Examples on Amortized Analysis

**Problem 1.** Consider a binary counter with two operations: INCREMENT, which updates the counter to the next number, and RESET, which resets the counter to 0.

The counter is implemented as a linked-list of nodes, each of which contains a binary digit (0 or 1). To help support the RESET operation, it also maintains a reference to the high-order 1.

For example, the following counter represents 3 ("11" in binary):

![Diagram of a binary counter representing 3]

INCREMENT changes the leading 1s to 0s and the next 0 to a 1. It also updates the reference to the high-order 1 node:

![Diagram of a binary counter after INCREMENT]

After calling RESET, the counter contains only zeroes, and the high-order reference points to the beginning of the list:

![Diagram of a binary counter after RESET]

(a) **Briefly** describe how to implement INCREMENT and RESET in this data structure. (Do not delete any nodes – just change their values. Remember to maintain the pointers.)

**Solution:** INCREMENT: Starting from the front (first) of the node list, change each 1 bit (if any) to zero, until reach the first zero bit. Change that bit to one and update the "high" pointer accordingly. In pseudocode:

```plaintext
curr = first
while (curr.value == 1) and (curr != high):
    curr.value = 0
    curr = curr.next
end while
if (curr==high)
    curr.value=0
    high=high.next
    high.value=1
else
    curr.value=1
end if
```
RESET: Starting at the front (first) of the node list, change each bit to zero. Stop when you reach the "high" pointer and update the "high" pointer accordingly. In pseudocode:

```python
curr = first
    while curr != high:
        curr.value = 0
        curr = curr.next
    end while
high = first
```

(b) After \( n \) INCREMENT operations on an initial-0 counter, what is the worst-case running time of the next RESET and the next INCREMENT? Explain (briefly)

**Solution:** After \( n \) increments, the value of the counter will be (at most) \( n \), which requires \( \lceil \log_2(n + 1) \rceil \) bits to represent in binary. Resetting the counter will therefore visit \( \lceil \log_2(n + 1) \rceil \) nodes, requiring \( O(\log n) \) time.

Similarly, INCREMENT in the worst case (e.g., all nodes storing 1s) needs to visit \( \lceil \log_2(n + 1) \rceil \) nodes, thus requires \( O(\log n) \) time.

(c) Use the accounting or the potential function method to show a sequence of \( n \) INCREMENT operations on an initial-0 counter takes \( O(n) \) time. Namely, the amortized complexity of INCREMENT is \( O(1) \).

**Solution:** One is free to choose either the accounting method or the potential function method. Note that there is only one operation (i.e., INCREMENT) that we need to consider.

**The accounting method.** Let us deposit $2 for each increment operation. An INCREMENT operation will possibly change a few 1s in the binary counter to 0 and one 0 to 1. We will show the following invariant holds after each INCREMENT operation,

the bank balance is the same as the number of 1s in the binary counter.

Let us see why it is the case. It trivially holds when the binary counter is 0. For each $2 deposit, we will spend $ 1 for changing 0 to 1 and save $1 for the future possible change of that 1 to 0. By the invariant statement, the money is already in the bank for all the possible change of 1s to 0s and that part of money is gone after these changes so that the invariant still holds. Thus, we show the amortized complexity is \( O(1) \) using the accounting method.

**The potential function method.** Let the potential function \( \Phi_i \) be the number of 1s in the binary counter. It is easy to see that \( \Phi_0 = 0 \) and \( \Phi_i \geq 0, \forall 1 \leq i \leq n \). Thus, for each INCREMENT operation, the actual running time is the number of changes from 1s to 0s (denoted \( k \)) plus one change of 0 to 1. The potential function will change by \( -k + 1 \). Then, the amortized complexity is

\[
t'_i = t_i + \Phi_i - \Phi_{i-1} = k + 1 + (-k + 1) = 2 \in O(1).
\]
Problem 2. Consider a modified queue that supports three operations:

- `enqueue(x)` – add an item x to the back of the queue
- `dequeue()` – remove and return the item at the front of the queue
- `removeEvens()` – remove the even-indexed items (2nd, 4th, 6th, 8th, etc.) in the queue. For example, if a queue contained \([a, b, d, g, x, y, z]\) before calling `removeEvens()`, then it would contain \([a, d, x, z]\) after calling `removeEvens()`.

This is identical to a standard queue except for the addition of a new operation.

(a) Describe how to implement `removeEvens()`, assuming that the queue is implemented as a linked list. You do not need detailed pseudocode, just a description and explanation of the basic algorithm you would use.

**Solution:** A simple algorithm is to go through the linked list and remove the even-indexed items one by one. The removal of each node in a linked list takes \(O(1)\) time.

(b) In a queue containing \(n\) items, what is the worst-case running time of `removeEvens()`? Express your answer in big-O notation. Briefly justify your answer.

**Solution:** The total time of the removal takes \(O(n/2)\) time. One also need to visit the odd-indexed items to reach the even-indexed items, which takes \(O(n/2)\) time. In the worst case, it takes \(O(n)\) time.

(c) Using the accounting or potential method, prove that `enqueue`, `dequeue`, and `removeEvens` operations run in \(O(1)\) amortized time. You may assume that `enqueue` and `dequeue` run in constant worst-case time.

**Solution:** The amortized analysis goes as follows.

**The accounting method.** Let us deposit \(\$3\) for each `enqueue` operation. For each `enqueue` operation, spend \(\$1\) for the actual cost of the enqueue and store \(\$2\) for the future operations. Out of \(\$2\), \(\$1\) is for the actual cost of the future removal of the node in the linked list, \(\$1\) for the cost of the visit of the node prior to the node (something that will happen in the `removeEvens`). Thus, for each `dequeue` operation, the cost is to remove the node in the front, which has been deposited in the `enqueue` operation. For each `removeEvens` operation, the actual cost is the sum of the visit costs to the odd-indexed items and the removals of the even-indexed items. The cost of all the removals has been already deposit. The visit cost for each odd-indexed item has also been deposited by the `enqueue` operation of the even-indexed item right after the odd-indexed item. Thus, all the cost has been deposited in the bank beforehand.

**The potential function method.** Define the potential function \(\Phi_i\) to be twice the number of items in the data structure after the \(i\)th operation. It is clear that \(\Phi_0 = 0\) and \(\Phi_i \geq 0, \forall i = 1, \cdots, n\). Let \(\Phi_i = 2k_i\) for each \(i\). For each operation, we have

- `enqueue`: The actual cost \(t_i = 1\) and \(k_i = k_{i-1} + 1\). Thus, we have the amortized complexity,
  \[t'_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2k_i - 2k_{i-1} = 3 \in O(1)\].

- `dequeue`: The actual cost \(t_i = 1\) and \(k_i = k_{i-1} - 1\). Thus, we have the amortized complexity,
  \[t'_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2k_i - 2k_{i-1} = -1 \in O(1)\].

- `removeEvens`: The actual cost \(t_i = k_{i-1}\) and \(k_i = k_{i-1}/2\). Thus, we have the amortized complexity,
  \[t'_i = t_i + \Phi_i - \Phi_{i-1} = k_{i-1} + 2k_i - 2k_{i-1} = 0 \in O(1)\].