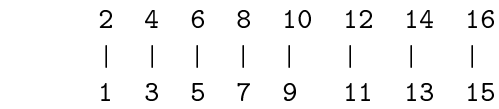
409 HW5 Solution Sketches

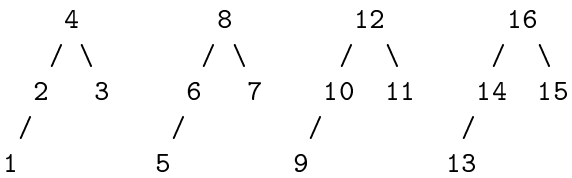
March 27, 2001

22.3-1 As pointed out on netnews, we have to assume that the UNION here is the one used in the text (or, equivalently, we have to do FINDs on the arguments of the UNION, since in general, the arguments are not the roots of trees. Sorry about the confusion here.

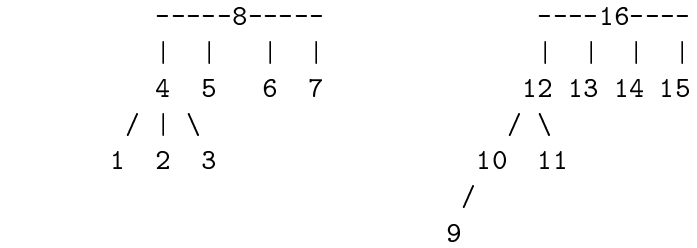
After line 4, we have the 8 trees:



After line 6, we have the 4 trees:

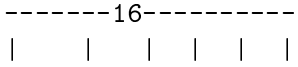


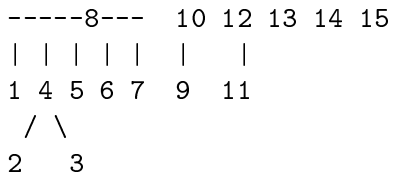
After line 8, we have the 2 trees:



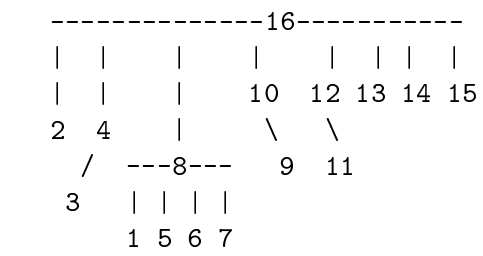
Note that 1 is the child of 4 because in the course of doing UNION(1,5), we had to do a FIND on 1, which resulted in path compression. For similar reasons, 5 is the child of 8, and 13 is the child of 16.

After line 9 we have 1 tree:

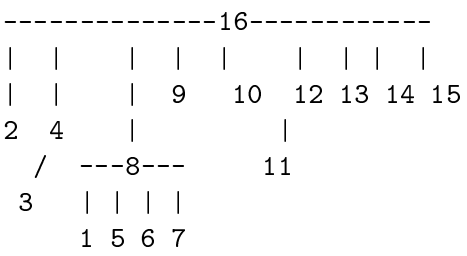




After line 10, we do path compression for 2 and 4 after finding 2, so we get the following hard-to-draw tree:



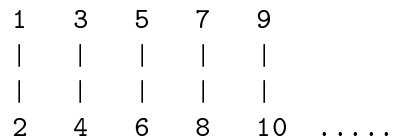
After line 11, we do path compression for 9, so we get the following even harder-to-draw tree:



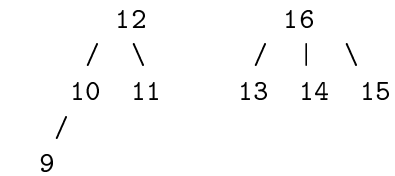
[Grading: 1 each for the trees that arise after lines 4, 6, 8, and 9. 2 each for the results of path compression after lines 10 and 11.]

Common mistakes:

1. A lot students don't do the path compression when executing FIND(x).
2. Some students didn't notice that the default root of UNION(x, y) is y if $rank(x) = rank(y)$, and they end up with trees like these:



3. When executing UNION(x_{11}, x_{13}) (where this version of UNION is the one in the text), we must really do UNION(FIND(x_{11}), FIND(x_{13}), using the UNION presented in class. Doing FIND(x_{11}) and FIND(x_{13}) results in the following trees (after compression):



We observe that the left tree is deeper. But FIND does not doesn't change the rank, so these two trees still have the same rank (since they had the same rank before we did the FIND). That's why we obtain the tree with 16 as its root. Several students made 12 the root.

23.1-3 Suppose $G = (V, E)$. For the adjacency list representation, given the array Adj represents G, construct the adjacency matrix Adj' that represents G^T by setting $Adj'[v] = \{u : v \in Adj[u]\}$. We can compute Adj' in time $O(|V| + |E|)$ by running through $Adj[u]$ for each $u \in V$, and constructing Adj' in the process, adding u to $Adj'[v]$ if we encounter v in $Adj[u]$.

For the adjacency matrix representation, given a matrix A that represents G, construct the matrix $B = (b_{ij})$ that represents G^T such that $b_{ij} = a_{ji}$. This clearly takes time $O(|V|^2)$, which is the size of the matrices A and B.

23.2-1 After running BFS-SEARCH[3], we have $\pi[3] = \text{NIL}$, $\pi[5] = 3$, $\pi[6] = 3$, $\pi[4] = 5$, $\pi[2] = 4$, $\pi[1] = \text{NIL}$, and $d[3] = 0$, $d[5] = d[6] = 1$, $d[4] = 2$, $d[2] = 3$.

23.2-3 The only line that changes in the algorithm presented in class is line 6 of BFS-SEARCH[s]. It becomes "**for** each v such that $a_{uv} = 1$". The running time is now $O(|V|^2)$, since we go through the loop at most $|V|^2$ times altogether, once for each entry in the matrix.

23.2-6 Suppose first that G is connected. Do the BFS algorithm, but now we have two types of gray (call them dark gray and light gray) and two types of black (dark black and light black). (If the graph is bipartite, then we will be able to partition it into light black and dark black vertices after running the algorithm.) Each time we call BFS-SEARCH(s), we color s light gray. If we are processing a light gray vertex u in the BFS algorithm (because it is at the head of Q), and $v \in Adj[u]$, we proceed as follows:

- if v is white, we color it dark gray and put it in Q (setting $\pi(v) = u$, as before)
- if v is dark gray or dark black, we do nothing
- if v is light gray or light black, return "G is *not* bipartite"

After checking all the vertices in $Adj[u]$, we color v light black.

If v is dark gray, we do the same thing, reversing the roles of "light" and "dark".

If, at the end of the algorithm, we have not returned "G is not bipartite", then return "G is bipartite".

To see that this works, suppose that the algorithm returns "G is not bipartite", and does so in the course of running BFS-SEARCH(s). Suppose, by way of contradiction, that G is biparitite. Then there must exist disjoint subsets V_1 and V_2 such that all the edges in E go between V_1 and V_2. Suppose that when we call BFS-SEARCH(s), that $s \in V_1$. ((An identical argument works if $s \in V_2$.) Then we show by induction on when a vertex is colored in the course of running BFS-SEARCH(s), that every vertex colored light gray (and later light black) must be in V_1 and every vertex colored dark gray (and later dark black) must be in V_2. To see this, suppose that a vertex v other than s is colored light gray. Then there must be an edge to it from a vertex u that was earlier colored dark gray. By the induction assumption, u must be in V_2. Since there is an edge from u to v, it cannot be the case that $v \in V_2$, so $v \in V_1$. The argument is identical if v is dark gray. But if the algorithm returns "G is not bipartite", there must be a vertex colored light gray that is adjacent to another one colored light gray or light

black, or a vertex colored dark gray that is adjacent to another one colored dark gray or dark black. That means there is an edge between two vertices in V_1 or an edge between two vertices in V_2, contradicting the assumption that G is bipartite.

On the other hand, if the algorithm returns "G is bipartite", then let V_1 consist of all the vertices colored light gray or light black and V_2 consist of all the vertices colored dark gray or dark black. (Remember, we are assuming that the graph is connected, so every vertex is colored.) Suppose there were an edge in E between two vertices in V_1, say u and v. That means that both u and v are colored light gray when they are discovered. Suppose v is discovered after u. That means that when v gets to the head of the queue, there will be a vertex (namely u) in $Adj[v]$ that is already colored light gray or light black. The algorithm will then return "G is not bipartite", contradicting the assumption that it returns "G is bipartite". A similar argument works if there is an edge between two vertices in V_2.

23.3-2 Here are the d/f pairs for each vertex: $q : 1/16$, $r : 17/20$, $s : 2/7$, $t : 8/15$, $u : 18/19$, $v : 3/6$, $w : 4/5$, $x : 9/12$, $y : 13/14$, $z : 10/11$. We'll also accept it if you started with $d = 0$ (in which case all your numbers will be 1 less than those given here.)

Extra problem: Suppose we are given a sequence σ of K MAKE-SETs + M FINDs + N UNIONs, where all the FINDs are performed at the end. Let σ' be the prefix of σ consisting of the MAKE-SETs and UNIONs. Thus, $\sigma = \sigma'\text{FIND}(v_1) \ldots \text{FIND}(v_k)$. Clearly it takes time $O(K+N)$ to perform all the operations in σ'. After this is done, we have a forest (collection of trees), say $T_1, \ldots, T_k$, where the trees have K nodes altogether. Let $r_1, \ldots, r_k$ be the roots of these trees. Since a tree with n nodes has $n-1$ edges (one for every node but the root, going to its parent), the total number of edges in these trees is $K-k$.

Now consider what happens to these trees after we perform FIND(v_1), ..., FIND(v_j), for $j \leq M$. We still have k trees, $T_1^j, \ldots, T_k^j$, where T_i^j has the same nodes and the same root as T_i. For any node $v \in T_i$, either it has the same parent in T_i and T_i^j, or its parent in T_i^j is r_i (if v was involved in some compression). To do the accounting, as we do FIND(v_1), FIND(v_2), ..., FIND(v_j), mark the edges in the original tree that get compressed. (That is, if v is in T_i and, as a result of compression following a FIND, the parent of v is r_i in T_i', we mark the edge from v to its parent in T_i.) Note if an edge in T_i is marked, so are all the edges from there on up to r_i, because of the way path compression works.

If v_{j+1} is in tree T_i, then the cost of FIND(v_{j+1}) is proportional to the cost of the path from v to r_i in T_i^j. The length of this path is 1 + the number of currently unmarked edges on the path from T_i. Thus, the total cost of all the M FINDs is $O(K+M)$ (since there are only $K-k$ edges to mark). That means the total cost of σ is $O(K+M+N)$. and FIND are $O(1)$ operations.

Grading: 1 point each for mentioning that UNION and FIND are $O(1)$ operations. 4 points for the proof that M FINDs take $O(K+M)$ (or $O(N+M)$).

Common mistakes:

1. Many students did an "average" case analysis of the running time of FIND. For some M number of FIND's, where M might even be a small num ber, this does not make

sense at all. This is probably a hangover from the probabilistic "expected" analysis of data-structures like skip lists. The homework question does not mention an average case analysis – by default, that means you should do a *worst case analysis*.

2. Many students proved that M finds take $O((K + M)lg^*K)$ (or something similar), following up with the assertion that the lg^*K is constant for all practical purposes. It might be, but not for this proof. The question did not say anything about making such an assumption, so you can't (neither do any of the proofs given in class).

3. Some students made the mistake of blindly following the proof given in class for the general case (where the FIND's are not at the end) - this was unnecessary (and leads you to the wrong assumption about lg^*K); the proof to the homework problem is much simpler.