

Sample Solution to Problem Set 2

1. (15 points) Exercise 21.3-4, page 509 (Exercise 22.3-4, page 450, of first edition).

Answer:

- (a) We are given a sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations. We use an amortized analysis to establish that such a sequence takes only $O(m)$ time if path compression is used.

The main idea is as follows: whenever a node “participates” in a find operation, its parent is the root and remains so since no link operations follow a find operation. Therefore, any node can only contribute once to a find operation that is applied to an element that is not the child of a root. Therefore, the cost of the find operations is $O(m + n)$, where n is the number of elements. Since $n \leq m$, we have the total cost being $O(m)$.

In the following, we formalize the above (essentially complete proof) by means of an accounting argument. The actual costs of the three operations are (accurate upto constant factors):

MAKE-SET	1	
LINK	1	where d is the depth of the argument supplied to
FIND-SET	$\max(2 \cdot (d - 1) + 1, 1)$,	

FIND-SET in its tree.

Whenever we execute an operation, we will inject some (at most 3) dollars into the system. In particular, we will inject the following amounts for the 3 operations:

MAKE-SET	3
LINK	1
FIND-SET	1

We will also maintain some (non-negative) credit on each element that is present in the system. For each operation, we will use part of the injected money and the total credit available to pay for the actual cost of the operation; any remainder money will be left as credit on the elements. We will always maintain a nonnegative credit for each element.

Now we show that given a sequence of m MAKE-SET, FIND-SET and LINK operations, where all the LINK operations appear before any FIND-SET operation, we can pay for all the operations. Suppose a dollar represents each unit of cost. For every MAKE-SET(u) operation, we pay a dollar from the 3 dollars we charge and associate the credit of 2 dollars with the element u . For every LINK operation, we charge the operation 1 dollar and use this to pay the actual cost.

Immediately after all the LINK operations have been executed, each element in the disjoint-set forest has 2 dollars associated with it. We consider two different cases for the FIND-SET(u) operations:

Case 1: u is at depth 1 or 0. The the actual cost is 1 dollar that is paid by charging the operation 1 dollar.

Case 2: u is at depth $d > 1$. Then we use the 2 dollars associated with each of the $d - 1$ nodes that are at depth > 1 in the path from the root of u to u alongwith the charge of 1 dollar to pay the actual cost of $2 \cdot (d - 1) + 1$ dollars.

Observe that a node “loses” its 2 dollars only in Case 2, and once this happens its parent pointer is set to the root of its tree by path compression, and this never changes as there are no LINK operations after any FIND-SET operation. Hence each node at a depth of 2 or greater possesses 2 dollars justifying the argument in Case 2. Since we have paid for every operation without running into debt, after m such operations, we have ensured that the credit is nonnegative. Thus the total amount of injected dollars, that is $O(m)$, is an upper bound on the total actual cost of the m operations.

- (b) Observe that in the above proof, we do not require the union by rank heuristic. So even in the case of path compression alone, the $O(m)$ upper bound holds.

2. (15 points) Problem 21-3, page 521 (Problem 22-3, page 460, of first edition).

Answer:

- (a) First we prove that for each node v , $LCA(v)$ is called exactly once. Observe that for any node v , $LCA(v)$ is called either initially (if v is root), or once during the execution of $LCA(u)$, where u is the parent of v . Since any node v has at most 1 parent, $LCA(v)$ is called at most once. We can establish that $LCA(u)$ is called for every u by induction on the depth of u . The base case is when u is root and $LCA(root[T])$ is the initial call. Let it be true that for every node u at depth less than d ($d \geq 0$), $LCA(u)$ is called (induction hypothesis). Consider any node v at depth d . Since its parent (say w) is at depth less than d , $LCA(w)$ is called. During the execution of $LCA(w)$, line 4 will call $LCA(v)$.

We note that once $LCA(u)$ has completed, u is colored black. Also observe that after a node u is colored black, before another call to LCA is issued, $LCA(u)$ completes execution. For any pair $\{u, v\} \in P$, line 9 is executed exactly twice - once during the execution of $LCA(u)$, and once during the execution of $LCA(v)$. W.l.o.g assume that $LCA(u)$ completes before $LCA(v)$. When line 9 of $LCA(u)$ is executed, $color[v]$ is WHITE because if $color[v]$ were BLACK, $LCA(v)$ would have completed before $LCA(u)$. Therefore line 10 will not be executed in $LCA(u)$. On the other hand, during the execution of $LCA(v)$, both u and v are colored black, so line 10 will be executed.

- (b) Observe that the LCA algorithm does a depth-first traversal of the tree. The following claim gives a very useful characterization of the disjoint-set data structure during the execution.

Claim 1 *Just when $LCA(u)$ is completed, the entire subtree rooted at u (call it T_u) is a set by itself with u as the ancestor of this set.*

Proof: We prove this by induction on the height, h_u , of u (i.e., maximum distance from u to a leaf in T_u).

Basis: $h_u = 0$ (i.e., u is a leaf) In this case, $LCA(u)$ does a $MakeSet(u)$, which creates a set with u as the ancestor. Thus, the claim is true for this basis.

Induction Hypothesis: Assume that the claim is true for all vertices of height at most $k - 1$.

Induction Step: Suppose $h_u = k$. Let the children of u be v_1, \dots, v_t , in the order in which they are traversed. Note that the heights of the children of u is at most $k - 1$, and hence the induction hypothesis can be applied to all of them. The following claim (with $i = t$) proves the induction step.

Claim: For $0 \leq i \leq t$, just when the execution of $LCA(v_i)$ is completed, the subtree containing $\{u, T_{v_1}, \dots, T_{v_i}\}$ is a set by itself with u as the ancestor of this set.

Proof of claim: This claim can be proved by induction on i . The base case ($i = 0$) can be verified easily.

Induction step for the claim: As $h_{v_i} \leq k - 1$, using the ind. hyp. of the claim, it follows that T_{v_i} is a separate set by itself. And by the ind. hyp of this claim, it follows that $\{u, T_{v_1}, \dots, T_{v_{i-1}}\}$ is a separate set by itself. Hence lines 3-6 of the algorithm combine these two sets and make u the ancestor of this set. This proves the claim, and completes the proof of the claim. ■

Claim 2 When $LCA(u)$ is called, the number of sets in the disjoint-set data structure is equal to the depth of u in T .

Proof: Each time we visit a new vertex, v (i.e., move down the tree), we increase the depth by one and create a new set (with $MakeSet(v)$). Thus each increase in the depth by one corresponds to an increase in the number of sets by one. Each time we move up the tree, say along the edge (v, w) (where w is the parent of v), the depth decreases by one and the execution of $LCA(v)$ is just completed. Hence by previous claim, T_v is a set by itself. In other words, v and w are in different sets. Hence the $Union(v, w)$ operation combines these two sets, and thus decreases the number of sets by one. Thus each decrease in the depth by one corresponds to a decrease in the number of sets by one.

Hence, at every stage of the traversal, the number of sets equal the depth of the current vertex. ■

(c) **Claim 3** For each pair $\{u, v\} \in P$, the algorithm correctly computes the least common ancestor of u and v .

Proof: Without loss of generality let v be the vertex visited before u in the LCA traversal. Let w be the least common ancestor of u and v . Also u' and v' be the children of w which are the ancestors of u and v respectively. **Remark:** It is possible that $u = u'$ or $u' = w$ or $v = v'$.

Let's look at the disjoint-set data structure just after the completion of $LCA(u)$. Both u and v are colored black. Also, the LCA of all the children of w which were visited before u' , has been completed. By the claim in part (b), it follows that w and v are in the same set, with w being the ancestor of that set, i.e., $ancestor[FindSet(v)] = w$. This is exactly the vertex output by line 10.

Hence, the algorithm is correct. ■

(d) Let n be the number of nodes in the tree T . Let m be the number of pairs in P . The total number of operations is,

- *MakeSet*: one for each node in T : total of n .
- *Union*: one for each edge in T : total of $n - 1$.
- *FindSet*: one for each node in T , one for each edge in T , and one for each pair in P : total of $2n + m - 1$.

Thus, the total number of operations is $4n + m - 2$. By using union by rank and path compression, this algorithm can be implemented in $O((4n + m - 2)\alpha(n))$ time, which is $O((n + m)\alpha(n))$.

3. (15 points) Bit vector agreement and $\log^* n$

Problem: Alice and Bob have an n -bit vector each and want to determine whether these vectors are identical; if the vectors differ, then they would also like to agree on a bit position that the two vectors differ in. Being seasoned agents who participate in all message exchange studies performed by cryptography researchers, they would like to minimize the total number of bits that they exchange.

Model: The model for communication is as follows. Each message from either Alice or Bob contains a header of $O(1)$ bits and a variable length sequence of bits, referred to as the *payload*.

A Protocol: One simple protocol to solve the given problem is for Alice to send her n -bit vector to Bob. Then, Bob determines whether the two vectors differ. If they do, then he sends a $\lceil \lg n \rceil$ -bit vector that gives the bit position in which the two vectors differ. The communication complexity of this protocol is $n + \lceil \lg n \rceil + O(1) = n + O(\lg n)$ bits ($O(1)$ bits for the header and $n + \lceil \lg n \rceil$ bits for the payload.)

- (a) Give a protocol that exchanges $O(\lg n)$ messages, the total payload for which is only $n + O(1)$. (*Hint*: Use a divide and conquer strategy in which Alice and Bob reduce the size of the vector to be compared by a factor of $1/2$ each step of the communication.)

Answer: Let the bit vectors of Alice and Bob be represented as $A[0..n]^1$ and $B[0..n)$, respectively. We will assume for simplicity that n is a power of 2; the protocol as well as the argument can be adapted to apply to general n in a straightforward manner. The protocol consists of a sequence of communication steps, numbered from 0. In even steps, Alice sends a message to Bob, and in odd steps Bob sends a message to Alice.

Consider a general step i . We consider three cases:

- $i = 0$: Alice sends $A[0..n/2)$ in the payload.
- $0 < i \leq \lg n - 3$: If i is odd, then Bob determines whether the payload sent in step $i - 1$ matches $B[n - n/2^{i-1}, n - n/2^i)$. If it does not, then Bob sends a payload containing the bit position where $B[n - n/2^{i-1}, n - n/2^i)$ differs from the payload sent in step $i - 1$. The bit position sent is an offset from the position $n - n/2^{i-1}$ and can be represented with $\lg(n/2^i)$ bits. If the payload matches, there are two cases: if $i < \lg n - 3$, then Bob sends a payload containing the vector $B[n - n/2^i, n - n/2^{i+1})$; otherwise, Bob sends a payload containing the vector $B[n - n/2^i, n)$ (that is, the entire remaining suffix of the vector).

For even i , Alice works analogously.

¹For $i < j$, the notation $X[i..j)$ represents the sequence $X[i], X[i + 1], \dots, X[j - 1]$.

- $i = \lg n - 2$: If i is odd, then Bob determines whether the payload sent in step $i - 1$ matches $B[n - n/2^{i-1}, n]$. If it does not, then Bob sends a payload containing the bit position where $B[n - n/2^{i-1}, n]$ differs from the payload sent in step $i - 1$. Otherwise, Bob sends an empty payload.

For even i , Alice works analogously.

On receipt of the message in step i , Alice (resp., Bob) checks the length of the received payload p . We have three cases:

- If $|p|$, the length of p , is non-zero and less than $n/2^{i+1}$, then Alice (resp., Bob) interprets this as an indicator of a mismatch since otherwise Bob (resp., Alice) would have sent a payload of size $n/2^{i+1}$. Note that for $i \leq \lg n - 3$, $n/2^i \geq 8$ and so $\lg(n/2^i) < n/2^{i+1}$. Alice (resp., Bob) determines the mismatch bit position as $n - n/2^i + p$ and terminates the protocol.
- If $|p|$ is at least $n/2^{i+1}$, then Alice (resp., Bob) interprets this as a continued partial match and proceeds to step $i + 1$. Note that in this case, there are two possibilities: $|p|$ equals $n/2^{i+1}$ or $|p|$ equals $n/2^i$. The latter case happens only when $i = \lg n - 3$.
- If p is empty, then Alice (resp., Bob) concludes that the two vectors are the same and terminates the protocol.

The total payload exchanged is at most n and the number of rounds is at most $\lg n - 1$. We have the desired result.

- (b) Generalize the approach suggested by part (a) to obtain a protocol with communication complexity $n + O(\lg^* n)$. (*Hint*: Use a divide and conquer strategy such that the protocol completes after $O(\lg^* n)$ message exchanges.)

Answer: We refine the above approach by replacing 2^i by $\lg^{(i)} n$. Thus, in the first step, a payload of length $n - \lg n$ is sent; in the second step that of length $\log n - \log \log n$ is sent, and so on. In general, in step i , if the protocol has not already terminated, the bits in positions $[n - \lg^{(i)} n, n - \lg^{(i+1)} n)$ are sent. The total size of the payload is at most

$$n - \log n + \log n - \log \log n + \log \log n - \log \log \log n \dots \leq n + O(1).$$

The number of steps is at most $\log^* n$. So the total communication complexity is $n + O(\log^* n)$.

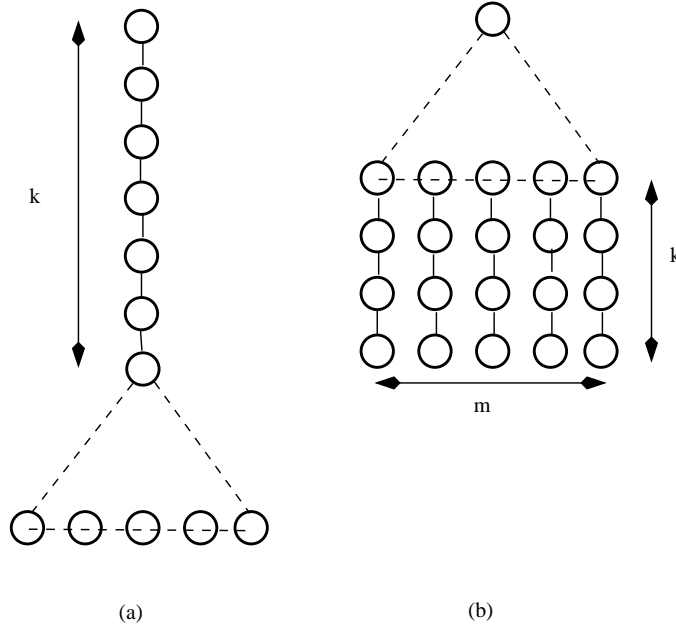
4. (15 points) Do ranks and heights induce the same notion of balance?

For any binary tree, define the *rank* $r(x)$ of a node x to be the logarithm (base 2) of the number of nodes (including x) in the subtree rooted at x . (Note that the rank, as defined above, need not be an integer.) Define the *total rank* of a binary tree to be the sum of the ranks of all of the nodes in the tree.

In the analysis of splay trees, we have seen that the total rank of a complete binary tree – a perfectly balanced binary tree – with n nodes is $\Theta(n)$, while the total rank of a line – a perfectly unbalanced tree! – of n nodes is $\Theta(n \lg n)$. We also know that the height of a complete binary tree is $\Theta(\lg n)$, while that of a line is $\Theta(n)$. We now ask whether the total rank and height measures are closely related in terms of the notion of balanced trees that they induce. Prove or disprove each of the following two statements.

(a) If the total rank of a binary tree is $O(n)$, then the height of the tree is $O(\lg n)$.

Answer: We disprove the above statement. Note that every node contributes to the total rank of the tree and every node has rank $O(\log n)$ even if its height is large. So we build a binary tree in which a few nodes have large height, but the vast majority of the nodes have height $O(\log n)$. We use the complete balanced binary tree and the line as the bases for our construction. In Figure (a), we have a line of length k “over” a complete binary tree of $n - k$ nodes. The total rank of the nodes in the line is $O(k \log n)$ and the total rank of the nodes in the balanced binary tree is $O(n)$. We can choose k to be as large as $n/\log n$ and still have total rank $O(n)$. In this case, the height is $k + \log(n - k) \geq n/\log n = \omega(\log n)$.



(b) If the height of a binary tree is $O(\lg n)$, then the total rank of the tree is $O(n)$.

Answer: Again we disprove the above statement. This part is slightly more tricky than part (a). Here we want the height to be $O(\log n)$ yet the total rank to be $\omega(n)$. In order for the total rank to be large, we need a majority of nodes to have high rank. This is true for a line; however, a line has large height. Instead of having several nodes in a single line, we have several separate lines, each of length k and connect them together using a complete binary tree as shown in Figure (b). We set $k = \log n$, since we want the height to be $O(\log n)$. Suppose we have m such lines, then the complete binary tree has $2m - 1$ nodes. So we have $2m - 1 + m(\log n)$ nodes, which should equal n . Therefore, we set $m = (n + 1)/(\log n + 2)$. (We are assuming that n is a power of 2 and such that m is an integer.)

The height of the tree is at most $\log m + \log n = O(\log n)$. The total rank is at least that of the nodes in the lines, which is $\Omega(mk \log k) = \Omega(n \log \log n) = \omega(n)$.