Partitions with Union-Find Operations

- **makeSet**(x): Create a singleton set containing the element x and return the position storing x in this set
- **union**(A, B): Return the set A \(\cup\) B, destroying the old A and B
- **find**(p): Return the set containing the element at position p
List-based Implementation

- Each set is stored in a sequence represented with a linked-list.
- Each node should store an object containing the element and a reference to the set name.

Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set.
  - Each time an element is moved it goes to a set of size at least double its old set.
  - Thus, an element can be moved at most O(log n) times.
- Total time needed to do n unions and finds is O(n log n).
Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name.
- A node $v$ whose set pointer points back to $v$ is also a set name.
- Each set is a tree, rooted at a node with a self-referencing set pointer.
- For example: The sets “1”, “2”, and “5”:

```
1
   4
   7

2
   3
   6

6
   8
   10
   9
   11
   12
```

Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other.
- To do a find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself.

```
2
   3
   6
   9

5
```

© 2014 Goodrich, Tamassia, Goldwasser
Union-Find Heuristic 1

- **Union by size:**
  - When performing a `union`, make the root of smaller tree point to the root of the larger
- **Implies $O(n \log n)$ time for performing $n$ union-find operations:**
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most $O(\log n)$ pointers for any `find`.

Union-Find Heuristic 2

- **Path compression:**
  - After performing a `find`, compress all the pointers on the path just traversed so that they all point to the root
- **Implies $O(n \log^* n)$ time for performing $n$ union-find operations:**
  - Proof is somewhat involved… (and not in the book)
Java Implementation

```java
/** A Union-Find structure for maintaining disjoint sets. */
public class Partition<E> {

    // nested Locator class
    private class Locator<E> implements Position<E> {
        public E element;
        public int size;
        public Locator<E> parent;

        public Locator(E elem) {
            element = elem;
            size = 1;
            parent = this; // convention for a cluster leader
        }

        public E getElement() { return element; }
    }

    /** Makes a new cluster containing element e and returns its position. */
    public Position<E> makeCluster(E e) {
        return new Locator<E>(e);
    }
}
```

Java Implementation, 2

```java
/** Finds the cluster containing the element identified by Position p 
and returns the Position of the cluster's leader. */
public Position<E> find(Position<E> p) {
    Locator<E> loc = validate(p);
    if (loc.parent != loc)
        loc.parent = (Locator<E>) find(loc.parent); // overwrite parent after recursion
    return loc.parent;
}

/** Merges the clusters containing elements with positions p and q (if distinct). */
public void union(Position<E> p, Position<E> q) {
    Locator<E> a = (Locator<E>) find(p);
    Locator<E> b = (Locator<E>) find(q);
    if (a != b)
        if (a.size > b.size) {
            b.parent = a;
            a.size += b.size;
        } else {
            a.parent = b;
            b.size += a.size;
        }
}
```
Proof of log* n Amortized Time

- For each node v that is a root:
  - Define n(v) to be the size of the subtree rooted at v (including v).
  - Identified a set with the root of its associated tree.
- We update the size field of v each time a set is unioned into v. Thus, if v is not a root, then n(v) is the largest the subtree rooted at v can be, which occurs just before we union v into some other node whose size is at least as large as v's.
- For any node v, then, define the rank of v, which we denote as r(v), as r(v) = \lceil \log n(v) \rceil.
- Thus, n(v) \geq 2^{r(v)}.
- Also, since there are at most n nodes in the tree of v, r(v) = \lceil \log n \rceil, for each node v.

Proof of log* n Amortized Time (2)

- For each node v with parent w:
  - r(v) > r(w)
- Claim: There are at most n/2^s nodes of rank s.
- Proof:
  - Since r(v) < r(w), for any node v with parent w, ranks are monotonically increasing as we follow parent pointers up any tree.
  - Thus, if r(v) = r(w) for two nodes v and w, then the nodes counted in n(v) must be separate and distinct from the nodes counted in n(w).
  - If a node v is of rank s, then n(v) \geq 2^s.
  - Therefore, since there are at most n nodes total, there can be at most n/2^s that are of rank s.
Proof of log* n Amortized Time (3)

Definition: Tower of two’s function:
- \( t(i) = 2^{t(i-1)} \)

Nodes \( v \) and \( u \) are in the same rank group \( g \) if
- \( g = \log*(r(v)) = \log*(r(u)) \)

Since the largest rank is \( \log n \), the largest rank group is
- \( \log*(\log n) = (\log* n) - 1 \)

Proof of log* n Amortized Time (4)

Charge 1 cyber-dollar per pointer hop during a find:
- If \( w \) is the root or if \( w \) is in a different rank group than \( v \), then charge the find operation one cyber-dollar.
- Otherwise (\( w \) is not a root and \( v \) and \( w \) are in the same rank group), charge the node \( v \) one cyber-dollar.

Since there are most \( (\log* n)-1 \) rank groups, this rule guarantees that any find operation is charged at most \( \log* n \) cyber-dollars.
Proof of log* n Amortized Time (5)

- After we charge a node v then v will get a new parent, which is a node higher up in v’s tree.
- The rank of v’s new parent will be greater than the rank of v’s old parent w.
- Thus, any node v can be charged at most the number of different ranks that are in v’s rank group.
- If v is in rank group g > 0, then v can be charged at most $t(g) - t(g-1)$ times before v has a parent in a higher rank group (and from that point on, v will never be charged again). In other words, the total number, $C$, of cyber-dollars that can ever be charged to nodes can be bounded by

$$C \leq \sum_{g=0}^{\log^* n} n(g) \cdot (t(g) - t(g-1))$$

Bounding $n(g)$:

$$n(g) \leq \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s}$$

$$= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s}$$

$$< \frac{n}{2^{t(g-1)+1}} \cdot 2$$

$$= \frac{n}{2^{t(g-1)}}$$

$$= \frac{n}{t(g)}$$

Returning to C:

$$C < \sum_{g=1}^{\log^* n} \frac{n}{t(g)} \cdot t(g)$$

$$\leq \sum_{g=1}^{\log^* n} \frac{n}{t(g)} \cdot t(g)$$

$$= \sum_{g=1}^{\log^* n} n$$

$$\leq n \log^* n$$