In both the regular lecture and the honors lecture, we discussed an implementation of array-lists whose `Insert` method doubles the size of the underlying data array if it is already full. In particular, we sketched a proof that each insertion into this array-list takes $O(1)$ amortized time.

Suppose we modify the `Delete` method to `shrink` the underlying array when it becomes too empty, to avoid wasting memory. Here is pseudocode for one implementation of this idea:

```plaintext
New():
   AL.num ← 0
   AL.cap ← 2
   AL.data ← new array[2]
   return AL

Insert(AL, x):
   if AL.num = AL.cap
      Resize(AL)  (Double!)
   (----- Next phase starts here -----
   AL.data[AL.num] ← x
   AL.num ← AL.num + 1

Delete(AL):  
   if AL.num = AL.cap/4
      Resize(AL)  (Half!)
   (----- Next phase starts here -----
   AL.num ← AL.num - 1
   return AL.data[AL.num]

Resize(AL):
   newdata ← new array[2*AL.num]
   for i ← 0 to AL.num - 1
      newdata[i] ← AL.data[i]
   delete AL.data
   AL.data ← newdata
   AL.cap ← 2*AL.num

Index(AL, i):
   return AL.data[AL.num - i - 1]
```

Here $AL.num$ is the actual number of items stored in the list (or what theoreticians would call $n$), $AL.cap$ is the size of the data array, and $AL.data$ is the actual data array. List items are stored in a prefix of the data array in the order they were inserted. Except when $n = 0$, all operations maintain the invariant $n \leq AL.cap \leq 4n$, so the size of our data structure is always $\Theta(n)$.

1. I claim that in any intermixed sequence of `Inserts` and `Deletes`, starting with a New array-list, each `Insert` and each `Delete` takes $O(1)$ amortized time. What this claim means is that the total time required to execute any intermixed sequence of $N_i$ `Inserts` and $N_d$ `Deletes` is at most $O(N_i + N_d)$.

   One way to prove this claim is partition the overall running time of the data structure into phases immediately after each `Resize`, as indicated in the pseudocode above, and analyze each phase separately.

   There are two cases to consider, depending on whether the phase ends by doubling or halving the data array. Let $n_0$ denote the value of $AL.num$ at the start of an phase; to avoid trivial boundary cases, assume $n_0 \geq 4$.

   (a) Suppose the phase ends by doubling the data array. What is the exact minimum number of `Insert` and `Delete` operations that phase can contain? Your answer should be a function of $n_0$.

   (b) Suppose the phase ends by halving the data array. What is the exact minimum number of `Insert` and `Delete` operations that phase can contain? Your answer should be a function of $n_0$.

   (c) Complete the amortized analysis: Prove that the total time to execute any phase containing $N_i$ `Inserts` and $N_d` `Deletes` is at most $O(N_i + N_d)$. 
2. **Think about this one on your own; do not submit solutions.**

A valid criticism of this implementation of array-lists is that in the worst case (just before an expensive DELETE) the data structure is using about four times as much space as necessary. Suppose we are allowed to maintain a tighter invariant $AL.cap \leq (1 + \varepsilon)n$, for some fixed constant $\varepsilon > 0$. We still want to keep the amortized time for each INSERT and DELETE as small as possible. For example, if $\varepsilon = 0.01$, we are allowed to use 1% more space than absolutely necessary; our reference implementation uses $\varepsilon = 3$.

(a) How would you change the INSERT, DELETE, and RESIZE algorithms?

(b) What is the amortized time for each INSERT and DELETE, as a function of $\varepsilon$? (You should see a tradeoff between space and time; the amortized time for each operation should increase as $\varepsilon$ approaches zero.)

3. **Think about this one on your own; do not submit solutions.**

Small amortized time bounds are sufficient if we're using data structures as part of a large batch computation, but in more interactive contexts—for example, video games, or self-driving cars— we really do need every operation to be fast in the worst case.

We can *deamortize* this version of array-lists by incrementally building larger and smaller versions of the current data array during INSERT and DELETE operations. The new data structure contains three data arrays:

- $AL.bigdata[0 \ldots 2 \cdot AL.cap - 1]$
- $AL.curdata[0 \ldots AL.cap - 1]$ — the actual data array
- $AL.weedata[0 \ldots AL.cap/2 - 1]$

We maintain these arrays as follows:

- Each INSERT spends $\Theta(1)$ additional time building or maintaining $AL.bigdata$.
- To “double” the data array, we delete $AL.weedata$, set $AL.weedata \leftarrow AL.curdata$, set $AL.curdata \leftarrow AL.bigdata$, and allocate a new bigdata array.
- Each DELETE spends $\Theta(1)$ time building or maintaining $AL.weedata$.
- To “halve” the data array, we delete $AL.bigdata$, set $AL.bigdata \leftarrow AL.curdata$, set $AL.curdata \leftarrow AL.weedata$, and allocate a new weedata array.

Work out the remaining details of this real-time implementation. What is the total worst-case size of the three arrays, as a function of $n = AL.num$? Exactly which operations are performed in each $\Theta(1)$-time increment? How do we guarantee that bigdata/weedata contain exactly the right data when we “resize” the array?

---

1“I'm sorry Dave, I can’t do that; Windows is updating.”

2I am assuming here that allocating an arbitrary block of memory takes $O(1)$ time in the worst case. That's not entirely realistic, but unpacking the details will have to wait until CS 340 or CS 341 or ECE 391.