

## Problem Set 4 Solutions

This problem set is due at **11:59pm** on **Thursday, March 5, 2015**.

---

**Exercise 4-1.** Read CLRS, Chapter 17.

**Exercise 4-2.** Exercise 17.1-3.

**Exercise 4-3.** Exercise 17.2-2.

**Exercise 4-4.** Exercise 17.3-2.

**Exercise 4-5.** Read CLRS, Chapter 7.

**Exercise 4-6.** Exercise 7.1-3.

**Exercise 4-7.** Exercise 7.2-5.

**Exercise 4-8.** Exercise 7.4-4.

---

### **Problem 4-1. Extreme FIFO Queues** [25 points]

Design a data structure that maintains a FIFO queue of integers, supporting operations ENQUEUE, DEQUEUE, and FIND-MIN, each in  $O(1)$  amortized time. In other words, any sequence of  $m$  operations should take time  $O(m)$ . You may assume that, in any execution, all the items that get enqueued are distinct.

- (a) [5 points] Describe your data structure. Include clear invariants describing its key properties. *Hint:* Use an actual queue plus auxiliary data structure(s) for bookkeeping.

**Solution:** For example, we might use a FIFO queue  $Main$  and an auxiliary linked list,  $Min$ , satisfying the following invariants:

1. Item  $x$  appears in  $Min$  if and only if  $x$  is the minimum element of some tail-segment of  $Main$ .
2.  $Min$  is sorted in increasing order, front to back.

- (b) [5 points] Describe carefully, in words or pseudo-code, your ENQUEUE, DEQUEUE and FIND-MIN procedures.

**Solution:**

ENQUEUE( $x$ )

- 1 Add  $x$  to the end of  $Main$
- 2 Starting at the end of the list, examine elements of  $Min$  and remove those that are larger than  $x$ ; stop ex
- 3 Add  $x$  to the end of  $Min$

DEQUEUE()

- 1 Remove and return the first element  $x$  of  $Main$
- 2 If  $x$  is the first element in  $Min$ , remove it.

FIND-MIN()

- 1 Return the first element of  $Min$ .

- (c) [5 points] Prove that your operations give the right answers. *Hint:* You may want to prove that their correctness follows from your data structure invariants. In that case you should also sketch arguments for why the invariants hold.

**Solution:** This solution is for the choices of data structure and procedures given above; your own may be different.

The only two operations that return answers are DEQUEUE and FIND-MIN. DEQUEUE returns the first element of  $Main$ , which is correct because  $Main$  maintains the actual queue. FIND-MIN returns the first element of  $Min$ . This is the smallest element of  $Min$  because  $Min$  is sorted in increasing order (by Invariant 2 above). The smallest element of  $Main$  is the minimum of the tail-segment consisting of all of  $Main$ , which is the smallest of all the tail-mins of  $Main$ . This is the smallest element in  $Min$  (by Invariant 1). Therefore, FIND-MIN returns the smallest element of  $Main$ , as needed.

*Proofs for the invariants:* The invariants are vacuously true in the initial state. We argue that ENQUEUE and DEQUEUE preserve them; FIND-MIN does not affect them.

It is easy to see that both operations preserve Invariant 2: Since a DEQUEUE operation can only remove an element from  $Min$ , the order of the remaining elements is preserved. For ENQUEUE( $x$ ), we remove elements from the end of  $Min$  until we find one that less than  $x$ , and then add  $x$  to the end of  $Min$ . Because  $Min$  was in sorted order prior to the ENQUEUE( $x$ ), when we stop removing elements, we know that all the remaining elements in  $Min$  are less than  $x$ . Since we do not change the order of any elements previously in  $Min$ , all the elements are still in sorted order.

So it remains to prove Invariant 1. There are two directions:

- The new  $Min$  list contains all the tail-mins.  
ENQUEUE( $x$ ):  $x$  is the minimum element of the singleton tail-segment of  $Main$  and it is added to  $Min$ . Additionally, since every tail-segment now contains the value  $x$ , all elements with value greater than  $x$  can no longer be tail-mins. So, after their removal,  $Min$  still contains all the tail-mins.

DEQUEUE of element  $x$ : The only element that could be removed from  $Min$  is  $x$ . It is OK to remove  $x$ , because it can no longer be a tail-min since it is no longer in  $Main$ . All other tail-mins are remain in  $Min$ .

- All elements of the new  $Min$  are tail-mins.  
 ENQUEUE( $x$ ):  $x$  is the only value that is added to  $Min$ . It is the min of the singleton tail-segment. Every other element  $y$  remaining in the  $Min$  list was a tail-min before the ENQUEUE and is less than  $x$ . So  $y$  is still a tail-min after the ENQUEUE.

DEQUEUE of element  $x$ : Then we claim that, if  $x$  is in  $Min$  before the operation, it is the first element of  $Min$  and therefore is removed from  $Min$  as well. Now, if  $x$  is in  $Main$ , it must be the minimum element of some tail of  $Main$ . This tail must include the entire queue, since  $x$  is the first element of  $Main$ . So  $x$  must be the smallest element in  $Min$ , which means it is the first element of  $Min$ . Every other element  $y$  in  $Min$  was a tail-min before the DEQUEUE, and is still a tail-min after the DEQUEUE.

- (d) [10 points] Analyze the time complexity: the worst-case cost for each operation, and the amortized cost of any sequence of  $m$  operations.

**Solution:** DEQUEUE and FIND-MIN are  $O(1)$  operations, in the worst case.

ENQUEUE is  $O(m)$  in the worst case. To see that the cost can be this large, suppose that ENQUEUE operations are performed for the elements  $2, 3, 4, \dots, m-1, m$ , in order. After these,  $Min$  contains  $\{2, 3, 4, \dots, m-1, m\}$ . Then perform ENQUEUE(1). This takes  $\Omega(m)$  time because all the other entries from  $Min$  are removed one by one. However, the amortized cost of any sequence of  $m$  operations is  $O(m)$ . To see this, we use a potential argument. First, define the actual costs of the operations as follows: The cost of any FIND-MIN operation is 1. The cost of any DEQUEUE operation is 2, for removal from  $Main$  and possible removal from  $Min$ . The cost of an ENQUEUE operation is  $2 + s$ , where  $s$  is the number of elements removed from  $Min$ . Define the potential function  $\Phi = |Min|$ .

Now consider a sequence  $o_1, o_2, \dots, o_m$  of operations and let  $c_i$  denote the actual cost of operation  $o_i$ . Let  $\Phi_i$  denote the value of the potential function after exactly  $i$  operations; let  $\Phi_0$  denote the initial value of  $\Phi$ , which here is 0. Define the amortized cost  $\hat{c}_i$  of operation instance  $o_i$  to be  $c_i + \Phi_i - \Phi_{i-1}$ .

We claim that  $\hat{c}_i \leq 2$  for every  $i$ . If we show this, then we know that the actual cost of the entire sequence of operations satisfies:

$$\sum_{i=1}^m c_i = \sum_{i=1}^m \hat{c}_i + \Phi_0 - \Phi_m \leq \sum_{i=1}^m \hat{c}_i \leq 2m.$$

This yields the needed  $O(m)$  amortized bound.

To show that  $\hat{c}_i \leq 2$  for every  $i$ , we consider the three types of operations. If  $o_i$  is a FIND-MIN operation, then

$$\hat{c}_i = 1 + \Phi_i - \Phi_{i-1} = 1 < 2.$$

If  $o_i$  is a DEQUEUE, then since the lengths of the lists cannot increase, we have:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \leq 2 + 0 \leq 2.$$

If  $o_i$  is an ENQUEUE, then

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \leq 2 + s - s = 2,$$

where  $s$  is the number of elements removed from *Min*. Thus, in every case,  $\hat{c}_i \leq 2$ , as claimed.

Alternatively, we could use the accounting method. Use the same actual costs as above. Assign each ENQUEUE an amortized cost of 3, each DEQUEUE an amortized cost of 2, and each FIND-MIN an amortized cost of 1. Then we must argue that

$$\sum_{i=1}^m \hat{c}_i \geq \sum_{i=1}^m c_i$$

for any sequence of operations and costs as above. This is so because each ENQUEUE( $x$ ) contributes an amortized cost of 3, which covers its own actual cost of 2 plus the possible cost of removing  $x$  from *Min* later.

#### Problem 4-2. Quicksort Analysis [25 points]

In this problem, we will analyze the time complexity of QUICKSORT in terms of error probabilities, rather than in terms of expectation. Suppose the array to be sorted is  $A[1..n]$ , and write  $x_i$  for the element that starts in array location  $A[i]$  (before QUICKSORT is called). Assume that all the  $x_i$  values are distinct.

In solving this problem, it will be useful to recall a claim from lecture. Here it is, slightly restated:

**Claim:** Let  $c > 1$  be a real constant, and let  $\alpha$  be a positive integer. Then, with probability at least  $1 - \frac{1}{n^\alpha}$ ,  $3(\alpha + c) \lg n$  tosses of a fair coin produce at least  $c \lg n$  heads.

**Note:** High probability bounds, and this Claim, will be covered in Tuesday's lecture.

- (a) [5 points] Consider a particular element  $x_i$ . Consider a recursive call of QUICKSORT on subarray  $A[p..p+m-1]$  of size  $m \geq 2$  which includes element  $x_i$ . Prove that, with probability at least  $\frac{1}{2}$ , either this call to QUICKSORT chooses  $x_i$  as the pivot element, or the next recursive call to QUICKSORT containing  $x_i$  involves a subarray of size at most  $\frac{3}{4}m$ .

**Solution:** Suppose the pivot value is  $x$ . If  $\lfloor \frac{m}{4} \rfloor + 1 \leq x \leq m - \lfloor \frac{m}{4} \rfloor$ , then both subarrays produced by the partition have size at most  $\frac{3m}{4}$ . Moreover, the number of values of  $x$  in this range is at least  $\frac{m}{2}$ , so the probability of choosing such a value is at least  $\frac{1}{2}$ . Then either  $x_i$  is the pivot value or it is in one of the two segments.

- (b) [9 points] Consider a particular element  $x_i$ . Prove that, with probability at least  $1 - \frac{1}{n^2}$ , the total number of times the algorithm compares  $x_i$  with pivots is at most  $d \lg n$ , for a particular constant  $d$ . Give a value for  $d$  explicitly.

**Solution:** We use part (a) and the Claim. By part (a), each time QUICKSORT is called for a subarray containing  $x_i$ , with probability at least  $\frac{1}{2}$ , either  $x_i$  is chosen as the pivot value or else the size of the subarray containing  $x_i$  reduces to at most  $\frac{3}{4}$  of what it was before the call. Let's say that a call is "successful" if either of these two cases happens. That is, with probability at least  $\frac{1}{2}$ , the call is successful.

Now, at most  $\log_{4/3} n$  successful calls can occur for subarrays containing  $x_i$  during an execution, because after that many successful calls, the size of the subarray containing  $x_i$  would be reduced to 1. Using the change of base formula for logarithms,  $\log_{4/3} n = c \lg n$ , where  $c = \log_{4/3} 2$ .

Now we can model the sequence of calls to QUICKSORT for subarrays containing  $x_i$  as a sequence of tosses of a fair coin, where heads corresponds to successful calls. By the Claim, with  $c = \log_{4/3} 2$  and  $\alpha = 2$ , we conclude that, with probability at least  $1 - \frac{1}{n^2}$ , we have at least  $c \lg n$  successful calls within  $d \lg n$  total calls, where  $d = 3(2 + c)$ . Each comparison of  $x_i$  with a pivot occurs as part of one of these calls, so with probability at least  $1 - \frac{1}{n^2}$ , the total number of times the algorithm compares  $x_i$  with pivots is at most  $d \lg n = 3(2 + c) \lg n = 3(2 + \log_{4/3} 2) \lg n$ . The required value of  $d$  is  $3(2 + \log_{4/3} 2) \leq 14$ .

- (c) [6 points] Now consider all of the elements  $x_1, x_2, \dots, x_n$ . Apply your result from part (b) to prove that, with probability at least  $1 - \frac{1}{n}$ , the total number of comparisons made by QUICKSORT on the given array input is at most  $d'n \lg n$ , for a particular constant  $d'$ . Give a value for  $d'$  explicitly. *Hint:* The Union Bound may be useful for your analysis.

**Solution:** Using a union bound for all the  $n$  elements of the original array  $A$ , we get that, with probability at least  $1 - n(\frac{1}{n^2}) = 1 - \frac{1}{n}$ , every value in the array is compared with pivots at most  $d \lg n$  times, with  $d$  as in part (b). Therefore, with probability at least  $1 - \frac{1}{n}$ , the total number of such comparisons is at most  $dn \lg n$ . Using  $d' = d$  works fine.

Since all the comparisons made during execution of QUICKSORT involve comparison of some element with a pivot, we get the same probabilistic bound for the total number of comparisons.

- (d) [5 points] Generalize your results above to obtain a bound on the number of comparisons made by QUICKSORT that holds with probability  $1 - \frac{1}{n^\alpha}$ , for any positive integer  $\alpha$ , rather than just probability  $1 - \frac{1}{n}$  (i.e.,  $\alpha = 1$ ).

**Solution:** The modifications are easy. The Claim and part (a) are unchanged. For part (b), we now prove that with probability at least  $1 - \frac{1}{n^{\alpha+1}}$ , the total number of times the algorithm compares  $x_i$  with pivots is at most  $d \lg n$ , for  $d = 3(\alpha + c)$ . The argument is the same as before, but we use the Claim with the value of  $\alpha$  instead of 2. Then for part (c), we show that with probability at least  $1 - \frac{1}{n^\alpha}$ , the total number of times the algorithm compares any value with a pivot is at most  $dn \lg n$ , where  $d = 3(\alpha + c)$ .

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.046J / 18.410J Design and Analysis of Algorithms  
Spring 2015

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.