Lecture: Amortized Analysis

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method

These notes are hand-written, unedited and sketchy. They are primarily used for, and based on my lectures.

If you find any bug, impreciseness, or a rare poor-/mis-interpretation of facts, please let me know. I will be grateful for any additional comments you have that are intended to make the quality of the notes better.

Please note that I will provide my hand-written lecture notes only for a subset of my lectures, not for all lectures. Therefore, it is your responsibility to attend all the lectures, take notes regularly, and ask me and/or the TAs if you have any questions.

Thank you!
--Chittu

Written using an Intuos 5 Touch Pen Tablet and Microsoft Office OneNote
Amortized Analysis

* Suppose we have a data structure that supports
  a set of basic operations, e.g., INSERT, DELETE, SEARCH.

* We have analyzed the worst case cost of these
  operations in many cases, e.g.,
  - $O(1)$ time for SEARCH in BST
  - $O(\log n)$ time for SEARCH in Red-Black Trees

* How about analyzing the average performance
  of each operation in the worst case?

  - Why? Although an individual operation can be
    expensive, on average the cost/operation is small
    in many cases.

* Note: “Average” here means “average cost/op.” We
  are NOT averaging over a distribution of inputs.
  Therefore,
  1) No probability involved
  2) Goal is to analyze average cost in the
     worst case.

* The amortized cost per operation for a sequence of
  $n$ operations is the total cost of these $n$ operations
  divided by $n$. 
Methods to Compute Amortized Cost:

- The Summation/Aggregate Method aka Engineer’s method
- The Taxation/Accounting Method aka Banker’s Method
- The Potential Method aka Physicist’s method

The Summation/Aggregate Method: aka Engineer’s Method

(Simplest) Idea:

Let \( T(n) = \) total worst case cost over a sequence of \( n \) operations.

Then, the amortized cost per operation = \( \frac{T(n)}{n} \).

Example: Implementing a queue using two stacks:

- \( S_1 \) and \( S_2 \) are two stacks
- Operations on \( S_1 \) and \( S_2 \):
  - \( S.\text{push}(x) \): \( O(1) \) cost/\( \text{op} \).
  - \( S.\text{pop}() \): \( O(n) \) cost/\( \text{op} \).
- Queue operation to be supported: \( \text{ENQUEUE} \) and \( \text{DEQUEUE} \).
- We use \( S_1 \) for \( \text{ENQUEUE} \) and \( S_2 \) for \( \text{DEQUEUE} \).
• Implementation:
  ° ENQUEUE is pushing the element to $S_1$.
  ° DEQUEUE is popping the element from $S_2$.

   • What if $S_2$ is empty?

   ° Transfer the entire content of $S_1$ to $S_2$.
   ° Then do pop operation.

• Observation 1: The order of elements placed in $S_2$ is just the opposite of the order when the elements were in $S_1$.

• Observation 2: Life cycle of an element $x$:

   ° Stage 1: $x$ is pushed into $S_1$, i.e., ENQUEUED.

   ° Stage 2: $x$ is popped from $S_1$, and is to be pushed into $S_2$.

   ° Stage 3: $x$ is pushed into $S_2$.

   ° Stage 4: $x$ is popped from $S_2$, i.e., DEQUEUED finally.

• Pseudocode:

  ENQUEUE($S_1, S_2, x$)

  1. PUSH($S_1, x$)

  DEQUEUE($S_1, S_2$)

  1. IF $\neg S_2$·EMPTY()

     2. RETURN $S_2$·POPC()

     3. ELSE // $S_2$ empty. Transfer from $S_1$ to $S_2$.

     4. IF $S_1$·EMPTY()

     5. Error "Queue is empty"

     6. ELSE WHILE $\neg S_1$·EMPTY()

     7. $x \leftarrow S_1$·POPC()

     8. $S_2$·PUSH($x$)

     9. RETURN $S_2$·POPC()
Claim: The amortized cost of ENQUEUE and DEQUEUE is O(1)

Proof:

- Each element goes through 2 Push + 2 Pop during its life time.
- Let cost of Push and Pop be 1 unit each.
- n ENQUEUES and ≤ n DEQUEUES.

- Cost (n ENQUEUE) = cost (n PUSH) ⇒ Cost (1 ENQUEUE) = \(\frac{n}{n} = 1\).
- Cost (n DEQUEUE) = Cost ((1 PUSH + 2 POP) * n) = 3n
  ⇒ Cost (1 DEQUEUE) = \(\frac{3n}{n} = 3\). \(\text{Constant}\)

∴ amortized cost for ENQUEUE and DEQUEUE is O(1)

The Taxation/Accounting Method: aka Banker's Method

Idea: Certain operations on the data structure charge you taxes, so that the total cost of maintaining the data structure is never more than the total taxes you pay.

∴ the amortized cost of an operation is the overall tax you pay during that operation.

Key to this method: Find an appropriate “tax schedule.”
- Different operations charge different taxes.
  - Some charge more than actual cost
  - Some charge less
- Amortized cost = amount of tax charged
- Credit is stored in the data structure, when amortized cost > actual cost.
- The stored credit is later used to pay for operations whose actual cost > amortized cost.
- Must note: Credit CANNOT go negative. Why?
  - Otherwise, over a sequence of n operations, the amortized cost is not an upper bound of actual cost.
  - So, analysis becomes meaningless!
- Let $c_i =$ actual cost of $i$-th op.
  
  $\hat{c} =$ amortized cost of $i$-th op.

  We require $\sum_{i=1}^{n} \hat{c}_i > \sum_{i=1}^{n} c_i$, for sequence of $n$ ops.

- Total credit stored = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i > 0$, any time.
Example: (Analysis of) Implementing a queue using two stacks.

- Analysis is based on Observation 2 we made earlier.
- Actual cost of `ENQUEUE` = $1.
- Actual cost of `DEQUEUE` = \( \begin{cases} $1 & \text{if } S_2 \text{ is not empty} \\ $(1 + 2 \times S_1 \cdot \text{size}(S_2)) & \text{otherwise} \end{cases} \)

- Pay \$3 more during `ENQUEUE` to defray:
  - \$1 to `POP` from \( S_1 \)
  - \$1 to `PUSH` to \( S_2 \)
  - \$1 to finally `POP` from \( S_2 \).
  
- \$4 total.

- Paying \$4, when an element is enqueued is sufficient. Make sure that we are paying enough to defray the cost of `DEQUEUE` operation.

  i.e., the amortized cost of `ENQUEUE` is \$4, and that of `DEQUEUE` is \$0.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual Cost</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ENQUEUE</code></td>
<td>1</td>
<td>4 ($3)</td>
</tr>
<tr>
<td><code>DEQUEUE</code></td>
<td>1 if ( S_2 ) is not empty</td>
<td>0 $1</td>
</tr>
<tr>
<td><code>DEQUEUE</code></td>
<td>1 + 2 \times S_1 \cdot \text{size}(S_2) \text{ if } S_2 \text{ is empty}</td>
<td>0 $1</td>
</tr>
</tbody>
</table>

- We can show that at any time credit never goes negative!
  
  i.e. \( \sum_{i=1}^{n} C_i \geq \frac{n}{2} \sum_{i=1}^{n} C_i \).
The Potential Method: aka The Physicist’s Method

Idea:

• Define a potential function $\phi$ for the entire data structure.
  
  • Initialize it to 0, i.e. $\phi_{init} = 0$.
  
  • Always maintain $\phi > 0$.
  
  • Then, the amortized cost of an operation is its actual cost plus the change in potential.
  
  That is, $\hat{C}_i = C_i + \phi_i - \phi_{i-1}$.

• Potential function $\phi : D \rightarrow R$.

  \[
  \hat{C}_i = C_i + \phi_i - \phi_{i-1} = C_i + \Delta \phi_i
  \]

  Change in potential due to $i$th op.

  Note $\phi_i = \phi(D_i)$, and $\phi_{i-1} = \phi(D_{i-1})$.

• Total amortized cost $= \sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} (C_i + \phi_i - \phi_{i-1}) = \phi_n - \phi_0 + \sum_{i=1}^{n} C_i$.

• We require $\phi_i > \phi_0, \forall i > 0$.

  So we can conveniently set $\phi_0 = 0$.

• Note: To show an upper bound, we must show

  amortized cost of a sequence of ops $\geq$ actual cost.

Accounting Method vs. Potential Method: (AM vs. PM)

- Credit/potential is stored in the entire data structure
  in PM, as opposed to individual elements/parts of it in AM.
- PM is most widely used and flexible method.
Example: Implementing a queue using two stacks.

- Accounting method gives a fair idea of setting up the potential function, which is defined as follows:
  \[ \phi_i = 3k + m. \]
  where \( \phi_i \) is potential after \( i \)th op

  \[ h = S_1. \text{size}(c), \]
  \[ m = S_2. \text{size}(c). \]

- Setting \( \phi_0 = 0 \), it is not difficult to check that \( \phi_i \geq \phi_0 = 0. \)

- Amortized cost of ENQUEUE:
  \[ \hat{c}_i = c_i + (\phi_i - \phi_{i-1}) = 1 + [3(k+1) + m] - (3k+m) = 4. \]

- Amortized cost of DEQUEUE:
  - when \( S_2 \) is not empty: \( \hat{c}_i = c_i + (\phi_i - \phi_{i-1}) = 1 + (3k+m-1) - (3k+m) = 0 \)
  - when \( S_2 \) is empty: \( \hat{c}_i = c_i + (\phi_i - \phi_{i-1}) = 1 + 2k + (0+k) + (3k-0) \)
    actual cost \( \Rightarrow 1. \)

Note: For any sequence of \( n \) operations

\[ \sum_{i=1}^{n} \hat{c}_i = \phi_n - \phi_0 + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i \geq 0. \]

Conclusion: Amortized costs of ENQUEUE and DEQUEUE operations are constant i.e. \( O(1) \). Upper bound only, remember!

Exercise: Pick \( \phi_i = 2 \times S_i. \text{size}(c) \). Will that also work?