Charging Arguments
[Notes for ESTR2102]

Yufei Tao

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

Today, we will learn a **charging argument** technique to prove amortized costs.
Consider \( n \) operations on a data structure. The \( i \)-th (\( 1 \leq i \leq n \)) operation incurs cost \( C_i \). Our goal is to prove:

\[
\sum_{i=1}^{n} C_i \leq f(n). \tag{1}
\]

Suppose that we can assign a “fake” cost \( \overline{C_i} \leq \frac{f(n)}{n} \) to the \( i \)-th operation such that

\[
\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \overline{C_i}. \tag{2}
\]

(1) will then follow from (2).
Recall: the 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.
Recall: The Expansion Algorithm

\[ n = 1 \]

\[ n = 2 \]

\[ n = 3 \]

\[ n = 4 \]

\[ n = 5 \]

... 

\[ n = 8 \]
Charging Argument

Earlier, we proved that each insertion has amortized cost $O(1)$. Next, we give an alternative analysis for proving the same.

Our algorithm ensures an invariant:

After an expansion, the new array has size $2n$, namely, there are $n$ empty positions.
Let $C_i$ be the actual cost of the $i$-th insertion.

We will assign an \textbf{amortized cost} $\bar{C}_i$ to the $i$-th insertion.
Charging Argument

For the $n$-th operation, first set $C_n = O(1)$.

If the array does not expand, done.

An array expansion takes at most $cn$ time for some constant $c$.
⇒ The previous expansion happened when $S$ had $n/2$ elements.
⇒ $n/2$ empty positions in the previous array.
⇒ $n/2$ insertions have taken place since the previous expansion.
⇒ **Charge the $cn$ cost over those $n/2$ insertions**: for each of those insertions, add $\frac{cn}{n/2} = 2c = O(1)$ to its amortized cost.
Example

\[
\begin{align*}
n &= 1 \\
\text{expanding cost charged on the insertion of the 2nd element}
\end{align*}
\]

\[
\begin{align*}
n &= 2 \\
\text{expanding cost charged on the insertion of the 2nd element}
\end{align*}
\]

\[
\begin{align*}
n &= 3 \\
\text{expanding cost charged on the insertions of elements 3, 4}
\end{align*}
\]

\[
\begin{align*}
n &= 4 \\
\text{expanding cost charged on the insertions of elements 3, 4}
\end{align*}
\]

\[
\begin{align*}
n &= 5 \\
\text{expanding cost charged on the insertions of elements 3, 4}
\end{align*}
\]

\[
\begin{align*}
n &= 8 \\
\text{expanding cost charged on the insertions of elements 5-8}
\end{align*}
\]

Each insertion is charged at most once.
Charging Argument

Convince yourself:

\[
\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \overline{C}_i
\]

and

\[
\overline{C}_i = O(1).
\]

Therefore, the total cost of all the \( n \) operations is \( O(n) \).
The Stack-with-Array Problem

Let $S$ be a collection of integers (not necessarily distinct). We want to support:

- **push($e$):** add an integer $e$ into $S$.
- **pop:** remove the most recently inserted integer 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 $O(m)$

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

1. A is **full** if all its cells are filled.
2. A is **sparse** if at most 1/4 of its cells are filled.

We will enforce an invariant:

At creation, an array is **half full** (i.e., half of its cells are filled).
Push

Carry out $\text{push}(e)$ in the same way we perform an insertion in the dynamic array problem.
Perform pop as follows:

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

Pop

Charging Arguments
Example

11 pushes followed by 9 pops on an initially empty stack:

\[ n = 1, \text{push} \]

\[ n = 2, \text{push} \]

\[ \ldots \]

\[ n = 4, \text{push} \]

\[ \ldots \]

\[ n = 8, \text{push} \]

\[ \ldots \]

\[ n = 11, \text{push} \]
Example

... 

\( n = 17 \) pop

\[
\begin{array}{cccccccccccccccc}
\text{Red} & \text{Pink} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} \\
\end{array}
\]

\( n = 18 \), pop

\[
\begin{array}{cccccccccccccccc}
\text{Red} & \text{Red} & \text{Pink} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} \\
\end{array}
\]

\( n = 19 \), pop

\[
\begin{array}{cccccccccccccccc}
\text{Red} & \text{Red} & \text{Red} & \text{Pink} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} \\
\end{array}
\]

\( n = 20 \), pop

\[
\begin{array}{cccccccccccccccc}
\text{Red} & \text{Red} & \text{Red} & \text{Red} & \text{Pink} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} & \text{Green} \\
\end{array}
\]
Think: how to prove that each operation incurs only $O(1)$ amortized cost?