Homework 3

Sam Daitzman // DSA Spring 2020

---

1. Queue implemented using stacks.

> 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.

A queue is a data structure that supports the enqueue (add to end of list) and enqueue (take from front of list) operations. Queues are first-in first-out linear data structures.

A stack is a data structure that supports pop (remove from start of list and return) and push (add to front of list) operations. Stacks are last-in first-out linear data structures.

To implement a queue with two stacks, you can use one stack to hold the incoming "enqueued" values. To dequeue a value, you reverse the input stack by popping each value and placing them into the other stack. The first item removed from that stack will be the first one added to the first stack, because it'll be the last that was added to the second, and so on. Functionally, this reverses the stack.

Amortized Analysis: Banker's Method

The two operations supported here are **enqueue** and **dequeue**. We know that the enqueue operation is basically free; it's $O(1)$ since enqueueing only adds it to the "in" stack. So we will assign its cost: 1 ruble. This means that n enqueues will cost n rubles.

Dequeue is more complicated. Our first dequeue is much more expensive–it costs $O(n)$ time, where n is the number of items on the "in" stack that must be moved to the "out" stack. Luckily, that first operation gives us unlimited future dequeue operations of items in that stack, up to n dequeues. So, we'll say that the first dequeue costs n rubles, where n is the number of items currently in that stack. And each dequeue after that, up to n, will cost 0 rubles, because it has already been paid for.

Across both operations, any sequence of n operations will cost n rubles on average, and therefore a sequence of n operations will take $O(n)$ time. Thus, the amortized time will be $\frac{O(n)}{n} = O(1)$.

2. Deque Analysis: Aggregate Method

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.
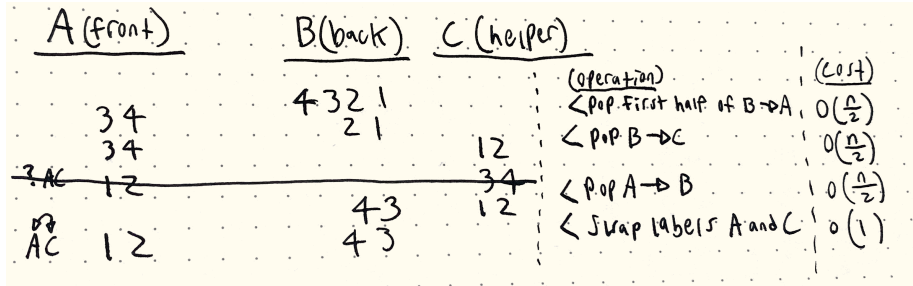

Figure 1: Cost analysis of three-stack deque

Operations (c) and (d), pushing to the front or back, will be constant-time because pushing to a stack is basically always $O(1)$. This means that across n operations, this structure will take $O(1) \times n = O(n)$ time.

The pop operations are more complicated. The procedure that balances the front and back of the queue takes a series of steps. First, we pop the first half of stack B onto stack A, which is $O(\frac{n}{2})$ because it takes n/2 operations, each of which is individually $O(1)$. Second, we pop the entire B stack onto the helper C stack, which again is $O(\frac{n}{2})$. Third, we pop the entire A stack onto the B stack to reverse it, which is also $O(\frac{n}{2})$. Lastly, we swap the labels of the A and C stacks, which is $O(1)$.

Considering the aggregate of these operations intuitively, it makes sense that this would be $O(n)$ because our most expensive step in a series of operations is always a constant (1/2) times $O(n)$.

Under the aggregate method:

(swap labels A and C) $\leq$ (pop first half) $\leq$ (pop B to C) $\leq$ (pop A to B)

$$O(1) \leq O(\frac{n}{2}) \leq O(\frac{n}{2}) \leq O(\frac{n}{2})$$

This operation gives us $\frac{n}{2}$ pops from a half of the list with $O(\frac{n}{2})$ time, so it comes out as:

$$\frac{O(\frac{n}{2})}{\frac{n}{2}} = O(1)\text{amortized}$$

This works for pops from both sides of the list, since the operational cost will be the same. Thus, we have that all operations on this form of queue will take $O(1)$ amortized time.

3. Data Structure Implementation

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) *findmin*: Returns the minimum value in the set.

See `./sdaitzman.py` for my implementation.

Prove the Correctness of Your Data Structure

Enqueue will always work as expected, since it's just adding items to the beginning of the queue.

At our first dequeue, we flip the entire "in" stack. In this base case, we know that we'll then get the first item that was added because it will be the last off the "in" stack, and therefore the last onto the "out" stack and first off of it. For every pop thereafter, the same holds. We know that this process can repeat until the final dequeue from the "out" stack, and at that point the flipping of the "in" stack will behave the same as it did previously.

Big-O Analysis: Potential Method

I've checked in with around 6 other students, and Dieter explained his approach to this problem to me in some detail.

The potential function $\Phi(DS)$, where h is the data structure at a particular state, tells us how much precharged time is available to pay for expensive operations down the road. Intuitively, I think of the potential function as the computer

science equivalent of riding a bicycle up a hilly road, where reaching the top of a series of small hills affects riding down other hills later.

In the case of this problem, at first I chose the following potential function:

$$\Phi(DS) = \text{length of "in" stack}$$

Since the length of the "in" stack will always be ≥ 0, but I'm confused about this approach's correctness because when the second stack contains some data, the structure as a whole is not empty but $\Phi(DS) \neq 0$, which I believe violates the potential function rules.

I moved on to the following potential function:

$$\Phi(DS) = I - O, \text{ for } I - O \geq 0$$

$$\Phi(DS) = 0, \text{ for } I - O < 0$$

$$I = \text{length of input stack}; O = \text{length of output stack}$$

This meets the following conditions:

- When the data structure is empty, $\Phi(DS) = 0$
- $\Phi(DS) \geq 0$

Enqueue

Enqueue needs to perform 2 appends, one for the value and one for the minimum tracker.

$$\text{Amortized Enqueue} = 2 + \Phi(DS_k) - \Phi(DS_{k-1}) = 3 \Rightarrow O(1)$$

Dequeue

Dequeue needs to perform an expensive operation every once in a while to flip the "in" stack, but the piecewise function ends up providing enough operations that the amortized time will still be $O(1)$.

$$\text{Amortized Dequeue} = 1 + \Phi(DS_k) - \Phi(DS_{k-1}) = 1 + 1 = 2 \Rightarrow O(1)$$

Find_Min

This operation is necessarily O(1) because it results in no change in the data structure.

$$\text{Amortized find\_min} = 1 + \Phi(DS_k) - \Phi(DS_{k-1}) = 1 + 0 = 1 \Rightarrow O(1)$$

Questions to Follow Up On/Think More About

- In functions that are of big-O time $O(n*log(n))$, what base is the logarithm? Does it matter?
- How do we choose a potential function? ~~Where can I find the table that Alice showed in class?~~
  - https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)
- How would I go about finding a useful potential function for the first exercise? What about the second one?
- I'm **really confused about the potential method**.