# Exercises: Disjoint Sets/ Union-find [updated Feb. 3]

## Questions

1) Suppose you have an implementation of union that is "by-size" and an implementation of find that does not use path compression. Give the parent map (or array) that results from the following sequence: union(1,2), union(3,4), union(3,5), union(1,7), union(3, 12), union(0,9), union(8,10), union(8,9), union(7,4), union(2,9) where the unions are:
   a) by size
   b) by height
   c) by size, but now with path compression

   *Ties should be broken arbitrarily, by choosing the smaller of the two roots as the new root. (Note: if the trees have more than two nodes each, then choosing the smaller of the two roots is different than choosing the smaller of the two arguments).*

2) Consider an algorithm that computes the connected components of an *undirected* graph G using a forest of trees and union-find: Start with a partition of the n vertices into a forest of n trees, each consisting of a single vertex. Then, for each edge (i,j) in the graph G, apply union(i,j). Prove that this algorithm is correct, in that it indeed computes the connected components of G. [This is one of those claims that seems so obvious that it doesn't need a proof. But see if you can formulate a proof anyhow.]

3) Consider a tree formed by union-by-height operations, *without path compression*. Suppose the tree has n nodes and the tree is of height h. Show that n is greater than or equal to 2^h.

   (Note that the tree need not be a binary tree, and so we cannot just apply properties of binary trees to this problem. Indeed, for binary trees of height h, we can only say that the number of nodes is less than 2^{h+1} – 1, which is looser than the bound stated in the question.)

4) Consider the same situation in the last question. Can we use a similar proof to show that n is less than or equal to 2^h? (This would imply that n is always equal to 2^h.)

5) Consider the set of all trees of height h that can be constructed by a sequence of "union-by-height" operations. How many such trees are there?

6) The data structures for quick-find and quick-union that were discussed in class both use an array, but quick-find's was called "rep" and quick-union's was called "p" (for parent). But I claim that they are in fact the same thing. Why?

7) To implement union-by-height(i, j), we need to compare heights of the trees containing i and j. We could store the height of each node at that node, but how can we do this in a time efficient way? That is, when we modify the tree by a union operation, how much time will it take to modify the heights of all the nodes that need to be modified?

8)

a) In Java, the hashCode() method should be related to the equals() method as follows: if two objects are equal (namely o1.equals(o2) returns true), then the o1.hashCode() should be the same number as o2.hashCode(). Why ?

b) Should the converse also hold, that is, if two objects o1 and o2 have the same hashCodes, then should we require o1.equals(o2) to return true?

## Answers

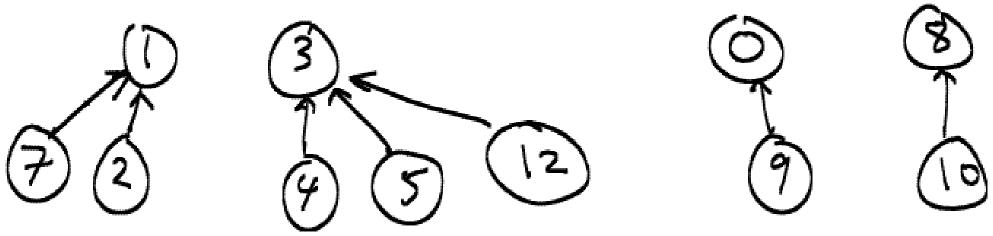
1) To make this as readable as possible, I am using python notation for maps where each pair means "index : parent[ array index]"

a) {0: 3, 1:3, 2:1, 3: -1 , 4:3, 5:3, 6:-1 , 7:1, 8:0, 9:0, 10:8 , 11:-1, 12:3}

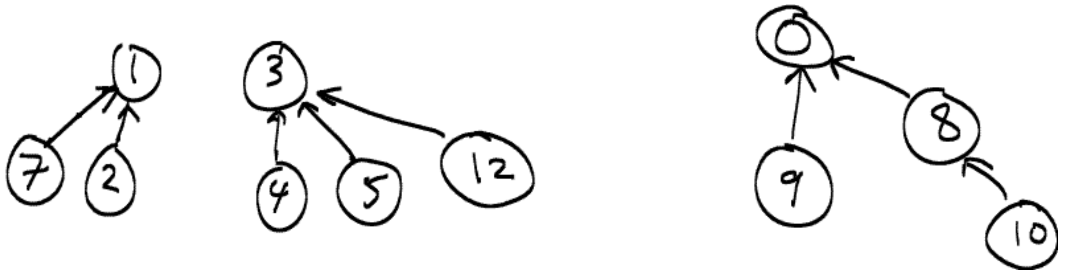
b) {0: -1, 1:0, 2:1, 3: 1 , 4:3, 5:3, 6:-1 , 7:1, 8:0, 9:0, 10:8 , 11:-1, 12:3}

c) {0: 3, 1:3, 2:3, 3: -1 , 4:3, 5:3, 6:-1 , 7:1, 8:0, 9:0, 10:8 , 11:-1, 12:3}

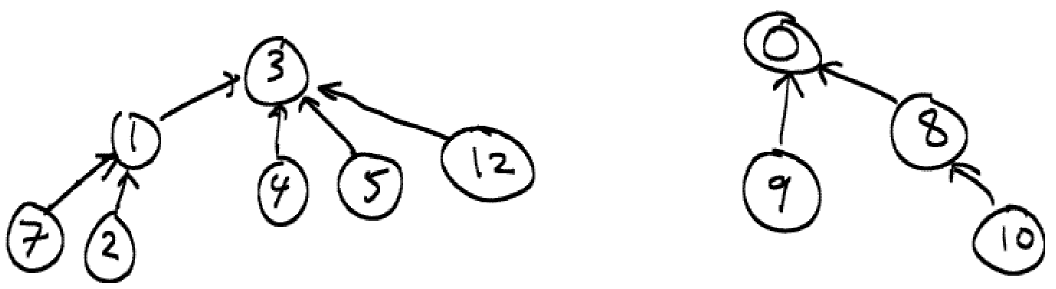
Here is a detailed explanation of c). The first seven unions are straightforward and give:



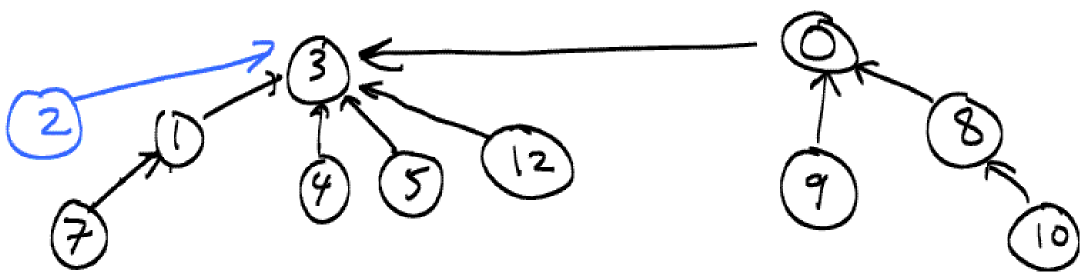
Union(8,9) then does a find on each argument (no path compression since they are already at depth 0 and 1 respectively). It then compares their roots (0 and 8) breaks the tie by choosing 0 to be the new root. This gives:



Union(7,4) then does a find on each, again no path compression is done since the 7 and 4 are already at depth 1. It compares the sizes of the two trees and since 3 has the greater size, it chooses 3 as the new rep. This yields:



Finally union(2,9) finds the roots of 2 and 9 (**doing path compression with the 2**). It chooses 3 as the new rep, yielding the final answer:



2) The first thing to realize is that the sequence of unions defines a graph F (the forest of trees), but it is typically a different graph than the original one since union(i, j) only joins nodes i and j by an edge in the new graph when these nodes happen to be the roots of the union tree.

You need to show that (1) two vertices are connected by a path in the graph G if and only (2) if they end up in the same tree in the forest F. This is an iff statement, so it has two directions, namely (1) implies (2) and (2) implies (1). You need to prove both. [If you didn't know how to do the proof, then stop here and put away this paper and see if you can state for yourself the two directions of the proof. Then try to think of how to prove each of them.]

- (1) implies (2). Suppose two vertices v_i and v_j are connected by a path in the graph G. The vertices in the path are (v_i, … , v_j) and so consider the list of edges defined by that path. Each of the edges produces a union call, and thus each pair of vertices in the path belongs in the same tree as its neighbor in the path, and thus (by transitivity) all vertices in the path end up in the same tree.

- (2) implies (1). Suppose two vertices in F are in the same tree. We want to show that they must be path connected in the graph G. Assume the opposite: namely that they are not path connected in G. Let A and B be the allegedly distinct connected components in G that contain these two vertices. But then there would be no edges (a,b) in G with a in A and b in B (since if there were, then all vertices in A and B would be path connected to all vertices in B). But if there were no edges in G between sets A and B (called "crossing edges"), then there would have been no union call to bring any vertices in A into the same tree as any vertex in B. Thus all vertices in A would be in a different tree in F than all vertices in B which is a contradiction.

3) Here is a proof by induction on the tree height k. The base case k=0 is easy, since a tree of height 0 always has just 1=2^0 node (the root). Suppose the claim is true for h=k. Now consider a union-by-height tree of height k+1. There must have been a union that brought two trees together and increased the height of one of them from k to k+1. Let those two trees (at the time of that union) be T1 and T2. We know that both T1 and T2 were of height k before the union. [Why? If one of them were of height less than k, then union-by-height would have changed the root of that shorter one to make it point to the root of the taller one, and the height of the unioned tree would still be k. But its not; the unioned tree is of height k+1.]
Now we can apply the induction hypothesis: the trees T1 and T2 each have at least 2^k nodes. Thus, the unioned tree has at least 2^k + 2^k = 2^(k+1) nodes.

4) No. When we considered the union of the two trees of height k to get a tree of height k + 1, there could have been unions that occurred afterwards that increased the number of nodes in the tree but that didn't increase the height of the tree beyond k+1.

5) It was a trick question. There are infinitely many of these trees, since there is no constraint given on the number of nodes and "union-by-height" trees (trees built by union-by-height operations) have no constraint on the number of children of any node.

6) union-by-find defines a tree, but it's a tree that has height at most 1. Each node either is a root, or it is the child of a root.

7) First you have to be clear on what we mean by the height of a node. It is the length of the longest path from that node to a leaf (or from a leaf to that node).

   Now to the problem: When you do union(i, j), you need to find the roots of the trees containing i and j and then you modify the parent p[ ] of one of them. Verify that, in doing so, at most only one node has its height modified, namely the rep of the new (merged) tree which was the rep of either i's or j's tree. So updating the heights is easy.

8)
   a) Hashcodes are used to allow us to index objects in HashMaps or HashSets. If two keys k1 and k2 are considered equal then we should be able to use either of these two keys when accessing

elements in the the HashMap or HashSet. (Otherwise, there would be not be much point in having an “equals” relation defined for keys.) If k1.equals(k2) were true but k1.hashCode() were different from k2.hashCode(), then these two keys most likely would bring us to two different slots in the table. So, if we put an item in the table with k1 and we are trying to get it with k2, we mostly likely would fail to get it.

b) The converse is not a problem. It could happen quite by accident that two keys have the same hashCode(). This would mean that they hash to the same slot in the hash table (since the compression function would be the same for them) and a collision happens. But this is not a big problem – it’s the price you pay for having O(1) access.