DSA Homework 3

Katie Foster

Part 1

A queue can be implemented using two stacks with one that you push to and one that you pop from. Once you try to dequeue an element, if the pop stack is empty, every element in the push stack is popped and pushed into the pop stack. This operation essentially copies the push stack into the pop stack but in reverse order.

I used the aggregate method to do amortized analysis on the runtime of enqueues and dequeues on this function.

$$number\ of\ pops \leq number\ of\ pushes \leq n$$
$$O(n) + O(n) = O(n)$$
$$O(n) / n\ operations = O(1)\ amortized$$

Whenever the program has to copy over the push list to the pop list once pop runs out, pop takes *O(current length of list * n)*. This worst case is not too bad since if it occurs often, the current length of the list is not very long, so the operation does not add too much time.

Part 2

I used the Banker's method to do amortized analysis on the double ended queue (deque) structure, which uses three stacks.

ℓ = length of list

If you try to pop but the list is empty, the following actions need to happen:

1. ½ of elements of B are pushed to A, which takes $O(n*2*½*\ell)$
2. Remaining ½ of elements are pushed off B to C, which takes $O(n*2*½*\ell)$
3. All elements from A are popped and added to B, putting them back in order, which takes $O(n*2*½*\ell)$
4. Labels of stack A and stack B are swapped, which takes $O(1)$

So the sum of paying for a pop in advance, taking into account this scenario, is:

$$O(n*2*½*\ell)*3 + O(1)$$
$$O(3*n*\ell)$$

Each time a pop necessitates the swapping of elements, one element is removed, it is possible that the remaining elements might be swapped around again. That means that whenever an item is popped, it should pay

$$O(3*n*\ell)$$

again to account for the further swapping of elements. So overall, n pushes take, $O(3*n*\ell)$ and n pops take, $O(3*n*\ell)$.

$$O(3*n*\ell)$$

This means that for n operations, it would cost a maximum of 3n.

$$O(3n) / n \text{ operations}$$
$$O(3)$$

So each element has to "pay" 3 rubles when added to the list in order to account for being moved around during the next shuffle, and has to "pay" 3 rubles when popped to account for further shuffling. In order to find the amortized cost, we can set the paid forward cost to less than or equal to the total cost, and calculate from there.

$$\text{total cost} \leq O(3n)$$
$$O(3n) \text{ simplifies to: } O(n)$$
$$O(n) / n \text{ operations} = O(1) \text{ amortized}$$

Part 3

In order to construct a data structure to allow the operations enqueue, dequeue, and find minimum, I would use two doubly linked lists. One of the lists is the main list which keeps track of all the elements in the queue. The other list is the minimum list, which keeps track of the minimum of the first list. Both lists have the head as the newest item and the tail as the oldest element.

When enqueue is called, the new item is added to the head of both the lists. Then the min list iterates through all items behind the head, and deletes any of these items with a smaller value than than the head.

When deque is called, the main list just removes its tail. The min list checks if the value of the item being removed is equal to the value of the tail of the min list. If the values are equal, the min list removes its tail. If the values are not equal, it does not do anything.

When find_min is called, the function returns the value of the tail of the min list.

This method will correctly keep track of the numbers in the main list because it uses a doubly linked list and always only adds elements at the head and remove elements at the tail. The minimum list will correctly keep track of the minimum because it will keep track of the current minimum and all other possible values that the minimum could be if other elements were dequeued.

I used the potential method to find the amortized cost of the operations in this data structure.

The enqueue operation takes:

$$Amortized\ cost = 2 + k + \Phi(DS_i) - \Phi(DS_{i-1})$$

where two is the cost to push the element to both lists, k is the number of elements that are removed from the minimum function, and $\Phi(DS_i)$ is the number of elements on the minimum list.

The change in the number of items on the minimum list $2 + k + \Phi(DS_i) - \Phi(DS_{i-1})$ is equal to k, so we can simplify the above equation to:

$$Amortized\ cost = 2 + k - k$$

$$Amortized\ cost = 2$$

which gives us the answer that the amortized cost of the enqueue operation is equal to two.

The dequeue operation takes:

$$Amortized\ cost = 1 + k + \Phi(DS_i) - \Phi(DS_{i-1})$$

where one is the cost to compare the items, k is the number of items being popped off, which is either 1 or 2, depending on whether or not an item is popped off the min list. Once again we can substitute the difference in the potential function for k, leaving us with the function:

$$Amortized\ cost = 1 + k - k$$

$$Amortized\ cost = 1$$

The function to find min is even simpler. First we write out the equation:

$$Amortized\ cost = 1 + \Phi(DS_i) - \Phi(DS_{i-1})$$

However there are no items being added or removed, so the difference in the potential function is equal to 0. We can use this substitution to simplify the function to:

$$Amortized\ cost = 1$$