601.433/633 Introduction to Algorithms **Lecturer:** Michael Dinitz
Topic: Union-Find **Date:** 9/24/24

9.1 Introduction and Problem Definition

In this lecture we'll talk about the Union-Find problem / data structure, which is also sometimes called the *disjoint sets* problem. Informally, we want to maintain a data structure of disjoint sets that allows to to take the union of sets, and to figure out which set a given element is in. There are many uses for this, particularly in graph algorithms (which we'll see later). For now, we're just doing it as an interesting data structure.

Slightly more formally, we want a data structure which supports the following operations:

1. Make-Set(x): create a new set containing just the element x, i.e. the set $\{x\}$.
2. Union(x, y): replace the set containing x (call it S) and the set containing y (call it T) with the single set $S \cup T$.
3. Find(x): return the representative for the set containing x

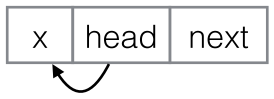
Let's examine these operations, particularly Find, in a little more detail. Suppose we're at some point in the operation of this data structure, so there is some collection of disjoint sets. We require every set to have a *unique* representative, which must be an element in the set. So if we call Find(x) and Find(y) and x and y are in the same set, the two calls must return the same representative (which could be x, y, or some other element z in the same set). And of course, since the representative must be in the set if x and y are in different sets then when we call Find on them we must get back different representatives. So we can think of the representative as being the "name" of the set.

We'll see a few ways of doing this reasonably efficiently, and in the textbook there is an extremely efficient method. One nice thing about Union-Find is that even simple things work pretty well, but we don't hit a limit to improvement until we get very, very strong bounds.

Let's fix some notation to start: there are m operations total, n of which are Make-Set operations (so the "number of elements" is n, like usual). One other thing to note is the assumption implicit in the Union and Find operations that we already have access to the elements that we care about: we're not concerned with *how* the algorithm known about some element x, but rather will simply assume that the algorithm has access to whatever elements it creates.

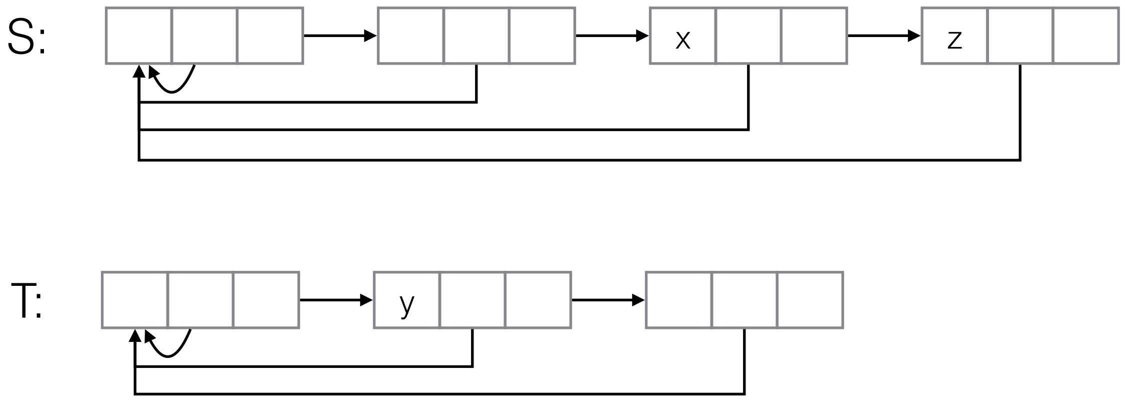
9.2 Lists

One simple data structure would to keep a linked list for each set. In the simplest version, we have a list for set and each element is an element in the list. We will also add a pointer from each element to the head of the list (the head will be the representative). So Make-Set(x) is easy: we just set $x \rightarrow head = x$ and $x \rightarrow next = NULL$, as the following figure shows.

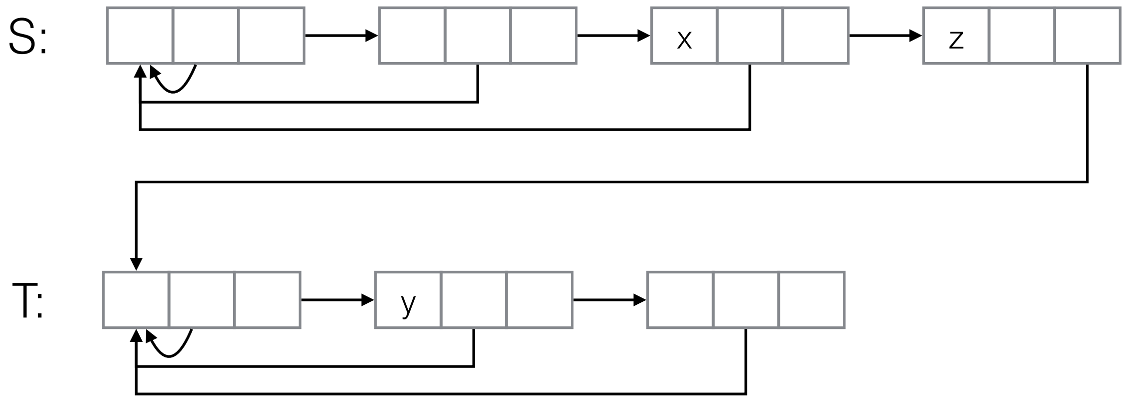


Find(x) is also easy: we can just return $x \to head$ (note that this uses the assumption that we have access to x: the algorithm does not need to search the lists for x, but rather by assumption already knows where it is).

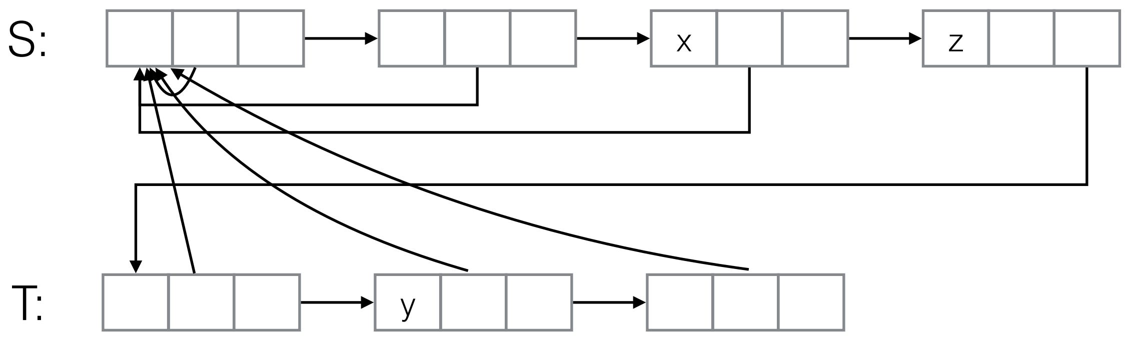
But what about Union(x, y)? Let S be the list containing x, and let T be the list containing y. Then initially the data looks like the following figure.



The obvious thing to do is traverse S until we get to the final element (say z), then set $z \to next = y \to head$, then walk down T (starting from $y \to head$) setting all of the head pointers to $x \to head$. After the traversal down S the data looks like

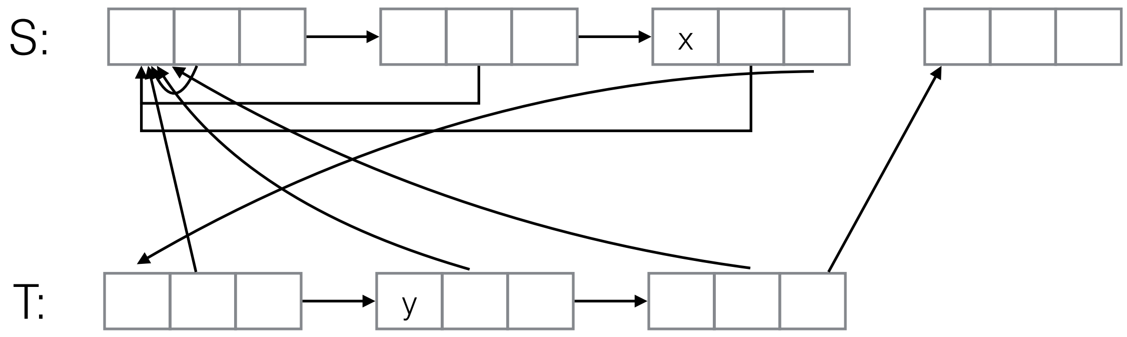


After the traversal down T our union is complete, and the data looks like



This takes $O(|S|+|T|)$ time. Since $|S|$ and $|T|$ might each be $\Omega(n)$, this gives a bound of only $O(n)$ on Unions (note that Make-Set and Find each take constant time).

Note that this can be bad even when $|S|$ or $|T|$ is small, say even a single element! Here's one obvious improvement: add T to the middle of S, rather than to the end. More precisely, add it right after x. Instead of walking down S to get z to be the final element, we could set $z = x \to next$, then set $x \to next = y \to head$, then walk down T (changing each head pointer to $x \to head$) until we get to the final element w and set $w \to next = z$. So T is "spliced" into S right after x. In our example, after doing this our data structure looks like



This decreases the time to $O(|T|)$, since we only need to traverse T and not S. Unfortunately, this can still be bad – clearly $|T|$ can be $\Omega(n)$.

Let's make one more modification. In the head node for each set we'll store the size of the set, i.e. the length of the list. This lets us always insert the shorter list into the longer one, so the time for a Union becomes $O(\min\{|S|, |T|\})$. Clearly this minimum can also be n, but now it turns out we can get a strong amortized bound!

Theorem 9.2.1 *The total running time is $O(m + n \log n)$ (recall that m is the total number of operations and n is the number of Make-Set operations).*

Proof: Find and Make-Set are clearly constant time, so their total cost is covered in the $O(m)$. So we need to argue that the total cost of all of the Union operations is $O(m + n \log n)$. In fact, we'll see that they only cost $O(n \log n)$.

To bound the total cost of all of the Unions, we can split up the cost among the elements (this is sometimes called *charging* the elements). Suppose we do a union of S and T where $|T| \leq |S|$. Then this takes time $O(|T|)$, so we can charge each element in T an $O(1)$ amount to pay for it (note that no elements in S get charged). For each element e, let $\alpha(e)$ be the total charge to e of all the operations. The total running time is thus

$$\sum_e \alpha(e),$$

so we just need to analyze how many times each element can be charged (note the similarity to what we did last class with the binary counter, where we switched to analyzing it bit-by-bit rather than operation-by-operation).

So how large can $\alpha(e)$ be? By definition, e is only charged when it is in the smaller of the two sets being unioned. This means that whenever e is charged, the size of the set containing it at least doubles! Thus $\alpha(e) \leq O(\log n)$, and thus the total running time of all of the Union operations is at most $O(n \log n)$. ■

It's also possible (and not too hard) to prove the above theorem by using a piggy bank argument. Suppose that we put $\log n$ tokens on every element when it is created through a Make-Set (note that this increases the amortized cost of Make-Set from $O(1)$ to $O(\log n)$). Then when we do a Union, we can take a token from each element in the smaller of S, T, which is enough to pay for the true cost (which is $\min\{|S|, |T|\}$). Note that by the discussion in the proof, since each element is charged only $\log n$ times, the number of tokens on any element is always nonnegative So now Union only has $O(1)$ amortized cost. And clearly Finds take only $O(1)$ time and do not change the banks, so their amortized cost is also $O(1)$. This implies the theorem.

Is this analysis tight? Let's give an example to show that it is, i.e. the Union operations might really cost $\Omega(n \log n)$ using this algorithm. Suppose we first do all n Make-Set operations, then use $n/2$ Unions to create $n/2$ sets of size 2, then $n/4$ unions to create $n/4$ sets of size 4, etc. Since the running time of Union(S, T) is $\Omega(\min\{|S|, |T|\})$, the time to union two equal sized sets is at least the size of the sets. Thus the first set of Unions takes time $\frac{n}{2} \cdot 1$, the second set takes time $\frac{n}{4} \cdot 2$, the third set takes times $\frac{n}{8} \cdot 4$, etc. At level i of this process, the total work is $\frac{n}{2^i} \cdot 2^{i-1} = n/2$. Since there are $\log n$ levels, this means the total running time is $\Omega(n \log n)$.

9.3 Better data structure: Trees

Even though the running time of the list-based data structure is pretty fast, let's think of ways we could make it even faster. How fast can we make a union-find data structure?

One idea is that instead of updating all the head pointers in list T (or whichever was shorter) when we perform a Union, we could do this in a lazy way, just pointing the head of T to the head of S and then waiting until we actually perform a find operation on some item x before updating its pointer. This will decrease the cost of the Union operations but will increase the cost of Find operations because we may have to take multiple hops. Notice that by doing this we have a collection of trees, with all links pointing up. The idea of updating the parent links when we do a Find operation is usually called *path compression.*

Another idea is that rather than deciding which of the two heads (or roots) should be the new one based on the size of their sets, perhaps there is some other quantity that would give us better performance. In particular, it turns out we can do better by setting the new root based on which tree has larger rank, which we will define in a minute.

We will prove that by implementing the two optimizations described above (lazy updates / path compression and union-by-rank), the total cost is bounded above by $O(m \log^* n)$, where $\log^* n$ is the number of times you need to take $\log_2$ until you get down to 1. For instance,

$$\log^*(2^{65536}) = 1 + \log^*(65536) = 2 + \log^*(16) = 3 + \log^*(4) = 4 + \log^*(2) = 5.$$

So, basically, $\log^* n$ is never bigger than 5. Technically, the running time of this algorithm is even better: $O(m \cdot \alpha(m, n))$ where α is the inverse-Ackermann function which grows even more slowly than $\log^*$. This is what's proved in the book, and I'd encourage you to read it. But the $\log^* n$ bound is hard enough to prove – let's not go completely overboard!

We now describe the procedure more specifically. Each element (node) will have two fields other than the element: a parent pointer that points to its parent in its tree (or itself if it is the root) and a rank, which is an integer used to determine which node becomes the new root in a Union operation. The operations are as follows.

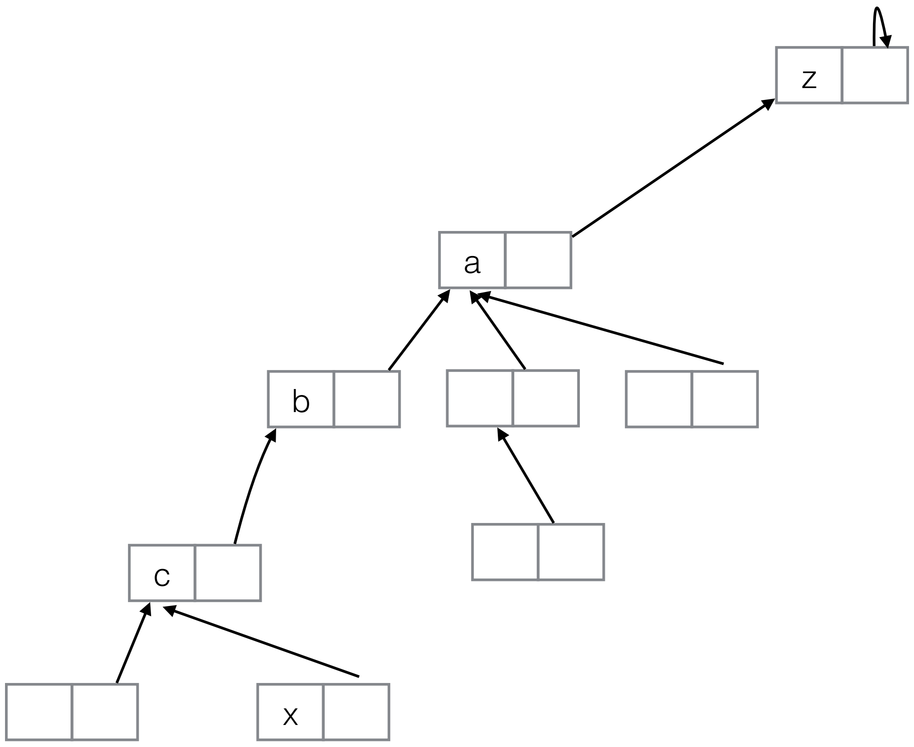
MakeSet(x): set $x \rightarrow rank = 0$ and $x \rightarrow parent = x$. This takes constant time.

Find(x): starting from x, follow the parent pointers until you reach the root. Update the parent pointers of x and all the nodes we pass over to point to the root. This is called *path compression.* The running time for Find(x) is proportional to (original) distance of x to its root.

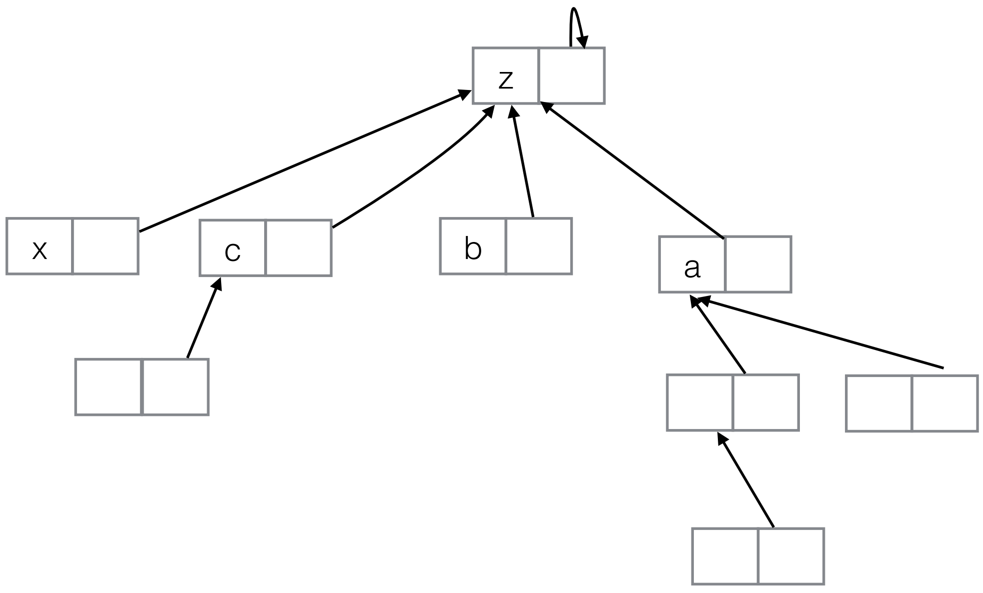
Link(root1, root2): This operation is only applied to root nodes. If root1 has strictly larger rank than root2, set root2→ *parent* to root1. If root2 has strictly larger rank than root1, set root1→ *parent* to root2. If they have the same rank, set root2→ *parent* to root1 and increase the root1→ *rank* by 1.

Union(x, y): Do Link(Find(x), Find(y)).

Let's do a quick example. First, a Find. If we do Find(x) on this:

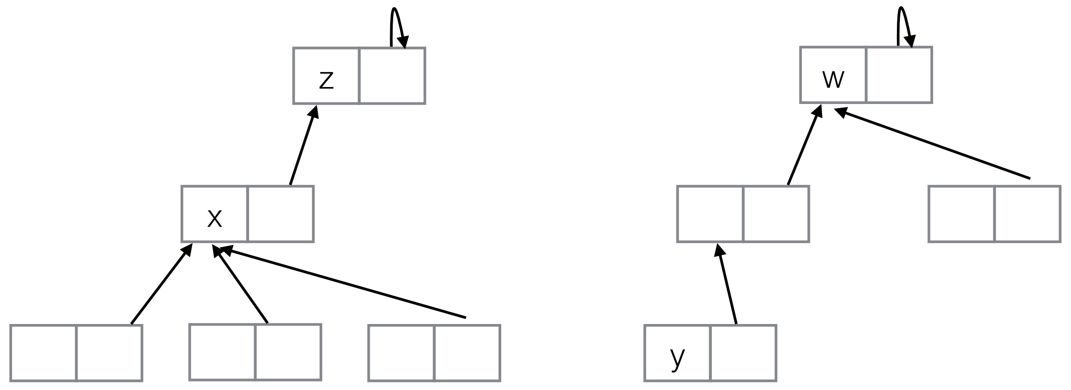


we end up with:

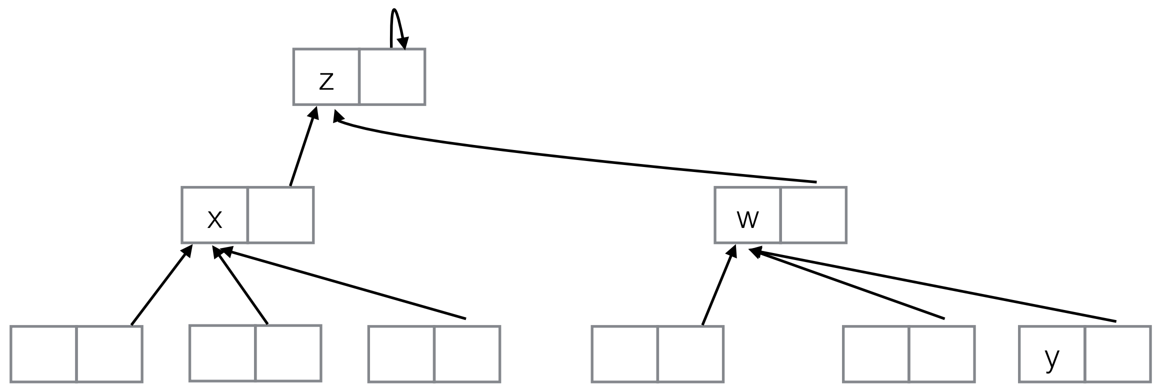


(where we are not showing ranks)

Now let's see a union. If we start with



and do Union(x, y), then if z has rank at least as large as the rank of w, we end up with



If z and w started out with exactly the same rank, then this would increase the rank of z by 1.

Properties of Ranks: We first develop some easy properties that ranks have – these will later help us analyze the full process, and are all easy to prove by induction.

1. If x is not a root, then its rank is strictly less than its parent's rank. Easy to see by induction: Union maintains this, and so does Find.

2. So when we do path compression, if the parent of x changes, then its new parent has rank strictly larger than its old parent.

3. The rank of x can only change if x is a root, and once x a non-root it never becomes a root again.

4. When x first reaches rank r, it must have at least 2^r nodes in its tree. Induction: Base case of $r = 0$ is true by Make-Set. Inductive step: when x first reaches r it must be because it had rank $r - 1$ and was the root of a tree when a union was taken with another tree whose root also had rank $r - 1$, and x became the root of the new tree. So by induction both trees had at least 2^{r-1} nodes, so when x got rank r it was the root of a tree with at least $2^{r-1} + 2^{r-1} = 2^r$ nodes.

Let's now prove an important lemma about ranks which uses these properties.

Lemma 9.3.1 *There are at most $n/2^r$ nodes of rank at least r.*

Proof: Let x be a node of rank at least r. By property 4 above, when x first reached rank r there were at least 2^r nodes in its tree. Let S_x be these elements. By property 1, they all had rank strictly less than r at that point in time, and by property 3 their rank never changes again, so they currently all have rank less than r.

Let z be a different element with rank at least r. We claim that S_x and S_z are disjoint. To see this, suppose WLOG that z achieved rank r after x did. Then when z first got rank r, its tree must have been disjoint from whatever tree contained x at that point, since the root of that tree already has rank at least r (by property 1). Hence S_x and S_z are disjoint.

So now for every node with rank at least r, we have found a set of at least 2^r nodes, and all of these sets are disjoint. Since the total number of nodes is only n, this means that there can be at most $n/2^r$ nodes of rank at least r. ■

We can now prove the main theorem.

Theorem 9.3.2 *The above algorithm has running time at most $O(m \log^* n)$.*

Proof: Let's begin with the easy parts. First, each Make-Set only takes $O(1)$ time, so we don't need to worry about them. Moreover, each Union does two Find operations plus a constant amount of extra work. So, we only need to worry about the time for the (at most $2m$) Find operations. As discussed earlier, the running time of Find(x) is proportional to the depth of x in its tree, so we'll just count the number of parent pointers examined and call this the running time. So we essentially want to prove that the total number of parent pointers examined is $O(m \log^* n)$.

To begin, note that on each Find we only examine a parent pointer of a root once and of a child of the root once. So there are at most $O(m)$ parent pointers examined that point *to* a root, so from now on we will only need to bound the number of parent pointers examined that *do not* point to a root.

The first thing we'll do is put the elements in buckets depending on their rank (note that this is only in the analysis, not in the actual algorithm). Let $2 \uparrow i$ denote a tower of i 2's. Bucket 0 will have all node of rank 0. In general, bucket i will have all nodes of rank $2 \uparrow (i-1)$ (tower of $i-1$ 2's) to $(2 \uparrow i) - 1$ (tower of i 2's minus 1). So, for example, bucket 1 has all nodes of rank 1, bucket 2 has ranks 2 through $2^2 - 1$, bucket 3 has ranks 2^2 through $2^{2^2} - 1$, bucket 4 has ranks 2^{2^2} through $2^{2^{2^2}} - 1$, etc. So there are $O(\log^* n)$ buckets. Let $B(i)$ denote the elements in bucket i.

Note that in bucket i, every node has rank at least $2 \uparrow (i-1)$ and hence by Lemma 9.3.1 there are at most $n/(2^{2 \uparrow (i-1)}) = n/(2 \uparrow i)$ elements in bucket i.

Now we can use this bucketing to help us upper bound the number of parent pointers examined. First, note that in each Find operation the number of times we can cross buckets is at most $O(\log^* n)$, since there are only that many buckets and property 1 guarantees that ranks are increasing as we walk up the tree. Thus the number of times we examine a parent pointer that crosses buckets is at most $O(m \log^* n)$ (as desired), and hence we will now only try to count the number of parent pointers we examine that *do not* cross buckets. This is the harder part – the proof is not hard, but it is subtle.

1. For each node x, let $\alpha(x)$ denote the number of times we examine the parent pointer from x where both x and its parent are in the same bucket and $x \to \mathit{parent}$ is not a root. So we are trying to prove that $\sum_x \alpha(x) \leq O(m \log^* n)$. We say that x is *charged* $\alpha(x)$ times.

2. Since x is not a root when it is charged, it is always the same rank whenever it is charged (by property 3).

3. Whenever x is charged, it gets a new parent because of path compression. Because of property 2, this new parent has rank strictly larger than its old parent. So if x was in bucket i when it started getting charged, the number of times it can get charged before its parent is from a different bucket is less than $2 \uparrow i$. Hence $\alpha(x) \leq 2 \uparrow i$.

4. Every node in bucket i has rank at least $2 \uparrow (i-1)$. So by Lemma 9.3.1 the number of nodes in bucket i is at most $n/(2 \uparrow i)$. Thus

$$\begin{aligned}\sum_x \alpha(x) &= \sum_{i=0}^{O(\log^* n)} \sum_{x \in B(i)} \alpha(x) \leq \sum_{i=0}^{O(\log^* n)} \sum_{x \in B(i)} (2 \uparrow i) \leq \sum_{i=0}^{O(\log^* n)} \frac{n}{2 \uparrow i}(2 \uparrow i) = O(n \log^* n) \\ &\leq O(m \log^* n),\end{aligned}$$

   as required.

■

So to sum up: The total running time is just

$$O(\text{number of parent pointers examined through at most } 2m \text{ Finds}).$$

The number of parent pointers examined where the parent is the root is at most $O(m)$ (two per Find), which is within our desired bound. When we bucket the elements based on their ranks, the number of parent pointers we examine that go between two elements in different buckets is at most $\log^* n$ per Find, for a total of $O(m \log^* n)$ as desired. So the only remaining thing is to bound the number of parent pointers we examine where the parent is not a root, and where both nodes are in the same bucket. For this, we prove that if x is in bucket i, the number of times such a parent pointer form x can be examined is at most $2 \uparrow i$. Since there are at most $n/(2 \uparrow i)$ elements in bucket i, this means the total number of such parent pointers examined from bucket i is at most n. Since there are only $O(\log^* n)$ buckets, this means that the total number of such parent pointers examined is at most $O(n \log^* n) \leq O(m \log^* n)$.