21 Data Structures for Disjoint Sets

21.1 Disjoint-set operations

21.1-1

> Suppose that CONNECTED-COMPONENTS is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ and the edges of E are processed in the order (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e). List the vertices in each connected component after each iteration of lines 3–5.

Edge processed											
initial	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$	$\{k\}$
(d, i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d, i\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$		$\{j\}$	$\{k\}$
(f, k)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d, i\}$	$\{e\}$	$\{f, k\}$	$\{g\}$	$\{h\}$		$\{j\}$	
(g, i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d, i, g\}$	$\{e\}$	$\{f, k\}$		$\{h\}$		$\{j\}$	
(b, g)	$\{a\}$	$\{b, d, i, g\}$	$\{c\}$		$\{e\}$	$\{f, k\}$		$\{h\}$		$\{j\}$	
(a, h)	$\{a, h\}$	$\{b, d, i, g\}$	$\{c\}$		$\{e\}$	$\{f, k\}$				$\{j\}$	
(i, j)	$\{a, h\}$	$\{b, d, i, g, j\}$	$\{c\}$		$\{e\}$	$\{f, k\}$					
(d, k)	$\{a, h\}$	$\{b, d, i, g, j, f, k\}$	$\{c\}$		$\{e\}$						
(b, j)	$\{a, h\}$	$\{b, d, i, g, j, f, k\}$	$\{c\}$		$\{e\}$						
(d, f)	$\{a, h\}$	$\{b, d, i, g, j, f, k\}$	$\{c\}$		$\{e\}$						
(g, j)	$\{a, h\}$	$\{b, d, i, g, j, f, k\}$	$\{c\}$		$\{e\}$						
(a, e)	$\{a, h, e\}$	$\{b, d, i, g, j, f, k\}$	$\{c\}$								

So, the connected components that we are left with are $\{a, h, e\}$, $\{b, d, i, g, j, f, k\}$, and $\{c\}$.

21.1-2

> Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected component if and only if they are in the same set.

First suppose that two vertices are in the same connected component. Then there exists a path of edges connecting them. If two vertices are connected by a single edge, then they are put into the same set when that edge is processed. At some point during the algorithm every edge of the path will be processed, so all vertices on the path will be in the same set, including the endpoints. Now suppose two vertices u and v wind up in the same set. Since every vertex starts off in its own set, some sequence of edges in G must have resulted in eventually combining the sets containing u and v. From among these, there must be a path of edges from u to v, implying that u and v are in the same connected component.

21.1-3

> During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

Find set is called twice on line 4, this is run once per edge in the graph, so, we have that find set is run $2|E|$ times. Since we start with $|V|$ sets, at the end only have k, and each call to UNION reduces the number of sets by one, we have that we have to made $|V| - k$ calls to UNION.

21.2 Linked-list representation of disjoint sets

21.2-1

> Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

The three algorithms follow the english description and are provided here. There are alternate versions using the weighted union heuristic, suffixed with WU.

```
MAKE-SET(x)
// Assume x is a pointer to a node contains .key .set .next
    Create a node S contains .head .tail .size
    x.set = S
    x.next = NIL
    S.head = x
    S.tail = x
    S.size = 1
    return S
```

```
FIND-SET(x)
    return x.set.head
```

```
UNION(x, y)
    S1 = x.set
    S2 = y.set
    if S1.size ⩾ S2.size
        S1.tail.next = S2.head
        z = S2.head
        while z ≠ NIL // z.next is incorrect, so S2.tail node should not be updated
            z.set = S1
            z = z.next
        S1.tail = S2.tail
        S1.size = S1.size + S2.size // Update the size of set
        return S1
    else
        same procedure as above
        change x to y
        change S1 to S2
```

21.2-2

> Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic.
>
> ```
> for i = 1 to 16
>     MAKE-SET(x[i])
> for i = 1 to 15 by 2
>     UNION(x[i], x[i + 1])
> for i = 1 to 13 by 4
>    UNION(x[i], x[i + 2])
> UNION(x[1], x[5])
> UNION(x[11], x[13])
> UNION(x[1], x[10])
> FIND-SET(x[2])
> FIND-SET(x[9])
> ```
>
> Assume that if the sets containing x_i and x_j have the same size, then the operation $\text{UNION}(x_i, x_j)$ appends x_j's list onto x_i's list.

Originally we have 16 sets, each containing x_i. In the following, we'll replace x_i by i. After the **for** loop in line 3 we have:

$$\{1,2\},\{3,4\},\{5,6\},\{7,8\},\{9,10\},\{11,12\},\{13,14\},\{15,16\}.$$

After the **for** loop on line 5 we have

$$\{1,2,3,4\},\{5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 7 results in:

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 8 results in:

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12,13,14,15,16\}.$$

Line 9 results in:

$$\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\}.$$

FIND-SET(x_2) and FIND-SET(x_9) each return pointers to x_1.

```
MAKE-SET-WU(x)
    L = MAKE-SET(x)
    L.size = 1
    return L
```

```
UNION-WU(x, y)
    L1 = x.set
    L2 = y.set
    if L1.size ≥ L2.size
        L = UNION(x, y)
    else L = UNION(y, x)
    L.size = L1.size + L2.size
    return L
```

21.2-3

> Adapt the aggregate proof of Theorem 21.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

During the proof of theorem 21.1, we concluded that the time for the n UNION operations to run was at most $O(n \lg n)$. This means that each of them took an amortized time of at most $O(\lg n)$. Also, since there is only a constant actual amount of work in performing MAKE-SET and FIND-SET operations, and none of that ease is used to offset costs of UNION operations, they both have $O(1)$ runtime.

21.2-4

> Give a tight asymptotic bound on the running time of the sequence of operations in Figure 21.3 assuming the linked-list representation and the weighted-union heuristic.

We call MAKE-SET n times, which contributes $\Theta(n)$. In each union, the smaller set is of size 1, so each of these takes $\Theta(1)$ time. Since we union $n-1$ times, the runtime is $\Theta(n)$.

21.2-5

> Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (head and tail), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (Hint: Use the tail of a linked list as its set's representative.)

For each member of the set, we will make its first field which used to point back to the set object point instead to the last element of the linked list. Then, given any set, we can find its last element by going ot the head and following the pointer that that object maintains to the last element of the linked list. This only requires following exactly two pointers, so it takes a constant amount of time. Some care must be taken when unioning these modified sets. Since the set representative is the last element in the set, when we combine two linked lists, we place the smaller of the two sets before the larger, since we need to update their set representative pointers, unlike the original situation, where we update the representative of the objects that are placed on to the end of the linked list.

21.2-6

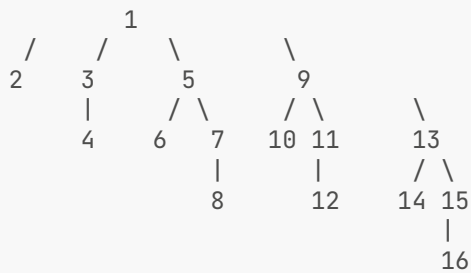
> Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the tail pointer to the last object in each list. Whether or not the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (Hint: Rather than appending one list to another, splice them together.)

Instead of appending the second list to the end of the first, we can imagine splicing it into the first list, in between the head and the elements. Store a pointer to the first element in S_1. Then for each element x in S_2, set $x.\text{head} = S_1.\text{head}$. When the last element of S_2 is reached, set its next pointer to the first element of S_1. If we always let S_2 play the role of the smaller set, this works well with the weighted-union heuristic and don't affect the asymptotic running time of UNION.

21.3 Disjoint-set forests

21.3-1

> Redo Exercise 21.2-2 using a disjoint-set forest with union by rank and path compression.



21.3-2

> Write a nonrecursive version of FIND-SET with path compression.

To implement FIND-SET nonrecursively, let x be the element we call the function on. Create a linked list A which contains a pointer to x. Each time we most one element up the tree, insert a pointer to that element into A. Once the root r has been found, use the linked list to find each node on the path from the root to x and update its parent to r.

21.3-3

> Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.

Suppose that $n' = 2^k$ is the smallest power of two less than n. To see that this sequences of operations does take the required amount of time, we'll first note that after each iteration of the **for** loop indexed by j, we have that the elements $x_1, \ldots, x_{n'}$ are in trees of depth i. So, after we finish the outer **for** loop, we have that $x_1, \ldots, x_{n'}$ all lie in the same set, but are represented by a tree of depth $k \in \Omega(\lg n)$. Then, since we repeatedly call FIND-SET on an item that is $\lg n$ away from its set representative, we have that each one takes time $\lg n$. So, the last **for** loop alltogther takes time $\Omega(m \lg n)$.

```
for i = 1 to n
    MAKE-SET(x[i])
for i = 1 to k
    for j = 1..n' - 2^{i = 1} by 2^i
        UNION(x[i], x[i + 2^{j - 1}])
for i = 1 to m
    FIND-SET(x[1])
```

21.3-4

> Suppose that we wish to add the operation $\text{PRINT-SET}(x)$, which is given a node x and prints all the members of x's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that $\text{PRINT-SET}(x)$ takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

In addition to each tree, we'll store a linked list (whose set object contains a single tail pointer) with which keeps track of all the names of elements in the tree. The only additional information we'll store in each node is a pointer $x.l$ to that element's position in the list.

- When we call $\text{MAKE-SET}(x)$, we'll also create a new linked list, insert the label of x into the list, and set $x.l$ to a pointer to that label. This is all done in $O(1)$.
- FIND-SET will remain unchanged.
- $\text{UNION}(x, y)$ will work as usual, with the additional requirement that we union the linked lists of x and y, since we don't need to update pointers to the head, we can link up the lists in constant time, thus preserving the runtime of UNION.
- Finally, $\text{PRINT-SET}(x)$ works as follows: first, set $s = \text{FIND-SET}(x)$. Then print the elements in the linked list, starting with the element pointed to by x. (This will be the first element in the list). Since the list contains the same number of elements as the set and printing takes $O(1)$, this operation takes linear time in the number of set members.

21.3-5 ⋆

> Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time if we use both path compression and union by rank. What happens in the same situation if we use only the path-compression heuristic?

Clearly each MAKE-SET and LINK operation only takes time $O(1)$, so, supposing that n is the number of FIND-SET operations occuring after the making and linking, we need to show that all the FIND-SET operations only take time $O(n)$.

To do this, we will ammortize some of the cost of the FIND-SET operations into the cost of the MAKE-SET operations. Imagine paying some constant amount extra for each MAKE-SET operation. Then, when doing a $\text{FIND-SET}(x)$ operation, we have three possibilities:

- First, we could have that x is the representative of its own set. In this case, it clearly only takes constant time to run.
- Second, we could have that the path from x to its set's representative is already compressed, so it only takes a single step to find the set representative. In this case also, the time required is constant.
- Third, we could have that x is not the representative and it's path has not been compressed. Then, suppose that there are k nodes between x and its representative. The time of this FIND-SET operation is $O(k)$, but it also ends up compressing the paths of k nodes, so we use that extra amount that we paid during the MAKE-SET operations for these k nodes whose paths were compressed. Any subsequent call to find set for these nodes will take only a constant amount of time, so we would never try to use the work that amortization amount twice for a given node.

21.4 Analysis of union by rank with path compression

21.4-1

> Prove Lemma 21.4.

The lemma states:

> For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$. The value of $x.rank$ is initially 0 and increases through time until $x \neq x.p$; from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

The initial value of $x.rank$ is 0, as it is initialized in line 2 of the $\text{MAKE-SET}(x)$ procedure. When we run $\text{LINK}(x, y)$, whichever one has the larger rank is placed as the parent of the other, and if there is a tie, the parent's rank is incremented. This means that after any $\text{LINK}(y, x)$, the two nodes being linked satisfy this strict inequality of ranks.

Also, if we have that $x \neq x.p$, then, we have that x is not its own set representative, so, any linking together of sets that would occur would not involve x, but that's the only way for ranks to increase, so, we have that $x.rank$ must remain constant after that point.

21.4-2

> Prove that every node has rank at most $\lfloor \lg n \rfloor$.

We'll prove the claim by strong induction on the number of nodes. If $n = 1$, then that node has rank equal to $0 = \lfloor \lg 1 \rfloor$. Now suppose that the claim holds for $1, 2, \ldots, n$ nodes.

Given $n + 1$ nodes, suppose we perform a UNION operation on two disjoint sets with a and b nodes respectively, where $a, b \leq n$. Then the root of the first set has rank at most $\lfloor \lg a \rfloor$ and the root of the second set has rank at most $\lfloor \lg b \rfloor$.

If the ranks are unequal, then the UNION operation preserves rank and we are done, so suppose the ranks are equal. Then the rank of the union increases by 1, and the resulting set has rank $\lfloor \lg a \rfloor + 1 \leq \lfloor \lg(n+1)/2 \rfloor + 1 = \lfloor \lg(n+1) \rfloor$.

21.4-3

> In light of Exercise 21.4-2, how many bits are necessary to store $x.rank$ for each node x?

Since their value is at most $\lfloor \lg n \rfloor$, we can represent them using $\Theta(\lg(\lg(n)))$ bits, and may need to use that many bits to represent a number that can take that many values.

21.4-4

> Using Exercise 21.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

MAKE-SET takes constant time and both FIND-SET and UNION are bounded by the largest rank among all the sets. Exercise 21.4-2 bounds this from about by $\lceil \lg n \rceil$, so the actual cost of each operation is $O(\lg n)$. Therefore the actual cost of m operations is $O(m \lg n)$.

21.4-5

> Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.rank > 0$ and $x.p$ is not a root, then $level(x) \leq level(x.p)$. Is the professor correct?

Professor Dante is not correct.

Suppose that we had that $x.p.rank > A_2(x.rank)$ but that $x.p.p.rank = 1 + x.p.rank$, then we would have that $level(x.p) = 0$, but $level(x) \geq 2$. So, we don't have that $level(x) \leq level(x.p)$ even though we have that the ranks are monotonically increasing as we go up in the tree. Put another way, even though the ranks are monotonically increasing, the rate at which they are increasing (roughly captured by the level values) doesn't have to be increasing.

21.4-6 ⋆

> Consider the function $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$. Show that $\alpha'(n) \leq 3$ for all practical values of n and, using Exercise 21.4-2, show how to modify the potential-function argument to prove that we can perform a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\alpha'(n))$.

First, observe that by a change of variables, $\alpha'(2^{n-1}) = \alpha(n)$. Earlier in the section we saw that $\alpha(n) \leq 3$ for $0 \leq n \leq 2047$. This means that $\alpha'(n) \leq 2$ for $0 \leq n \leq 2^{2046}$, which is larger than the estimated number of atoms in the observable universe.

To prove the improved bound $O(m\alpha'(n))$ on the operations, the general structure will be essentially the same as that given in the section.

First, modify bound 21.2 by observing that $A_{\alpha'(n)}(x.\,rank) \geq A_{\alpha'(n)}(1) \geq \lg(n+1) > x.\,p.\,rank$ which implies $level(x) \leq \alpha'(n)$.

Next, redefine the potential replacing $\alpha(n)$ by $\alpha'(n)$. Lemma 21.8 now goes through just as before. All subsequent lemmas rely on these previous observations, and their proofs go through exactly as in the section, yielding the bound.

Problem 21-1 Off-line minimum

> The ***off-line minimum problem*** asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array $extracted[1..m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "off-line" in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.
>
> **a.** In the following instance of the off-line minimum problem, each operation $INSERT(i)$ is represented by the value of i and each EXTRACT-MIN is represented by the letter E:
>
> $$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.$$
>
> Fill in the correct values in the *extracted* array.
>
> To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by
>
> $$I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1},$$
>
> where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j , we initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. We then do the following:
>
> ```
> OFF-LINE-MINIMUM(m, n)
>     for i = 1 to n
>         determine j such that i ∈ K[j]
>         if j ≠ m + 1
>             extracted[j] = i
>             let l be the smallest value greater than j for which set K[l] exists
>             K[l] = K[j] ∪ K[l], destroying K[j]
>     return extracted
> ```
>
> **b.** Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.
>
> **c.** Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

a.

index	value
1	4
2	3
3	2
4	6
5	8
6	1

b. As we run the **for** loop, we are picking off the smallest of the possible elements to be removed, knowing for sure that it will be removed by the next unused EXTRACT-MIN operation. Then, since that EXTRACT-MIN operation is used up, we can pretend that it no longer exists, and combine the set of things that were inserted by that segment with those inserted by the next, since we know that the EXTRACT-MIN operation that had separated the two is now used up. Since we proceed to figure out what the various extract operations do one at a time, by the time we are done, we have figured them all out.

c. We let each of the sets be represented by a disjoint set structure. To union them (as on line 6) just call UNION. Checking that they exist is just a matter of keeping track of a linked list of which ones exist(needed for line 5), initially containing all of them, but then, when deleting the set on line 6, we delete it from the linked list that we were maintaining. The only other interaction with the sets that we have to worry about is on line 2, which just amounts to a call of $\text{FIND-SET}(j)$. Since line 2 takes amortized time $\alpha(n)$ and we call it exactly n times, then, since the rest of the **for** loop only takes constant time, the total runtime is $O(n\alpha(n))$.

Problem 21-2 Depth determination

In the ***depth-determination problem***, we maintain a forest $\square = \{T_i\}$ of rooted trees under three operations:

$\text{MAKE-TREE}(v)$ creates a tree whose only node is v.

$\text{FIND-DEPTH}(v)$ returns the depth of node v within its tree.

$\text{GRAFT}(r, v)$ makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree than r but may or may not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that we implement $\text{GRAFT}(r, v)$ by setting $r.p = v$ and $\text{FIND-DEPTH}(v)$ by following the find path up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, we can reduce the worst-case running time. We use the disjoint-set forest $\square = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\square$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

> **d.** Show how to implement $\text{GRAFT}(r, v)$, which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.
>
> **e.** Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

a. MAKE-TREE and GRAFT are both constant time operations. FINDDEPTH is linear in the depth of the node. In a sequence of m operations the maximal depth which can be achieved is $m/2$, so FIND-DEPTH takes at most $O(m)$. Thus, m operations take at most $O(m^2)$. This is achieved as follows: Create $m/3$ new trees. Graft them together into a chain using $m/3$ calls to GRAFT. Now call FIND-DEPTH on the deepest node $m/3$ times. Each call takes time at least $m/3$, so the total runtime is $\Omega((m/3)^2) = \Omega(m^2)$. Thus the worst-case runtime of the m operations is $\Theta(m^2)$.

b. Since the new set will contain only a single node, its depth must be zero and its parent is itself. In this case, the set and its corresponding tree are indistinguishable.

```
MAKE-TREE(v)
    v = ALLOCATE-NODE()
    v.d = 0
    v.p = v
    return v
```

c. In addition to returning the set object, modify FIND-SET to also return the depth of the parent node. Update the pseudodistance of the current node v to be $v.d$ plus the returned pseudodistance. Since this is done recursively, the running time is unchanged. It is still linear in the length of the find path. To implement FIND-DEPTH, simply recurse up the tree containing v, keeping a running total of pseudodistances.

```
FIND-SET(v)
    if v ≠ v.p
        (v.p, d) = FIND-SET(v.p)
        v.d = v.d + d
        return (v.p, v.d)
    return (v, 0)
```

d. To implement GRAFT we need to find v's actual depth and add it to the pseudodistance of the root of the tree S_i which contains r.

```
GRAFT(r, v)
    (x, d_1) = FIND-SET(r)
    (y, d_2) = FIND-SET(v)
    if x.rank > y.rank
        y.p = x
        x.d = x.d + d_2 + y.d
    else
        x.p = y
        x.d = x.d + d_2
        if x.rank == y.rank
            y.rank = y.rank + 1
```

e. The three implemented operations have the same asymptotic running time as MAKE, FIND, and UNION for disjoint sets, so the worst-case runtime of m such operations, n of which are MAKE-TREE operations, is $O(m\alpha(n))$.

Problem 21-3 Tarjan's off-line least-common-ancestors algorithm

> The ***least common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***off-line least-common-ancestors problem***, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and we wish to determine the least common ancestor of each pair in P.
>
> To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call $\text{LCA}(T.\text{root})$. We assume that each node is colored WHITE prior to the walk.
>
> ```
> LCA(u)
>     MAKE-SET(u)
>     FIND-SET(u).ancestor = u
>     for each child v of u in T
>         LCA(v)
>         UNION(u, v)
>         FIND-SET(u).ancestor = u
>     u.color = BLACK
>     for each node v such that {u, v} ∈ P
>         if v.color == BLACK
>             print "The least common ancestor of" u "and" v "is" FIND-
> SET(v).ancestor
> ```
>
> **a.** Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.
>
> **b.** Argue that at the time of the call $\text{LCA}(u)$, the number of sets in the disjoint-set data structure equals the depth of u in T.
>
> **c.** Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.
>
> **d.** Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure in Section 21.3.

a. Suppose that we let $\leq_{\text{LCA}}$ to be an ordering on the vertices so that $u \leq_{\text{LCA}} v$ if we run line 7 of $\text{LCA}(u)$ before line 7 of $\text{LCA}(v)$. Then, when we are running line 7 of $\text{LCA}(u)$, we immediately go on to the **for** loop on line 8.

So, while we are doing this **for** loop, we still haven't called line 7 of $\text{LCA}(v)$. This means that $v.\text{color}$ is white, and so, the pair $\{u, v\}$ is not considered during the run of $\text{LCA}(u)$. However, during the **for** loop of $\text{LCA}(v)$, since line 7 of $\text{LCA}(u)$ has already run, $u.\text{color} = \text{black}$. This means that we will consider the pair $\{v, u\}$ during the running of $\text{LCA}(v)$.

It is not obvious what the ordering $\leq_{\text{LCA}}$ is, as it will be implementation dependent. It depends on the order in which child vertices are iterated in the **for** loop on line 3. That is, it doesn't just depend on the graph structure.

b. We suppose that it is true prior to a given call of LCA, and show that this property is preserved throughout a run of the procedure, increasing the number of disjoint sets by one by the end of the procedure. So, supposing that u has depth d and there are d items in the disjoint set data structure before it runs, it increases to $d + 1$ disjoint sets on line 1. So, by the time we get to line 4, and call LCA of a child of u, there are $d + 1$ disjoint sets, this is exactly the depth of the child. After line 4, there are now $d + 2$ disjoint sets, so, line 5 brings it back down to $d + 1$ disjoint sets for the subsequent times through the loop. After the loop, there are no more changes to the number of disjoint sets, so, the algorithm terminates with $d + 1$ disjoint sets, as desired. Since this holds for any arbitrary run of LCA, it holds for all runs of LCA.

c. Suppose that the pair u and v have the least common ancestor w. Then, when running $\text{LCA}(w)$, u will be in the subtree rooted at one of w's children, and v will be in another. WLOG, suppose that the subtree containing u runs first.

So, when we are done with running that subtree, all of their ancestor values will point to w and their colors will be black, and their ancestor values will not change until $\text{LCA}(w)$ returns. However, we run $\text{LCA}(v)$ before $\text{LCA}(w)$ returns, so in the **for** loop on line 8 of $\text{LCA}(v)$, we will be considering the pair $\{u, v\}$, since $u.\text{color} = \text{black}$. Since $u.\text{ancestor}$ is still w, that is what will be output, which is the correct answer for their LCA.

d. The time complexity of lines 1 and 2 are just constant. Then, for each child, we have a call to the same procedure, a UNION operation which only takes constant time, and a FIND-SET operation which can take at most amortized

inverse Ackerman's time. Since we check each and every thing that is adjacent to u for being black, we are only checking each pair in P at most twice in lines 8-10, among all the runs of LCA. This means that the total runtime is $O(|T|\alpha(|T|) + |P|)$.