1 Building a Queue from Stacks (33 points)

Assume we are given an implementation of a stack in which the Push and Pop operations each take constant time. For concreteness, let’s suppose that Push and Pop each take exactly 1 unit of time. We now implement a queue using two stacks $A$ and $B$ as follows:

**ENQUEUE**(x):

1. Push x onto $A$

**DEQUEUE**(x):

1. If $B$ is nonempty, return $B$.Pop()

2. Otherwise, pop each element from $A$ and push it into $B$, except return the final element of $A$ (the last one popped) rather than push it onto $B$.

(a) Prove that this algorithm is correct, i.e. it does implement a Queue. Hint: think about which invariants are maintained by this algorithm.

(b) Prove that the *amortized* runtime of ENQUEUE and DEQUEUE is $O(1)$.

2 More Stacks (33 points)

In this problem we again have two stacks $A$ and $B$. In what follows, we will use $n$ to denote the number of elements in stack $A$ and use $m$ to denote the number of elements in stack $B$. Suppose that we use these stacks to implement the following operations:

- **PUSHA**(x): Push element $x$ onto $A$.
- **PUSHB**(x): Push element $x$ onto $B$.
- **MULTIPOPA**(k): Pop $\min(n, k)$ elements from $A$.
- **MULTIPOPB**(k): Pop $\min(m, k)$ elements from $B$. 
• **Transfer**\(_k\): Repeatedly pop one element from \(A\) and push it into \(B\), until either \(k\) elements have been moved or \(A\) is empty.

We are using the stacks as a black box – you may assume that Push\(_A\), Push\(_B\), MultiPop\(_A\)(1), and MultiPop\(_B\)(1) each take one unit of time (i.e. it takes one time step to push or pop a single element).

(a) What is the worst-case running time of MultiPop\(_A\), MultiPop\(_B\), and Transfer?

(b) Design a potential function \(\Phi(n, m)\), and use it to prove that the amortized running time is \(O(1)\) for every operation.

3 More counters (34 points)

We saw in class that if we have a binary counter which we increment \(n\) times the total cost (measured in terms of the number of bits that are flipped) is \(O(n)\), i.e. the amortized cost of an increment is \(O(1)\). What if we also want to be able to decrement the counter? Throughout this problem we will assume that the counter never goes negative – at every point in time the number of increments up to that point is at least as large as the number of decrements.

(a) Show that it is possible for a sequence of \(n\) operations (increments and decrements) to have amortized cost of \(\Omega(\log n)\) per operation (so the total cost is \(\Omega(n \log n)\)). This should hold even if we start from 0 and the counter never goes negative.

(b) To decrease this cost, let’s consider a new way of representing numbers: a redundant ternary number system. A number is represented as a sequence of trits (as opposed to the more usual bits or digits), each of which is 0, -1, or +1. The value of the number represented by \(t_{k-1}, \ldots, t_0\) (where each \(t_i\) is a trit) is defined to be \(\sum_{i=0}^{k-1} t_i 2^i\).

Note that the same number might have multiple representations. This is why this system is a redundant ternary system. For example, 1 0 1 and 1 1 -1 both represent the number 5.

Incrementing and decrementing work as you would expect. When incrementing, we add 1 to the low order trit. If the result is 2, then we change it to 0 and propagate a carry to the next trit. This is repeated until no carry results. Similarly, when we decrement we subtract 1 from the low order trit. If the result is -2, we set it to 0 and propagate a borrow (i.e. subtract 1 from the next lowest order trit). Again, we repeat this until no borrow is necessary.

The cost of an increment or decrement is the number of trits that change in the process. Suppose that we perform a sequence of \(n\) increments and decrements, starting from 0. Prove that the amortized cost of each operation is \(O(1)\), i.e. the total cost is \(O(n)\). Hint: think about a “potential function” or “bank account” argument.