Disjoint set (Union-Find)

For Kruskal's algorithm for the minimum spanning tree problem, we found that we needed a data structure for maintaining a collection of disjoint sets. That is, we need a data structure that can handle the following operations:

- MAKESET(x) - create a new set containing the single element x
- UNION(x, y) - replace two sets containing x and y by their union.
- FIND(x) - return the name of the set containing the element x

Naturally, this data structure is useful in other situations, so we shall consider its implementation in some detail.

Within our data structure, each set is represented by a tree, so that each element points to a parent in the tree. The root of each tree will point to itself. In fact, we shall use the root of the tree as the name of the set itself; hence the name of each set is given by a canonical element, namely the root of the associated tree.

It is convenient to add a fourth operation LINK(x, y) to the above, where we require for LINK that x and y are two roots. LINK changes the parent pointer of one of the roots, say x, and makes it point to y. It returns the root of the now composite tree y. With this addition, we have UNION(x, y) = LINK(FIND(x),FIND(y)), so the main problem is to arrange our data structure so that FIND operations are very efficient.

Notice that the time to do a FIND operation on an element corresponds to its depth in the tree. Hence our goal is to keep the trees short. Two well-known heuristics for keeping trees short in this setting are UNION BY RANK and PATH COMPRESSION. We start with the UNION BY RANK heuristic. The idea of UNION BY RANK is to ensure that when we combine two trees, we try to keep the overall depth of the resulting tree small. This is implemented as follows: the rank of an element x is initialized to 0 by MAKESET. An element's rank is only updated by the LINK operation. If x and y have the same rank r, then invoking LINK(x, y) causes the parent pointer of x to be updated to point to y, and the rank of y is then updated to $r+1$. On the other hand, if x and y have different rank, then when invoking LINK(x, y) the parent point of the element with smaller rank is updated to point to the element with larger rank. The idea is that the rank of the root is associated with the depth of the tree, so this process keeps the depth small. (**Exercise:** Try some examples by hand with and without using the UNION BY RANK heuristic.)

The idea of PATH COMPRESSION is that, once we perform a FIND on some element, we should adjust its parent pointer so that it points directly to the root; that way, if we ever do another FIND on it, we start out much closer to the root. Note that, until we do a FIND on an element, it might not be worth the effort to update its parent pointer, since we may never access it at all. Once we access an item, however, we must walk through every pointer to the root, so modifying the pointers only changes the cost of this walk by a constant factor.

```
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
```

In our analysis, we show 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 $\log_2$ function on n before getting a number less than or equal to 1. (So $\log^* 4 = 2, \log^* 16 = 3, \log^* 65536 = 4$.) We should note that this is not the tightest analysis possible; however, this analysis is already somewhat complex!

Note that we are going to do an *amortized analysis* here. That is, we are going to consider the cost of the algorithm over a sequence of steps, instead of considering the cost of a single operation. In fact a single UNION or FIND operation could require $O(\log n)$ operations. (**Exercise:** Prove this!) Only by considering an entire sequence

of operations at once can obtain the above bound. Our argument will require some interesting accounting to total the cost of a sequence of steps.

We first make a few observations about 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 with rank k is at most $\frac{n}{2^k}$
- the number of elements with rank at least k is at most $\frac{n}{2^{k-1}}$

The first two assertions are immediate from the description of the algorithm. The third assertion follows from the fact that the rank of an element v changes only if $\text{LINK}(v,w)$ is executed, $\text{rank}(v) = \text{rank}(w)$, and v remains the root of the combined tree; in this case v's rank is incremented by 1. A simple induction then yields that when $\text{rank}(v)$ is incremented to k, the resulting tree has at least 2^k elements. The last assertion then follows from the third assertion, as $\sum_{j=k}^{\infty} \frac{n}{2^j} = \frac{n}{2^{k-1}}$.

Exercise: Show that the maximum rank an item can have is $\log n$.

As soon as an element becomes a non-root, its rank is fixed. Let us divide the (non-root) elements into groups according to their ranks. Group $i+1$ contains all elements whose rank r satisfies $\log^* r = i$. (Group 0 contains elements of rank 0.) For example, elements in group 4 have ranks in the range $(4,16]$, and the range of ranks associated with group i is

$$(\underbrace{2^{2^{\cdots^2}}}_{i-2\ 2\text{s}}, \underbrace{2^{2^{\cdots^2}}}_{i-1\ 2\text{s}}].$$

For convenience we shall write this more simply by saying group $(k, 2^k]$ to mean the group with these ranks.

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

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

Let us assign 2^k tokens to each element in group $(k, 2^k]$. The total number of tokens assigned to all elements from that group is then at most $2^k \frac{n}{2^k} = n$, and the total number of groups is at most $\log^* n$, so the total number of tokens given out is $n \log^* n$. We use these tokens to account for the work done by FIND operations.

Recall that the number of steps for a FIND operation is proportional to the number of pointers that the FIND operation must follow up the tree. We separate the pointers into two groups, depending on the groups of u and $p(u) = v$, as follows:

- Type 1: a pointer is of Type 1 if u and v belong to different groups, or v is the root.
- Type 2: a pointer is of Type 2 if u and v belong to the same group.

We account for the two Types of pointers in two different ways. Type 1 links are "charged" directly to the FIND operation; Type 2 links are "charged" to u, who "pays" for the operation using one of the tokens. Let us consider these charges more carefully.

The number of Type 1 links each FIND operation goes through is at most $\log^* n$, since there are only $\log^* n$ groups, and the group number increases as we move up the tree.

What about Type 2 links? We charge these links directly back to u, who is supposed to pay for them with a token. Does u have enough tokens? The point here is that each time a FIND operation goes through an element u, its parent pointer is changed to the current root of the tree (by PATH COMPRESSION), so the rank of its parent increases by at least 1. If u is in the group $(k, 2^k]$, then the rank of u's parent can increase fewer than 2^k times before it moves to a higher group. Therefore the 2^k tokens we assign to u are sufficient to pay for all FIND operations that go through u to a parent in the same group.

We now count the total number of steps for m UNION and FIND operations. Clearly LINK requires just $O(1)$ steps, and since a UNION operation is just a LINK and 2 FIND operations, it suffices to bound the time for at most $2m$ FIND OPERATIONS. Each FIND operation is charged at most $\log^* n$ for a total of $O(m \log^* n)$. The total number of tokens used at most $n \log^* n$, and each token pays for a constant number of steps. Therefore the total number of steps is $O((m+n)\log^* n)$.

Let us give a more equation-oriented explanation. The total time spent over the course of m UNION and FIND operations is just

$$\sum_{\text{all FIND ops}} (\text{\# links passed through}).$$

We split this sum up into two parts:

$$\sum_{\text{all FIND ops}} (\text{\# links in same group}) + \sum_{\text{all FIND ops}} (\text{\# links in different groups}).$$

(Technically, the case where a link goes to the root should be handled explicitly; however, this is just $O(m)$ links in

total, so we don't need to worry!) The second term is clearly $O(m \log^* n)$. The first term can be upper bounded by:

$$\sum_{\text{all elements } u} (\#\text{ ranks in the group of } u),$$

because each element u can be charged only once for each rank in its group. (Note here that this is because the links to the root count in the second sum!) This last sum is bounded above by

$$\sum_{\text{all groups}} (\#\text{ items in group}) \cdot (\#\text{ ranks in group}) \leq \sum_{k=1}^{\log^* n} \frac{n}{2^k} 2^k \leq n \log^* n.$$

This completes the proof.