1 The basic data structure

The union-find data structure maintains a collection of disjoint sets and supports the following three operations:

- MAKESET(x) - create a new set containing the single element x.
- UNION(x,y) - replace the two sets containing x and y by their union.
- FIND(x) - return the name of the set containing the element x. For our purposes this will be a canonical element in the set containing x.

We will represent each set by a tree, where each element has a pointer to its parent in the tree. The root points to itself, and is the canonical element (or name) of the set. It is convenient to add a fourth operation LINK(x,y) where x and y are required to be canonical elements. LINK changes the parent pointer of one of these elements, say x, and makes it point to y. It returns the root of the composite tree y. Then UNION(x,y) = LINK(FIND(x), FIND(y)).

We must be careful in carrying out the LINK operation so that the new tree remains roughly balanced. This is because FIND(x) requires time proportional to the length of the path from x to the root of the tree. The balancing heuristic we shall use is called UNION BY RANK. For each element we have a number that we call its rank. Intuitively, the rank is the height of this element in the tree, the length of the longest path from it to a leaf. The rank of x is initialized to 0 by MAKESET. Rank is only updated by the operation LINK as follows: if x and y have the same rank r, then invoking LINK(x,y) causes the parent pointer of x to be updated to y, and the rank of y to be updated to r+1. On the other hand, if LINK is invoked on two elements x and y of different rank, then the parent pointer of the smaller rank element is updated to point to the larger rank element. We shall show below that, thanks to UNION BY RANK, the depth of each tree is bounded by $\log n$.

procedure makeset(x)
    p(x) := x
    rank(x) := 0
end
function find(x)
    if $x \neq p(x)$ then $p(x) := \text{find}(p(x))$
    return(p(x))
end

function link(x, y)
    if $\text{rank}(x) > \text{rank}(y)$ then $x \link y$
    if $\text{rank}(x) = \text{rank}(y)$ then $\text{rank}(y) = \text{rank}(y) + 1$
    $p(x) := y$
    return(y)
end

procedure union(x, y)
    link(find(x), find(y))
end

Let us first make a few observations about $\text{rank}$:

- if $v \neq p(v)$ then $\text{rank}(p(v)) > \text{rank}(v)$.
- whenever $p(v)$ is updated, $\text{rank}(p(v))$ increases.
- The number of elements of $\text{rank} k$ is at most $\frac{n}{2^k}$.
- The number of elements of $\text{rank} > k$ is at most $\frac{n}{2^k}$.

We will prove below that any sequence of $m$ UNION and FIND operations on $n$ elements take at most $O((m + n)\log^* n)$ steps, where $\log^* n$ is the number of times you must iterate the log function on $n$ before you get a number less than or equal to 1.

First a few observations about $\text{rank}$:

- if $v \neq p(v)$ then $\text{rank}(p(v)) > \text{rank}(v)$.
- whenever $p(v)$ is updated, $\text{rank}(p(v))$ increases.
- $\text{rank}(v)$ is precisely the height of $v$, i.e., the length of the longest path from $v$ to a leaf.
- At the time an element acquires rank $k$, it is the root of a tree with at least $2^k$ elements.
- The number of elements of $\text{rank} k$ is at most $\frac{n}{2^k}$. 
• The rank of any element is at most $\log n$.

• The number of elements of $\text{rank} > k$ is at most $\frac{n}{2^k}$.

The third and fourth assertions are proved by an easy induction in the rank. They both hold when rank is initialized to 0. And $\text{rank}(y)$ is updated to $k$ only when $\text{rank}(x) = \text{rank}(y) = k - 1$, and $\text{LINK}(x,y)$ causes $x$ to point to $y$. By induction, both trees had at least $2^{k-1}$ elements, and height $k - 1$, and so the overall tree will have at least $2^k$ elements and height $k$. The next assertion is now easy: Since for each element of rank $k$ we have at least $2^k - 1$ elements of lesser rank (the nodes of the tree at the time the element got its rank), the number of rank $k$ elements is at most $\frac{n}{2^k}$. The $\log n$ bound on rank now follows easily: If there is at least one element of rank $k$, it must be that $\frac{n}{2^k} \geq 1$, or $k \leq \log n$. And the last assertion follows by a calculation: the number of elements of $\text{rank} > k$ is at most $\sum_{j=k+1}^{\infty} \frac{n}{2^j} = \frac{n}{2^k}$.

From all this we obtain the following result:

• The union-find data structure has worst-case complexity of any operation $O(\log n)$

This is because the operations of link and makeset are $O(1)$. As for find, its complexity is determined by the length of the path that find has to follow, from $x$ to the root. But this path is at most $\log n$ long.

2 Path Compression

There is another heuristic that will help cut down the time required to carry out FIND operations even further. Whenever we carry out the operation FIND(x), we may as well update x’s parent pointer to point directly to the root of the tree. In fact, we could do better still by updating all the parent pointers in the path from x to the root of the tree. This heuristic, which we will call PATH COMPRESSION, only doubles the time required to carry out this FIND operation, but can potentially save a lot of time in future FIND operations.

The time requirements of union-find with PATH COMPRESSION are related to an extremely slowly-growing function of $n$, $\log^* n$. $\log^* n$ is the number of times you have to take $\lceil \log(\cdot) \rceil$ starting from $n$ until the number becomes one. For $n = 1$, $\log^* n$ is of course 0 — no logs need be taken — and for $n = 2$, $\log^* n = 1$. Then for $n = 3, 4$, $\log^* n = 2$. Then for $n$ in the range 5, \ldots, 16, $\log^* n = 3$. For $n$ in the range 17, \ldots, $2^{16} = 64K$, $\log^* n = 4$. And for any number larger than $64K$ and smaller than $2^{64K}$ — an unimaginably large number, much larger than
the number of nodes of any graph to be stored in a computer—$\log^* n = 5$. In every meaningful sense, $\log^* n$ is at most 5—a constant.

Note that path compression does not affect the rank of an element. Therefore, the rank of the root is what the height of the tree would have been had there been no path compression. As soon as an element becomes a non-root vertex, its rank is forever fixed. Now let us divide the (non-root) elements into groups according to their ranks: we will assign to group $i$ all vertices whose rank $r$ satisfies $\log^* r = i$. Thus each group will consist of all vertices with ranks in the interval $(k, 2^k]$ where $k$ is itself an iterated power of 2 (the first six such intervals were given above).

It is easy to establish the following assertions about these groups:

- The number of distinct groups is at most $\log^* n$.
- The number of elements in the group $(k, 2^k]$ is at most $\frac{n}{2^k}$.

Suppose that we consider a sequence of $m$ FIND operations on these elements. Think that they are executed on a coin-operated computer, which requires one coin for each pointer chased in a FIND operation. Notice that this captures the time requirements of the operations within a constant, because all other operations are at most proportional to the number of pointer chases.

How are we to pay for these pointer chases? Let us give $2^k$ tokens to each element in group $(k, 2^k]$. Then the total number of tokens assigned to all the elements in that group is at most $2^k \frac{n}{2^k} = n$. Moreover, since the total number of groups is at most $\log^* n$, the number of tokens assigned to all elements in all groups is at most $n \log^* n$. Also, each FIND operation is assigned $\log^* n$ tokens. Thus the total number of tokens is $O((m + n) \log^* n)$. We pay for each pointer $(u, v)$ that FIND chases as follows:

- if $u$ and $v$ belong to different rank groups then FIND uses one of its $\log^* n$ tokens to pay for chasing this pointer.
- if $u$ and $v$ belong to the same rank group then $u$ pays using one of its tokens.

The main point here is that each time a FIND operation goes through an element $u$, its parent pointer is changed to the root of the current tree (due to path compression), and therefore $\text{rank}(p(u))$ increases by at least 1. If $u$ is in the group $(k, 2^k]$, then the rank of the parent can increase fewer than $2^k$ times before it moves up to a higher group. Therefore the $2^k$ tokens assigned to $u$ are sufficient to pay for all the FIND operations that go through $u$. 
The tokens assigned to the FIND operations are sufficient to pay for the remaining pointer jumps since there are at most $\log^* n$ groups. Thus the total number of steps for $m$ FIND operations on $n$ elements is bounded by the total number of tokens, which is $O((m + n) \log^* n)$. Finally, since LINK requires $O(1)$ steps, and a UNION is implemented using two FIND operations and a LINK operation, the total time for $m$ UNION and FIND operations on $n$ elements is also $O((m + n) \log^* n)$. 