The Basic Data Structure

This 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*. For now, think of the rank of an element as its height in the tree. i.e. it is 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. This heuristic, called UNION BY RANK, tries to keep the trees short and bushy. We shall show below that, thanks to UNION BY RANK, the depth of each tree is bounded by $\log n$.

Here is the pseudocode for what we've discussed so far:

```
procedure makeset(x)
    p(x) := x
    rank(x) := 0
end
```

```
function find(x)
    if x ≠ p(x) then p(x):= find(p(x))
    return(p(x))
end

function link(x,y)
    if rank(x) > rank(y) then x ↔ y
if rank(x) = rank(y) then rank(y) = rank(y) + 1
p(x) := y
return(y)
end

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

First a few observations about *rank*:

- **Lemma 1:** If $v \neq p(v)$ then $rank(p(v)) > rank(v)$.

  Proof: Note that we update the rank of the root of the tree only; once an element has a parent other than itself, its rank is forever fixed. Now observe that when v's parent pointer was updated to $p(v)$, either $p(v)$ had a higher rank, or they had the same rank and $rankp(v)$ was incremented by one. In either case, $rank(p(v)) > rank(v)$.

- **Lemma 2:** At the time an element acquires rank k, it is the root of a tree with at least 2^k elements (all but the root having rank strictly smaller than k).

  Proof: $rank(v)$ is updated to k only when v is a root vertex, $rank(w) = rank(v) = k-1$, and LINK(v,w) causes w to point to v. By a simple induction on k, after the update the tree rooted at v contains at least 2^k elements. By **Lemma 1** all these vertices except v have rank strictly less than k.

- **Lemma 3:** The number of elements of rank strictly greater than k is at most $\frac{n}{2^k}$.

  Proof: This assertion follows easily from the previous one. For each element v of rank k we have at least $2^k - 1$ elements of lesser rank - the nodes of the tree rooted at v when v got its rank. Since there are n elements in all, there can be only $\frac{n}{2^k}$ such trees each of which contains at most a single element of rank k. Thus, there are at most $\frac{n}{2^k}$ elements of rank k.

The number of elements of rank stricly greater than k is at most:

$$\sum_{j=k+1}^{\infty} \frac{n}{2^j} = \frac{n}{2^k}(1/2 + 1/4 + 1/8 + \ldots) = \frac{n}{2^k}$$

- **Lemma 4:** The rank of any element is at most $\log n$

  Proof: Since $\frac{n}{2^k}$ is an upper bound on the number of elements of rank k, if there is at least one element of rank k, then $\frac{n}{2^k} \geq 1$ and $k \leq \log n$.

Path Compression

There is another heuristic that will further reduce the time required, on average, to carry out FIND operations. 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.

When we do path compression, we do not update rank. Thus, although the rank of an element may no longer be the height of that element in the tree, the UNION BY RANK heuristic still LINKs according to the ranks of the elements involved.

Also note that the lemmas above are still correct: their proofs depend only on rank which behaves as if there was no path compression.

Time Requirements

The time requirement of union-find with path compression depend on a very slowly-growing function of n called $\log^* n$. $\log^* n$ is the ceiling of the number of times you must iterate the log function on n before you get one. Thus, $\log^*(1) = 0$, $\log^*(2) = 1$, for $n = 3, 4$, $\log^*(n) = 2$, for $2^2 = 4 < n \leq 2^{2^2} = 16$, $\log^*(n) = 3$, for $2^{2^2} = 16 < n \leq 2^{16}$, $\log^*(n) = 4$, and for $64K < n \leq 2^{64K}$, $\log^*(n) = 5$. Since 2^{64K} is an unimaginably large number - much larger than the number of bits that can be stored in a computer - for all practical purposes, $\log^*(n)$ is constant.

Now we will argue that any sequence of m UNION and FIND operations on n elements take $O((m+n)log^*n)$ steps - or, since typically $m \geq n$, $O(m \log^* n)$. This indicates that while some of these operations may take as much as $\log n$ time, the *average* time per operation is only $\log^*(n)$, this is the savings from path compression!

This type of time analysis - where the average cost of an operation is small if we average over a sequence of operations even though the cost of a single operation may be quite large - is known as *amortized analysis.* We say that the union-find data structure with path compression takes $O((m+n)log^*n)$ *amortized time.*

First note that once an element is a non-root vertex, its rank is forever fixed (recall that path compression doesn't affect rank). Now divide the non-root elements into groups according to the $\log^*$ of their ranks. Thus group i is defined to be the set of vertices whose rank r satisfies $\log^* r = i$. More simply, each group consists of all vertices with ranks in the interval $(k, 2^k]$ where k is itself an iterated power of 2 (the first six such groups were given above).

- **Observation 1:** The number of distinct groups is at most $\log^*(n)$.

  Proof: In **Lemma 4** we proved that the maximum rank attainable for a given element is $\log n$, and $\log^*(\log n) < \log^* n$.

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

  Proof: This follows immediately from **Lemma 3** which states that the number of elements of rank greater than k is at most $\frac{n}{2^k}$.

Let us imagine assigning 2^k tokens to each element in group $(k, 2^k]$. Then, by Observation 1, the total number of tokens assigned to all the elements in that group is at most $2^k \frac{n}{2^k} = n$. Moreover, by Observation 2, the number of tokens assigned to all elements in all groups is at most $n \log^* n$.

In addition, we assign $\log^* n$ tokens to each FIND operation. Since we have m UNIONs, each requiring 2 FINDS, the total number of FINDs is $O(m)$. Putting this together with the tokens assigned to the elements, the grand total number of tokens is $O((m+n)\log^* n)$.

Recall that the number of steps to carry out the operation FIND(x) is proportional to the length of the path from x to the root of the tree. We shall use a token to pay for each pointer along this path that FIND must traverse. Our payment scheme is the following:

- if u and v belong to different groups then FIND uses one of its tokens to pay for chasing this pointer.

- if u and v belong to the same group then u pays using one of its tokens.

Recall that every time the parent pointer of an element v is updated via path compression, the rank of $p(v)$ increases by at least 1. If u and its initial parent are in group $(k, 2^k])$, the rank of the parent can increase fewer than 2^k times before the parent is an element in a higher group. Once the parent of u is in a higher group, FIND operations pay every time we chase that pointer; so the 2^k tokens assigned to u are sufficient to pay for all FIND operations through u.

The tokens assigned to the FIND operations are sufficient to pay for the remaining (inter-group) pointer jumps: since there are at most $\log^* n$ groups, a given FIND operation can cross at most $\log^* n$ boundaries between 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 $O((m+n)\log^* n)$.