CS4522 Advanced Algorithms

Batch 09, L4S1

Lecture 6: (21 June 2013) Amortized Analysis

N. H. N. D. DE SILVA
DEPT. OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF MORATUWA
Announcement

- Assignment 1 is out
- Due on 28th June
An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we’re taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the worst case.
Amortized Analysis: Why?

- How much does it cost per day to maintain a car? (Not a perfect example)
Amortized Analysis: Why?

- Some days; Almost zero cost!
Amortized Analysis: Why?

- Some days; You need to buy fuel
Amortized Analysis: Why?

- And some days you have to pay a LOT more
Amortized Analysis: Why?

- Dynamic data structures
  - a succession of inserts
  - Removes
  - find/retrieves (or overwrites)
Amortized Analysis: Why?

- A worst-case analysis can be too pessimistic.
  - For both the total cost and the average cost per operation of "maintaining" the structure.
  - Particularly true for self-adjusting structures.
  - Such as?

- Better approach: use amortized analysis which determines an amortized ("time" averaged) cost per operation.
Note the Differences...

- Average-case analysis
  - We average over all possible inputs

- Probabilistic analysis
  - We average over all possible random choices

- Amortized analysis
  - We average over a sequence of operations
  - Assumes worst-case input and typically does not allow random choices
Amortized Analysis: What? (Again)

More accurate analysis for dynamic sets and their operations (than typical analysis)

“amortized”:

- from accounting practice of spreading a large cost (incurred in one time period) over multiple time periods
- These other time periods are related to the reason for incurring the cost
Example: Dynamic Tables

- We don’t know how big table (or array) we might need when computation begins
- **Naïve solution:** allocating largest possible
- **Better solution:**
  - Allocate a small array initially
  - Double its size when we feel it’s too small
  - Need to keep track of the number of elements
Example ... contd

- Generally, doubling the array may mean:
  1. creating a new array of twice the size, and
  2. transferring elements to the new larger array (this can be expensive)

- What is the total cost for inserting n items?
  - Doubling and transferring happens at times
  - Other times, constant time insertion

- If items deleted, table may be contracted
First consider table with only insertions

\textbf{TABLE-INSERT} \((T, y)\)

\begin{itemize}
  \item if \(\text{size}[T] = 0\)
    \begin{itemize}
      \item allocate \(\text{table}[T]\) with 1 slot; \(\text{size}[T] \leftarrow 1\)
    \end{itemize}
  \item if \(\text{num}[T] = \text{size}[T]\)
    \begin{itemize}
      \item allocate \(\text{new\_table}\) with \(2 \times \text{size}[T]\)
      \item insert all items in \(\text{table}[T]\) into \(\text{new\_table}\)
      \item \(\text{table}[T] \leftarrow \text{new\_table}\); \(\text{size}[T] \leftarrow 2 \times \text{size}[T]\)
    \end{itemize}
  \item insert \(y\) into \(\text{table}[T]\)
  \item \(\text{num}[T] \leftarrow \text{num}[T] + 1\)
\end{itemize}
First consider table with only insertions

TABLE-INSERT (T, y)

if size[T] = 0
allocate table[T] with 1 slot; size[T] = 1
if num[T] = size[T]
allocate new_table with 2xsize[T]
insert all items in table[T] into new_table
insert y into table[T]
num[T] = num[T] + 1

size[T] = 0  num[T] = 0
First consider table with only insertions

**TABLE-INSERT (T, y)**

if $\text{size}[T] = 0$

allocate $\text{table}[T]$ with 1 slot; $\text{size}[T] \leftarrow 1$

if $\text{num}[T] = \text{size}[T]$

allocate $\text{new_table}$ with $2 \times \text{size}[T]$

insert all items in $\text{table}[T]$ into $\text{new_table}$

$\text{table}[T] \leftarrow \text{new_table}$; $\text{size}[T] \leftarrow 2 \times \text{size}[T]$

insert $y$ into $\text{table}[T]$

$\text{num}[T] \leftarrow \text{num}[T] + 1$

size[T] = 1  num[T] = 0
First consider table with only insertions

TABLE-INSERT \((T, y)\)

if \(\text{size}[T] = 0\)

allocate table\([T]\) with 1 slot; size\([T]\) $\leftarrow$ 1

if \(\text{num}[T] = \text{size}[T]\)

allocate new\(_{}\)\_{}table with 2x\(\text{size}[T]\)

insert all items in table\([T]\) into new\(_{}\)\_{}table

\(\text{table}[T] \leftarrow \text{new table};\) size\([T]\) $\leftarrow$ 2x\(\text{size}[T]\)

\(\text{insert } y \text{ into table}[T]\)

\(\text{num}[T] \leftarrow \text{num}[T] + 1\)

size\([T]\) = 1 \quad \text{num}[T] = 0
Example ... contd

First consider table with only insertions

TABLE-INSERT (T, y)
if size[T] = 0
    allocate table[T] with 1 slot; size[T] ← 1
if num[T] = size[T]
    allocate new_table with 2xsize[T]
    insert all items in table[T] into new_table
    table[T] ← new_table; size[T] ← 2xsize[T]
insert y into table[T]
num[T] ← num[T] + 1

size[T] = 1   num[T] = 1
First consider table with only insertions

\[
\text{TABLE-INSERT} \ (T, y) \\
\text{if } \text{size}[T] = 0 \ \\
\text{allocate table}[T] \text{ with 1 slot; size}[T] \leftarrow 1 \\
\text{if } \text{num}[T] = \text{size}[T] \\
\text{allocate new_table with 2xsize}[T] \\
\text{insert all items in table}[T] \text{ into new_table} \\
\text{table}[T] \leftarrow \text{new table; size}[T] \leftarrow 2x\text{size}[T] \\
\text{insert } y \text{ into table}[T] \\
\text{num}[T] \leftarrow \text{num}[T] + 1
\]

size[T] = 1 \quad \text{num}[T] = 1
First consider table with only insertions

**TABLE-INSERT** \((T, y)\)

- if \(\text{size}[T] = 0\)
  - allocate \(\text{table}[T]\) with 1 slot; \(\text{size}[T] \leftarrow 1\)
  - if \(\text{num}[T] = \text{size}[T]\)
    - allocate new_table with \(2 \times \text{size}[T]\)
    - insert all items in \(\text{table}[T]\) into new_table
    - \(\text{table}[T] \leftarrow \text{new table}; \text{size}[T] \leftarrow 2 \times \text{size}[T]\)
  - insert \(y\) into \(\text{table}[T]\)
  - \(\text{num}[T] \leftarrow \text{num}[T] + 1\)

\(\text{size}[T] = 1\)  \(\text{num}[T] = 1\)
Example ...contd

First consider table with only insertions

TABLE-INSERT \((T, y)\)

if size\([T]\) = 0

allocate table\([T]\) with 1 slot; size\([T]\) $\leftarrow 1$

if num\([T]\) = size\([T]\)

allocate new_table with 2xsize\([T]\)

insert all items in table\([T]\) into new_table

table\([T]\) $\leftarrow$ new_table; size\([T]\) $\leftarrow$ 2xsize\([T]\)

insert \(y\) into table\([T]\)

num\([T]\) $\leftarrow$ num\([T]\) + 1

size\([T]\) = 1     num\([T]\) = 1
Example ...contd

First consider table with only insertions

\[
\text{TABLE-INSERT} (T, y) \\
\text{if } \text{size}[T] = 0 \\
\quad \text{allocate } \text{table}[T] \text{ with 1 slot; } \text{size}[T] \leftarrow 1 \\
\text{if } \text{num}[T] = \text{size}[T] \\
\quad \text{allocate new_table with 2xsize}[T] \\
\quad \text{insert all items in table}[T] \text{ into new_table} \\
\quad \text{table}[T] \leftarrow \text{new table; } \text{size}[T] \leftarrow 2\times\text{size}[T] \\
\text{insert } y \text{ into table}[T] \\
\text{num}[T] \leftarrow \text{num}[T] + 1
\]

size[T] = 2 \quad \text{num}[T] = 1
First consider table with only insertions

TABLE-INSERT (T, y)

if size[T] = 0
    allocate table[T] with 1 slot; size[T] ← 1
if num[T] = size[T]
    allocate new_table with 2xsize[T]
    insert all items in table[T] into new_table
    table[T] ← new_table; size[T] ← 2xsize[T]
insert y into table[T]
num[T] ← num[T] + 1

size[T] = 2  num[T] = 1
First consider table with only insertions

TABLE-INSERT (T, y)

if size[T] = 0
    allocate table[T] with 1 slot; size[T] \leftarrow 1
if num[T] = size[T]
    allocate new_table with 2xsize[T]
    insert all items in table[T] into new_table
    table[T] \leftarrow new_table; size[T] \leftarrow 2xsize[T]
    insert y into table[T]
    num[T] \leftarrow num[T] + 1

size[T] = 2 \quad num[T] = 2
Example ...contd

Cost of inserting:
• 1st Element (A) = 1
• 2nd Element (B) = 2
• 3rd Element (C) = 3
• 4th Element (D) = 1

Copying A; Inserting B
Copying A, B; Inserting C
Cost of inserting:
- 1\textsuperscript{st} Element (A) = 1
- 2\textsuperscript{nd} Element (B) = 2 → Copying A ; Inserting B
- 3\textsuperscript{rd} Element (C) = 3 → Copying A,B ; Inserting C
- 4\textsuperscript{th} Element (D) = 1
- 5\textsuperscript{th} Element (E) = 5 → Copying A,B,C,D ; Inserting D

Pattern?
Example ...contd

Consider a sequence of $n$ insertions

- Initially empty table
- What is the cost $c_i$ of $i$-th insert operation?
  - $c_i = 1$ if table is not full
  - $c_i = i$ if table is full (1 insertion + $i - 1$ items copied)
- For $n$ insertions, worst-case operation is $O(n)$; so $O(n^2)$ for total running time
  - Is this correct, or tight enough?
  - Not really, as expanding table is infrequent
Example ...contd

Consider a sequence of $n$ insertions (...contd)

Total cost for $n$ insertions can be proved to be in $O(n)$
Example ...contd

- Consider a sequence of $n$ insertions (...contd)
- Expansion at $i$-th operation if $i-1$ is power of 2

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$size_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>$c_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Example ...contd

- Consider a sequence of \( n \) insertions (...contd)
  - Expansion at \( i \)-th operation if \( i-1 \) is power of 2

\[
c_i = \begin{cases} 
  i & \text{if } i - 1 \text{ is an exact power of 2} \\
  1 & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>size( i )</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>( c_i )</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( c_i )</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
Example ...contd

- Consider a sequence of \( n \) insertions (...contd)
- Total cost of \( n \) insertions is therefore

\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n
\]

Amortized cost of a single operation is 3

Thus, the average cost of each dynamic-table operation is \( \Theta(n)/n = \Theta(1) \).
Amortized analysis: What? (3rd Time!)

- An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

- Even though we’re taking averages, however, probability is not involved!
  - An amortized analysis guarantees the average performance of each operation in the worst case.
Techniques

3 most common techniques
1. Aggregate analysis method
2. Accounting method
3. Potential method

- CLRS book discuss these 3 using 2 e.g.
  - A stack with `multipop` operation
  - A binary counter counting up from 0
1. Aggregate Analysis

- Show for all \( n \), a sequence of \( n \) operations takes total worst-case \( T(n) \) time.
- In the worst-case, the amortized (average) cost per operation is \( \frac{T(n)}{n} \).
  - The same cost applies to each operation.
  - There can be several types of operations.
- This is the method shown in previous Example (insertions into dynamic table).
2. Accounting Method

- Assign differing charges to different operations
  - Some charged more/less than actual cost
  - Amount we charge is called its amortized cost
  - When amortized cost exceeds actual cost, difference assigned to objects in data structure as credit
  - Credit can be later used to pay for operations whose amortized cost is less than actual cost
2. Accounting Method

...contd

- Amortized cost of an operation split
  - between actual cost, and
  - credit that is either deposited or used up
- [note the difference from aggregate method]
Accounting method

• Charge \( i \) th operation a fictitious *amortized cost* \( \hat{c}_i \), where $1$ pays for $1$ unit of work (i.e., time).
• This fee is consumed to perform the operation.
• Any amount not immediately consumed is stored in the *bank* for use by subsequent operations.
• The bank balance must not go negative! We must ensure that

\[
\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i
\]

for all \( n \).
• Thus, the total amortized costs provide an upper bound on the total true costs.
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the $i$th insertion.

- $\$1$ pays for the immediate insertion.
- $\$2$ is stored for later table doubling.

When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

**Example:**

```
$0 \, $0 \, $0 \, $0 \, $2 \, $2 \, $2 \, $2$  overflow
```

```
[Yellow blocks]
```

```
[Yellow blocks]
```

```
[Yellow blocks]
```
Accounting analysis of dynamic tables

Charge an amortized cost of $c_i = 3$ for the $i$th insertion.
- $1$ pays for the immediate insertion.
- $2$ is stored for later table doubling.

When the table doubles, $1$ pays to move a recent item, and $1$ pays to move an old item.

Example:

```
$0$ $0$ $0$ $0$ $0$ $0$ $0$
```

overflow
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = 3$ for the $i$th insertion.

- $1$ pays for the immediate insertion.
- $2$ is stored for later table doubling.

When the table doubles, $1$ pays to move a recent item, and $1$ pays to move an old item.

**Example:**

```
  $0$ $0$ $0$ $0$ $0$ $0$ $0$ $0$ $0$ $2$ $2$ $2$
```
Accounting analysis (continued)

**Key invariant:** Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$size_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>$c_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>$\hat{c}_i$</td>
<td>2*</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$bank_i$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

*Okay, so I lied. The first operation costs only $2, not $3.*
3. Potential Method

- Represents the prepaid work as “potential energy” (or “potential”)
- Can be released to pay for future operations
- Potential is associated with the data structure as a whole
- In contrast: in accounting method, pre-paid work as credit is associated with specific objects in the data structure
3. Potential Method ...

- Start with an initial data structure $D_0$
- Perform $n$ operations
- For each $i=1, 2, \ldots, n$
  - $c_i$ is the actual cost
  - $D_i$ is the data structure that results after applying $i$-th operation to data structure $D_{i-1}$
- A potential function $\Phi$ maps each data structure $D_i$ to a real number $\Phi(D_i)$
  - It is the potential associated with data structure $D_i$
The amortized cost \( \langle c_i \rangle \) of the \( i \)-th operation w.r.t potential function \( \Phi \) is

\[
\langle c_i \rangle = c_i + \Phi(D_i) - \Phi(D_{i-1})
\]

That is, the actual cost plus the increase in potential due to the operation.

The total amortized cost for \( n \) operations can be computed by taking summation over \( n \).
3. Potential Method (contd)

\[ \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \]

potential difference \( \Delta \Phi_i \)

- If \( \Delta \Phi_i > 0 \), then \( \hat{c}_i > c_i \). Operation \( i \) stores work in the data structure for later use.

- If \( \Delta \Phi_i < 0 \), then \( \hat{c}_i < c_i \). The data structure delivers up stored work to help pay for operation \( i \).
The total amortized cost of $n$ operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides.
3. Potential Method...contd

The total amortized cost of $n$ operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

The series telescopes.
3. Potential Method ...contd

The total amortized cost of $n$ operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \quad \text{since } \Phi(D_n) \geq 0 \text{ and } \Phi(D_0) = 0.$$
Define the potential of the table after the \( i \)th insertion by \( \Phi(D_i) = 2i - 2^{[\lg i]} \). (Assume that \( 2^{[\lg 0]} = 0 \).)

**Note:**
- \( \Phi(D_0) = 0 \),
- \( \Phi(D_i) \geq 0 \) for all \( i \).

**Example:**

\[
\begin{array}{cccccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
$0$ & $0$ & $0$ & $0$ & $2$ & $2$
\end{array}
\]

\( \Phi = 2 \cdot 6 - 2^3 = 4 \) (accounting method)
The amortized cost of the $i$th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \begin{cases} 
  i & \text{if } i - 1 \text{ is an exact power of 2,} \\
  1 & \text{otherwise;}
\end{cases}$$

$$+ (2i - 2^{\lfloor \log i \rfloor}) - (2(i-1) - 2^{\lfloor \log (i-1) \rfloor})$$

$$= \begin{cases} 
  i & \text{if } i - 1 \text{ is an exact power of 2,} \\
  1 & \text{otherwise;}
\end{cases}$$

$$+ 2 - 2^{\lfloor \log i \rfloor} + 2^{\lfloor \log (i-1) \rfloor}.$$
3. Potential Method ...contd

Case 1: $i-1$ is an exact power of 2

\[
\hat{c}_i = i + 2 - 2 \left\lfloor \log_2 i \right\rfloor + 2 \left\lfloor \log_2 (i-1) \right\rfloor \\
\hat{c}_i = i + 2 - 2(i-1) + (i-1) \\
\hat{c}_i = i + 2 - 2i + 2 + i - 1 \\
\hat{c}_i = 3
\]
3. Potential Method ...contd

Case 2: $i-1$ is not an exact power of 2

\[
\hat{C}_i = 1 + 2 - 2^\left\lfloor \log i \right\rfloor + 2^\left\lfloor \log (i-1) \right\rfloor
\]

\[
\hat{C}_i = 1 + 2 - (i-1) + (i-1)
\]

\[
\hat{C}_i = 1 + 2 - i + 1 + i - 1
\]

\[
\hat{C}_i = 3
\]
3. Potential Method ...contd

Therefore, $n$ insertions cost $\Theta(n)$ in the worst case.

**Exercise:** Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.
Discussion

- Refer to 2 slidesets
  - 6-page note titled “Lecture 7 Amortized Analysis” from CMU (Online)
  - 42-slide presentation by Demaine and Leiserson of MIT (Online. Most of them were discussed in this presentation)

- Also read
  - Slides by Kevin Wayne at Princeton
    - Analysis of splay trees and other trees (Online)
Application: Splay Trees

- Review
  - Binary trees that are not balanced
  - Individual operations can take linear time
  - As operations are performed, tree tends to balance itself
  - In the long run, the amortized complexity is $O(lg n)$ per operation
  - See handout last week on Splay Trees
Other applications

- To analyze
  - Binomial heaps, Fibonacci heaps
  - Dictionaries and dynamic tables
  - KMP algorithm on string matching
  - Some graph algorithms
  - Several others....
Conclusion

- Amortized analysis
  - Introduction, why?
  - Examples, techniques

- Next class
  - Part 2: randomized algorithms
References

- Amortized Analysis [CLRS Chapter 17]
- The lecture slides are based on the slides prepared by Prof. Sanath Jayasena for this class in previous years.
- Presentation by Demaine and Leiserson of MIT