More on Dynamic Array and Amortized Analysis

CSCI2100 Tutorial 7

Jianwen Zhao

Department of Computer Science and Engineering
The Chinese University of Hong Kong

Adapted from the slides of the previous offerings of the course

Introduction

In the previous lectures, we have introduced the dynamic array problem and solved it by making use of some clever tricks which allow us to perform n operations in $O(n)$ time, namely, each operation takes $O(1)$ amortized time. We also implemented the data structure stack by exploiting dynamic array.

In this tutorial, we will introduce a new version of dynamic array with smaller space consumption, while each operation still costs $O(1)$ amortized time. We will also try to implement another data structure – the queue – with dynamic array.

Recap: 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).

Recall that, while performing insertions to a dynamic array A, once A is full, we expand A by doubling the current length. We proved that each insertion costs $O(1)$ amortized time and that the space consumption is $O(n)$ at any moment.

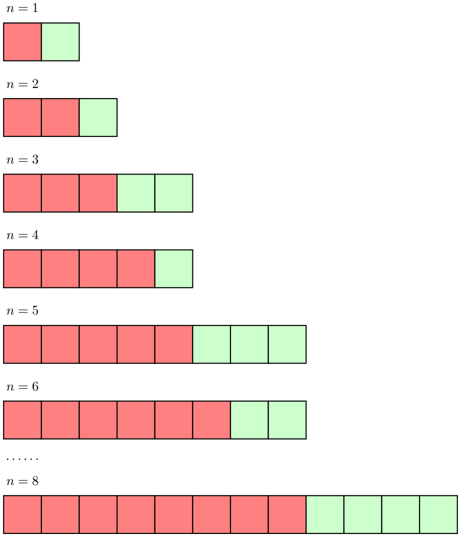
In fact, it is not necessary to restrict the expansion to doubling. In the following, we will show a new version of dynamic array which expands the length of A to $1.5n$ once A is full, while each operation still costs $O(1)$ amortized time.

Dynamic Array – A New Version

Perform insert(e) as follows:

- If $n = 0$, then set n to 1. Initialize an array A with 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 $\lceil 1.5n \rceil$.
  - Copy all the n elements of A over to A'.
  - Destroy A, and replace it with A'.

Example



Cost Analysis

Lemma: When $n \geq 15$, at least $n/4$ elements must have been inserted since the last expansion.

Proof: Let x be the number of elements when the last expansion happened. Hence, $n = \lceil 1.5x \rceil$, meaning that $n - x$ elements have been inserted since the last expansion. It suffices to prove $n - x \geq n/4$ when $n \geq 15$. Towards this purpose, since $n - x \geq 1.5x - x = 0.5x$, it suffices to prove:

$$\begin{aligned} 0.5x &\geq n/4 = \lceil 1.5x \rceil / 4 \\ \Leftrightarrow 2x &\geq \lceil 1.5x \rceil \end{aligned}$$

whose correctness can be easily verified for $n \geq 15$. □

Cost Analysis

Suppose that the array expansion occurs when A is full with n elements, and that expansion takes $c \cdot n$ time. When $n \leq 15$, $cn = O(1)$. For $n > 15$,

- There were $n/4$ insertions have taken place since the previous expansion.
- Each of those insertions bears additional $\frac{cn}{n/4} = 4c = O(1)$ cost.

The Stack-with-Array Problem

Push

We can perform push(e) in the same way as an insertion in the dynamic array problem.

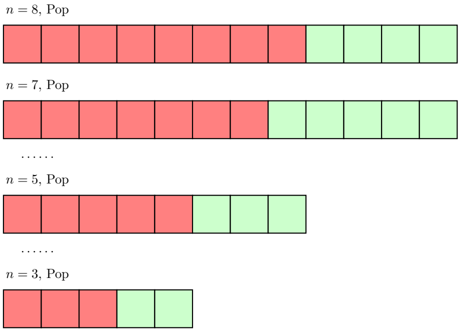
Pop

We say that A is sparse if its length is at least 2, and the number of integers therein drops below $4/9$ of its length.

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 $\lceil 1.5n \rceil$.
  - Copy all the elements of A over to A'.
  - Destroy A, and replace it with A'.

Example



Cost Analysis

The analysis follows the same ideas explained in the lecture. The crux is to show that, when an overhaul (i.e., expansion/shrinking) happens, $\Omega(n)$ operations must have occurred since the last overhaul. As each overhaul takes $O(n)$ time, each of those operations is amortized $O(1)$ time.

The Queue-with-Array Problem

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

- En-queue(e): Inserts an integer e into S.
- De-queue: Removes the least recently inserted element from S.

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, $\cdots$, $A[m]$ the most recently inserted.

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

The Queue-with-Array Problem

We will explain how to maintain a dynamic array that ensures minimum occupancy of 50%. You may apply the techniques explained earlier to increase the minimum occupancy at the tradeoff of higher amortized update cost.

The Queue-with-Array Problem

En-queue

Perform en-queue(e) as follows:

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

The Queue-with-Array Problem

De-queue

Perform de-queue as follows:

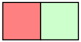
- Return the first element of A, and decrease m by 1. If A is sparse, shrink the array as follows:
    - Initialize an array A' of length $2m$.
    - Copy all the elements of A over to A'.
    - Destroy A, and replace it with A'.

We say that A is sparse if the number of integers therein is equal to $1/4$ of its length.

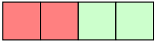
Example

Next, we use the algorithm to perform 11 en-queues and 9 de-queues on an initially empty queue.

$n = 1$, En-queue

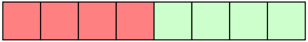


$n = 2$, En-queue



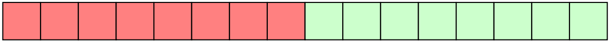
......

$n = 4$, En-queue



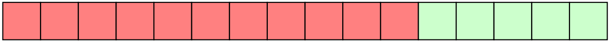
......

$n = 8$, En-queue



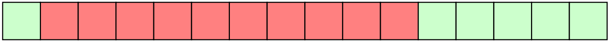
......

$n = 11$, En-queue



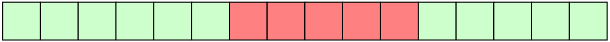
Example

$n = 12$, De-queue

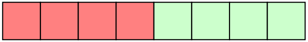


⋮

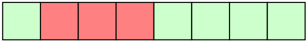
$n = 17$, De-queue



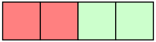
$n = 18$, De-queue



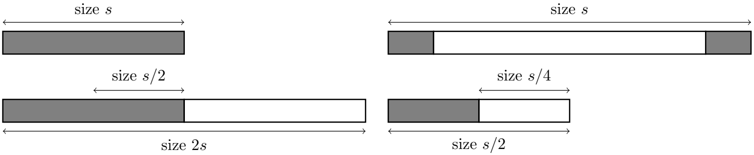
$n = 19$, De-queue



$n = 20$, De-queue



Cost Analysis



Array Expansion

Array Shrinking (take the array as circular)

The cost of expansion is at most $c_1 \cdot s$ for some constant c_1. By charging the cost over the $s/2$ en-queue operations as indicated above, each operation bears at most $2c_1$ cost.

The cost of shrinking is at most $c_2 \cdot s$ for some constant c_2. By charging the cost over the $s/4$ de-queue operations as indicated above, each operation bears at most $4c_2$ cost.

Hence, performing any sequence of operations using $O(n)$ time in total, and each operation (either an en-queue or a de-queue) is guaranteed to cost $O(1)$ amortized time.