Homework 3: Stacks and Queues

Elias Gabriel

1. Explain how a queue can be implemented using two stacks. Then, prove that your stack-based queue has the property that any sequence of n operations (where an operation is either an `enqueue` or `dequeue`) takes a total of O(n) time resulting in amortized O(1) time for each of these operations. You may use any of the amortized analysis techniques that we discussed in class.

Solution: To implement a queue using two stacks, one of them can serve as a inbox and the other as an outbox. Procedurally, the functions would follow a very simple set of steps. Given two stacks, A and B respectively:

- To enqueue an item onto the queue, simply push it onto stack A.
- To dequeue an item:
  (a) If stack B is not empty, simply pop the top-most element of that stack.
  (b) If stack B is empty, then pop all the items off of A one by one and push them onto B. Once A is empty, pop B as normal.

Following the procedure above, the enqueue operation will always be O(1) since pushing onto a stack is constant time. Therefore, a series of n enqueues will always be n operations in O(n) time.

The dequeue operation is different, as it requires occasionally moving items from stack A to B. However, each unique element only has to move to between A and B once, meaning that every element gets pushed twice and popped twice. Using the aggregate method:

$$nO(1) = \frac{O(n)}{n} = O(1) \text{ amortized} \qquad (\texttt{enqueue})$$

$$nO(1) + nO(2) + nO(1) = \frac{O(4n)}{n} = O(1) \text{ amortized} \qquad (\texttt{dequeue})$$

2. (6 points) A deque (a double-ended queue) is an abstract data type that generalizes a queue. It supports the following operations
   (a) `pop_back:` remove the element at the back of the deque.
   (b) `pop_front:` remove the element at the front of the deque.

(c) `push_back:` add a given element to the back of the deque.

(d) `push_front:` add a given element to the front of the deque.

We can implement a deque using three stacks A, B, and C. Stack A will correspond to the front of the deque, stack B will correspond to the back of the deque, and stack C will be a helper stack, which we will use to ensure that stack A and B are balanced. Whenever we want to push an element x to the front of the deque, we push it onto stack A. Similarly, whenever we want to push an element to the back of the deque, we push it onto to stack B.

If A is not empty, we can pop an element from the front of the deque by popping off stack A. However, if A is empty and B is non-empty, we first transfer half the elements off of B and push them onto A, which reverses the order. Second, we pop the remaining half of elements on B and add them to C, again reversing the order. Third, we pop the elements off of A and add them to B, putting them back into the original order. Last, we swap the labels of stacks A and C. Thus, the items on the new stack A are the front half of the deque from front to middle, and the items on stack B are the back half of the deque from back to middle. See below. Similarly, if B is not empty, we pop an element from the back of the deque by popping off of stack B. Otherwise, we use C to split the elements between the two stacks as before.

$$A = [\,], B = [4, 3, 2, 1], C = [\,]$$
$$A = [3, 4], B = [2, 1], C = [\,]$$
$$A = [3, 4], B = [\,], C = [1, 2]$$
$$A = [\,], B = [4, 3], C = [1, 2]$$
$$C = [\,], B = [4, 3], A = [1, 2]$$

Prove that any sequence of n operations (where an operation is either a pop_front, pop_back, push_front, or push_back) takes a total of O(n) time resulting in amortized O(1) time for each of these operations. You should use an amortized analysis technique different from what you used in problem 1.

Solution: To prove that all the operations in a deque can be O(1), we can take advantage of the Banker's method of analysis. Each of the operations has a cost associated with it:

operation	cost
push_front	6
push_back	6
pop_front	0
pop_back	0

The costs assigned above correspond with the number of operations that any element in a list needs to undergo throughout an entire process of enqueueing and dequeueing.

Much like in a single-ended queue, pushing elements does not require anything more than a single operation so its amortized cost is 1. Applying the pay-it-forward mentality, the cost of popping the elements needs to be taken into account, since a push will necessarily always result in an eventual pop.

In the worst case scenario, the entire queue will have to undergo the described "juggling" method for however many times necessary to keep the internal stacks in balance. We know that a series of k pushes and k pops from either the front or end would be the worst case for the internal balancing, as well as that a juggle will have to occur every time one of the stack are empty and that one of the stacks are empty only after half of the queue elements are gone. Counting the number of times required to halve n elements repeatedly until left with a single number is equivalent to $\log_2(n)$. Using that knowledge, we can estimate the amortized number of times a given element will have to be juggled. Using an example where $n = 16$:

$$\sum_{i=1}^{\lceil \log_2(n) \rceil - 1} \frac{2^i}{n} = \frac{8+4+2}{8} = 1 + \frac{1}{2} + \frac{1}{4} \longrightarrow 2 \tag{1}$$

As the number of elements in the deque tends toward infinity, the amortized number of juggles that happen approaches the constant value 2 rather than increasing linearly. Logically, that indicates that there is a constant upper bound. Establishing that, the cost for the `pop` operations is simply $2 * p$, where p is the number of constant-time operations in a single juggle. In the method described above, each element goes through a maximum of 2 pushes and 2 pops as it is juggled between stacks A, B, and C, making $p = 4$. Thus, the price for the `pop` operation is amortized $4 + 1$ (4 from the juggle and 1 for the actual pop), which is added to the constant 1 `push` price for a total cost of 6.

Finally, calculating the actual amortized cost using the Banker's method shows that all the `push` and `pop` operations are effectively constant in O(1):

$$\begin{aligned} nO(6) &= \frac{O(6n)}{n} = O(1) \text{ amortized} && (\texttt{push\_front \& push\_back}) \\ nO(0) &= \frac{O(n)}{n} = O(1) \text{ amortized} && (\texttt{pop\_front \& pop\_back}) \end{aligned} \tag{2}$$

3. (12 points) Construct a data structure that implements the following operations:

   (a) `enqueue`: Adds an element to the set.

   (b) `dequeue`: Removes and returns the oldest element in the set.

   (c) `find_min`: Returns the minimum value in the set.

   This is a queue with the added operation of finding the minimum element. Prove the correctness of your data structure and analyze the complexity using amortized analysis. You

should use an amortized analysis technique different from what you used in problems 1 and 2.

Solution:

A simple queue can be implemented using a doubly-linked list and a separate array (or Python list) to keep track of the queue minimum(s). In order to enqueue something, it simply needs to be pushed to the front of the DLL, updating the head node for consistency and sanity. To dequeue, the tail node needs to be popped and updated accordingly. In that way, the order of the elements can be preserved and the structure can adhere to the FIFO principle.

To keep track of the minimum, the internal list can be used. Whenever an element is enqueued, the elements would need to be compared to all the existing elements in the minimum list. Then, each element that is less than the new value can simply be removed. After pruning, the new low value can be added. Following this principle, the queue's minimum value will always be the first element in the list, and thus can be accessed in constant time.

The correctness of my queue implementation can be proved by showing that it behaves as intended for 1+ elements. If a single element is added to the queue, we can guarantee that it will be the first the exit when it is popped. In the case where multiple items are pushed, we know that they are always inserted at the head of the internal linked list. Because they are always inserted at the head and since their insertion doesn't affect the ordering or stability of the rest of the DLL, we know that the order of elements in the queue during insertion is preserved for both the i^{th} push and the $i^{th}+1$ push. Thus, the elements are popped from the queue in the order that they were pushed.

In terms of time complexity, we can use the Potential Method to calculate amortized cost for the implemented `enqueue` and `dequeue` methods:

$$\Phi(Q) = \text{length of internal minimum list} \tag{3}$$

$$\text{amortized cost} = \text{actual cost} + \Phi(Q_k) - \Phi(Q_{k-1}) \tag{4}$$

$$O_{\texttt{enqueue}} = 1 + n + 1 - n = 2 = O(1) \text{ amortized} \qquad (\texttt{enqueue})$$

$$O_{\texttt{dequeue}} = 1 + n - 1 - n = 0 = O(1) \text{ amortized} \qquad (\texttt{dequeue})$$

$$O_{\texttt{find\_min}} = 1 + n - n = 1 = O(1) \text{ amortized} \qquad (\texttt{find\_min})$$