3

Programming 2 Novel Union find

Running cost:

Find: time proportional to depth of p and q

Union: takes) constant time, given roots

Depth of any node x is at most $\lg N$.

	Initialize	Union	Find (worst case)
Quick find	N	N	1
Quick union	N	N *	N
Weight QU	N	$\lg N$ *	$\lg N$

* includes cost of finding roots

Path compression:

QU with PC: just after computing the root of P, set the id of each examined node to point to that root.

Give FIND a side job of compressing the tree

~~Singular~~ One-pass variant: Make every other node in path point to its grandparent (halving path length)

```
private int root(int i) {
    while (i != id[i]) {
        id[i] = id[id[i]]; ← one extra line
        i = id[i];
    }
    return i;
}
```

SUMMARY:

	Worst case time
Quick find	MN
Quick Union	MN
Weighted QU	$N + M \log N$
QU + path compression	$N + M \log N$
Weighted QU + path compr.	$N + \log^* N$

1.

7/4/14. # Programming 2 Notes - Union Find

- Given a fixed number of N elements
- Each element belongs to exactly one set.
- We want to be able to union sets

Union Find interface

UnionFind(N) initialize a union-find data structure with N-elements each numbered 0 .. N-1

Void union(int x, int y) unions the sets of x and y.

int Find(int x) returns the REPRESENTATIVE ELEMENT of the set where x is in

boolean connected(int x, int y) returns true when x is connected to y

Main idea of Array Implementation

- Element x is represented by position x in an array id[].
- Connection from x to y
  - → id[x] = y
  - → NOTE: don't need to record id[y] = x
  - → Note: Direction of connection unimportant.
  - → No connects ⇒ id[x] = x

element:	0	1	2	3	4	5
id[]	0	1	2	4	4	2

Interpretation id[i] is parent of i.

```
private int root(int i) {
    while (i != id[i]) {
        i = id[i];
    }
    return i;
}
```

2.

```
public void union (int p, int q) {
    int i = root (p);
    int j = root(q);
    id[i] = j;
```

Cost

	initialize	union	Find
Quick find	N	N	1
Quick Union	N	N	N

Quick find defect
- Union too expensive
- trees are flat → expensive

Quick union defect
- trees get tall
- find too expensive

Weighted quick union
- Modify quick union to avoid tall trees
- keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree

Data Structure: Same as quick union, but maintain extra array sz[i] to count number of objects in the tree rooted at i.

union method:

```
int rootP = find(p).
int rootQ = find(q))
if (rootP == rootQ) { return };
if (sz [rootP] < sz[rootQ]){
    id [rootP] = rootQ;
    sz [rootQ] += sz [rootP];
} else {
    id[rootQ] = rootP;
    sz[rootP] += sz[rootQ];
}
```