Chapter 17

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 17.1-1

It woudn't because we could make an arbitrary sequence of $MULTIPUSH(k), MULTIPOP(k)$. The cost of each will be $\Theta(k)$, so the average runtime of each will be $\Theta(k)$ not $O(1)$.

Exercise 17.1-2

Suppose the input is a 1 followed by $k-1$ zeros. If we call DECREMENT we must change k entries. If we then call INCREMENT on this it reverses these k changes. Thus, by calling them alternately n times, the total time is $\Theta(nk)$.

Exercise 17.1-3

Note that this setup is similar to the dynamic tables discussed in section 17.4. Let n be arbitrary, and have the cost of operation i be $c(i)$. Then,

$$\sum_{i=1}^{n} c(i) = \sum_{i=1}^{\lceil \lg(n) \rceil} 2^i + \sum_{i \leq n \text{ not a power of 2}} 1 \leq \sum_{i=1}^{\lceil \lg(n) \rceil} 2^i + n = 2^{1+\lceil \lg(n) \rceil} - 1 + n \leq 4n - 1 + n \leq 5n \in O(n)$$

So, since to find the average, we divide by n, the average runtime of each command is $O(1)$.

Exercise 17.2-1

To every stack operation, we charge twice. First we charge the actual cost of the stack operation. Second we charge the cost of copying an element later on. Since we have the size of the stack never exceed k, and there are always k operations between backups, we always overpay by at least enough. So, the amortized cost of the operation is constant. So, the cost of the n operation is $O(n)$.

Exercise 17.2-2

Assign the cost 3 to each operation. The first operation costs 1, so we have a credit of21. Now suppose that we have nonnegative credit after having performed the 2^ith operation. Each of the $2^i - 1$ operations following has cost 1. Since we pay 3 for each, we build up a credit of 2 from each of them, giving us $2(2^i - 1) = 2^{i+1} - 2$ credit. Then for the 2^{i+1}th operation, the 3 credits we pay gives us a total of $2^{i+1} + 1$ to use towards the actual cost of 2^{i+1}, leaving us with 1 credit. Thus, for any operation we have nonnegative credit. Since the amortized cost of each operation is $O(1)$, an upper bound on the total actual cost of n operations is $O(n)$.

Exercise 17.2-3

For each time we set a bit to 1, we both pay a dollar for eventually setting it back to zero (in the usual manner as the counter is incremented). But we also pay a third dollar in the event that even after the position has been set back to zero, we check about zeroing it out during a reset operation. We also increment the position of the highest order bit (as needed). Then, while doing the reset operation, we will only need consider those positions less significant than the highest order bit. Because of this, we have at least paid one extra dollar before, because we had set the bit at that position to one at least once for the highest order bit to be where it is. Since we have only put down a constant amortized cost at each setting of a bit to 1, the amortized cost is constant because each increment operation involves setting only a single bit to 1. Also, the amortized cost of a reset is zero because it involves setting no bits to one. It's true cost has already been paid for.

Exercise 17.3-1

Define $\Phi'(D) = \Phi(D) - \Phi(D_0)$. Then, we have that $\Phi(D) \geq \Phi(D_0)$ implies $\Phi'(D) = \Phi(D) - \Phi(D_0) \geq \Phi(D_0) - \Phi(D_0) = 0$. and $\Phi'(D_0) = \phi(D_0) - \Phi(D_0) = 0$. Lastly, the amortized cost using Φ' is $c_i + \Phi'(D_i) - \Phi'(D_{i-1}) = c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_i) - \Phi(D_{i-1})) = c_i + \Phi(D_i) - \Phi(D_{i-1})$ which is the amortized cost using Φ.

Exercise 17.3-2

Let $\Phi(D_i) = k + 3$ if $i = 2^k$. Otherwise, let k be the largest integer such that $2^k \leq i$. Then define $\Phi(D_i) = \Phi(D_{2^k}) + 2(i - 2^k)$. Also, define $\Phi(D_0) = 0$. Then $\Phi(D_i) \geq 0$ for all i. The potential difference $\Phi(D_i) - \Phi(D_{i-1})$ is 2 if i is not a power of 2, and is $-2^k + 3$ if $i = 2^k$. Thus, the total amortized cost of n operations is $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} 3 = 3n = O(n)$.

Exercise 17.3-3

Make the potential function be equal to $n \lg(n)$ where n is the size of the min-heap. Then, there is still a cost of $O(\lg(n))$ to insert, since only an amount

of ammortization that is about $\lg(n)$ was spent to increase the size of the heap by 1. However, since extract min decreases the size of the heap by 1, the actual cost of the operation is offset by a change in potential of the same order, so only a constant amount of work is needed.

Exercise 17.3-4

Since $D_n = s_n$, $D_0 = s_0$, and the amortized cost of n stack operations starting from an empty stack is is $O(n)$, equation 17.3 implies that the amortized cost is $O(n) + s_n - s_0$.

Exercise 17.3-5

Suppose that we have that $n \geq cb$. Since the counter begins with b 1's, we'll make all of our amortized cost $2 + \frac{1}{c}$. Then the additional cost of $\frac{1}{c}$ over the course of n operations amounts to paying an extra$\frac{n}{c} \geq b$ which was how much we were behind by when we started. Since the amortized cost of each operation is $2 + \frac{1}{c}$ it is in $O(1)$ so the total cost is in $O(n)$.

Exercise 17.3-6

We'll use the accounting method for the analysis. Assign cost 3 to the ENQUEUE operation and 0 to the DEQUEUE operation. Recall the implementation of 10.1-6 where we enqueue by pushing on to the top of stack 1, and dequeue by popping from stack 2. If stack 2 is empty, then we must pop every element from stack 1 and push it onto stack 2 before popping the top element from stack 2. For each item that we enqueue we accumulate 2 credits. Before we can dequeue an element, it must be moved to stack 2. Note: this might happen prior to the time at which we wish to dequeue it, but it will happen only once overall. One of the 2 credits will be used for this move. Once an item is on stack 2 its pop only costs 1 credit, which is exactly the remaining credit associated to the element. Since each operation's cost is $O(1)$, the amortized cost per operation is $O(1)$.

Exercise 17.3-7

We'll store all our elements in an array, and if ever it is too large, we will copy all the elements out into an array of twice the length. To delete the larger half, we first find the element m with order statistic $\lceil |S|/2 \rceil$ by the algorithm presented in section 9.3. Then, scan through the array and copy out the elements that are smaller or equal to m into an array of half the size. Since the delete half operation takes time $O(|S|)$ and reduces the number of elements by $\lfloor |S|/2 \rfloor \in \Omega(|S|)$, we can make these operations take amortized constant time by selecting our potential function to be linear in $|S|$. Since the insert operation only increases $|S|$ by one, we have that there is only a constant amount of work going towards satisfying the potential, so the total amortized cost of an insertion

is still constant. To output all the elements just iterate through the array and output each.

Exercise 17.4-1

By theorems 11.6-11.8, the expected cost of performing insertions and searches in an open address hash table approaches infinity as the load factor approaches one, for any load factor fixed away from 1, the expected time is bounded by a constant though. The expected value of the actual cost my not be O(1) for every insertion because the actual cost may include copying out the current values from the current table into a larger table because it became too full. This would take time that is linear in the number of elements stored.

Exercise 17.4-2

First suppose that $\alpha_i \geq 1/2$. Then we have

$$\begin{aligned}\hat{c_i} &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + 2 \cdot num_i - size_i - 2 \cdot num_i - 2 + size_i \\ &= -2.\end{aligned}$$

On the other hand, if $\alpha_i < 1/2$ then we have

$$\begin{aligned}\hat{c_i} &= 1 + (size_i/2 - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + size_i/2 - num_i - 2 \cdot num_i - 2 + size_i \\ &= -1 + \frac{3}{2}(size_i - 2 \cdot num_i) \\ &\leq -1 + \frac{3}{2}(size_i - (size_i - 1)) \\ &\leq 1.\end{aligned}$$

Either way, the amortized cost is bounded above by a constant.

Exercise 17.4-3

If a resizing is not triggered, we have

$$\begin{aligned}\hat{c_i} &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\ &= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_i + 2 - size_i| \\ &\leq 1 + |2 \cdot num_i - size_i| - |2 \cdot num_i - size_i| + 2 \\ &= 3\end{aligned}$$

However, if a resizing is triggered, suppose that $\alpha_{i-1} < \frac{1}{2}$. Then the actual cost is $num_i + 1$ since we do a deletion and move all the rest of the items. Also,

since we resize when the load factor drops below $\frac{1}{3}$, we have that $size_{i-1}/3 = num_{i-1} = num_i + 1$.

$$\begin{aligned}
\hat{c}_i &= c_i + \phi_i - \Phi_{i-1} \\
&= num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&\leq num_i + 1 + \left(\frac{2}{3} size_{i-1} - 2 \cdot num_i\right) - (size_{i-1} - 2 \cdot num_i - 2) \\
&= num_i + 1 + (2 \cdot num_i + 2 - 2 \cdot num_i) - (3 \cdot num_i + 3 - 2 \cdot num_i) \\
&= 2
\end{aligned}$$

The last case, that we had the load factor was greater than or equal to $\frac{1}{2}$, will not trigger a resizing because we only resize when the load drops below $\frac{1}{3}$.

Problem 17-1

a. Initialize a second array of length n to all trues, then, going through the indices of the original array in any order, if the corresponding entry in the second array is true, then swap the element at the current index with the element at the bit-reversed position, and set the entry in the second array corresponding to the bit-reversed index equal to false. Since we are running $rev_k < n$ times, the total runtime is $O(nk)$.

b. Doing a bit reversed increment is the same thing as adding a one to the leftmost position where all carries are going to the left instead of the right. See the algorithm BIT-REVERSED-INCREMENT(a)

Algorithm 1 BIT-REVERSED-INCREMENT(a)

```
let m be a 1 followed by k-1 zeroes
while m bitwise-AND a is not zero do
    a = a bitwise-XOR m
    shift m right by 1
end whilereturn m bitwise-OR a
```

By a similar analysis to the binary counter (just look at the problem in a mirror), this BIT-REVERSED-INCREMENT will take constant amortized time. So, to perform the bit-reversed permutation, have a normal binary counter and a bit reversed counter, then, swap the values of the two counters and increment. Do not swap however if those pairs of elements have already been swapped, which can be kept track of in a auxiliary array.

c. The BIT-REVERSED-INCREMENT procedure given in the previous part only uses single shifts to the right, not arbitrary shifts.

Problem 17-2

a. We linearly go through the lists and binary search each one since we don't know the relationship between one list an another. In the worst case, every list is actually used. Since list i has length 2^i and it's sorted, we can search it in $O(i)$ time. Since i varies from 0 to $O(\lg n)$, the runtime of SEARCH is $O((\lg n)^2)$.

b. To insert, suppose we must change the first m 1's in a row to 0's, followed by changing a 0 to a 1 in the binary representation of n. Then we must combine lists $A_0, A_1, \ldots, A_{m-1}$ into list A_m, then insert the new item into list A_m. Since merging two sorted lists can be done linearly in the total length of the lists, the time this takes is $O(2^m)$. In the worst case, this takes time $O(n)$ since m could equal k. However, since there are a total of 2^m items the amortized cost per item is $O(1)$.

c. Find the smallest m such that $n_m \neq 0$ in the binary representation of n. If the item to be deleted is not in list A_m, remove it from its list and swap in an item from A_m, arbitrarily. This can be done in $O(\lg n)$ time since we may need to search list A_k to find the element to be deleted. Now simply break list A_m into lists $A_0, A_1, \ldots, A_{m-1}$ by index. Since the lists are already sorted, the runtime comes entirely from making the splits, which takes $O(m)$ time. In the worst case, this is $O(\lg n)$.

Problem 17-3

a. Since we have $O(x.size)$ auxiliary space, we will take the tree rooted at x and write down an inorder traversal of the tree into the extra space. This will only take linear time to do because it will visit each node thrice, once when passing to its left child, once when the nodes value is output and passing to the right child, and once when passing to the parent. Then, once the inorder traversal is written down, we can convert it back to a binary tree by selecting the median of the list to be the root, and recursing on the two halves of the list that remain on both sides. Since we can index into the middle element of a list in constant time, we will have the recurrence $T(n) = 2T(n/2)+1$, which has solution that is linear. Since both trees come from the same underlying inorder traversal, the result is a BST since the original was. Also, since the root at each point was selected so that half the elements are larger and half the elements are smaller, it is a 1/2-balanced tree.

b. We will show by induction that any tree with $\leq \alpha^{-d} + d$ elements has a depth of at most d. This is clearly true for $d = 0$ because any tree with a single node has depth 0, and since $\alpha^0 = 1$, we have that our restriction on the number of elements requires there to only be one. Now, suppose that in some inductive step we had a contradiction, that is, some tree of

depth d that is α balanced but has more than α^{-d} elements. We know that both of the subtrees are alpha balanced, and by being alpha balanced at the root, we have $root.left.size \leq \alpha \cdot root.size$ which implies $root.right.size > root.size - \alpha \cdot root.size - 1$. So, $root.right.size > (1-\alpha)root.size - 1 > (1-\alpha)\alpha^{-d} + d - 1 = (\alpha^{-1} - 1)\alpha^{-d+1} + d - 1 \geq \alpha^{-d+1} + d - 1$ which is a contradiction to the fact that it held for all smaller values of d because any child of a tree of depth d has depth $d-1$.

c. The potential function is a sum of $\Delta(x)$ each of which is the absolute value of a quantity, so, since it is a sum of nonnegative values, it is nonnegative regardless of the input BST.

   If we suppose that our tree is $1/2-$ balanced, then, for every node x, we'll have that $\Delta(x) \leq 1$, so, the sum we compute to find the potential will be over no nonzero terms.

d. Suppose that we have a tree that has become no longer α balanced because it's left subtree has become too large. This means that $x.left.size > \alpha x.size = (\alpha - \frac{1}{2})x.size + \frac{1}{2}\alpha.size$. This means that we had at least $c(\alpha - \frac{1}{2})x.size$ units of potential. So, we need to select $c \geq \frac{1}{\alpha - \frac{1}{2}}$.

e. Suppose that our tree is α balanced. Then, we know that performing a search takes time $O(\lg(n))$. So, we perform that search and insert the element that we need to insert or delete the element we found. Then, we may have made the tree become unbalanced. However, we know that since we only changed one position, we have only changed the Δ value for all of the parents of the node that we either inserted or deleted. Therefore, we can rebuild the balanced properties starting at the lowest such unbalanced node and working up. Since each one only takes amortized constant time, and there are $O(\lg(n))$ many trees made unbalanced, tot total time to rebalanced every subtree is $O(\lg(n))$ amortized time.

Problem 17-4

a. If we insert a node into a complete binary search tree whose lowest level is all red, then there will be $\Omega(\lg n)$ instances of case 1 required to switch the colors all the way up the tree. If we delete a node from an all-black, complete binary tree then this also requires $\Omega(\lg n)$ time because there will be instances of case 2 at each iteration of the while-loop.

b. For RB-INSERT, cases 2 and 3 are terminating. For RB-DELETE, cases 1 and 3 are terminating.

c. After applying case 1, z's parent and uncle have been changed to black and z's grandparent is changed to red. Thus, there is a ned loss of one red node,

so $\Phi(T') = \Phi(T) - 1$.

d. For case 1, there is a single decrease in the number of red nodes, and thus a decrease in the potential function. However, a single call to RB-INSERT-FIXUP could result in $\Omega(\lg n)$ instances of case 1. For cases 2 and 3, the colors stay the same and each performs a rotation.

e. Since each instance of case 1 requires a specific node to be red, it can't decrease the number of red nodes by more than $\Phi(T)$. Therefore the potential function is always non-negative. Any insert can increase the number of red nodes by at most 1, and one unit of potential can pay for any structural modifications of any of the 3 cases. Note that in the worst case, the call to RB-INSERT has to perform k case-1 operations, where k is equal to $\Phi(T_i) - \Phi(T_{i-1})$. Thus, the total amortized cost is bounded above by $2(\Phi(T_n) - \Phi(T_0)) \leq n$, so the amortized cost of each insert is $O(1)$.

f. In case 1 of RB-INSERT, we reduce the number of black nodes with two red children by 1 and we at most increase the number of black nodes with no red children by 1, leaving a net loss of at most 1 to the potential function. In our new potential function, $\Phi(T_n) - \Phi(T_0) \leq n$. Since one unit of potential pays for each operation and the terminating cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-INSERT-FIXUP $O(1)$.

g. In case 2 of RB-DELETE, we reduce the number of black nodes with two red children by 1, thereby reducing the potential function by 2. Since the change in potential is at least negative 1, it pays for the structural modifications. Since the other cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-DELETE-FIXUP $O(1)$.

h. As described above, whether we insert or delete in any of the cases, the potential function always pays for the changes made if they're nonterminating. If they're terminating then they already take constant time, so the amortized cost of any operation in a sequence of m inserts and deletes is $O(1)$, making the toal amortized cost $O(m)$.

Problem 17-5

a. Since the heuristic is picked in advance, given any sequence of requests given so far, we can simulate what ordering the heuristic will call for, then, we will pick our next request to be whatever element will of been in the last position

of the list. Continuing until all the requests have been made, we have that the cost of this sequence of accesses is $= mn$.

b. The cost of finding an element is $= rank_L(x)$ and since it needs to be swapped with all the elements before it, of which there are $rank_L(x) - 1$, the total cost is $2 \cdot rank_L(x) - 1$.

c. Regardless of the heuristic used, we first need to locate the element, which is left where ever it was after the previous step, so, needs $rank_{L_{i-1}}(x)$. After that, by definition, there are t_i transpositions made, so, $c_i^* = rank_{L_{i-1}}(x) + t_i^*$.

d. If we perform a transposition of elements y and z, where y is towards the left. Then there are two cases. The first is that the final ordering of the list in L_i^* is with y in front of z, in which case we have just increased the number of inversions by 1, so the potential increases by 2. The second is that in L_I^* z occurs before y, in which case, we have just reduced the number of inversions by one, reducing the potential by 2. In both cases, whether or not there is an inversion between y or z and any other element has not changed, since the transposition only changed the relative ordering of those two elements.

e. By definition, A and B are the only two of the four categories to place elements that precede x in L_{i-1}, since there are $|A| + |B|$ elements preceding it, it's rank in L_{i-1} is $|A| + |B| + 1$. Similarly, the two categories in which an element can be if it precedes x in L_{i-1}^* are A and C, so, in L_{i-1}^*, x has rank $|A| + |C| + 1$.

f. We have from part d that the potential increases by 2 if we transpose two elements that are being swapped so that their relative order in the final ordering is being screwed up, and decreases by two if they are begin placed into their correct order in L_i^*. In particular, they increase it by at most 2. since we are keeping track of the number of inversions that may not be the direct effect of the transpositions that heuristic H made, we see which ones the Move to front heuristic may of added. In particular, since the move to front heuristic only changed the relative order of x with respect to the other elements, moving it in front of the elements that preceded it in L_{i-1}, we only care about sets A and B. For an element in A, moving it to be behind A created an inversion, since that element preceded x in L_i^*. However, if the element were in B, we are removing an inversion by placing x in front of it.

g. First, we apply parts b and f to the expression for $\hat{c}_i$ to get $\hat{c}_i \leq 2 \cdot rank_L(x) - 1 + 2(|A| - |B| + t_i^*)$. Then, applying part e, we get this is $= 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t_i^*) = 4|A| - 1 + 2t_i^* \leq 4(|A| + |C| + 1) + 4t_i^* = 4(rank_{L_{i-1}^*}(x) + t_i^*)$. Finally, by part c, this bound is equal to $4c_i^*$.

h. We showed that the amortized cost of each operation under the move to front heuristic was at most four times the cost of the operation using any other heuristic. Since the amortized cost added up over all these operation is at most the total (real) cost, so we have that the total cost with movetofront is at most four times the total cost with an arbitrary other heuristic.