1. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

2. Recall that a standard FIFO queue supports the following operations:

   - PUSH(x): Add item x to the back of the queue.
   - PULL(): Remove and return the first item at the front of the queue.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time per operation is $O(1)$.

   (a) Now suppose we want to support the following operation instead of PULL:

   - MULTIPULL(k): Remove the first k items from the front of the queue, and return the kth item removed.

   Suppose we use the obvious algorithm to implement MULTIPULL:

```
MULTIPULL(k):
  for i ← 1 to k
      x ← PULL()
  return x
```

   Prove that in any intermixed sequence of PUSH and MULTIPULL operations, the amortized cost of each operation is $O(1)$

   (b) Now suppose we *also* want to support the following operation instead of PUSH:

   - MULTIPUSH(x, k): Insert k copies of x into the back of the queue.

   Suppose we use the obvious algorithm to implement MULTIPUUSH:

```
MULTIPUSH(k, x):
  for i ← 1 to k
      PUSH(x)
```

   Prove that for any integers ℓ and n, there is a sequence of ℓ MULTIPUSH and MULTIPULL operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

   (c) Describe a data structure that supports arbitrary intermixed sequences of MULTIPUSH and MULTIPULL operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure should use only $O(1)$ space per item.

3. (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The ***only*** access you have to the stacks is through the standard subroutines PUSH and POP. You may not implement your own nontrivial data structure or examine the internal contents of the stacks.

   (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - QUACKPUSH(x): add a new item x to the left end of the list
   - QUACKPOP(): remove and return the item on the left end of the list;
   - QUACKPULL(): remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPUSH, QUACKPOP, or QUACKPULL operation is $O(1)$. In particular, each element in the quack must be stored in *exactly one* of the three stacks. Again, you are ***only*** allowed to access the component stacks through the interface functions PUSH and POP.