并查集总结

一、并查集是什么：并查集是对一堆具有相互关联的数据中快速找出两个对象是否具有关联关系的数据集合，这个问题看似简单实际上牵扯到大量的计算。

二、并查集的解决思路：

a、先初始化一个数组，然后再让这个数组的内容指向本身。

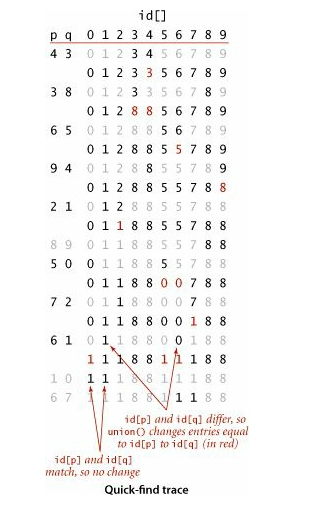
b、若这个数组某个下标与某个下标具有关联性，那么就讲某个下标的内容指向另一个下标。

解决这个问题有3（4）种解决思路。

**1、quick find: 顾名思义；就是快速查找的意思，它的优点就是快速判定两个元素是否具有关联性。**

原理：就是将具有关联关系的元素内容改为相同的一个元素即可。

图：



代码：

**public** **class** QuickFind {

**int**[] unionSet;

**int** count;

**public** QuickFind(**int** count) {

**this**.count = count;

unionSet = **new** **int**[count];

**for** (**int** i = 0; i < unionSet.length; i++) {

unionSet[i] = i;

}

}

**void** union(**int** p1, **int** p2) {

**int** p1id = unionSet[p1];

**int** p2id = unionSet[p2];

**if** (p1id == p2id)

**return**;

//缺点就在这里，每次插入的时候都要对这个数组进行一次遍历，消耗大量的时间

**for** (**int** i = 0; i < count; i++) {

**if** (p1id == unionSet[i]) {

unionSet[i] = p2id;

}

}

}

//优点就在这里，只需要倡廉时间就可以快速查出关联关系

**boolean** connected(**int** p1, **int** p2) {

**return** unionSet[p1] == unionSet[p2];

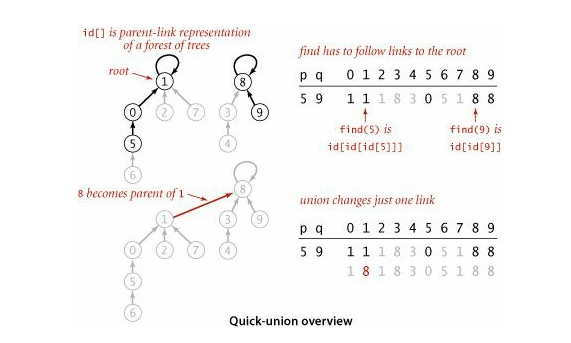
}

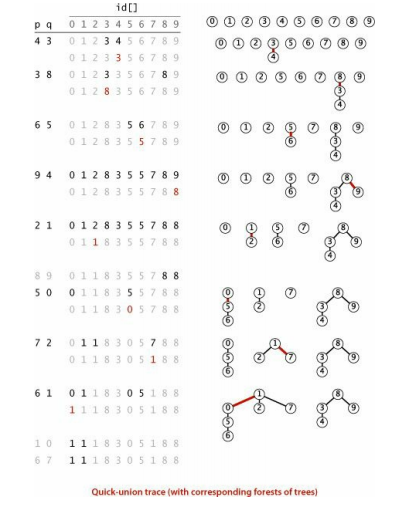
}

**2 quick union 顾名思义，就是快速建立关联关系**

这里的原理是，首先将每个点都建立在指向自身的基础上，以后每次添加一个点就使其指向上个节点。

图：





代码：

**public** **class** QuickUnion {

**int**[] unionSet;

**public** QuickUnion(**int** count) {

unionSet = **new** **int**[count];

**for** (**int** i = 0; i < count; i++) {

unionSet[i] = i;

}

}

//缺点 每次查询关联关系都要寻找根节点

**boolean** connected(**int** p1, **int** p2) {

**int** p1Root = getRoot(p1);

**int** p2Root = getRoot(p2);

**return** p1Root == p2Root;

}

//优点 快速建立关联关系，而非遍整个数组

**void** union(**int** p1, **int** p2) {

**int** p1Root = getRoot(p1);

**int** p2Root = getRoot(p2);

unionSet[p1Root] = unionSet[p2Root];

}

**int** getRoot(**int** p) {

**while** (unionSet[p] != p)

p = unionSet[p];

**return** p;

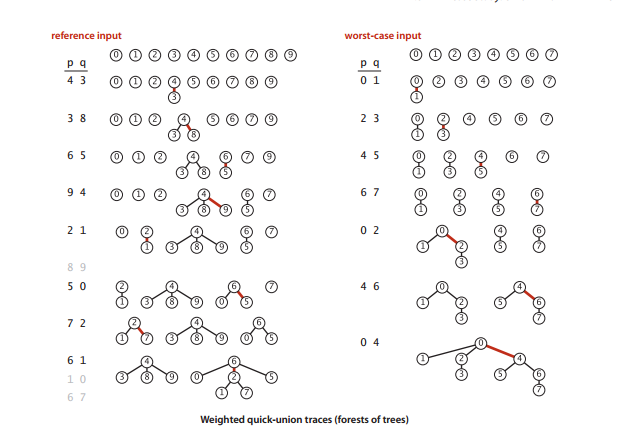
}

}

3 unionfind 顾名思义：它结合了 1 2两种方法的优点。

实现思路，再建立一个数组，用来记录当前节点数的数目，这个就尽量避免了高度的快速增长。

图：



代码：

**public** **class** UnionFind {

**int**[] sz;

**int**[] unionSet;

**public** UnionFind(**int** count) {

sz = **new** **int**[count];

unionSet = **new** **int**[count];

**for** (**int** i = 0; i < count; i++) {

unionSet[i] = i;

sz[i] = 1;

}

}

**void** union(**int** p, **int** q) {

**int** pid = root(p);

**int** qid = root(q);

**if** (pid == qid)

**return**;

//优点 ，判断子节点的数目，将小节点添加到大节点上，缺点就是没有办法避免高度 //的增长。

**if** (sz[pid] > sz[qid]) {

unionSet[qid] = pid;

sz[pid] += sz[qid];

} **else** {

unionSet[pid] = qid;

sz[qid] += sz[pid];

}

}

**boolean** connected(**int** p, **int** q) {

**return** root(p) == root(q);

}

**int** root(**int** p) {

**while** (unionSet[p] != p)

p = unionSet[p];

**return** p;

}

}

4 我们可以对 3 进行优化，优化的地方在于每次遍历的时候发现根节点过长，就将根节点上移，减小根节点的遍历深度。

**public** **class** UnionFind2 {

**int**[] sz;

**int**[] unionSet;

**public** UnionFind2(**int** count) {

sz = **new** **int**[count];

unionSet = **new** **int**[count];

**for** (**int** i = 0; i < count; i++) {

unionSet[i] = i;

sz[i] = 1;

}

}

**void** union(**int** p, **int** q) {

**int** pid = root(p);

**int** qid = root(q);

**if** (pid == qid)

**return**;

**if** (sz[pid] > sz[qid]) {

unionSet[qid] = pid;

sz[pid] += sz[qid];

} **else** {

unionSet[pid] = qid;

sz[qid] += sz[pid];

}

}

**boolean** connected(**int** p, **int** q) {

**return** root(p) == root(q);

}

**int** root(**int** p) {

**while** (unionSet[p] != p) {

// This is a improvement 这里就是优化点

// beccause it use last resul

// it change the parant to the root it will shorter the path

unionSet[p] = unionSet[unionSet[p]];

p = unionSet[p];

}

**return** p;

}

}

下面是自己写的测试数据

使用随机插入500000个数据，然后随机查询500000次

对照结果

1 quick find 用时196768 毫秒

2 quick union用时845483毫秒

3 union find 用时128 毫秒

4 union find 的优化用时103毫秒

性能差距如此之大，可见往往更快的电脑硬件不能解决性能上的瓶颈。所以说我们已改从算法上对其优化