Lecture 14

Dynamic Arrays

Consider data structure: array of fixed size and two operations:
`append` → store element in first free position;
`delete` → remove element in last occupied position
(Note: standard stack implementation using array)

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. Clearly bad in terms of worst case for single operation: $\Theta(n)$ (when array full and `append` executed). But consider: worst case only happens after n `append` operations. Think about the cost of performing n `append` operations, starting from an empty array of size 1, in the ammortized sense.

The Accounting Method

First we tried charging \$2 for every `append` and we saw that we didn't have enough credit. We realized that since we have added elements into the last half of the array since the last append, we need to charge enough so that the credit will pay for the copying of these newly added elements and *also* the ones that were copied the last time we grew. Consider charging \$3 for every `append`. The costs of `append`s which increase the array size depend on how many elements are copied to the larger array. The true cost is \$1 per copied element and \$1 additionally to add the new element.

Consider again `append`ing to an empty array with size 1. The cost for this operation is \$1 and the charge is \$3.

\$\$
x

The next `append` causes the array size to double. This operations costs \$2 (\$1 for the copy and \$1 for the new append) but we charge \$3.

\$	\$\$
x	x

The next `append` causes the array size to double again. This operation costs \$1 per copy and \$1 to append the new item. Notice that each of the items which needs to be copied, has associated with it at least \$1 credit. The new picture after the copy but before the insert of the new item is:

\$			
x	x		

After inserting the item we have

\$		\$\$	
x	x	x	

After inserting the fourth item we have

\$		\$\$	\$\$
x	x	x	x

When the 5^{th} item is to be inserted, the array first doubles in size. The "cost" of the doubling is paid for by the "credits" on the last half of the current array. Each element in the upper half has a \$2 credit. Once the doubling has happene but the new element hasn't been inserted:

$							
x	x	x	x				

Now the insertion of each element into the empty upper half will cost \$1 and charge \$3 giving a \$2 credit.

For an array of length n (where $n = 2^k$ for some integer k) the credit after n insertions will be $2n + 1$. At no point does the credit ever become negative. The cost per insertion is 3. So for n ammortized insertions the ammortized cost is $O(n)$.

Credit invariant

"Each element in the second half of the array has a credit of \$2"

Consider the array of length 1 after 1 `append`. It has a credit of \$2 so the invariant holds. 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. We have \$2 credit on each element in the second half so we have enough money to pay for the cost of copying all the elements into the new array of twice the size. Afterwards, we use the \$3 as before, to pay for storing the new element and keep \$2 credit on that new element. Thus the invariant is maintained

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, that is, the ammortized cost per operation is no more than $3m/m = 3$.

Shrinking on `delete`

What if we delete many elements from that array? We could be left with a large array that only stores a small number of elements, wasting memory.

Can we modify the `delete` operation so that it will "shrink" the array when it becomes "too empty", while keeping the amortized cost of each operation constant? It's possible, but only by doing it carefully: the solution that works best is to shrink the array only when it becomes less than 1/4 full. By charging \$3 for each `append` (as before) and \$2 for each `delete`, it is possible to show that the total credit never gets negative, so that the amortized cost of each operation is $O(1)$.

Why do we want to wait until the array is less than 1/4 full? 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 `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. Then, no matter how many elements the array had to start with, we must delete *at least* 1/2 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 credit

invariant:

"Every element in the second half of the array has $2 of credit and if the array is less than half full, there are at least as many dollars of credit in the first quarter of the array as there are elements by which the array is less than half full"

Another equivalent way to state this (that we used in the 2nd class) is

"Every element in the second half of the array has a $2 credit and every empty location in the first half of the array has a $1 credit"

In other words, if the array is at least half full, there aren't necessarily any credits in the first half of the array, but for each element deleted from the first half of the array, one more dollar of credit is added to first half of the array, so that by the time the array is less than 1/4 full, the first half of the array has one credit for each element. This is exactly the amount needed to copy the elements from the first quarter into a new smaller array.

More precisely, we charge `append` $3 and `append` $2. Initially, the array is empty and has size 0, so the credit invariant is vacuously true. After a certain number of operations have been performed, assuming that the credit invariant holds, consider the next operation.

`append`:

Treat it almost like before. If the array is full, there are enough credits to double the size and copy all the elements over, and of the $3, $1 pays for the new element and $2 stays as credit with the new element. If the element is being added to the first half of the array, "throw away" both the credit and any existing $1 credit that was in the empty space.

If `delete` is performed, we have 3 cases:

1. 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 of credit that was stored with the element.

2. If the element deleted is in the first half of the array but not in the first quarter, then pay for the cost of the deletion using $1 and put the other $1 as credit in the empty position.

3. 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 location in the second quarter has $1 credit: we use those credits to pay for the cost of copying each element into a new array of half the size, and then use $1 to pay for the deletion and $1 to give as credit to the newly-emptied location.

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 is $\leq 3m$, the amortized cost of each operation in a sequence of m operations is always $\leq 3m/m = 3$.

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. In other words, if we try to charge less than $3 for `append` and $2 for `delete`, we can end up in situations where we don't have enough money to pay for the cost of an operation