CMSC 420: Lecture 4
Disjoint Set Union-Find

Equivalence relations: An *equivalence relation* over some set S is a relation that satisfies the following properties for all elements of $a, b, c \in S$.

Reflexive: $a \equiv a$.

Symmetric: $a \equiv b$ then $b \equiv a$

Transitive: $a \equiv b$ and $b \equiv c$ then $a \equiv c$.

Equivalence relations arise in numerous applications. An example includes any sort of "grouping" operation, where every object belongs to some group (perhaps in a group by itself) and no object belongs to more than one group. More formally these groups are called *equivalent classes* and the subdivision of the set into such classes is called a *partition*. For example, suppose we are maintaining a bidirectional communication network. The ability to communicate is an equivalence relation, since if machine a can communicate with machine b, and machine b can communicate with machine c, then machine a can communicate with machine c (e.g. by sending messages through b). Now suppose that a new link is created between two groups, which previously were unable to communicate. This has the effect of *merging* two equivalence classes into one class.

We discuss a data structure that can be used for maintaining equivalence partitions with two operations: (1) *union*, merging to groups together, and (2) *find*, determining which group an element belongs to. This data structure should *not* be thought of as a general purpose data structure for storing sets. In particular, it cannot perform many important set operations, such as splitting two sets, or computing set operations such as intersection and complementation. And its structure is tailored to handle just these two operations. However, there are many applications for which this structure is useful. As we shall see, the data structure is simple and amazingly efficient.

Union-Find ADT: We assume that we have an underlying finite set of elements S. We want to maintain a *partition* of the set. In addition to the constructor, the (abstract) data structure supports the following operations.

`Set s = find(Element x):` Return an *set identifier* of the set s that contains the element x. A set identifier is simply a special value (of unspecified type) with the property that `find(x) == find(y)` if and only if x and y are in the same set.

`Set r = union(Set s, Set t):` Merge two sets named s and t into a single set r containing their union. We assume that s, t and r are given as set identifiers. This operations destroys the sets s and t.

Note that there are *no key values* used here. The arguments to the find and union operations are pointers to objects stored in the data structure. The constructor for the data structure is given the elements in the set S and produces a structure in which every element $x \in S$ is in a singleton set $\{x\}$ containing just x.

Inverted-Tree Implementation: We will derive our implementation of a data structure for the union-find ADT by starting with a simple structure based on a forest of *inverted trees*. You think of an inverted tree as a multiway tree in which we only store parent links (no child or

sibling links). A root's parent pointer is `null`. There is no limit on how many children a node can have. The sets are represented by storing the elements of each set in separate tree. For example, suppose that $S = \{1, 2, 3, \ldots, 13\}$ and the current partition is:

$$\{1, 6, 7, 8, 11, 12\}, \{2\}, \{3, 4, 5, 13\}, \{9, 10\}.$$

This might be stored in an inverted tree as shown in Fig. 1(a). Note that there is no particular order to how the individual trees are structured, as long as they contain the proper elements.
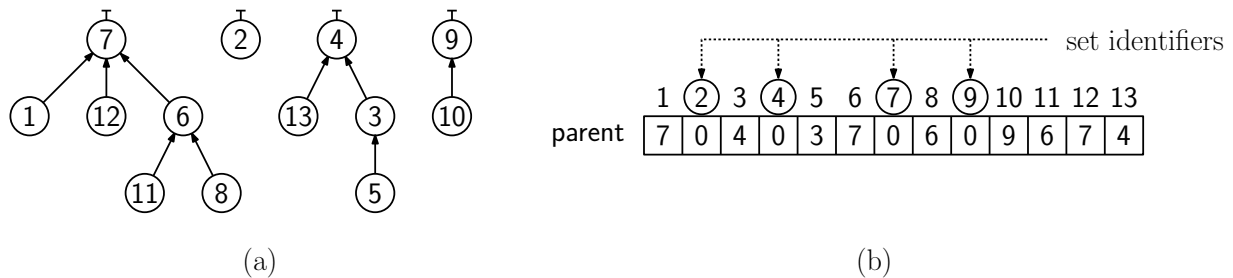

Fig. 1: (a) Partition stored as a forest of inverted trees and (b) array-based representation.

Array representation: You might wonder, "How do I find the node containing a given element?" Let's assume that the set elements are integers $\{1, 2, \ldots, n\}$ as shown above. Rather than using a standard node and pointer structure, we will think of the elements as being entries in an array `parent[1..n]`. We define `parent[i]` to hold the index the parent of element `i` in our tree or zero if `i` is a root of a tree. (We will sometimes call this zero and sometimes call this `null`, but they mean the same thing.) Even though we will use the tree representations in our illustrations, the actual implementation of the data structure is the array.

Initially, every element is in its own set, which we can set up by setting every element of the `parent` array to zero.

Set identifiers: Using this representation, each `Element` of the set is just an integer x, where $1 \leq x \leq n$. Each set is represented by a root in an inverted tree. A *set identifier*, called `Set`, is just an integer index x, where $1 \leq x \leq n$ and `parent[i] == 0`. For example, in Fig. 1(a) the set $\{3, 4, 5, 13\}$ is represented by the set identifier (root) `4`.

If we want to know whether two elements are in the same set, we simply find the roots of their associated trees (by following parent links) and then check whether these two roots are the same. For example, when we trace the parent links from `12` and `8` both stop at `7`, implying that `12` and `8` are in the same set. On the other hand, when we trace the parent links from `6` and `10`, they stop at the roots `7` and `9`, respectively. Since these are different roots, these elements are in different sets.

Find Operation: As mentioned above, given any element x, we perform the operation `find(x)` by walking along parent links until reaching the root of its tree. We return the root of the tree as the desired set identifier. This root element is set identifier for the set containing x. Notice that this satisfies the above requirement, since two elements are in the same set if and only if they have the same root. We call this `simple-find()`. (Later we will propose an improvement.)

For example, in Fig. 1(b)), operation `simple-find(11)` would start at node `11` and trace the path up through `6` up to the root `7`. It returns the index `7` as the answer.

Find operation (without path compression)

```
Set simple-find(Element x) {
    while (parent[x] != null) x = parent[x]  // follow chain to root
    return x;                                // return the root
}
```
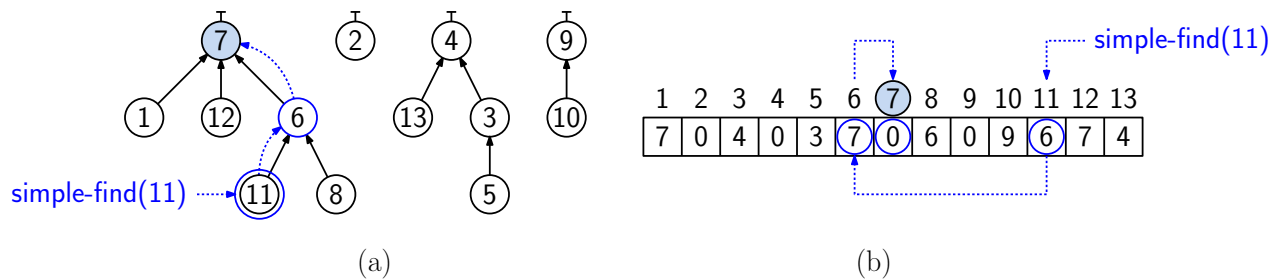

(a)   (b)

Fig. 2: The operation `simple-find(11)`.

Union Operation: A union is similarly straightforward to implement. To perform the union of two sets we just link the root of one tree into the root of the other tree. But here is where it pays to be smart. Recall that *height* of a tree is the maximum number of edges from any leaf to the root. In Fig. 3, we label each tree with its height. If we link the root of `i` as a child of `b`, the height of the resulting tree will be 2, whereas if we do it the other way the height of the tree will only be 1. Clearly, it is better to link the lower height tree under the other to keep the final tree's height as small as possible. This will make future `find` operations run faster. In contrast, if we take the union of two trees of equal height, say `g` and `d`, we can make either the root. (You might wonder why we use rank rather than some other property, and this is a good question for further thought.)
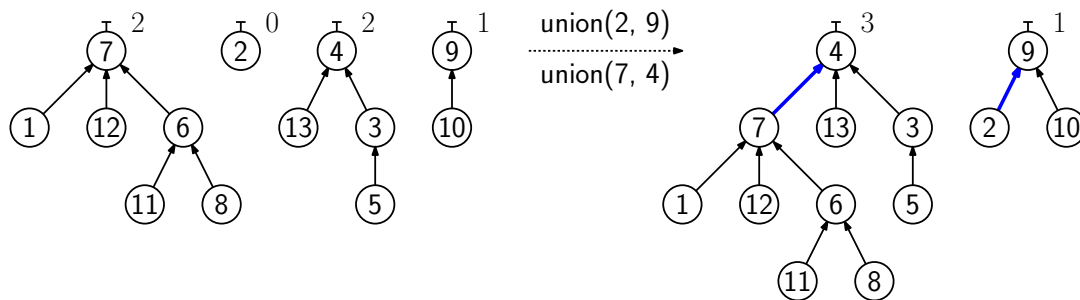

Fig. 3: Union-find with ranks.

In our later enhancements, we will compress paths, and this will change the tree height. So, we will invent an abstract concept, called *rank*, which behaves similar to tree height. We assume that the rank is stored as a field in each node, but we only use it for root nodes. The operation `union(s, t)` is given the roots of two trees. It swaps them if necessary so that `t`'s rank is at least as large as `s`'s. It then links `s` as a child of `t` and update's `t`'s rank to be the maximum of `t`'s rank and 1 plus `s`'s rank. The implementation is shown in the following code block.

Storing Ranks: There is a cool trick for storing ranks. As suggested in the code block, we could just implement a separate array. But since ranks are only needed for tree roots, we can do

_____Union operation

```
Set union(Set s, Set t) {
    if (rank[s] > rank[t]) {            // s has higher rank?
        swap s and t                    // swap so that t's rank is larger
    }
    parent[s] = t                       // link s as child of t
    rank[t] = max(rank[t], 1 + rank[s]) // update t's rank
    return t
}
```

all of this with just a single array. Rather than setting `parent[x] = 0` to indicate a root, we can set `parent[x]` to be the negation of the rank of `x`. When searching for the root of a tree, rather than testing whether `parent[x] == null` (or equivalently zero), we can check whether `parent[x] < 0`. If so, then the rank can be extracted by negating its value.

Analysis of Running Time: Consider a sequence of m union-find operations, performed on a domain with n total elements. Observe that the running time of the initialization is proportional to n, the number of elements in the set, but this is done only once. Each union takes only constant time, $O(1)$.

In the worst case, find takes time proportional to the height of the tree. The key to the efficiency of this procedure is the following observation, which implies that the tree height is never greater than $\lg m$. (Recall that $\lg m$ denotes the logarithm base 2 of m.)

Lemma: Using the union-find procedures described above any tree with height h has at least 2^h elements.

Proof: Given a union-find tree T, let h denote the height of T, let n denote the number of elements in T. We will show that $n \geq 2^h$.

It will help to introduce an intermediate quantity to help drive the proof. Let $u \geq 0$ denote the number of union operations used to build T. Our proof is based on induction on u. For the basis (no unions, or $u = 0$) we have a tree with $n = 1$ element of height $h = 0$. Since $1 = 2^0$, we have $n \geq 2^h$, which establishes the basis case.

For the induction step, suppose that we form a tree T through u union operations by merging two trees T' and T''. Let n' and n'' be their respective numbers of elements, let h' and h'' be their respective heights, and let u' and u'' denote the number of union operations to build each. The number unions to build T is clearly $u = 1 + u' + u''$. Thus, u' and u'' are both smaller than u, which means that we can apply the induction hypothesis to each of them, implying that $n' \geq 2^{h'}$ and $n'' \geq 2^{h''}$.

Following the merge we have a total $n = n' + n''$ elements. The final height depends on h' and h''. We may assume without loss of generality that $h' \leq h''$. (If not, swap the trees T' and T''.)

There are two possibilities. If $h' = h''$, then the resulting tree has height $h = h' + 1 = h'' + 1$ (see Fig. 4(a)). The number of elements in the final tree is

$$n \;=\; n' + n'' \;\geq\; 2^{h'} + 2^{h''} \;=\; 2^{h-1} + 2^{h-1} \;=\; 2 \cdot 2^{h-1} \;=\; 2^h.$$

On the other hand, if $h' < h''$, then the final tree has height equal to the larger tree, that is, $h = h''$ (see Fig. 4(b)). The number of elements is

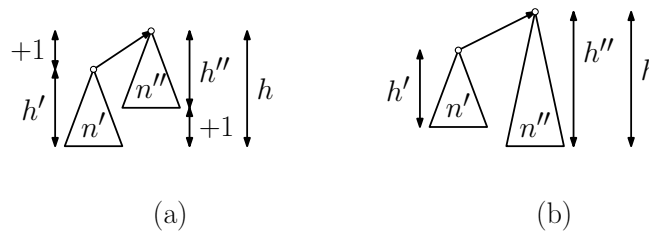$$n \;=\; n' + n'' \;\geq\; n'' \;\geq\; 2^{h''} \;=\; 2^h.$$

Fig. 4: Proof that $n \geq 2^h$.

In either case we obtain the desired conclusion.

Since the unions's take $O(1)$ time each, we immediately have the following.

Theorem: After initialization, any sequence of m union's and find's can be performed in time $O(m \log m)$. In other words, the amortized time for union-find operations is $O(\log m)$.

Path Compression: It is possible to apply a very simple heuristic improvement to this data structure which provides a significant improvement in the running time. Here is the intuition. If the user of your data structure repeatedly performs find's on a leaf at a very low level in the tree then each such find takes as much as $O(\log n)$ time. Can we improve on this?

Once we know the result of the find, we can go back and "short-cut" each pointer along the path to point directly to the root of the tree. This only increases the time needed to perform the find by a constant factor, but any subsequent find on this node (or any of its ancestors) will take only $O(1)$ time. The operation of short-cutting the pointers so they all point directly to the root is called *path-compression* and an example is shown below. Notice that only the pointers along the path to the root are altered. We present a slick recursive version below as well. Trace it to see how it works.
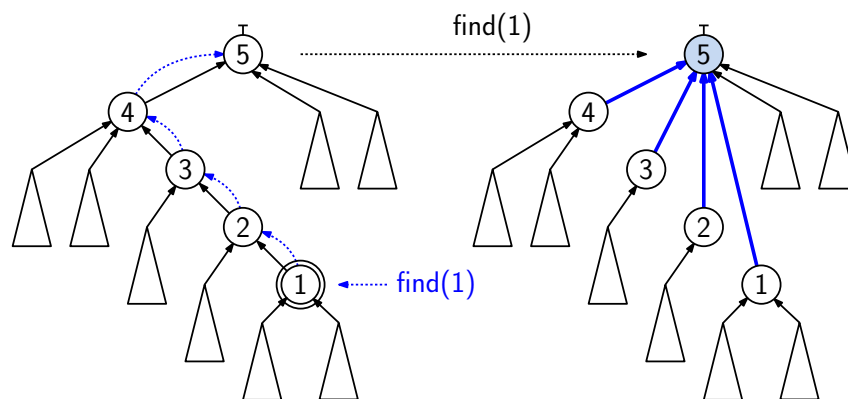

Fig. 5: Find using path compression.

The running time of `find-compress` is still proportional to the depth of node being found, but observe that each time you spend a lot of time in a find, you flatten the search path. Thus the work you do provides a benefit for later find operations. (This is the sort of thing that we observed in earlier amortized analyses.)

Does the savings really amount to anything? The answer is yes, but it is not easy to analyze. In 1975, Robert Tarjan proved that the amortized running time is strictly more than $O(1)$,

Find operation (with path compression)
```
Set find-compress(Element x) {
    if (parent[x] == null) return x            // return root
    else {
        mySet = find-compress(parent[x])       // find
        parent[x] = mySet                      // compress the link
        return mySet
    }
}
```

it is much less than $O(\log m)$.

Analyzing this algorithm is quite tricky. In order to create a bad situation you need to do lots of unions to build up a tree with some real depth. But as soon as you start doing finds on this tree, it very quickly becomes very flat again. In the worst case we need to an immense number of unions to get high costs for the finds.

To give the analysis (which we won't prove) we introduce two new functions, $A(i, j)$ and $\alpha(i)$. For $i, j \geq 0$, the function $A(i, j)$ is called *Ackerman's function* (discovered by Wilhelm Ackerman way back in 1928).

$$A(i, j) = \begin{cases} j + 1 & \text{if } i = 0, \\ A(i - 1, 1) & \text{if } i > 0 \text{ and } j = 0, \\ A(i - 1, A(i, j - 1)) & \text{otherwise.} \end{cases}$$

It is famous for being just about the fastest growing function imaginable. It is not obvious from the definition, so here are a few examples as the value of i increases

$$\begin{aligned} A(0, j) &= j + 1 \\ A(1, j) &= j + 2 \\ A(2, j) &= 2j + 3 \\ A(3, j) &= 2^{j+3} - 3 \\ A(4, j) &= \left(2^{2^{\cdot^{\cdot^{\cdot^2}}}} \right) - 3, \end{aligned}$$

where the tower of 2's in $A(4, j)$ is of height $j + 3$. As the value of i increases, the function increases to insanely large values very quickly. Even modest values, such as $A(4, 5) \approx 2^{65,536}$, which is already much larger than the number of particles in the observable universe.

Ackerman's function grows unbelievably fast. To create correspondingly slow growing function, we will defined its inverse, which we call α. Define

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

This definition is somewhat hard to interpret, but the important bottom line is that assuming $\lfloor m/n \rfloor \geq 1$, we have $\alpha(m, n) \leq 4$ as long as m is less than the number of particles in the universe, which is certainly true for any input set your program will ever encounter! The following result shows that an sequence of union-find operations take amortized time $O(\alpha(m, n))$. While we cannot formally state that this is constant time, it is constant time for all practical purposes. (The proof is quite complicated, and we will not present the proof.)

Theorem: After initialization, any sequence of m union's and find's (using path compression) on an initial set of n elements can be performed in time $O(m \cdot \alpha(m, n))$ time. Thus the amortized cost of each union-find operation is $O(\alpha(m, n))$.