The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
The Accounting Method

Principle

▶ Every primitive operation costs 1-unit money.
▶ Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
The Accounting Method

Principle

- Every primitive operation costs 1-unit money.
- Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
- Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!
The Accounting Method

Principle

▶ Every primitive operation costs 1-unit money.
▶ Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
▶ Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!

Correctness

#all primitive ops \leq #all money deposited
= amortized complexity \times \# ops
The Accounting Method

Principle

▶ Every primitive operation costs 1-unit money.
▶ Deposit money whenever performing an operation (amortized complexity). Money spent after every primitive operation.
▶ Your bank starts with zero-balance and remains non-negative during the whole procedure. No loan!

Correctness

\[
\text{#all primitive ops} \leq \text{#all money deposited} \leq \text{amortized complexity} \times \text{# ops}
\]

\leq \text{due to your balance being non-negative all the time!}
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
The Accounting Method: Example

Push() & Multi-pop()

▶ deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
▶ deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.

Credit Invariant

▶ Invariant: # of (bank) credits = # of items in the stack.
▶ Prove the invariant for each operation: push(), multi-pop().
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
- deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.
- A formal proof requires showing the non-negativity of your balance.
The Accounting Method: Example

Push() & Multi-pop()

- deposit 2$ for each Push(): 1$ is spent to execute the push operation, 1$ is left in the bank for later.
- deposit 0$ for each Multi-pop(): its cost is paid for by the deposit made at the push operation.
- A formal proof requires showing the non-negativity of your balance.

Credit Invariant

- Invariant: # of (bank) credits = # of items in the stack.
- Prove the invariant for each operation: push(), multi-pop().
The Potential Function Method

Principle

▶ Every primitive operation costs 1-unit energy.
The Potential Function Method

Principle

- Every primitive operation costs 1-unit energy.
- For each operation, energy cost + potential energy change = amortized complexity.

Mathematics

Let $\Phi_i$ denote the potential energy right after the $i$th op. $\Phi_0 = 0$, $\Phi_i \geq 0$, $\forall i$. Let $t_i$ denote the actual running time of the $i$th op. Then its amortized running time $t'_i$ is defined to be $t'_i = t_i + \Phi_i - \Phi_{i-1}$. 
The Potential Function Method

**Principle**

- Every primitive operation costs 1-unit energy.
- For each operation, energy cost + potential energy change = amortized complexity.
- Potential energy starts with 0 and remains non-negative.
The Potential Function Method

 Principle

- Every primitive operation costs 1-unit energy.
- For each operation, energy cost + potential energy change = amortized complexity.
- Potential energy starts with 0 and remains non-negative.

 Mathematics

- Let $\Phi_i$ denote the potential energy right after the $i$th op. $\Phi_0 = 0$, $\Phi_i \geq 0$, $\forall i$.
- Let $t_i$ denote the actual running time of the $i$th op. Then its amortized running time $t'_i$ is defined to be

$$t'_i = t_i + \Phi_i - \Phi_{i-1}$$
Correctness: total actual running time $\leq$ total amortized running time

\[ T = \sum_i t_i \]

\[ = \sum_i (t_i' + \Phi_{i-1} - \Phi_i) \]

\[ = \sum_i t_i' + \sum_i (\Phi_{i-1} - \Phi_i) \]

\[ = T' + (\Phi_0 - \Phi_n) \]

\[ \leq T' \]

where $T' = \sum_i t_i'$, the total amortized time of all operations. The second summation simplifies to $(\Phi_0 - \Phi_n)$ due to the telescoping sum.
The Potential Function Method: Example

Setup

- Set $\Phi_i = \# \text{ of items in the stack}$. $\Phi_0 = 0$ and $\Phi_i \geq 0$, $\forall i$. 
The Potential Function Method: Example

Setup

- Set $\Phi_i =$ # of items in the stack. $\Phi_0 = 0$ and $\Phi_i \geq 0, \forall i$.  
- Push() : $t' = t_i + \Phi_i - \Phi_{i-1} = 1 + 1 = 2$, which is $O(1)$. The change in the potential is an increase in one, which combines with the constant-time operations of push to yield a total amortized cost of 2.
The Potential Function Method: Example

Setup

- Set $\Phi_i = \# \text{ of items in the stack}$. $\Phi_0 = 0$ and $\Phi_i \geq 0, \forall i$.
- Push() : $t' = t_i + \Phi_i - \Phi_{i-1} = 1 + 1 = 2$, which is $O(1)$. The change in the potential is an increase in one, which combines with the constant-time operations of push to yield a total amortized cost of 2.
- Multi-pop(): Multipop(k): $t' = t_i + \Phi_i - \Phi_{i-1} = k + -k = 0$, which is $O(1)$. The potential decrease cancels the running time of multi-pop().
Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).
Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).

- **Rank(e):** # of elements that precede $e$ in $S$ (start with rank 0). Similar to an array index. Different in the sense that it does not necessarily point to a physical location.
Vectors & Lists

Insertion and removal in the middle

- How to refer to an item in the middle of $S$? We already know how to operate at the top (stack), front, rear (queue).
- **Rank**(e): # of elements that precede e in $S$ (start with rank 0). Similar to an array index. Different in the sense that it does not necessarily point to a physical location.
- **Position**(e): relative "place" of e to others in $S$. In the object-oriented design, a position is an ADT that supports: element()

  \[\text{element()}: \text{Return the element stored at this position.}\]
Vector: insertion and removal based on rank

Implementation with Array
An array of size $N$: $A[i]$ stores the element with rank $i$.
$n$: # elements ($n < N$).
Vector: insertion and removal based on rank

Implementation with Array
An array of size $N$: $A[i]$ stores the element with rank $i$. $n$: # elements ($n < N$).

Insert $e$ at rank $r$

**Algorithm** insertAtRank($r, e$)

for $i = n - 1, n - 2, \ldots, r$ do

$A[i + 1] \leftarrow A[i]$: make room for the new element

end for

$A[r] \leftarrow e$: insert $e$ at the rank $r$

$n \leftarrow n + 1$: maintain # elements
Vector: insertion and removal based on rank

Implementation with Array

An array of size $N$: $A[i]$ stores the element with rank $i$.
$n: \# \text{ elements (} n < N\).$

Insert $e$ at rank $r$

Algorithm insertAtRank$(r, e)$
for $i = n - 1, n - 2, \cdots, r$ do
  $A[i + 1] \leftarrow A[i]$: make room for the new element
end for
$A[r] \leftarrow e$: insert $e$ at the rank $r$
$n \leftarrow n + 1$: maintain $\# \text{ elements}$

Time: $O(n)$.
Remove $e$ at rank $r$

**Algorithm** `removeAtRank(r, e)`

```
e ← A[r]:
for $i = r, r + 1, \cdots, n - 2$ do
end for
$n ← n - 1$: maintain # elements
return $e$.
```

**Time:** $O(n)$. 
Vector: insertion and removal based on rank

Remove $e$ at rank $r$

**Algorithm** `removeAtRank(r, e)`

- $e \leftarrow A[r]$
- for $i = r, r + 1, \ldots, n - 2$ do
  - $A[i] \leftarrow A[i + 1]$: fill in for the removed element
- end for
- $n \leftarrow n - 1$: maintain $\#$ elements
- return $e$

**Time:** $O(n)$.

<table>
<thead>
<tr>
<th></th>
<th><code>size()</code></th>
<th><code>elemAtRank(r)</code></th>
<th><code>insertAtRank(r, e)</code></th>
<th><code>removeAtRank(r, e)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
List: insertion and removal based on **position**

More about position

A position is defined *relatively*, in terms of its neighbors. i.e., a position $p$ is **after** position $q$ and **before** position $s$.
List: insertion and removal based on position

More about position
A position is defined relatively, in terms of its neighbors. i.e., a position \( p \) is after position \( q \) and before position \( s \)

A doubly linked list implementation

- Header and Trailer node.
- Other nodes: a next link, a prev link, and element stored.
List: insertion and removal based on position

Insert $e$ after position $p$

**Algorithm** `insertAfter(p, e)`
- Create a new node $v$
  - $v$.element $\leftarrow e$
  - $v$.prev $\leftarrow p$
  - $v$.next $\leftarrow p$.next
  - $(p$.next).prev $\leftarrow v$
- $p$.next $\leftarrow v$
- **return** $v$.

Remove the element at position $p$

**Algorithm** `remove(p)`
- $t \leftarrow p$.element
- $(p$.prev).next $\leftarrow p$.next
- $(p$.next).prev $\leftarrow p$.prev
- $p$.prev $\leftarrow null$
- $p$.next $\leftarrow null$
- **return** $t$. 

Comparison: Vector vs List

**Question:** Get rank for List?
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$. 

**Access:** Vector better than List rank-based
Vector equal List position-based

**Update:** Vector equal List rank-based
Vector worse than List position-based

**Space:** array-based $O(N)$ vs. doubly linked list $O(n)$. 
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$.

- **Access:** Vector *better than* List
  Vector *equal* List

  rank-based

  position-based
Comparison: Vector vs List

Question: Get rank for List? $\Theta(n)$.

- **Access:** Vector **better than** List
  Vector **equal** List
- **Update:** Vector **equal** List
  Vector **worse than** List

rank-based  position-based
Comparison: Vector vs List

**Question:** Get rank for List? $\Theta(n)$.

- **Access:** Vector better than List  \hspace{1cm} \text{rank-based}
  Vector equal List \hspace{1cm} \text{position-based}
- **Update:** Vector equal List  \hspace{1cm} \text{rank-based}
  Vector worse than List \hspace{1cm} \text{position-based}
- **Space:** array-based $O(N)$ v.s. doubly linked list $O(n)$. 