1 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.

2 More Stacks (33 points)

In this problem we 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:

- **PUSH\(_A\)(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 **PushA**, **PushB**, **MultipopA**(1), and **MultipopB**(1) each take one unit of time (i.e. it takes one time step to push or pop a single element).

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

What if **Transfer** was not restricted to moving elements from \(A\) to \(B\), but could instead move elements in either direction? In other words, suppose instead of **Transfer**\((k)\) we had two different operations:

• **TransferA**\((k)\): Repeatedly pop one element from \(A\) and push it into \(B\), until either \(k\) elements have been moved or \(A\) is empty.

• **TransferB**\((k)\): Repeatedly pop one element from \(B\) and push it into \(A\), until either \(k\) elements have been moved or \(B\) is empty.

(b) Now do all operations have amortized running time of \(O(1)\)? If yes, prove it. If no, prove the strongest lower bound you can on the amortized running time. In other words, find the largest function \(f(t)\) you can so that there is a series of \(t\) operations (starting from both stacks being empty) where the total running time of all \(t\) operations is at least \(\Omega(t \cdot f(t))\).

### 3 Splay Trees (33 points)

(a) Suppose that \(T\) is a binary search tree on three items: an item with key 1, an item with key 2, and an item with key 3. How many configurations can \(T\) be in? In other words: how many binary search trees are there on keys 1, 2, 3? Draw them.

(b) Now let \(T\) be a splay tree on the same three keys (not an arbitrary binary search tree). Now how many configurations can \(T\) be in? Assume that at least one Lookup has been performed on each element after they were all inserted. Draw them, and prove that the other configurations are not possible.

(c) Now let \(T\) be a general splay tree on \(n\) elements (not the specific splay tree on 3 elements of the last two parts). Prove that if we perform Lookups on the elements in a splay tree in sequential ascending order (i.e., we first do a Lookup on the smallest element, then a Lookup on the second-smallest element, etc.) then in the end the splay tree is simply a path of left-children (i.e. every element in the tree has an empty right child).