Dynamic Arrays and Amortized Analysis

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
To create an array, you need to specify a size, i.e., how many elements you can store in the array. Increasing the size is expensive because it means creating a new array and moving all the elements over.

This lecture will discuss clever tricks to change the array size efficiently! Our discussion introduces the method of **amortized analysis**.
Dynamic Array Problem

Let $S$ be a collection of integers (not necessarily distinct). $S$ is empty in the beginning. Integers are then added to $S$ one by one with insertions.

Let $n$ be the number of elements in $S$ currently. We want to maintain an array $A$ satisfying:

1. $A$ has length $O(n)$.
2. For each $i \in [1, n]$, $A[i] = x$ if $x$ is the $i$-th integer added to $S$.

The above requirements need to be satisfied after every insertion.
Naive Algorithm

Perform insert(e) (which inserts an integer e to S) as follows:

- If \( n = 0 \), set \( n \) to 1 and initialize \( A \) to have length 1 to store \( e \).
- Otherwise (\( n \geq 1 \)):
  - Increase \( n \) by 1.
  - Initialize an array \( A' \) of length \( n \).
  - Copy all the \( n - 1 \) elements of \( A \) 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 reduce the time of inserting \( n \) elements dramatically to \( O(n) \). Our array \( A \) may have a length up to \( 2n \).
A Better Algorithm

A is **full** if its cells are all filled.

Perform insert(e) as follows:

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

\begin{itemize}
  \item $n = 1$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ]}
          \end{array} \]
  \end{itemize}
  \item $n = 2$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ] [ ]}
          \end{array} \]
  \end{itemize}
  \item $n = 3$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ] [ ] [ ]}
          \end{array} \]
  \end{itemize}
  \item $n = 4$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ] [ ] [ ] [ ]}
          \end{array} \]
  \end{itemize}
  \item $n = 5$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ] [ ] [ ] [ ] [ ]}
          \end{array} \]
  \end{itemize}
  \item $n = 8$
  \begin{itemize}
    \item \[ \begin{array}{c}
            \text{[ ] [ ] [ ] [ ] [ ] [ ] [ ]}
          \end{array} \]
  \end{itemize}
\end{itemize}
Analysis

Cost of inserting the $n$-th element:

- if $A$ is not full after the insertion, $O(1)$;
- otherwise, $O(n)$, i.e., the time of expanding $A$. 

Yufei Tao

Dynamic Arrays and Amortized Analysis
Analysis

Array expansions are infrequent:

- Initially, size 2.
- 1st expansion: size from 2 to 4.
- 2nd expansion: from 4 to 8.
- ...
- \(i\)-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, at most \(\log_2 n\) expansions.
The total cost of $n$ insertions is bounded by:

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

where

- the first term captures the $O(1)$ time compulsory for each insertion;
- the second term captures all the expansion cost.

(1) evaluates to $O(n)$.
We have shown that the total cost of $n$ insertions is $O(n)$. In other words, each insertion entails $O(1)$ cost “on average”. This does not mean that every insertion can be performed in $O(1)$ time. The cost of some insertions can reach $\Omega(n)$. 
In general, if a data structure can process any $n$ operations in $f(n)$ time, we say that it guarantees an amortized cost of $\frac{f(n)}{n}$ per operation.

The dynamic array guarantees $O(1)$ amortized cost per insertion.