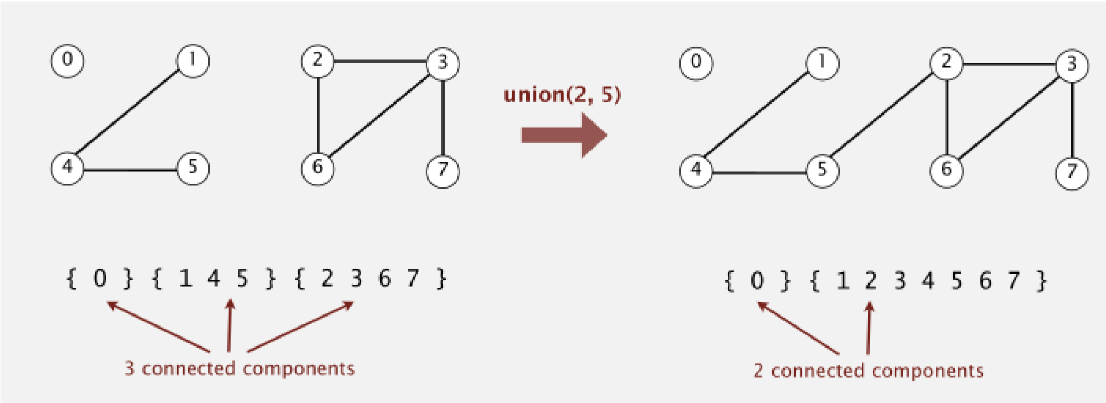
1、 Union-find

Goal. Design efficient data structure for union-find.

- Number of objects N can be huge.
- Number of operations M can be huge.
- Find queries and union commands may be intermixed.



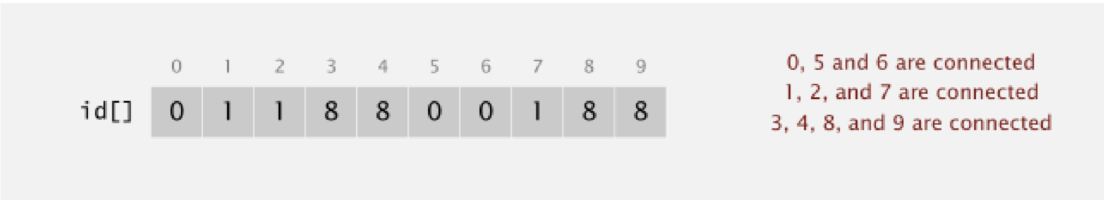
Union-find data type API:

```
public class UF{
    //initialize union-find data structure
    //with N objects (0 to N - 1)
    UF(int N)
    //add connection between p and q
    void union(int p, int q)
    //are p and q in the same component?
    boolean connected(int p, int q)
    //component identifier for p (0 to N - 1)
    int find(int p)
    //number of components
    int count()
}
```

Quick-find

Data structure:

- Integer array id[] of length N.
- Interpretation: p and q are connected iff they have the same id.



Find: check if p and q have the same id.

Union: To merge components containing p and q, change all entries whose id equals id[p] to id[q].

Java implementation:

```java
public class QuickFindUF {
    private int[] id;

    public QuickFindUF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public boolean connected(int p, int q) {
        return id[p] == id[q];
    }

    public void union(int p, int q) {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++)
            if (id[i] == pid)
                id[i] = qid;
    }
}
```

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	N	N	1

order of growth of number of array accesses

Union is too expensive. It takes N^2 array accesses to process a sequence of N union commands on N objects.

Quick-union

Data structure. (Tree)

- Integer array id[] of length N.
- Interpretation: id[i] is parent of i.
- Root of I is id[id[...id[i]...]].

Find. Check if p and q have the same root.

Union. To merge components containing p and q, set the id of p's root to the id of q's root.

Java implementation.

```java
public class QuickUnionUF {
    private int[] id;

    public QuickUnionUF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

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

    public boolean connected(int p, int q) {
        return root(p) == root(q);
    }

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

Cost model. Number of array accesses (for read or write)

algorithm	initialize	union	find
quick-find	N	N	1
quick-union	N	N †	N

← worst case

† includes cost of finding roots

Quick-find defect.

- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

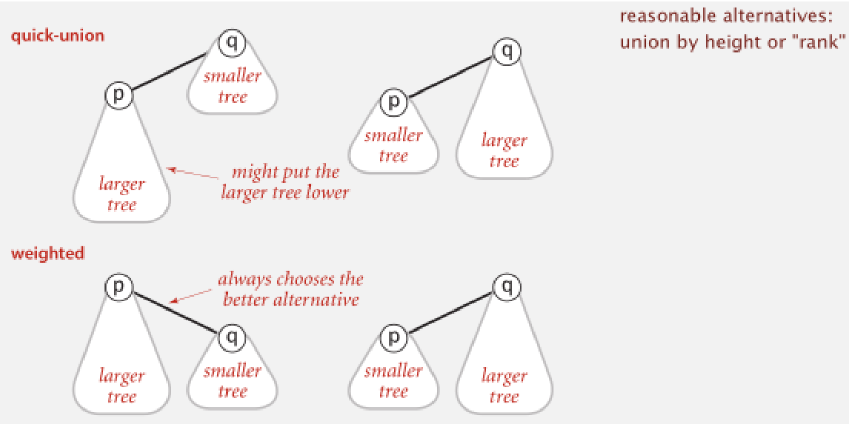
Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N array accesses).

Improvement 1: weighting

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.



Running time.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, give roots.

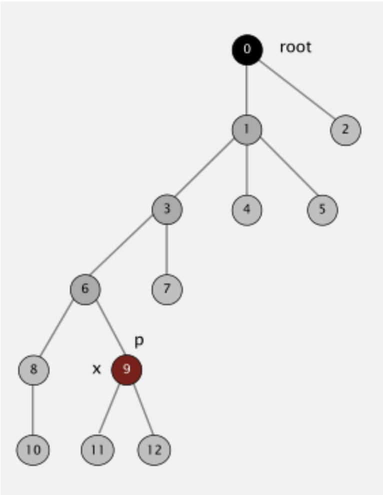
Proposition. Depth of any node x is at most lgN.

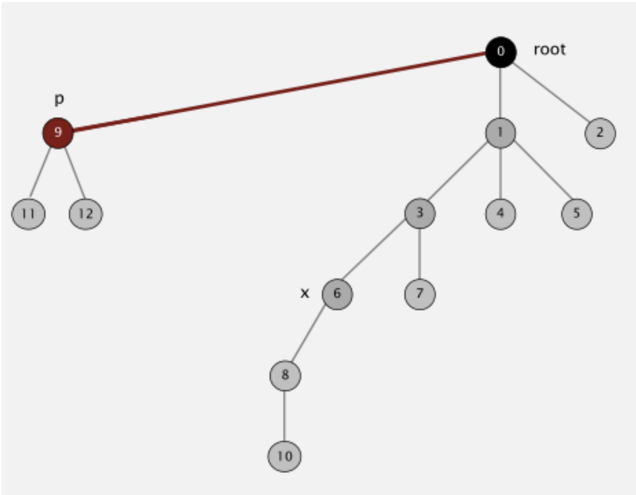
algorithm	initialize	union	connected
quick-find	N	N	1
quick-union	N	N †	N
weighted QU	N	lg N †	lg N

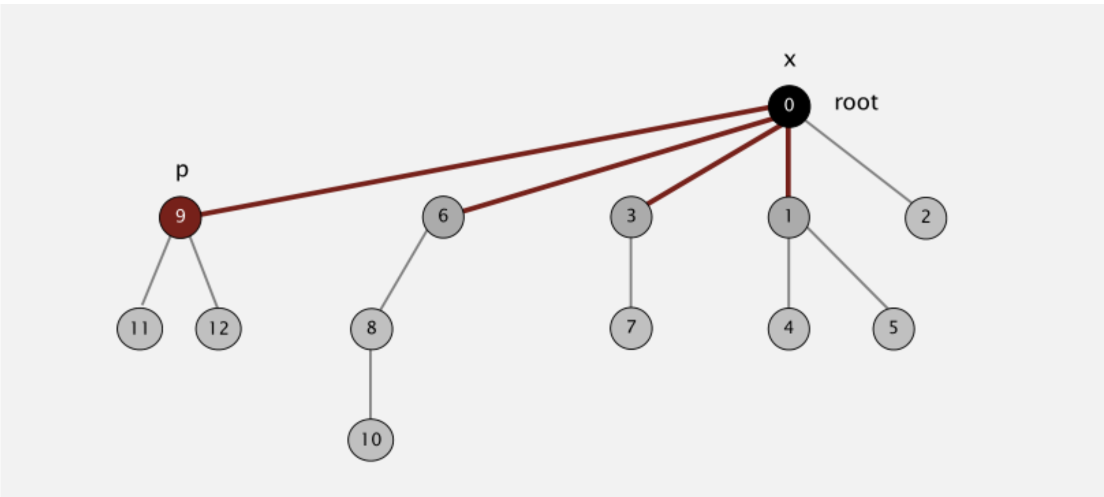
† includes cost of finding roots

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.







Java implementation

```java
private int root(int i){
   while (i != id[i]){
      //only one extra line of code
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

Summary

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

Application:

- Percolation.
- Games (Go, Hex).
- ✓ Dynamic connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's bwlabel() function in image processing.