分裂优化后，可是时间复杂度降低至O(a(n)),其中a(n) < 5

泛型下的并查集

非整形无法使用数组作为存储结构，可以使用链表+映射来处理并查集的问题

- 每个元素初始化时封装成node对象并自己成为一个集合

```
class UFNode<E: Equatable>: Equatable {

        var parent:UFNode?
        var value:E
        var rank:Int


        init(value:E) {
            self.value = value
            rank = 1
        }

        static func == (lhs: UFNode<E>, rhs: UFNode<E>) -> Bool {
            return lhs.value == rhs.value
        }
    }

    func makeSet(e:E){
        let node = UFNode(value: e)
        node.parent = node
        map[e] = node
    }
```

- 联合

```
func union(e1:E,e2:E){
        guard findNode(e: e1) != nil && findNode(e: e2) != nil else {
            return
        }
        let p1 = findNode(e: e1)!
        let p2 = findNode(e: e2)!
        if p1.value == p2.value  { return }

        if p1.rank < p2.rank {
            p1.parent = p2
        }else if p1.rank > p2.rank{
            p2.parent = p1
        }else{
            p1.parent = p2
            p2.rank += 1
        }
    }
```

- 查询是否在同一集合：即根节点相同

```
func isSame(e1:E,e2:E) -> Bool{
        return findNode(e: e1) == findNode(e: e2)
    }
```

- 查找链表根节点

```
/// 返回元素所在链表的根节点
    /// - Parameter e: <#e description#>
    /// - Returns: <#description#>
    private func findNode(e:E) -> UFNode<E>?{
        guard changeToNode(e: e) != nil else {
            return nil
        }
        var node = changeToNode(e: e)!

        while node.parent!.value != node.value {
            node.parent = node.parent!.parent!
            node = node.parent!
        }

        return node
    }
```