Dynamic Arrays and Amortized Analysis

Yufei Tao

ITEE
University of Queensland

As mentioned earlier, one drawback of arrays is that their lengths are fixed. This makes it difficult when you want to use an array to store a set that may continuously grow and shrink with time.

In this lecture, we will discuss clever tricks that allow us to design an array whose size can be varied efficiently! Our discussion also serves as a golden opportunity to introduce the method of amortized analysis.

Dynamic Array Problem

Let S be a multi-set of integers that grows with time. At the beginning, S is empty. Over time, the integers of S are added by the following operation:

- insert(e): which adds an integer e into S.

At any moment, let n be the number of elements in S. We want to store all the elements of S in an array A satisfying:

1. A has length $O(n)$
2. If an integer x was the i-th ($i \geq 1$) inserted, then $A[i] = x$ (i.e., x is at the i-th position of the array).

> This problem is dynamic, namely, the value of n continuously grows over time (initially, $n = 0$). The above requirements must be satisfied after every insertion.

Naive Algorithm

Perform insert(e) as follows:

- If $n = 0$, then set n to 1. Initialize an array A of length 1, containing just e itself.
- Otherwise (i.e., $n \geq 1$):
  - Increase n by 1.
  - Initialize an array A' of length n.
  - Copy all the $n - 1$ elements of A over to A'.
  - Set $A'[n] = e$.
  - Destroy A, and replace it with A'.

This algorithm spends $O(n)$ time on the n-th insertion. Altogether, it takes $O(n^2)$ time to do n insertions.

We will improve the time of inserting n elements dramatically to $O(n)$ (this is clearly optimal because every insertion must take at least constant time)! As a tradeoff, our array A may have a length up to $2n$, which is still $O(n)$.

A Better Algorithm

We say that an array A is full, if the number of integers therein is already equal to its length.

- For example, if A was initialized with length 8, it is non-full if it has only up to 7 integers.

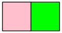
A Better Algorithm

Perform insert(e) as follows:

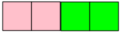
- If $n = 0$, then set n to 1. Initialize an array A of length 2, containing just e itself.
- Otherwise (i.e., $n \geq 1$), append e to A, and increase n by 1. If A is full, do the following
    - Initialize an array A' of length $2n$.
    - Copy all the n elements of A over to A'.
    - Destroy A, and replace it with A'.

Example

$n = 1$

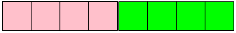


$n = 2$



$n = 3$

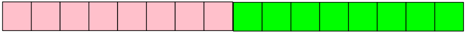
$n = 4$



$n = 5$

...

$n = 8$



Analysis

Cost of insertion when inserting the n-th element:

- If A is non-full after the insertion, $O(1)$.
- Otherwise, $O(n)$—the time incurred in expanding A.
    - Suppose that the expansion time is at most cn, for some constant $c > 0$.

Analysis

Array expansions happen infrequently:

- Initially, size 2.
- First expansion: size from 2 to 4.
- Second expansion: from 4 to 8.
- ...
- The i-th expansion: from 2^i to 2^{i+1}.

After n insertions, the size of A is at most $2n$. Hence:

$$2^{i+1} \leq 2n \quad \Rightarrow \quad i \leq \log_2 n$$

That is, there can be no more than $\log_2 n$ array expansions.

Analysis

Therefore, the total cost of n insertions is bounded by:

$$\left(\sum_{i=1}^{n} O(1)\right) + \left(\sum_{i=1}^{\log_2 n} c \cdot 2^i\right) \tag{1}$$

where the first term corresponds to the compulsory constant time spent on each insertion, and the second term corresponds to the total cost of expanding.

Formula (1) evaluates to $O(n)$.

Cleverer Analysis

Next, we give an alternative analysis that proves the same conclusion with an elegant charging argument.

Our algorithm maintains an invariant:

- After an array expansion, the new array has size $2n$, namely, offering n empty positions.

Cleverer Analysis

Suppose that an array expansion occurs at n, which takes $c \cdot n$ time.

$\Rightarrow$ The previous expansion happened at $n/2$.

$\Rightarrow$ $n/2$ empty positions in the previous array.

$\Rightarrow$ $n/2$ insertions have taken place since the previous expansion.

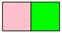
$\Rightarrow$ Charge the $c \cdot n$ cost over those $n/2$ insertions.

$\Rightarrow$ Each insertion bears additional $\frac{c \cdot n}{n/2} = 2c = O(1)$ cost.

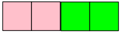
Therefore, the total cost of n insertions is $O(n)$.

Example

$n = 1$



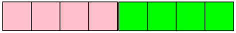
$n = 2$



expanding cost charged on the 2nd element

$n = 3$

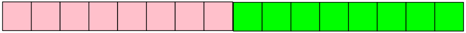
$n = 4$ expanding cost charged on elements 3-4



$n = 5$

...

$n = 8$ expanding cost charged on elements 5-8



The Stack-with-Array Problem

Let S be a multi-set of integers that grows with time. At the beginning, S is empty. We must support the following stack operations:

- push(e): which adds an integer e into S.
- pop: which removes from S the most recently inserted integer.

At any moment, let m be the number of elements in S. We want to store all the elements of S in an array A satisfying:

1. A has length $O(m)$
2. $A[1]$ is the least recently inserted element, $A[2]$ the second least recently inserted, ..., $A[m]$ the most recently inserted.

We will denote by n the number of operations processed so far.

The Stack-with-Array Problem

We will give an algorithm for maintaining such an array by handling n operations in $O(n)$ time, namely, each operation is processed in $O(1)$ amortized time.

The Stack-with-Array Problem

We say that

1. (Same as before) an array A is full, if the number of integers therein is equal to its length.
2. A is sparse if the number of integers therein is equal to $1/4$ of its length (we will ensure that the length is a multiple of 4).

Stack-with-Array Algorithm

If $m \leq 4$, simply keep all the elements of S in an array of length 4, where the elements are sorted in the same order by which they were inserted.

Next, we assume that $m > 4$.

Push ($m \geq 4$)

Perform push(e) in the same way as an insertion in the dynamic array problem.

Pop ($m \geq 4$)

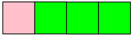
Perform pop as follows:

- Return the last element of A, and decrease n by 1. If A is sparse, shrink the array as follows:
  - Initialize an array A' of length $2n$.
  - Copy all the n elements of A over to A'.
  - Destroy A, and replace it with A'.

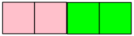
Example

Next, we use the algorithm to perform 11 pushes and then 9 pops on an initially empty stack.

$n = 1$, push

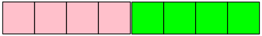


$n = 2$, push



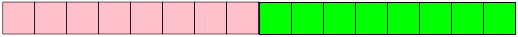
...

$n = 4$, push



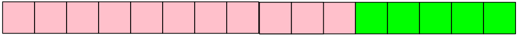
...

$n = 8$, push



...

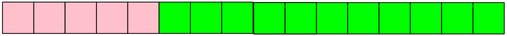
$n = 11$, push



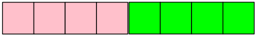
Example

...

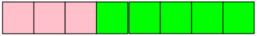
$n = 17$ pop



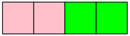
$n = 18$, pop



$n = 19$, pop



$n = 20$, pop



Analysis

We will leave as an exercise for you to prove that our algorithm performs any sequence of n operations (each being either a push or a pop) using $O(n)$ time in total.

Hint: Use a charging argument following the steps below:

- Every array expansion/shrinking takes $O(m)$ time (where m is the size of the current S.
- Charge the cost on $\Omega(m)$ appropriate operations.