Homework #3

Megan Ku
Data Structures and Algorithms

February 14, 2020

1

Implementing a Queue using Two Stacks

A stack is a "first in, last out" data type, while a queue is a "first in, first out" data type. Implementing the `enqueue` operation is easy; I can simply `push` to the first stack. The second stack becomes useful when implementing the `dequeue` operation. When I need to `dequeue`, I simply `pop` every element in the first stack, `push`-ing each element that I `pop` into the second stack. The second stack should then contain the elements of the first stack, but in reverse order. Now, I can simply `pop` the items from the stack that I wanted to `dequeue` in the first place. When `dequeue` is called and the second stack is empty, all the elements of the first stack must be `pop`ped and `push`-ed into the second stack. While the second stack has more than zero elements, though, `dequeue`-ing can be done by simply `pop`ing from the second stack.

Runtime Analysis

The runtime will be analyzed using the aggregate method. Let n be the number of elements in the stack-based queue. The number of elements will be greater than or equal to the number of times `enqueue` is called, which is greater than or equal to the number of times `dequeue` is called.

$$\#\texttt{dequeue} \leq \#\texttt{enqueue} \leq n \tag{1}$$

The `enqueue` operation runs in O(1), or constant time, because `pop` runs in O(1) time.

The `dequeue` operation runs in O(1) amortized, because the operation runs in O(1) until the `dequeue` stack is empty. When the `dequeue` stack is empty, the operation takes O(n) time to move the elements from the `enqueue` stack into the `dequeue` stack.

Thus, the total runtime for n operations is:

$$O(n) + O(n) = O(n) \tag{2}$$

Dividing over n operations gives an amortized runtime of O(1).

2

Three-Stack Deque Runtime Analysis

I will use the banker's (or accounting) method to do amortized analysis on the three-stack deque. Let n be the number of elements in the deque. I will assign costs to three operations: `push` (which covers both `push_front` and `push_back`), `pop` (covers both `pop_front` and `pop_back`), and `transfer`, which accounts for the reshuffling of stacks when stacks A or B are empty and the `pop` operation is called. I will assign the `transfer` operation a cost of 0 rubles, as I know that it is the most costly operation. I'll then assign `push` a cost of 7 rubles: 1 ruble for the `push`, and 6 rubles for the three pairs of `push`-es and `pop`-s that happen as the element gets shifted. Finally, I'll assign `pop` a cost of 7 rubles: 1 ruble for the `pop`, and 6 rubles for the potential movement that an element on the other stack may execute if a stack is empty.

operation	cost (rubles)
`push`	7
`pop`	7
`transfer`	0

At any time, the total cost will be greater than or equal to the actual cost. Our total cost for n operations would be 7*n rubles, which means that the actual cost will be less than or equal to 7*n rubles. This equates to a total runtime of O(n). Dividing this over n operations gives us an amortized runtime of O(1).

3

Data Structure with Enqueue, Dequeue, and Find Min

I can implement a data structure that can `enqueue`, `dequeue`, and `find_min` with two doubly-linked lists (DLLs). One DLL will actually hold the elements of the queue, while the other list will keep track of the current minimum value.

If I want to `enqueue` into the DLL, I can simply make it the new head node of the DLL. This requires the following operations: create a node that stores the given value and assigns it to be the new head of the DLL, point the old head.prev attribute to the new head, and point the new head.next to the old head. This ensures that the elements remain doubly-linked.

For the `dequeue` operation, I can remove the tail by assigning the new tail to be tail.prev. Then, I can assign the new tail.next to be None. Like with `enqueue`, these assignments preserve the two-way pointers.

To keep track of the minimum value, I can use a second DLL to store the current and future minimum values, with the head node containing the current minimum value. When

I `enqueue` an element to the queue DLL, I can `enqueue` that same element into the second DLL as the new tail node. I then compare the tail node to tail.prev. If tail.prev is greater than the tail node, I remove tail.prev from the DLL. I then check to see if the new tail.prev is larger and remove it if it is. I repeat this process, comparing tail with tail.prev, until tail.prev is less than or equal to the tail.

If an element gets dequeued, I can check to see if the element has the same value as the head of the DLL. If so, I remove the head node from the minimum DLL.

Proof of Correctness

Because elements are only added to the head of the DLL and removed from the tail of the DLL, order is preserved, and the queue keeps its property of being "first in, first out". Also, the pointers to each node's previous and next node are also preserved, keeping the functionality of the doubly-linked list.

The minimum DLL holds the current and future minimums of the queue, and accounts for the `dequeue` of the current minimum by dropping the head node when they match. By `enqueue`ing every element into the minimum DLL, we can ensure that we are adding the minimum, which would be the last element if all other elements were to be `dequeue`d from the main DLL. If the new node is lower than previous values in the minimum DLL, we can guarantee that those previous values will never be the minimum, because those elements will be `dequeue`d before the new element will. Therefore, removing values greater than the most recently added element will not result in the DLL losing the current or future minimums.

Runtime Analysis: Enqueue

The runtime will be analyzed using the potential method. Let Let

$$\Phi = \text{number of elements in minimum DLL} \tag{3}$$

The amortized cost of the kth operation is the following:

$$k^{th}\text{cost} = \text{cost} + \Phi(DS_k) - \Phi(DS_{k-1}) \tag{4}$$

For `enqueue`-ing, I considered the worst case scenario for this data structure, which involves `dequeue`-ing the entire min DLL to update the minimum value: `enqueue`-ing a new minimum after adding several elements that were strictly ascending. The potential function is the following:

$$\text{cost} = (2 + n) + 1 - n = 3 \tag{5}$$

where the actual cost of the function is 2 + n (enqueueing to both the main DLL and the DLL that keeps track of the minimum costs 2, while dequeue-ing the min list to update the minimum costs the length of the queue). Because the cost of the kth operation is a constant, I can conclude that `enqueue` has a runtime of O(1) amortized.

Runtime Analysis: Dequeue

For `dequeue`-ing, the potential function is the following:

$$\text{cost} = 2 + (n - 1) - n = 1 \tag{6}$$

where the actual cost of the function is 2 (dequeueing from both the main DLL and the min DLL if the element was the current min). Because the cost of the k^{th} operation is constant, then I can conclude that `enqueue` has a runtime of O(1) amortized.

Runtime Analysis: Find Min

For `find_min`, the potential function is the following:

$$\text{cost} = 1 + (n + 1) - n = 2 \tag{7}$$

where the actual cost of the function is 1 (returning the value of the head node of the DLL). Because the cost of the k^{th} operation is constant, then I can conclude that `enqueue` has a runtime of O(1) amortized.