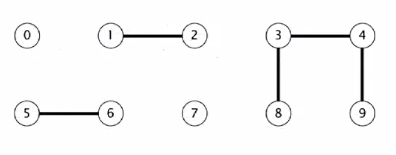
**Union Find**

Dynamic connectivity

Explained: dynamic connectivity is whether or not there is a link between multiple points. For example, there are several points below and three connections of more than one point (there are 5 connected components).

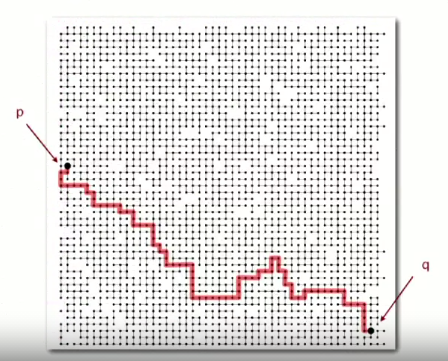


You need to:

1. Union command (connect objects)
2. Find (is there a path connecting objects)

Note that connections are not necessarily direct connections. In fact, 8 is connected to 9. We can see that 8 -> 3 -> 4 -> 9. Using computers to find these connections is extremely useful when the problem is larger. Connections are:

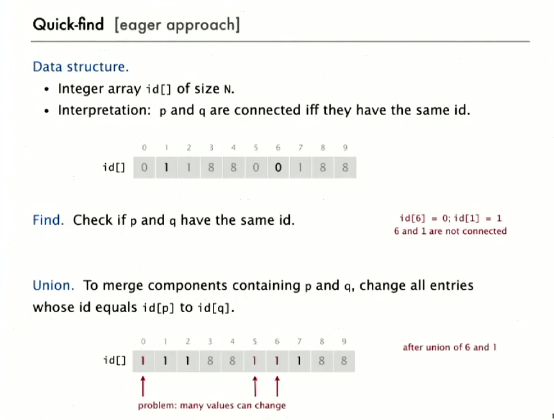
* Reflexive (p is connected to p)
* Symmetric (if p is connected to q, then q is connected to p)
* Transitive (if p is connected to q and q is connected to r, p is connected to r)



Quick Find

**Explanation:**

The structure is an integer array indexed by object where P and Q are connected iff they have the same id. To merge components containing p and q, all entries whose id equals id[p] must change to id[q].



**Why does it exist?**

The reason for this data structure is to determine whether there is any connection between two points.

This can be used in computer networks, social networks, computer chips (circuit elements), variable names and other types of objects. With programming we associate objects with a names and those are integers 0 – N-1, with integers as indices in an array. This can work well with searching algorithms.

**What is this optimized for?**

This is optimized to find links between different objects, i.e.:

* Access
* Search

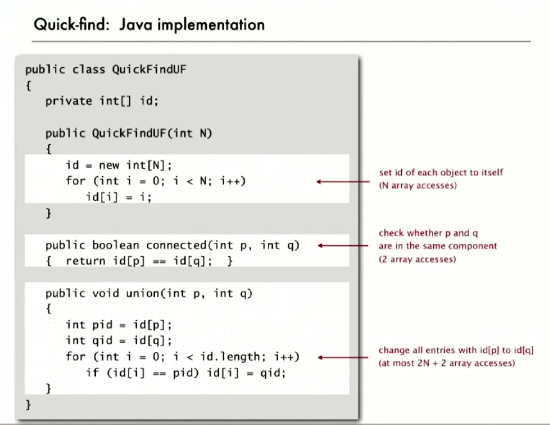
**The weaknesses:**

Unions are very expensive. Therefore the algorithm is very slow and can easily have **quadratic time** (N union commands on N objects is N2 time), which is far too small (especially for large problem); quadratic algorithms do not scale.

**Function time costs**

|  |  |
| --- | --- |
| Function | Time |
| Initialize | N |
| Union | N |
| Find | 1 |

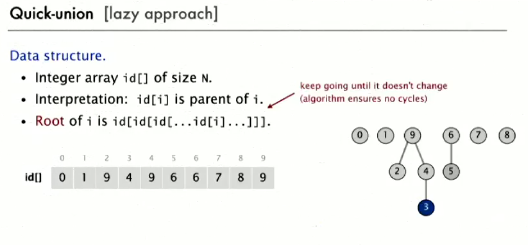
**Java implementation:**



Quick Union

**Explanation:**

This uses the same data structure of the array, but each entry contains a reference to its parent in the tree. Base roots are self-referential, but everything else leads back to a root. To merge different components, simply change the id of p’s root to the id of q’s root.



This means that unions are therefore faster because instead of having to change the indices of all objects in the case of a union, only one index in changed.

**What is this optimized for?**

This improves upon Quick Find to increase the speed of unions.

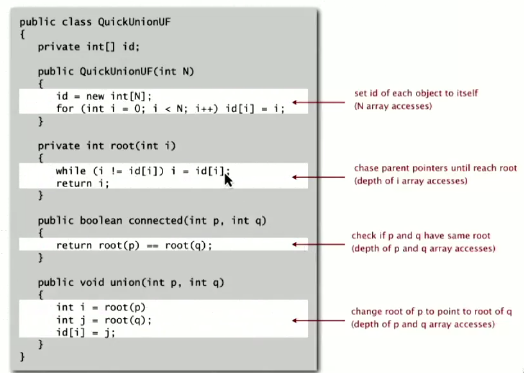
**The weaknesses:**

There are similar weaknesses to quick find, except that unions are faster and finds cost too many array accesses (too expensive); the trees can become very tall.

**Function time costs**

|  |  |
| --- | --- |
| Function | Time |
| Initialize | N |
| Union | N (must find roots) |
| Find | N |

**Java implementation:**



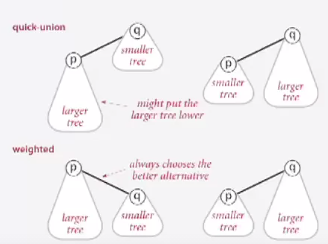
Weighted Quick Union

**Explanation**

Track the size of each tree and link the smaller tree to the larger tree for the sake of balance (minimize time cost of find). The data structure is the same, but must also maintain an array to count number of objects rooted at i.

**What is this optimized for?**

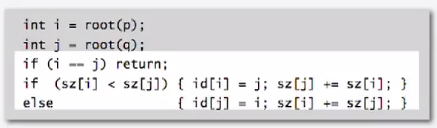
This is to keep the tree balanced so that no trees become too tall. Note the following example:



**Java implementation**

Updates:

* Link the smaller tree’s root to the larger tree’s root
* Update the ‘size’ array.

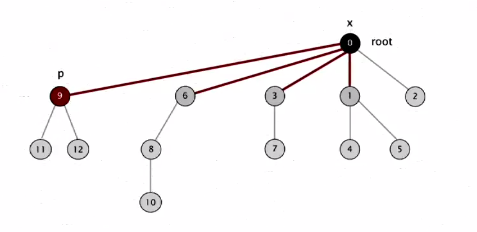
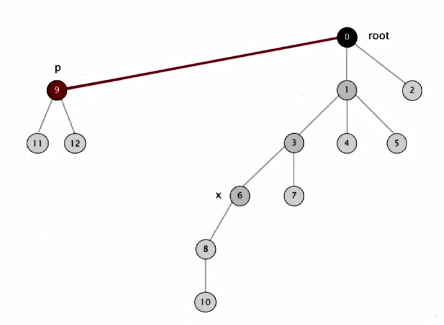
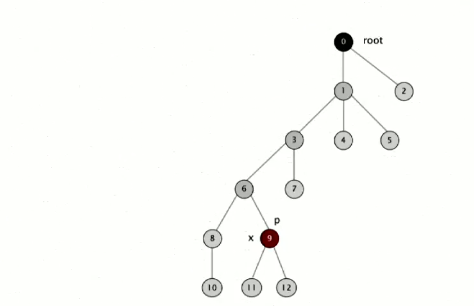


|  |  |
| --- | --- |
| Function | Time |
| Initialize | N |
| Find | Log N |
| Union | Log N |

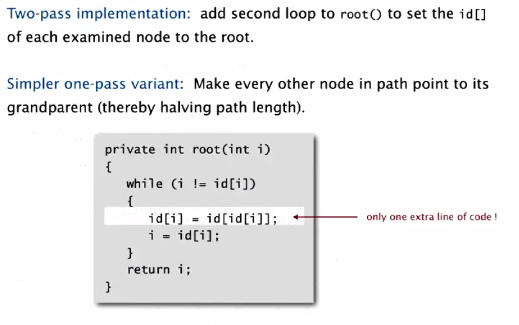
Find and Union times are proportional to the depth of ‘p’ and ‘q.’ The depth of any node x is at most log N. X’s depth only increases when its tree is merged into another tree, which only happens if that tree is larger than/ equal to X’s tree.

Quick Union with Path Compression

After computing the root of p, set the id of each examined node to point to that root (and other roots on that path).



Only adds one line of code.

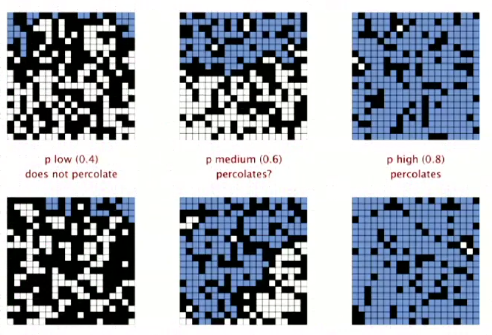


**Summary of Big O**

|  |  |
| --- | --- |
| Algorithm | Worst-Case Time |
| Quick-find | N |
| Quick-union | N |
| Weighted QU | log N |
| QU + path compression | log N |
| Weighted QU + path compression | log\* N |

Union-Find Applications

* Percolation (does one point at the top connect to any point at the bottom?):
  + Water flowing through a porous material
  + Social network connections
  + Electricity through conductors
* Percolation visualized with different levels of percolation:



Depending on the probability of percolation, there is significant percolation or no percolation. There is a small threshold where there is some percolation, but not significant percolation.

This cannot be found with any mathematical model, but fast union-find algorithms can make a close approximation.

How can we use our dynamic connectivity model to do this?

