Lecture 17 (Amortized Analysis) - Feb. 9, 2018

CLRS 17-1, 17-2, 17-3, 17-4

University of Manitoba
Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of operations.
Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of operations.
- Average case analysis relies on probabilistic assumptions about the input or the data structure.
  - There is an underlying probability distribution.
  - The worst-case might be met with some small chance (you can be ‘lucky’ or not).
Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of operations.

- Average case analysis relies on probabilistic assumptions about the input or the data structure
  - There is an underlying probability distribution.
  - The worst-case might be met with some small chance (you can be ‘lucky’ or not).

- Amortized analysis consider consider a sequence of consecutive operations.
  - Bound the total cost for $m$ operations
  - This gives the amortized cost $B(n)$ per operation
  - The amortized cost is only a function of $n$, the size of stored data
  - Unlike average case analysis, there is no probability distribution
  - Every sequence of $m$ operations is guaranteed to have worst-case time at most $mB(n)$, regardless of the input or the sequence of operations (regardless of how luck you are).
Amortized vs Average Analysis

- Let’s compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ on average.
- B performs operations which take $\Theta(n)$ time in the worst case and amortized $\Theta(\log n)$.

<table>
<thead>
<tr>
<th></th>
<th>worst-case time per operation</th>
<th>average/amortized time per operation</th>
<th>worst-case time for $m$ operations</th>
<th>average time for $m$ operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ average</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ amortized</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let's compare two algorithms A and B.

A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ on average.

B performs operations which take $\Theta(n)$ time in the worst case and amortized $\Theta(\log n)$.

<table>
<thead>
<tr>
<th></th>
<th>worst-case time per operation</th>
<th>average/amortized time per operation</th>
<th>worst-case time for $m$ operations</th>
<th>average time for $m$ operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ average</td>
<td>$\Theta(m \cdot n)$</td>
<td></td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ amortized</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let’s compare two algorithms A and B

A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ on average.

B performs operations which take $\Theta(n)$ time in the worst case and amortized $\Theta(\log n)$.

<table>
<thead>
<tr>
<th></th>
<th>worst-case time per operation</th>
<th>average/amortized time per operation</th>
<th>worst-case time for $m$ operations</th>
<th>average time for $m$ operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ average</td>
<td>$\Theta(m \cdot n)$</td>
<td></td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ amortized</td>
<td>$\Theta(m \log n)$</td>
<td></td>
</tr>
</tbody>
</table>
Let’s compare two algorithms A and B

- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ on average.
- B performs operations which take $\Theta(n)$ time in the worst case and amortized $\Theta(\log n)$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst-case time per operation</th>
<th>average/amortized time per operation</th>
<th>worst-case time for $m$ operations</th>
<th>average time for $m$ operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ average</td>
<td>$\Theta(m \cdot n)$</td>
<td>$\Theta(m \log n)$</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ amortized</td>
<td>$\Theta(m \log n)$</td>
<td></td>
</tr>
</tbody>
</table>
Amortized vs Average Analysis

- Let’s compare two algorithms A and B

- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ on average.

- B performs operations which take $\Theta(n)$ time in the worst case and amortized $\Theta(\log n)$.

<table>
<thead>
<tr>
<th></th>
<th>worst-case time per operation</th>
<th>average/amortized time per operation</th>
<th>worst-case time for $m$ operations</th>
<th>average time for $m$ operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm A</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ average</td>
<td>$\Theta(m \cdot n)$</td>
<td>$\Theta(m \log n)$</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$ amortized</td>
<td>$\Theta(m \log n)$</td>
<td>$\Theta(m \log n)$</td>
</tr>
</tbody>
</table>
Bit Counter

- Start from an initial configuration where all bits are ‘0’
- Each operation increments the encoded number
- We want to know how many bits are flipped per operation
Bit Counter

- Start from an initial configuration where all bits are ‘0’
- Each operation increments the encoded number
- We want to know how many bits are flipped per operation
- The $i$’th bit from right is flipped iff all $i - 1$ bits on its right are 1 before the increment ($i \geq 0$)
  - After the flip all bits on the right will be 0.
  - In the next $2^i - 1$ operations after the flip the bit is not flipped.
  - The $i$’th bit is flipped once in $2^i$ operations

<table>
<thead>
<tr>
<th>Log m</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
<td>initial configuration</td>
<td>1 bit flipped</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 1</td>
<td>after 1st increment</td>
<td>2 bits flipped</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 1 0</td>
<td>after 2nd increment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0 1 1 0 1 1 1 1</td>
<td>after 111th increment</td>
<td>1 bit flipped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0 0 0 0</td>
<td>after 112th increment</td>
<td>5 bits flipped</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bit Counter

For a sequence of \( m \) operations, the \( i \)'th bit is flipped \( \frac{m}{2^i} \) times.

Total number of flips will be at most

\[
\begin{align*}
\frac{m}{2^0} + \frac{m}{2^1} + \ldots + \frac{m}{2^{\lceil \log m \rceil}} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m
\end{align*}
\]

The amortized number of flips per operation is \( \Theta(1) \) flips.

The worst case number of flips is \( \Theta(\log m) \); but it never happens that a sequence of \( m \) operations have \( m \Theta(\log m) \) flips!
Bit Counter

- For a sequence of $m$ operations, the $i$'th bit is flipped $\frac{m}{2^i}$ times.
- Total number of flips will be at most

$$m + \frac{m}{2} + \ldots + \frac{m}{2^{\lceil \log m \rceil}} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

- The amortized number of flips per operation is $2 = \Theta(1)$ flips.
Bit Counter

- For a sequence of $m$ operations, the $i$’th bit is flipped $\frac{m}{2^i}$ times.
- Total number of flips will be at most

\[
\frac{m}{2} + \frac{m}{2^2} + \ldots + \frac{m}{2^{\lceil \log m \rceil}} \leq m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m
\]

- The amortized number of flips per operation is $2 = \Theta(1)$ flips.
- The worst case number of flips is $\Theta(\log m)$; but it never happens that a sequence of $m$ operations have $m\Theta(\log m)$ flips!

<table>
<thead>
<tr>
<th>$\log m$</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial configuration</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>after 1st increment</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>after 2nd increment</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>after 111th increment</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>after 112th increment</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Amortized Analysis Review

- Considering a sequence of \( m \) operations for sufficiently large \( m \):
  - Some operations are more ‘expensive’ and most are ‘inexpensive’.
  - Amortized cost is the average cost over all operations
  - There is no probability distribution or randomness
Considering a sequence of \( m \) operations for sufficiently large \( m \):

- Some operations are more ‘expensive’ and most are ‘inexpensive’.
- Amortized cost is the average cost over all operations
- There is no probability distribution or randomness

We saw the amortized number of flips when incrementing a number \( m \) times is \( \Theta(1) \)

- Some increment operation need \( \Theta(\log m) \) flips while most operation take less flips.
- On average, each operation needs \( \Theta(1) \) flips.
Methods for Amortized Analysis

There are three frameworks for amortized analysis.

- **Aggregate method**:
  - Sum the total cost of $m$ operations
  - Divide by $m$ to get the amortized cost
  - This is what we did for bit flips
There are three frameworks for amortized analysis.

**Aggregate method:**
- Sum the total cost of $m$ operations
- Divide by $m$ to get the amortized cost
- This is what we did for bit flips

**Accounting method**
- Analogy with a bank account, where there are fixed deposits and variable withdrawals
Methods for Amortized Analysis

- There are three frameworks for amortized analysis.

  **Aggregate method:**
  - Sum the total cost of $m$ operations
  - Divide by $m$ to get the amortized cost
  - This is what we did for bit flips

- **Accounting method**
  - Analogy with a *bank account*, where there are *fixed deposits* and variable *withdrawals*

- **Potential method**
  - Define amortized cost through *potential function* which maps the sequence of operations to an integer
Methods for Amortized Analysis

- There are three frameworks for amortized analysis.

  - **Aggregate method:**
    - Sum the total cost of \( m \) operations
    - Divide by \( m \) to get the amortized cost
    - This is what we did for bit flips

  - **Accounting method**
    - Analogy with a bank account, where there are fixed deposits and variable withdrawals

  - **Potential method**
    - Define amortized cost through potential function which maps the sequence of operations to an integer

  - Let’s review these methods with an example!
Problem: implement a stack stored in an array to support push (insert) operations.

The problem is **online** in the sense that we do not know how many operations to expect.
Dynamic Arrays

- Problem: implement a stack stored in an array to support push (insert) operations.
- The problem is **online** in the sense that we do not know how many operations to expect.
- How large the array should be? There is a trade-off:
  - Larger array: less likely to run out of space, more unused/wasted memory
  - Smaller array: more likely to run out of space, less unused/wasted memory
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.

- If the array runs out of space ($n > a$):
  - allocate a new array of size $2n$
  - copy all $n$ items to the new array

  \[
  i \quad \text{operation}
  \]
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.

- If the array runs out of space ($n > a$):
  - allocate a new array of size $a 2n$
  - copy all $n$ items to the new array

  \[ i \] operation
Dynamic Arrays

Possible solution: allocate an array of size $a = 2n$.

If the array runs out of space ($n > a$):
- allocate a new array of size $2n$
- copy all $n$ items to the new array

\[
\begin{array}{ll}
i & \text{operation} \\
1 & \text{insert}(a)
\end{array}
\]
Dynamic Arrays

Possible solution: allocate an array of size $a = 2^n$.

If the array runs out of space ($n > a$):

- allocate a new array of size $2^n$
- copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
</tr>
<tr>
<td></td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $2n$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $2a$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $2n$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th>no space: allocate array of size 2, copy 1 item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $a = 2n$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th>details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>6</td>
<td>insert(f)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $2n$
  - copy all $n$ items to the new array

---

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>6</td>
<td>insert(f)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>insert(g)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size \( a = 2^n \).
- If the array runs out of space \( (n > a) \):
  - allocate a new array of size \( a 2^n \)
  - copy all \( n \) items to the new array

<table>
<thead>
<tr>
<th>( i )</th>
<th>operation</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>6</td>
<td>insert(f)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>insert(g)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>insert(h)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

Possible solution: allocate an array of size $a = 2^n$.

If the array runs out of space ($n > a$):

- allocate a new array of size $a = 2^n$
- copy all $n$ items to the new array

$i$ operation
1 insert(a)
2 insert(b) no space: allocate array of size 2, copy 1 item
3 insert(c) no space: allocate array of size 4, copy 2 item
4 insert(d)
5 insert(e) no space: allocate array of size 8, copy 4 item
6 insert(f)
7 insert(g)
8 insert(h)
9 insert(i) no space: allocate array of size 16, copy 8 item
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2^n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $a = 2^n$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>6</td>
<td>insert(f)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>insert(g)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>insert(h)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>insert(i)</td>
<td>no space: allocate array of size 16, copy 8 item</td>
</tr>
<tr>
<td>10</td>
<td>insert(j)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Possible solution: allocate an array of size $a = 2^n$.
- If the array runs out of space ($n > a$):
  - allocate a new array of size $a = 2^n$
  - copy all $n$ items to the new array

<table>
<thead>
<tr>
<th>$i$</th>
<th>operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert(a)</td>
<td>no space: allocate array of size 2, copy 1 item</td>
</tr>
<tr>
<td>2</td>
<td>insert(b)</td>
<td>no space: allocate array of size 4, copy 2 item</td>
</tr>
<tr>
<td>3</td>
<td>insert(c)</td>
<td>no space: allocate array of size 8, copy 4 item</td>
</tr>
<tr>
<td>4</td>
<td>insert(d)</td>
<td>no space: allocate array of size 16, copy 8 item</td>
</tr>
<tr>
<td>5</td>
<td>insert(e)</td>
<td>no space: allocate array of size 32, copy 16 item</td>
</tr>
<tr>
<td>6</td>
<td>insert(f)</td>
<td>no space: allocate array of size 64, copy 32 item</td>
</tr>
<tr>
<td>7</td>
<td>insert(g)</td>
<td>no space: allocate array of size 128, copy 64 item</td>
</tr>
<tr>
<td>8</td>
<td>insert(h)</td>
<td>no space: allocate array of size 256, copy 128 item</td>
</tr>
<tr>
<td>9</td>
<td>insert(i)</td>
<td>no space: allocate array of size 512, copy 256 item</td>
</tr>
<tr>
<td>10</td>
<td>insert(j)</td>
<td>no space: allocate array of size 1024, copy 512 item</td>
</tr>
<tr>
<td>11</td>
<td>insert(k)</td>
<td>no space: allocate array of size 2048, copy 1024 item</td>
</tr>
</tbody>
</table>
The worst-case cost occurs when the whole array is copied to a new array:

- $\Theta(n)$ worst-case time per insert.

Rough estimate: a sequence of $m$ insert operations takes $O(m \cdot n)$ time.
Dynamic Arrays

- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.
- Rough estimate: a sequence of $m$ insert operations takes $O(m \cdot n)$ time.
  - We can obtain a much better (smaller) bound.
Dynamic Arrays

The worst-case cost occurs when the whole array is copied to a new array:

- \(\Theta(n)\) worst-case time per insert.

Rough estimate: a sequence of \(m\) insert operations takes \(O(m \cdot n)\) time.

- We can obtain a much better (smaller) bound.

<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>array size ((a))</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>
</tbody>
</table>
Dynamic Arrays

- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.

- Rough estimate: a sequence of $m$ insert operations takes $O(m \cdot n)$ time.
  - We can obtain a much better (smaller) bound.

- Let $c(i)$ denote the cost of the $i$th insertion (cost = number of insert/copies).
  
  \[
  c(i) = \begin{cases} 
  i & \text{if } i = 2^k + 1 \text{ for some integer } k \\
  1 & \text{if otherwise}
  \end{cases}
  \]

<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>array size ($a$)</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>
</tbody>
</table>
Aggregate Method for Dynamic Arrays

- Aggregate method: find total cost of $m$ operations and divide by $m$

$$c(i) = \begin{cases} 
  i & \text{if } i = 2^k + 1 \text{ for some integer } k \\
  1 & \text{if otherwise}
\end{cases}$$

The amortized cost is hence $\Theta(1)$.
Aggregate Method for Dynamic Arrays

- Aggregate method: find total cost of $m$ operations and divide by $m$

$$c(i) = \begin{cases} 
  i & \text{if } i = 2^k + 1 \text{ for some integer } k \\
  1 & \text{if otherwise}
\end{cases}$$

Cost of $m$ insertions \[\sum_{i=1}^{m} c(i) \leq \underbrace{m}_{\text{insert new item}} + \underbrace{\sum_{j=0}^{\lfloor \log(m-1) \rfloor} 2^j}_{\text{copy old items to new array}}\]

\[= m + 2^{\lfloor \log(m-1) \rfloor} + 1 - 1\]

\[\leq m + 2^{\log m + 1} - 1\]

\[= m + 2m - 1\]

\[= 3m - 1\]

\[\in \Theta(m)\]
Aggregate Method for Dynamic Arrays

- Aggregate method: find total cost of \( m \) operations and divide by \( m \)

\[
c(i) = \begin{cases} 
  i & \text{if } i = 2^k + 1 \text{ for some integer } k \\
  1 & \text{if otherwise}
\end{cases}
\]

Cost of \( m \) insertions:

\[
\sum_{i=1}^{m} c(i) \leq m + \sum_{j=0}^{[\log(m-1)]} 2^j
\]

\[
= m + 2^{[\log(m-1)]+1} - 1 \\
\leq m + 2^{\log m + 1} - 1 \\
= m + 2m - 1 \\
= 3m - 1 \\
\in \Theta(m)
\]

- The amortized cost is hence \( \frac{\Theta(m)}{m} = \Theta(1) \)
Aggregate Method for Dynamic Arrays

- Aggregate method: find total cost of \( m \) operations and divide by \( m \)

\[
c(i) = \begin{cases} 
  i & \text{if } i = 2^k + 1 \text{ for some integer } k \\
  1 & \text{otherwise}
\end{cases}
\]

Cost of \( m \) insertions:

\[
\sum_{i=1}^{m} c(i) \leq \underbrace{m}_{\text{insert new item}} + \underbrace{\sum_{j=0}^{[\log(m-1)]} 2^j}_{\text{copy old items to new array}}
\]

\[
= m + 2^{[\log(m-1)]+1} - 1
\leq m + 2^{\log m+1} - 1
= m + 2m - 1
= 3m - 1
\in \Theta(m)
\]

- The amortized cost is hence \( \frac{\Theta(m)}{m} = \Theta(1) \)
- The aggregate is useful for simple amortized analysis.
- Sometimes require a different technique to obtain amortized cost.
Assume you want to prove that your average (amortized) daily cost is no more than 100$. 

One way to do that is to assume every day 100$ is deposited into your account. On days which you spend more than 100$, you should use accumulated credit from previous days. If your balance remains positive at the end of each day, your average cost is at most 100$.

In $m$ consecutive days your expenditure has been at most 100$, amortized cost at most 100$. 

Accounting Method
Accounting Method

Assume you want to prove that your average (amortized) daily cost is no more than 100$.

Some days you might spend much more but on average it is at most 100$.
Assume you want to prove that your average (amortized) daily cost is no more than 100$.

- Some days you might spend much more but on average it is at most 100$

One way to do that is to assume every day 100$ is deposited into your account.

On days which you spend more than 100$, you should use accumulated credit from previous days.
Accounting Method

- Assume you want to prove that your average (amortized) daily cost is no more than 100$.
  - Some days you might spend much more but on average it is at most 100$
- One way to do that is to assume every day 100$ is deposited into your account
- On days which you spend more than 100$, you should use accumulated credit from previous days
- If your balance remains positive at the end of each day, your average cost is at most 100$
  - In $m$ consecutive days your expenditure has been at most $100m \rightarrow$ amortized cost at most 100$. 

Accounting Method

Accounting method overview:

- Each operation **deposits** a fixed credit into an account (This amount is an upper bound on the amortized cost.)
- Each operation uses ‘credit’ to pay its **cost**
- Inexpensive operations save more than their cost
- Expensive operations cost more more than they save
- Account must remain positive
Accounting Method for Dynamic Arrays

- We prove the amortized cost for insertion is 3

![Insertion Diagram]

<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>array size ($a$)</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>total deposited</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>total spent</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>available credit</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Accounting Method for Dynamic Arrays

- We prove the amortized cost for insertion is 3
  - Each operation deposits $3
  - Each write/move operation costs $1

<table>
<thead>
<tr>
<th>$i$</th>
<th>array size ($a$)</th>
<th>$c(i)$</th>
<th>total deposited</th>
<th>total spent</th>
<th>available credit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10</td>
<td>1 2 3 1 5 1 1 1 9 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 2 4 4 8 8 8 8 16 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Inexpensive insertion deposits } 3 \text{ and spends } 1 = 2 \text{ saved}
\]
\[
\text{Expensive insertion deposits } 3 \text{ and spends } m \rightarrow (m - 3) \text{ spent}
\]

Number of consecutive inexpensive insertions before expensive insertion:

\[
m/2 - 1
\]

\[
2(\frac{m}{2} - 1) = (m - 2) \text{ accumulated credit since last expensive insertion}
\]

\[
m - 2 > m - 3 \rightarrow \text{account remains positive}
\]
Accounting Method for Dynamic Arrays

- We prove the amortized cost for insertion is 3
  - Each operation deposits $3
  - Each write/move operation costs $1
  - Inexpensive insertion deposits $3 and spends $1 = $2 saved

<table>
<thead>
<tr>
<th>$i$</th>
<th>array size ($a$)</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>$c(i)$</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>
<td></td>
</tr>
<tr>
<td>total deposited</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>total spent</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>26</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>available credit</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Accounting Method for Dynamic Arrays

We prove the amortized cost for insertion is 3

- Each operation deposits $3
- Each write/move operation costs $1
- Inexpensive insertion deposits $3 and spends $1 = $2 saved
- Expensive insertion deposits $3 and spends $m → $(m - 3) spent

<table>
<thead>
<tr>
<th>$i$</th>
<th>array size $(a)$</th>
<th>$c(i)$</th>
<th>total deposited</th>
<th>total spent</th>
<th>available credit</th>
</tr>
</thead>
</table>
| 1   | 1 2 3 4 5 6 7 8 9 10 | 1 2 3 1 5 1 1 1 9 1 | 3 6 9 12 15 18 21 24 27 30 | 1 3 6 7 12 13 14 15 26 27 | 2 3 3 5 3 7 9 1 3
Accounting Method for Dynamic Arrays

- We prove the amortized cost for insertion is 3
  - Each operation deposits $3
  - Each write/move operation costs $1
  - Inexpensive insertion deposits $3 and spends $1 = $2 saved
  - Expensive insertion deposits $3 and spends $m → $(m - 3)$ spent
  - Number of consecutive inexpensive insertions before expensive insertion: $m/2 - 1$
  - → $2(m/2 - 1) = $(m - 2) accumulated credit since last expensive insertion

<table>
<thead>
<tr>
<th>$i$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
<th>$i$</th>
<th>$j$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>array size $(a)$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></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>
<td></td>
</tr>
<tr>
<td>total deposited</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>total spent</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>26</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>available credit</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Accounting Method for Dynamic Arrays

- We prove the amortized cost for insertion is 3
  - Each operation deposits $3
  - Each write/move operation costs $1
  - Inexpensive insertion deposits $3 and spends $1 = $2 saved
  - Expensive insertion deposits $3 and spends $m \rightarrow $(m - 3) spent
  - Number of consecutive inexpensive insertions before expensive insertion: $m / 2 - 1$
  - $2(m / 2 - 1) = $(m - 2) accumulated credit since last expensive insertion
  - $m - 2 > m - 3 \rightarrow$ account remains positive

<table>
<thead>
<tr>
<th>i</th>
<th>array size (a)</th>
<th>c(i)</th>
<th>total deposited</th>
<th>total spent</th>
<th>available credit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10</td>
<td>1 2 3 1 5 1 1 1</td>
<td>3 6 9 12 15 18 21 24 27 30</td>
<td>1 3 6 7 12 13 14 15 26 27</td>
<td>2 3 3 5 3 5 7 9 1 3</td>
</tr>
</tbody>
</table>
Potential method

- Define a potential function \( \Phi \) that maps the state of the structure and the index of an operation to an integer.
  - Potential is basically the available credit in accounting method:
    \[
    \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1)
    \]
  - \( \hat{c}(i) \rightarrow \) amortized cost of operation \( i \)
  - \( c(i) \rightarrow \) actual cost of operation \( i \)
- Total amortized cost will be total cost plus a constant independent of \( m \).
Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- $a_i$ denotes the size of the array after operation $i$
Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- $a_i$ denotes the size of the array after operation $i$
- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be
    $$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) = 1 + [2i - a_i] - [2(i - 1) - a_{i-1}] = 3$$
Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$

- $a_i$ denotes the size of the array after operation $i$

- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be
    \[
    \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) = 1 + [2i - a_i] - [2(i - 1) - a_{i-1}] = 3
    \]

- For expensive operation $i$, table size changes from $a_{i-1} = (i - 1)$ to $a_i = 2(i - 1)$ and we have $c_i = i$.
  - the amortized cost will be
    \[
    \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) = i + [2i - a_i] - [2(i - 1) - a_{i-1}] \\
    = i + 2i - 2(i - 1) - 2i + 2 + (i - 1) = 3
    \]
Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- $a_i$ denotes the size of the array after operation $i$
- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be
    \[
    \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) = 1 + [2i - a_i] - [2(i - 1) - a_{i-1}] = 3
    \]
- For expensive operation $i$, table size changes from $a_{i-1} = (i - 1)$ to $a_i = 2(i - 1)$ and we have $c_i = i$.
  - the amortized cost will be
    \[
    \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) = i + [2i - a_i] - [2(i - 1) - a_{i-1}] \\
    = i + 2i - 2(i - 1) - 2i + 2 + (i - 1) = 3
    \]
- Potential method is often the strongest method for amortized analysis
Methods for Amortized Analysis

- There are three frameworks for amortized analysis.

  - **Aggregate method**: Sum the total cost of \( m \) operations. Divide by \( m \) to get the amortized cost.

  - **Accounting method**: Analogy with a bank account, where there are fixed deposits and variable withdrawals.

  - **Potential method**: Define amortized cost through potential function which maps the sequence of operations to an integer.

- Let’s review these methods with another example!
Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

$Op(n, x)$: pop $n$ items from the stack and push $x$ to it.
Special Stacks

Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

$Op(n, x)$: pop $n$ items from the stack and push $x$ to it.
Special Stacks

Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

$Op(n, x)$: pop $n$ items from the stack and push $x$ to it.

What is the time complexity of each operation?

Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).
Special Stacks

- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.
  $Op(n, x)$: pop $n$ items from the stack and push $x$ to it.
- What is the time complexity of each operation?
  - Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).
- Assume $m - 1$ operations pop nothing and the $m$'th operation pops everything
  - A single operation can have a cost of $\Theta(m)$ in the worst case.
Special Stacks

Consider a stack with one operation \( Op(n, x) \), where \( n \geq 0 \).

\( Op(n, x) \): pop \( n \) items from the stack and push \( x \) to it.

What is the time complexity of each operation?

- Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).

Assume \( m - 1 \) operations pop nothing and the \( m \)'th operation pops everything
  - A single operation can have a cost of \( \Theta(m) \) in the worst case.
  - The amortized time is much better!
Aggregate Method for Special Stacks

- Review of aggregate method:
  - Sum the total cost of $m$ consecutive operations
  - Divide by $m$ to get the amortized cost

Unlike bit flips and dynamic arrays, we cannot predict the cost of the $i^{th}$ operation. The aggregate method is limited and cannot help for amortized analysis of special stacks!
Aggregate Method for Special Stacks

- Review of aggregate method:
  - Sum the total cost of $m$ consecutive operations
  - Divide by $m$ to get the amortized cost

- Unlike bit flips and dynamic arrays, we cannot predict the cost of the $i$'th operation.

- The aggregate method is limited and cannot help for amortized analysis of special stacks!
Accounting Method for Special Stacks

- Review of accounting method:
  - Each operations comes with a **fixed deposit** that is added to the **account** (defines the amortized cost).
  - For each operation, we subtract the cost of the operation from the account
    - Inexpensive operations contribute to the account
    - Expensive operations take away from the account
  - Iff the account is positive after each operation, the amortized cost is at most the fixed deposit.

Accounting Method for Special Stacks

- Review of accounting method:
  - Each operation comes with a **fixed deposit** that is added to the **account** (defines the amortized cost).
  - For each operation, we subtract the cost of the operation from the account
    - Inexpensive operations contribute to the account
    - Expensive operations take away from the account
  - Iff the account is positive after each operation, the amortized cost is at most the fixed deposit.

- Often, the account can be imagined as sum of ‘credits’ assigned to different components of data structure

```
[ ] [ ] [ ] [ ]
  a  b  c  d  e  f  g  h
a  a  b  b  a  a  f  e  a

op(0,a)  op(0,b)  op(0,c)  op(1,d)  op(2,e)  op(0,f)  op(0,g)  op(4,h)
```
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack → account is the number of items in the stack.
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack → account is the number of items in the stack.
- \( OP(n, x) \) where \( n \geq 0 \):

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>op(0,a)</td>
<td>op(0,c)</td>
<td>op(1,d)</td>
<td>op(2,e)</td>
<td>op(0,f)</td>
</tr>
<tr>
<td>g</td>
<td>f</td>
<td>e</td>
<td>a</td>
<td>h</td>
</tr>
<tr>
<td>op(0,g)</td>
<td>op(4,h)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bal:4</td>
<td>Bal:1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack → account is the number of items in the stack.
- \( OP(n, x) \) where \( n \geq 0 \):
  - Pop \( n \) items: there is a credit of 1 for each item that is popped; so the cost that the algorithm pays for pops is the same as the consumed credit → account remains positive

```
1 2 3 3 2 3 4 1
```

```
op(0,a) op(0,b) op(0,c) op(1,d) op(2,e) op(0,f) op(0,g) op(4,h)
```

Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack → account is the number of items in the stack.
- \( OP(n, x) \) where \( n \geq 0 \):
  - Pop \( n \) items: there is a credit of 1 for each item that is popped; so the cost that the algorithm pays for pops is the same as the consumed credit → account remains positive
  - Push(\( x \)): there is a cost of 1 and fixed deposit of 2; the extra saving is stored as the credit for the item.
Accounting Method for Special Stacks

- With a fixed deposit of 2 per operation, we showed that the balance remains positive after each operation.
- The balance was the accumulated credits stored in each item in the stack.
- We conclude that the amortized cost of each operation is at most 2.
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation $i$ to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.

- Amortized cost is the summation of actual cost and the difference in potential function:
  \[ \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) \]
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation $i$ to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  \[
  \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1)
  \]
- Define the potential to be the number of items in the stack

Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation $i$ to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  \[ \hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1) \]
- Define the potential to be the number of items in the stack
  - Assume operation $i$ is $OP(n, x)$. The actual cost is $c(i) = n + 1$. 

```
OP(0, a)  OP(0, b)  OP(0, c)  OP(1, d)  OP(2, e)  OP(0, f)  OP(0, g)  OP(4, h)
  a  b  c  d  e  f  g  h
```

Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation $i$ to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.

- Amortized cost is the summation of actual cost and the difference in potential function:
  $$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1)$$

- Define the potential to be the number of items in the stack
  - Assume operation $i$ is $OP(n, x)$. The actual cost is $c(i) = n + 1$.
  - After the operation, the number of items is increased by $1 - n$, i.e., $\Phi(i) - \Phi(i - 1) = 1 - n$. 
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation $i$ to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.

- Amortized cost is the summation of actual cost and the difference in potential function:
  $$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i - 1)$$

- Define the potential to be the number of items in the stack
  - Assume operation $i$ is $OP(n, x)$. The actual cost is $c(i) = n + 1$.
  - After the operation, the number of items is increased by $1 - n$, i.e., $\Phi(i) - \Phi(i - 1) = 1 - n$.
  - The amortized cost is $\hat{c}(i) = (n + 1) + (1 - n) = 2$. 

```plaintext
op(0,a)  op(0,b)  op(0,c)  op(1,d)  op(2,e)  op(0,f)  op(0,g)  op(4,h)
```
More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).
More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).

- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression
More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).

- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression
  - Splay trees: dynamic binary trees which move an accessed item closer to the root.
    - Ideal for real-world scenarios where there is locality in accesses
    - Dynamic optimality conjecture: the amortized cost of accessing an item in a splay tree is within a constant ratio of any other tree (a challenging open question).
More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).

- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression
  - Splay trees: dynamic binary trees which move an accessed item closer to the root.
    - Ideal for real-world scenarios where there is locality in accesses
    - Dynamic optimality conjecture: the amortized cost of accessing an item in a splay tree is within a constant ratio of any other tree (a challenging open question).

- The whole field of online algorithms!