

CSC 263 Lecture 7

July 13, 2004

For another reference, see chapter 17 of the textbook (CLRS).

1 Dynamic Arrays

Consider the following data structure: we have an array of some fixed size, and two operations, **APPEND** (store an element in the first free position of the array) and **DELETE** (remove the element in the last occupied position of the array). This data structure is the standard way to implement stacks using an array.

It has one main advantage (accessing elements is very efficient), and one main disadvantage (the size of the structure is fixed). We can get around the disadvantage with the following idea: when trying to **APPEND** an element to an array that is full, first create a new array that is twice the size of the old one, copy all the elements from the old array into the new one, and then carry out the **APPEND** operation.

Let's look at the cost of performing **APPEND** operations, starting from an array with size 0. We'll only count the cost of assigning a value to an element of the array, disregarding the cost of allocating memory for each one (since most languages can usually allocate large chunks of memory efficiently, independently of the size of the memory required, and also because counting this cost would only add a constant factor more).

A: <Empty>

APPEND X1 →

X1

 COST=1

APPEND X2 →

X1	X2
----	----

 COST=2

APPEND X3 →

X1	X2	X3	
----	----	----	--

 COST=3

APPEND X4 →

X1	X2	X3	X4
----	----	----	----

 COST=1

APPEND X5 →

X1	X2	X3	X4	X5			
----	----	----	----	----	--	--	--

 COST=5

APPEND X6 →

X1	X2	X3	X4	X5	X6		
----	----	----	----	----	----	--	--

 COST=1

So generally, operation number i will cost 1, except if $i = 2^k + 1$ for any natural number k ; then the cost will be i .

We want to analyze the amortized complexity of a sequence of m APPEND operations, starting with an array of size 0. As in the binary counter example, let's try charging 2 for each APPEND operation; then we should have enough to pay 1 for the cost of assigning the new element, and 1 to save with the element to pay for the cost of copying it later on.

A: <Empty>

APPEND X1 →

X1

 COST=1 CHARGE=2 TOTAL CREDIT=1

APPEND X2 →

X1	X2
----	----

 COST=2 CHARGE=2 TOTAL CREDIT=1

APPEND X3 →

X1	X2	X3	
----	----	----	--

 COST=3 CHARGE=2 TOTAL CREDIT=0

APPEND X4 →

X1	X2	X3	X4
----	----	----	----

 COST=1 CHARGE=2 TOTAL CREDIT=1

APPEND X5 →

X1	X2	X3	X4	X5			
----	----	----	----	----	--	--	--

COST=5 CHARGE=2 TOTAL CREDIT=-2

As you can see, we run into a problem: we don't have enough credits to copy over all of the old elements! In fact, what ends up happening is that only the elements in the second half of the array (the ones added since the last size increase) have a credit on them. This suggests the following solution to our problem: make each element in the second half responsible for paying to copy both itself and one other element from the first half of the array.

So, if we charge 3 for each APPEND operation, we can prove the following credit invariant: *Each element in the second half of the array has a credit of 2.*

Proof of credit invariant: Initially, the array has size 0 and no elements, so the invariant is trivially true. Assume that the invariant is true after a certain number of APPEND operations have been performed, and consider the next APPEND operation:

- If the size of the array does not need to be changed, simply use 1 to pay for storing the new element, and keep 2 as credit with that new element. Since new elements are only added in the second half of the array, the credit invariant is maintained.
- If the size of the array needs to be changed, then this means that the array is full. Since the number of elements in the first half of the array is the same as the number of elements in the second half of the array, and since we have 2 credits on each element in the second half, we have exactly enough money to pay for the cost of copying all the elements into the new array of twice the size. Then, we use the 3 as before, to pay for storing the new element and keep 2 credits on that new element. As before the invariant is maintained.

Hence, the number of credits in the array never becomes negative, so the total charge for the sequence is an upper bound on the total cost of the sequence, i.e., the sequence complexity of m APPEND operations is at most $3m$ and the amortized cost of a single operation in this sequence is $3m/m = 3$.

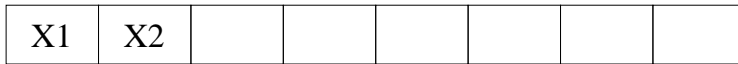
If we look at what happens when we include DELETE operations, we can simply charge each DELETE operation 1 to pay for the cost of removing one element. Notice that this does not affect the credit invariant. Now, since we charge 3 for the most expensive operation (namely APPEND), the worst-case sequence complexity of m operations is $3m$ and the amortized sequence complexity is 3.

2 Dynamic Arrays with Reducing

If many DELETE operations are performed, the array could become very empty, which wastes memory space. We would like to contract the size of the array when it becomes "too empty", which involves creating a new array with a smaller size and copying every element over into the new array. Consider the following policy:

Suppose that we reduce the size of the array in half when a DELETE operation causes the array to become less than half full. Unfortunately, this leads to the following situation: consider a sequence of $n = 2^k$ operations, where the first $n/2$ operations are APPEND, followed by the sequence APPEND, DELETE, DELETE, APPEND, APPEND, DELETE, DELETE, ... The first APPEND in the second half of the sequence of operations will cause the array to grow (so $n/2$ elements need to be copied over), while the two DELETE operations will cause the new array to become less than half full, so that it shrinks (copying $n/2 - 1$ elements), the next two APPEND operations cause it to grow again (copying $n/2$ elements), etc. Hence, the total cost for this sequence of n operations is $\Omega(n^2)$, which gives an amortized cost of n .

Intuitively, we need to perform more deletions before contracting the array size. Consider what happens if we wait until the array becomes less than 1/4 full before reducing its size by half.

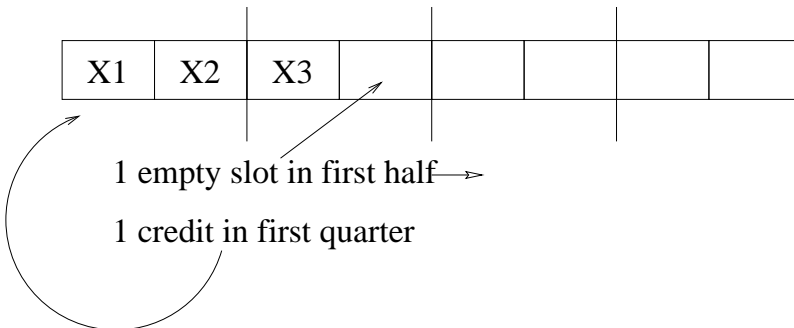


DELETE X2



Then, no matter how many elements the array had to start with, we must delete at least 1/4 of the elements in the array before a contraction occurs, and once a contraction occurs, we must add at least as many elements as there are left before an expansion occurs. This gives us enough time to amass credit, and to maintain the following two-part credit invariant:

1. Every element in the second half of the array has credit 2 (this is the same as the non-reducing case).
2. If the array is less than half full, the amount of credit in the first quarter of the array is at least the number of elements by which the array is less than half full.



Basically, as we delete elements from the array and thereby get closer to reducing it, we want to build up credit in the first quarter of the array because these are the elements that will need to be copied during the reduction.

To achieve this credit invariant, we use the following charging scheme: we charge 3 for APPEND and 2 for DELETE.

Proof of credit invariant:

- **Base Case:** Initially, the array is empty and has size 0, so the credit invariant is vacuously true.
- **Inductive Step:** After a certain number of operations have been performed, assume that the credit invariant holds. Now consider the next operation:
 - **Case A:** If APPEND is performed, treat it just like before (if the array is full, there are enough credits to double the size and copy all the elements over, and of the charge of 3, 1 pays for the new element and 2 stays as credit with the new element).
 - **Case B:** If DELETE is performed, there are three cases to consider:
 - * **Case B1:** If the element deleted is in the second half of the array, we pay for the deletion using 1 of the 2 charged and simply "throw away" the remaining 1 as well as the 2 credits that were stored with the element.

- * **Case B2:** If the element deleted is in the first half of the array but not in the first quarter, then we pay for the cost of the deletion using 1 and put the other 1 as credit on the first element in the array that has 0 credit (if there is no such element, we just "throw away" the extra 1).
- * **Case B3:** If the element deleted is in the first quarter, it must be the last element in the first quarter of the array, so by the credit invariant, every element in the first quarter has at least 1 credit: we use those credits to pay for the cost of copying each element into a new array of half the size. The 1 that was stored with the deleted element can be used to pay for the DELETE. That leaves 2 more from the charge of DELETE. We give one of these to the first element in the array if it has no credits. Otherwise we just throw them out. Note that the array is now one element short of half full. In accordance with the credit invariant, we have made sure that there is at least 1 credit in the first quarter.

In all cases, the credit invariant is maintained, so the total credit of the data structure is never negative, meaning that the total charge is an upper bound on the total cost of the sequence of operations. Since the total charge for m operations is $\leq 3m$, the amortized cost of each operation is $\leq 3m/m = 3$.

Notice that in this case, we really are overcharging for some of the operations (because we sometimes throw away credits that are not needed), but this is necessary if we want to ensure that we always have enough credits in all possible cases.