1. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - PUSH(x): Add item x to the end of the sequence.
   - PULL(): Remove and return the item at the beginning of the sequence.
   - SIZE(): Return the current number of items in the sequence.

   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 for each of these operations is $O(1)$.

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
DECIMATE():
  n ← SIZE()
  for i ← 0 to n − 1
      if i mod 10 = 0
          PULL()    ⟨⟨result discarded⟩⟩
      else
          PUSH(PULL())
```

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

2. ***This problem is extra credit***, because the original problem statement had several confusing small errors. I believe these erors are corrected in the current revision.

   Deleting an item from an open-addressed hash table is not as straightforward as deleting from a chained hash table. The obvious method for deleting an item x simply empties the entry in the hash table that contains x. Unfortunately, the obvious method doesn't always work. (Part (a) of this question asks you to prove this.)

   Knuth proposed the following *lazy* deletion strategy. Every cell in the table stores both an *item* and a *label*; the possible labels are **EMPTY**, **FULL**, and **JUNK**. The DELETE operation marks cells as **JUNK** instead of actually erasing their contents. Then FIND pretends that **JUNK** cells are occupied, and INSERT pretends that **JUNK** cells are actually empty. In more detail:

```
FIND(H, x):
  for i ← 0 to m − 1
      j ← h_i(x)
      if H.label[j] = FULL and H.item[j] = x
          return j
      else if H.label[j] = EMPTY
          return NONE
```

```
INSERT(H, x):
  for i ← 0 to m − 1
      j ← h_i(x)
      if H.label[j] = FULL and H.item[j] = x
          return        ⟨⟨already there⟩⟩
      if H.label[j] ≠ FULL
          H.item[j] ← x
          H.label[j] ← FULL
          return
```

```
DELETE(H, x):
  j ← FIND(H, x)
  if j ≠ NONE
      H.label[j] ← JUNK
```

Lazy deletion is always *correct*, but it is only *efficient* if we don't perform too many deletions. The search time depends on the fraction of non-**EMPTY** cells, not on the number of actual items stored in the table; thus, even if the number of items stays small, the table may fill up with **JUNK** cells, causing unsuccessful searches to scan the entire table. Less significantly, the data structure may use significantly more space than necessary for the number of items it actually stores. To avoid both of these issues, we use the following rebuilding rules:

- After each INSERT operation, if less than $1/4$ of the cells are **EMPTY**, rebuild the hash table.
- After each DELETE operation, if less than $1/4$ of the cells are **FULL**, rebuild the hash table.

To rebuild the hash table, we allocate a new hash table whose size is twice the number of **FULL** cells (unless that number is smaller than some fixed constant), INSERT each item in a **FULL** cell in the old hash table into the new hash table, and then discard the old hash table, as follows:

```
REBUILD(H):
  count ← 0
  for j ← 0 to H.size − 1
      if H.label[j] = FULL
          count ← count + 1
  H′ ← new hash table of size max{2 · count, 32}
  for j ← 0 to H.size − 1
      if H.label[j] = FULL
          INSERT(H′, H.item[j])
  discard H
  return H′
```

Finally, here are your actual homework questions!

(a) Describe a ***small*** example where the "obvious" deletion algorithm is incorrect; that is, show that the hash table can reach a state where a search can return the wrong result. Assume collisions are resolved by linear probing.

(b) Suppose we use Knuth's lazy deletion strategy instead. Prove that after several INSERT and DELETE operations into a table of arbitrary size m, it is possible for a single item x to be stored in *almost half* of the table cells. (However, at most one of those cells can be labeled **FULL**.)

(c) For purposes of analysis,[1] suppose FIND and INSERT run in $O(1)$ time when at least $1/4$ of the table cells are **EMPTY**. Prove that in any intermixed sequence of INSERT and DELETE operations, using Knuth's lazy deletion strategy, the amortized time per operation is $O(1)$.

*3. ***Extra credit.*** Submit your answer to Homework 4 problem 3.

[1] In fact, FIND and INSERT run in $O(1)$ *expected* time when at least $1/4$ of the table cells are **EMPTY**, and therefore each INSERT and DELETE takes $O(1)$ *expected* amortized time. But probability doesn't play any role whatsoever in the amortized analysis, so we can safely ignore the word "expected".

CS 473 Fall 2013 — Homework 5 Problem 1

Name:	NetID:
Name:	NetID:
Name:	NetID:
Section: T4 T5 W2 W3 W5 None	

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

CS 473 Fall 2013 — Homework 5 Problem 2

Name:	NetID:
Name:	NetID:
Name:	NetID:
Section: T4 T5 W2 W3 W5 None	

(a) Describe a ***small*** example where the "obvious" deletion algorithm is incorrect, assuming collisions are resolved by linear probing.

(b) Prove that after several INSERT and DELETE operations, it is possible for a single item x to be stored in *almost half* of the table cells.

(c) Prove that in any intermixed sequence of INSERT and DELETE operations, using Knuth's lazy deletion strategy, the amortized time per operation is $O(1)$.
