# Homework #11
# Introduction to Algorithms/Algorithms 1
# 600.363/463
# Spring 2014

Solutions

April 22, 2014

## 1 Problem 1 (20 points)

(a) (10 points) Suppose we have a set of $n$ vertices $V = \{v_1, v_2, v_3, \ldots, v_n\}$. and that we have two binary trees $T_1 = (V, E_1)$ and $T_2 = (V, E_2)$. Prove that we can make tree $T_1$ "look like" tree $T_2$ using $O(n)$ rotations (i.e., some mix of left- and right-rotations). What me mean by a tree $T$ "looking like" another tree $T'$ is that they are the same up to a relabeling of the nodes– so, for example, the root of $T$ might be vertex $v_i$ and the root of tree $T'$ might be vertex $v_j$, but if we ignored the subscripts on the vertices, $T$ and $T'$ are the same tree. Put another way, $T = (V, E)$ "looks like" $T' = (V, E')$ if there exists a permutation $\pi : V \mapsto V$ such that $E' = \{(\pi(a), \pi(b)) : (a, b) \in E\}$.

Hint: first, show that we can turn tree $T_1$ into a linear chain via some sequence of $O(n)$ rotations. Then show that we can turn a linear chain into a tree of any "shape" we please via a sequence of $O(n)$ rotations. Specifically, we'll be able to turn the linear chain into a tree with the same "shape" as $T_2$, i.e., we'll make a tree that "looks like" $T_2$ in the sense described above. You may find it useful here to describe trees (and subtrees) in terms of two numbers– the number of nodes that are descendants of the left child of the root and the number of nodes that are descendants of the right child of the root.

**Solution:** Let us first show that we can turn any tree into a linear chain using $O(n)$ rotations. We will do this by induction on the size of the tree, which we will denote by $n$. The case where $n = 1$ is trivial, as is the case where $n = 2$. A binary tree on 3 nodes that is not already a linear chain can be turned into a linear chain via a single right-rotation. Let these serve as our base cases.

Suppose that for all $m < n$, it is the case that we can turn a binary tree on $m$ nodes into a linear chain in $O(m)$ operations. Let $T$ be a binary tree on $n$ nodes. Denote its root node by $t$, and let $\ell$ and $r$ denote the left and right child, respectively, of node $t$ (assume for now that both $\ell$ and $r$ are indeed present in the tree– we'll deal with the case where $t$ has only one child below). The subtree rooted at $r$ contains at most $n - 1$ nodes, and thus by the induction hypothesis we can transform it into a linear chain in $O(n)$ rotations. Similarly, by the induction hypothesis, we can turn the subtree rooted at $\ell$ into a linear chain via $O(n)$ right-rotations. The result of these operations will be two linear chains, rooted at two children of $t$ (note that our rotation operations may have made it so that $\ell$ and $r$ are no longer the roots of their respective subtrees– the labels $\ell$ and $r$ may now refer to different nodes– $\ell$ and $r$ are now the *new* children of $t$). Note now that any right rotation called on $t$ makes $\ell$ the new root of the tree, $t$ the new left child, and moves $r$ lower in the linear chain rooted at the right child of the root. Repeating this operation once for each node in the linear chain that was originally rooted at $\ell$ gives the linear chain we are after. Since the left linear chain contained at most $n - 1$ nodes, we require again at most $O(n)$ right-rotations. Thus, we have performed $O(n)$ rotations 3 times, for a total of $O(n)$ right-rotations. It should be clear now that the case where $t$ does not have two children is easily dealt with– simply treat the child of $t$ as the root of its own tree and perform the required number of rotations to turn that tree into a linear chain.

(b) (10 points) Prove that red-black trees are indeed balanced by showing that for any two simple paths $P_1$ and $P_2$ from the root to a leaf, we have $\frac{1}{2}|P_2| \leq |P_1| \leq 2|P_2|$, where $|P|$ denotes the length of path $P$.

**Solution:** Let $T$ be a red-black tree with $n$ internal nodes and black height $k$ Let $r$ denote the root of this tree. By the properties of red-black trees, we know that any path from $r$ to a leaf of the tree has exactly $k$ black nodes. Let $P_1$ be the shortest path from $r$ to a leaf of $T$. This path must contain at least $k$ nodes, since there must exist $k$ black nodes on path $P_1$. Thus, $|P_1| \geq k$. Let $P_2$ be the longest path from $r$ to a leaf of $T$. Again by the properties of red-black trees $P_2$ must contain at least $k$ nodes. Note, however, that $P_2$ must contain $|P_2| - k$ red nodes, since $P_2$ must contain exactly $k$ black nodes. However, for every red node on path $P_2$, the next node on the path must be black because the children of red nodes must be black. Thus, $|P_2| - k \leq k$ (note that if we were a little more rigorous in our counting we could get a slightly tigher bound here, but this will do). Thus, $|P_2| \leq 2k$. Putting our two results together, we have

$$k \leq |P_1| \leq |P_2| \leq 2k,$$

which implies that the longest path from $r$ to a leaf is at most twice the length of the shortest path, as we set out to show.

## 2 Problem 2 (20 points)

(a) (5 points) Prove that for any node $a$ in a disjoint-set forest (also known as Union-Find with path-compression) the rank of $a$ is less than or equal to the rank of $a.p$, with inequality when $a \neq a.p$.

**Solution:** First, let us note that when $a = a.p$, then it is trivially the case that $\text{rank}(a) \leq \text{rank}(a.p)$. Thus, assume $a \neq a.p$. If this is the case, then it must be that at some point, we had $a.p = a$ and $\text{rank}(a) = 0$ (i.e., just after initializing the data structure), and at some later point we called $\text{LINK}(a, b)$. But if we called $\text{LINK}(a, b)$, then either it was because (1) $\text{rank}(b) > \text{rank}(a)$, in which case our desired result follows by induction (because we must show that it is also the case that $\text{rank}(b.p) \geq \text{rank}(b)$, since it is possible that we had to perform path compression to connect $a$ to the root of the tree), or (2) $\text{rank}(b) = \text{rank}(a)$, in which case, by our assumption that $a.p \neq a$, we can assume without loss of generality that we updated $a.p = b$. As part of calling $\text{LINK}(a, b)$ in this latter case we would have incremented $\text{rank}(b)$, so our hypothesis holds. Again, the case where $b$ is not the root of its tree in the disjoint set forest is easily handled by performing induction up the tree to the root.

(b) (5 points) Suppose we have a disjoint-set forest (that is, the data structure for Union-Find with path compression) on $n$ elements. Prove that every node in the disjoint-set forest has rank $O(\log n)$.

**Solution** Recall that we only increment the rank of a node when we union it with another node of the same rank. Thus, if we have a node $a$ with $\text{rank}(a) = k$, then there must have, at an earlier time, have existed two nodes, say $b$ and $c$, such that at that time $\text{rank}(c) = \text{rank}(b) = k - 1$. Repeating this argument, we see that there must have been, at some time, $2^k$ nodes each of rank 0. Since it must be that $2^k \leq n$, we have $k \leq \log n$. Letting $k$ be the rank of the highest-ranked node gives the desired result.

(c) (10 points) In a previous homework, you were asked to give an algorithm that took as input an undirected graph $G = (V, E)$ and determined whether or not $G$ contained a cycle. Let's revisit that problem from a different angle. As before, we would like to determine whether or not the graph $G = (V, E)$ is a forest. It is likely that your algorithm the first time you had to solve this problem used union-find (or something similar) implicitly as a black box. Devise an algorithm that takes as input a graph $G = (V, E)$ and returns `True` if $G$ is a forest and returns `False` otherwise. Put another way, your algorithm should return `False` if the input graph $G = (V, E)$ contains a cycle, and `True` otherwise. Your algorithm should make explicit use of the union-find data structure and should run in (amortized) $O\left(|V| + |E| + |E|\alpha(|V|)\right)$ time, where $\alpha$ is defined to be

$$\alpha(n) = \min\{k : A_k(1) \geq n\},$$

and where $A_k(j)$ is the Ackermann function, defined in CLRS on page 573 (section 21.4) (note that the $|V| + |E|$ term in the runtime is simply the time that we need to read the input).

Prove the correctness and the runtime of your algorithm.

**Solution** We will build a disjoint set forest over the vertices of graph $G$. Initially, we have one node in the forest for every vertex $v \in V$, with $v.p = v$ and $\mathrm{rank}(v) = 0$. We will read in the edges $E$, one at a time. Unions on our data structure will correspond to connected components (based on the edges we've read so far). At the time that we encounter edge $\{u, v\} \in E$, if FIND-SET$(u)$ = FIND-SET$(v)$, then we know that $u$ and $v$ must be in the same connected component and are thus connected by a path. The presence of edge $\{u, v\}$ in additionto this path then implies the existence of a cycle containing edge $\{u, v\}$. If FIND-SET$(u) \neq$ FIND-SET$(v)$, then we can call $\mathrm{UNION}(u, v)$ to join the connected components containing $u$ and $v$ into a single connected component.

The runtime of the algorithm is simply the time required to read the input, initialize a Union-Find data structure on $V$, and make $O(|E|)$ calls to $\mathrm{UNION}(\cdot, \cdot)$, which is precisely $O(|V| + |E| + |E|\alpha(|V|))$, as we wished to show.

## Optional exercises

Solve the following problems and exercises from CLRS: 21.3-2, 21.3-3, 13.1-6, 13.1-4, 13.1-3, 17.1-3, 17.2-2, 17.3-2.