Prob. 1	Prob. 2	Prob. 3

Problem 1.

Stack is capable of inserting and deleting at the end of a list but queue accepts elements from one end and elements are removed from the other end so they have quite different operational semantics. However, a queue can easily be simulated by two stacks, namely *inbox*, S_i and *outbox*, S_o.

We will use S_i for Insert-Queue and S_o for Delete-Queue operation. Therefore, we push the element into S_i if we want to insert a new element into queue. Delete operation contains two scenarios according to state of S_o. If it isn't empty, we pop its top element; otherwise, we transfer all elements from S_i to S_o respectively by popping from S_i and pushing into S_o. Then, we have the required element at the top of S_o so we just pop it.

For analysis, we define potential function: $\Phi = 2n_i$, where n_i is the number of elements in stack S_i. There is a multiplier 2 since when *outbox* is empty, we transfer elements from *inbox* by pop and push, where each one has a cost of 1 unit, totally 2. Then, we have the following amortized cost for each Insert-Queue operation:

$$actual\ cost + \Delta\Phi = stack\ push + change\ of\ inbox = 1 + 2 \leq O(1)$$

For each Delete-Queue operation, we have to consider two cases. First, easier one, when S_o is not empty, the amortized cost is:

$$actual\ cost + \Delta\Phi = stack\ pop + change\ of\ inbox = 1 + 0 \leq O(1)$$

When S_o is empty, the amortized cost is:

$$\begin{aligned} actual\ cost + \Delta\Phi &= n\ stack\ pop + (n-1)\ stack\ push + change\ of\ inbox \\ &= (2n_i - 1) + (0 - 2n_i) \leq O(1) \end{aligned}$$

As a result, amortized cost of each Insert-Queue and Delete-Queue operations is constant.

Double-ended queue can also be implemented by two stack that are mounted each other as in the following figure.

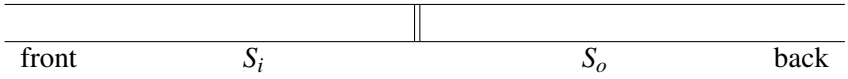

Table 1: Double-ended queue via two regular stacks

For each operation, we execute following steps:

- insert-front: push the element into S_i
- insert-back: push the element into S_o
- delete-back: check if there is an element in S_o. If exists, pop top element and return it. If S_o is empty and S_i is *not* empty, pop and push all elements from S_i to S_o one by one, and at the end pop top element of S_o and return it. If both are empty, delete operation couldn't be completed, error is returned.
- delete-front: this operation is exactly opposite of delete-back operation. Firstly, check S_i, if it isn't empty, pop top element and return it. Otherwise, if S_o isn't empty, transfer elements from S_o to S_i one by one, and at the end pop top element and return it. If both are empty, error is returned.

With above implementation, we can easily find a sequence whose amortized cost isn't $O(1)$. Sequence:

- n insert-back
- $\frac{n}{2}$ delete-front and $\frac{n}{2}$ delete-back alternating between each other

We have done $2n$ operation totally and total cost is:

$$n + (2n+1) + [2(n-1)+1] + [2(n-2)+1] + \cdots + [2*1+1] = n^2 + 3n = \Theta(n^2)$$

In this equation, first term is the cost of n insert-back operations and following terms are consecutive delete operations. Since the number of operations is the order of $\Theta(n)$ and cost is the order $\Theta(n^2)$, this is a counterexample to show that amortized cost of each four operation isn't the order of $O(1)$, constant. Costly operations are delete operations so one of them should be removed and in another way, we have used one insert type so removing it wouldn't help us to reduce our cost. However, one of the delete operations is removed, it can be shown that remaining three operations have constant amortized cost0. For analysis, let's remove delete-back operation and use potential function $\Theta = 2n_o$ where n_o is the number of elements in S_o. Therefore, we have following amortized cost for insert-back:

$$actual\ cost + \Delta\Phi = 1 + 2 \leq O(1)$$

Insert-front:

$$actual\ cost + \Delta\Phi = 1 + 0 \leq O(1)$$

When S_i isn't empty, delete-front:

$$actual\ cost + \Delta\Phi = 1 + 0 \leq O(1)$$

When S_i is empty, delete-front:

$$actual\ cost + \Delta\Phi = (2n_o + 1) + (0 - 2n_o) \leq O(1)$$

As seen above, each operation has constant time amortized cost.

Problem 2.

Problem 3.