Math 590 Homework 9 Fall 2019

Due: **Thursday**, December 5

1. **Two Stacks can make a Queue**

   A well implemented stack has two operations that it can support in $O(1)$ time: `Push` and `Pop`. It turns out that it is possible to implement an efficient queue using only two stacks. A queue has two supported operations:

   - `Enqueue`: where a value is pushed into the rear of the queue.
   - `Dequeue`: where a value is popped from the front of the queue.

   Your task will be to describe how a queue can be implemented using two stacks, and prove that your stack-based queue has the property that any sequence of n operations (selected from `Enqueue` and `Dequeue`) takes a total of $O(n)$ time resulting in amortized $O(1)$ time for each of these operations!

   (a) [2pts] Describe how a queue can be implemented using two stacks. (Hint: a more standard queue implementation typically tracks **two** values: front and rear. You have **two** stacks to use.)

   ANSWER:
   First, let's name our two stacks. We will have a Rear stack and a Front stack.

   When we have an Enqueue operation, we will Push the value onto the Rear stack. Now, clearly that means that the stack will store the values with the first-in element at the bottom of the Rear stack.

   So, when we have a Dequeue operation, we will need to access the element from the bottom of the Rear stack. We will Pop each element off the Rear stack and Push them one-by-one onto the Front stack. This reverses their order, meaning that the top of the Front stack is the first-in element of the queue. We can now Pop from the Front stack for each Dequeue operation to access the next most recently Enqueued element. Whenever the Front stack is empty, we will refill it by popping the elements from the Rear stack and Pushing them onto the Front stack.

   (b) [2pts] Using the aggregate method from class to prove that `Enqueue` and `Dequeue` each have $O(1)$ amortized runtimes.

   ANSWER: Note that each element in the queue can be pushed onto the Rear stack once, popped from the Rear stack once, pushed onto the Front stack once, and popped from the Front stack once. So, with n elements at most in the queue after n Enqueue and Dequeue operations, we could see a total of $4n \in O(n)$ work. Averaging over the n operations gives $O(1)$ amortized cost.

(c) [2pts] Using the accounting method from class to prove that `Enqueue` and `Dequeue` each have $O(1)$ amortized runtimes.

ANSWER: We first note that at most, each element will get pushed onto the Rear stack once when it is Enqueued, popped from the Rear stack and pushed onto the Front stack once when the Front stack is empty (which must happen before this element can be Dequeued), and popped from the Front stack once when Dequeued. So, assign each element 4 rubles. Spend one ruble for the Push onto the Rear stack, two rubles for the Pop from the Rear stack and the Push onto the Front stack, then the final one ruble for the Pop from the Front stack. This covers all the possible operations, and so a series of n operations will cost at most $4n$ rubles. So we have $O(1)$ amortized time.

(d) [2pts] Using the potential method from class to prove that `Enqueue` and `Dequeue` each have $O(1)$ amortized runtimes.

ANSWER: We will define our potential function as the difference between the size of the Rear stack and the Front stack, i.e.,

$$\Phi = \text{size(Rear) - size(Front)}.$$

For an Enqueue operation, we push 1 value onto the Rear stack. This costs 1 actual work, and increases our potential by 1. So $O(1)$ work.

For a simple Dequeue (when the Front stack is not empty), we simply need to pop an element from the Front stack. This costs 1 actual work, and decreases our potential by 1. So still $O(1)$ work.

For a complex Dequeue operation (when the Front stack is empty), we need to pop everything from the Rear stack and push it onto the Front stack. If there were n things in our queue when this occurred, this would take $2n$ work. However, our potential function would have gone from n to $-n$, making up for it. So this is still $O(1)$ work.

Note: this is not the only potential function we could have chosen, and, in fact, there are people who would criticize this potential function because it can have a negative value. To correct for that, you could simply add a value that will guarantee that it will be positive. For example, we know that the total number of elements in the stack will be greater than or equal to the size of the Front stack. So add *size(Rear)+size(Front)* to the potential to get $\Phi = 2 \cdot \text{size(Rear)}$.

2. [4pts] **Fast Food Coloring**

   Say there is a group of towns that are all connected by roads, given as a graph where the weights of the edges are the distances between the towns.

   Suppose we are owners of k different fast food chains f_1, f_2, $\ldots$, f_k (for example, f_1 might be Pizza Shack, f_2 might be Burger Prince, f_3 might be Pusheye's Chicken, etc). We would like to place exactly one restaurant per town such that no two towns within a distance d of each other have the same restaurant, or determine if this is not possible.

   Meanwhile, the 3 Color Problem is defined by an undirected, unweighted graph and a set of 3 colors. The goal of the problem is to determine if each vertex in the graph can be colored with one of the 3 colors in such a way that no adjacent vertices have the same color.

   Prove that the Fast Food Problem is NP Complete. Do this by reducing the 3 Color Problem **to** the Fast Food Problem.

   In other words, suppose you are given an instance of the 3 Color Problem. Describe how you could, in polynomial time, create an instance of the Fast Food Problem (*specifying all necessary inputs to the problem*). Then describe how you could, in polynomial time, construct a solution for the 3 Color Problem *given an optimal solution* from your created Fast Food Problem. Finally, prove that this returned solution is optimal.

   ANSWER: For this reduction, we start with a 3 Color problem. We are given an undirected graph G and a set of 3 colors c_1, c_2, c_3.

   We will create a new graph H that is the same as graph G, but where each edge has a weight of 1. This graph H will be the undirected, weighted graph of towns and distances for the Fast Food problem. We then set the list of fast food chains to be the three colors, i.e., $f_1 = c_1$, $f_2 = c_2$, and $f_3 = c_3$. Finally, we set $d = 1.5$ (so that connected towns cannot have the same chain, but disconnected towns are fine). We then solve the Fast Food problem on graph H with fast food chains f_1, f_2, f_3 and distance $d = 1.5$.

   If there is a way to assign the fast food chains to the towns, then there was a way to color graph G with the 3 colors. Note: each vertex in G corresponds to a town in H, and if two vertices were connected in G, then there were less than distance $d = 1.5$ away from each other in H. Finally, each fast food chain corresponded to a color. So if we could assign fast food chains to towns in H so that no two towns closer than $d = 1.5$ had the same chain, we are able to assign the corresponding colors to the corresponding vertices in G so that no two vertices that were connected had the same color.

   If no such fast food assignment was possible, then no 3 coloring is possible. Consider the contrapositive 'if a 3 coloring is possible, then a fast food assignment is possible'. This is clearly true because we would assign the color's associated chain to the vertices' associated towns, with no two towns within distance 1.5 of each other if they had been adjacent vertices.

   And clearly everything took polynomial time (one scan of the vertices+edges).