Chapter 17 overview

Amortized analysis

- Analyze a sequence of operations on a data structure.
- **Goal**: Show that although some individual operations may be expensive, *on average* the cost per operation is small.

*Average* in this context does not mean that we’re averaging over a distribution of inputs.

- No probability is involved.
- We’re talking about *average cost in the worst case*.

Organization

We’ll look at 3 methods:

- aggregate analysis
- accounting method
- potential method

Using 3 examples:

- stack with multipop operation
- binary counter
- dynamic tables (later on)

Aggregate analysis

**Stack operations**

- **PUSH**($S, x$): $O(1)$ each $\Rightarrow O(n)$ for any sequence of $n$ operations.
- **POP**($S$): $O(1)$ each $\Rightarrow O(n)$ for any sequence of $n$ operations.
MULTIPOP$(S, k)$

while $S$ is not empty and $k > 0$
  do POP$(S)$
  $k \leftarrow k - 1$

Running time of MULTIPOP:
- Linear in # of POP operations.
- Let each PUSH/POP cost 1.
- # of iterations of while loop is $\operatorname{min}(s, k)$, where $s =$ # of objects on stack.
- Therefore, total cost $= \operatorname{min}(s, k)$.

Sequence of $n$ PUSH, POP, MULTIPOP operations:
- Worst-case cost of MULTIPOP is $O(n)$.
- Have $n$ operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

**Observation**
- Each object can be popped only once per time that it’s pushed.
- Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPS, including those in MULTIPOP.
- Therefore, total cost $= O(n)$.
- Average over the $n$ operations $\Rightarrow O(1)$ per operation on average.

Again, notice no probability.
- Showed worst-case $O(n)$ cost for sequence.
- Therefore, $O(1)$ per operation on average.

This technique is called **aggregate analysis**.

**Binary counter**
- $k$-bit binary counter $A[0..k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.
  - Value of counter is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, counter value is 0, so $A[0..k-1] = 0$.
- To increment, add 1 (mod $2^k$):
  
  \[
  \text{INCREMENT}(A, k) \\
  i \leftarrow 0 \\
  \text{while } i < k \text{ and } A[i] = 1 \\
  \text{do } A[i] \leftarrow 0 \\
  \text{ if } i < k \\
  \text{ then } A[i] \leftarrow 1
  \]
Example: $k = 3$

[Underlined bits flip. Show costs later.]

<table>
<thead>
<tr>
<th>counter</th>
<th>A</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Cost of INCREMENT = $\Theta(\# \text{ of bits flipped})$.

**Analysis:** Each call could flip $k$ bits, so $n$ INCREMENTS takes $O(nk)$ time.

**Observation**

Not every bit flips every time.

[Show costs from above.]

<table>
<thead>
<tr>
<th>bit</th>
<th>flips how often</th>
<th>times in $n$ INCREMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>every time</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>1/2 the time</td>
<td>$\lfloor n/2 \rfloor$</td>
</tr>
<tr>
<td>2</td>
<td>1/4 the time</td>
<td>$\lfloor n/4 \rfloor$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>1/2$^i$ the time</td>
<td>$\lfloor n/2^i \rfloor$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i \geq k$</td>
<td>never</td>
<td>0</td>
</tr>
</tbody>
</table>

Therefore, total # of flips = $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$

$< n \sum_{i=0}^{\infty} 1/2^i$

$= n \left( \frac{1}{1 - 1/2} \right)$

$= 2n$.

Therefore, $n$ INCREMENTS costs $O(n)$.

Average cost per operation = $O(1)$. 
Accounting method

Assign different charges to different operations.
- Some are charged more than actual cost.
- Some are charged less.

**Amortized cost** = amount we charge.

When amortized cost > actual cost, store the difference on specific objects in the data structure as credit.

Use credit later to pay for operations whose actual cost > amortized cost.

Diffs from aggregate analysis:
- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

Need credit to never go negative.
- Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
- Amortized cost would tell us nothing.

Let $c_i = \text{actual cost of } i^{th} \text{ operation}$,
\[ \hat{c}_i = \text{amortized cost of } i^{th} \text{ operation}. \]

Then require \[ \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \text{ for all sequences of } n \text{ operations.} \]

Total credit stored \[ = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0. \]

had better be

Stack

<table>
<thead>
<tr>
<th>operation</th>
<th>actual cost</th>
<th>amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>POP</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MULTIPOP</td>
<td>min(k, s)</td>
<td>0</td>
</tr>
</tbody>
</table>

**Intuition:** When pushing an object, pay $2.
- $1 pays for the PUSH.
- $1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has $1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, = $O(n)$, is an upper bound on total actual cost.
Binary counter

Charge $2 to set a bit to 1.
• $1 pays for setting a bit to 1.
• $1 is prepayment for flipping it back to 0.
• Have $1 of credit for every 1 in the counter.
• Therefore, credit \( \geq 0 \).

Amortized cost of INCREMENT:
• Cost of resetting bits to 0 is paid by credit.
• At most 1 bit is set to 1.
• Therefore, amortized cost \( \leq 2 \).
• For \( n \) operations, amortized cost = \( O(n) \).

Potential method

Like the accounting method, but think of the credit as potential stored with the entire data structure.
• Accounting method stores credit with specific objects.
• Potential method stores potential in the data structure as a whole.
• Can release potential to pay for future operations.
• Most flexible of the amortized analysis methods.

Let \( D_i \) = data structure after \( i \)th operation,
\( D_0 \) = initial data structure,
\( c_i \) = actual cost of \( i \)th operation,
\( \hat{c}_i \) = amortized cost of \( i \)th operation.

**Potential function** \( \Phi : D_i \rightarrow \mathbb{R} \)

\( \Phi(D_i) \) is the potential associated with data structure \( D_i \).

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

increase in potential due to \( i \)th operation

Total amortized cost = \( \sum_{i=1}^{n} \hat{c}_i \)
\[
= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))
\]
(telescoping sum: every term other than \( D_0 \) and \( D_n \)
is added once and subtracted once)
\[
= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .
\]
If we require that \( \Phi(D_i) \geq \Phi(D_0) \) for all \( i \), then the amortized cost is always an upper bound on actual cost.

In practice: \( \Phi(D_0) = 0, \Phi(D_i) \geq 0 \) for all \( i \).

**Stack**

\( \Phi = \) # of objects in stack

\[ \phi \approx \text{# of$1 bills in accounting method} \]

\( D_0 = \) empty stack \( \Rightarrow \Phi(D_0) = 0 \).

Since # of objects in stack is always \( \geq 0 \), \( \Phi(D_i) \geq 0 = \Phi(D_0) \) for all \( i \).

<table>
<thead>
<tr>
<th>operation</th>
<th>actual cost</th>
<th>( \Delta \Phi )</th>
<th>amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>( s + 1 )</td>
<td>( s - 1 = 1 )</td>
<td>( 1 + 1 = 2 )</td>
</tr>
<tr>
<td>POP</td>
<td>( s - 1 )</td>
<td>( s - 1 = -1 )</td>
<td>( 1 - 1 = 0 )</td>
</tr>
<tr>
<td>MULTIPOP</td>
<td>( k' = \min(k, s) )</td>
<td>( s - k' = -k' )</td>
<td>( k' - k' = 0 )</td>
</tr>
</tbody>
</table>

Therefore, amortized cost of a sequence of \( n \) operations = \( O(n) \).

**Binary counter**

\( \Phi = b_i = \) # of 1’s after \( i \)th INCREMENT

Suppose \( i \)th operation resets \( t_i \) bits to 0.

\( c_i \leq t_i + 1 \) (resets \( t_i \) bits, sets \( \leq 1 \) bit to 1)

\[ \begin{align*}
\text{• If } b_i &= 0, \text{ the } i \text{th operation reset all } k \text{ bits and didn’t set one, so } \\
& b_i = t_i = k \Rightarrow b_i = b_{i-1} - t_i.
\end{align*} \]

\[ \begin{align*}
\text{• If } b_i &> 0, \text{ the } i \text{th operation reset } t_i \text{ bits, set one, so } \\
& b_i = b_{i-1} - t_i + 1.
\end{align*} \]

Either way, \( b_i \leq b_{i-1} - t_i + 1 \).

Therefore,

\[ \begin{align*}
\Delta \Phi(D_i) & \leq (b_{i-1} - t_i + 1) - b_{i-1} \\
& = 1 - t_i . \\
\end{align*} \]

\[ \begin{align*}
\tilde{c}_i &= c_i + \Delta \Phi(D_i) \\
& \leq (t_i + 1) + (1 - t_i) \\
& = 2 . \\
\end{align*} \]

If counter starts at 0, \( \Phi(D_0) = 0 \).

Therefore, amortized cost of \( n \) operations = \( O(n) \).

**Dynamic tables**

A nice use of amortized analysis.
Scenario

• Have a table—maybe a hash table.
• Don’t know in advance how many objects will be stored in it.
• When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
• When it gets sufficiently small, might want to reallocate with a smaller size.

Details of table organization not important.

Goals

1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

Load factor $\alpha = \frac{\text{num}}{\text{size}}$, where $\text{num} = \# \text{ items stored}, \text{size} = \text{allocated size}.$

If $\text{size} = 0$, then $\text{num} = 0$. Call $\alpha = 1$.

Never allow $\alpha > 1$.

Keep $\alpha > a$ constant fraction $\Rightarrow$ goal (2).

Table expansion

Consider only insertion.

• When the table becomes full, double its size and reinsert all existing items.
• Guarantees that $\alpha \geq 1/2$.
• Each time we actually insert an item into the table, it’s an elementary insertion.

TABLE-INSERT $(T, x)$

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

    then allocate table[T] with 1 slot
    size[T] $\leftarrow 1$

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

    then allocate new-table with $2 \cdot \text{size}[T]$ slots
    insert all items in table[T] into new-table $\triangleright \text{num}[T]$ elem insertions
    free table[T]
    table[T] $\leftarrow$ new-table
    size[T] $\leftarrow 2 \cdot \text{size}[T]$

insert $x$ into table[T] $\triangleright 1$ elem insertion

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

Initially, $\text{num}[T] = \text{size}[T] = 0$. 
**Running time:** Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.

$c_i =$ actual cost of $i$th operation

- If not full, $c_i = 1$.
- If full, have $i - 1$ items in the table at the start of the $i$th operation. Have to copy all $i - 1$ existing items, then insert $i$th item $\Rightarrow c_i = i$.

$n$ operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for $n$ operations.

Of course, we don’t always expand:

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

Total cost $= \sum_{i=1}^{n} c_i$  

$\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$  

$= n + \frac{2^{\lceil \lg n \rceil + 1} - 1}{2 - 1}$  

$< n + 2n$  

$= 3n$

Therefore, **aggregate analysis** says amortized cost per operation $= 3$.

**Accounting method**

Charge $3$ per insertion of $x$.

- $\$1$ pays for $x$’s insertion.
- $\$1$ pays for $x$ to be moved in the future.
- $\$1$ pays for some other item to be moved.

Suppose we’ve just expanded, $size = m$ before next expansion, $size = 2m$ after next expansion.

- Assume that the expansion used up all the credit, so that there’s no credit stored after the expansion.
- Will expand again after another $m$ insertions.
- Each insertion will put $\$1$ on one of the $m$ items that were in the table just after expansion and will put $\$1$ on the item inserted.
- Have $2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!
Potential method

\[ \Phi(T) = 2 \cdot \text{num}[T] - \text{size}[T] \]
- Initially, \(\text{num} = \text{size} = 0 \Rightarrow \Phi = 0\).
- Just after expansion, \(\text{size} = 2 \cdot \text{num} \Rightarrow \Phi = 0\).
- Just before expansion, \(\text{size} = \text{num} \Rightarrow \Phi = \text{num} \Rightarrow \) have enough potential to pay for moving all items.
- Need \(\Phi \geq 0\), always.

Always have

\[ \text{size} \geq \text{num} \geq \frac{1}{2} \cdot \text{size} \Rightarrow \\
2 \cdot \text{num} \geq \text{size} \Rightarrow \\
\Phi \geq 0. \]

\textbf{Amortized cost of \(i\)th operation:}

\[ \text{num}_i = \text{num} \text{ after } i \text{th operation} , \]
\[ \text{size}_i = \text{size} \text{ after } i \text{th operation} , \]
\[ \Phi_i = \Phi \text{ after } i \text{th operation} . \]

- If no expansion:
  \[ \text{size}_i = \text{size}_{i-1} , \]
  \[ \text{num}_i = \text{num}_{i-1} + 1 . \]
  \[ c_i = 1 . \]

Then we have

\[ \hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \]
\[ = 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \]
\[ = 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2(\text{num}_i - 1) - \text{size}_i) \]
\[ = 1 + 2 \]
\[ = 3 . \]

- If expansion:
  \[ \text{size}_i = 2 \cdot \text{size}_{i-1} , \]
  \[ \text{size}_{i-1} = \text{num}_{i-1} = \text{num}_i - 1 . \]
  \[ c_i = \text{num}_{i-1} + 1 = \text{num}_i . \]

Then we have

\[ \hat{c}_i = c_i + \Phi_i + \Phi_{i-1} \]
\[ = \text{num}_i + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \]
\[ = \text{num}_i + (2 \cdot \text{num}_i - 2(\text{num}_i - 1)) - (2(\text{num}_i - 1) - (\text{num}_i - 1)) \]
\[ = \text{num}_i + 2 - (\text{num}_i - 1) \]
\[ = 3 . \]
Expansion and contraction

When \( \alpha \) drops too low, contract the table.

- Allocate a new, smaller one.
- Copy all items.

Still want

- \( \alpha \) bounded from below by a constant,
- amortized cost per operation = \( O(1) \).

Measure cost in terms of elementary insertions and deletions.

"Obvious strategy":

- Double size when inserting into a full table (when \( \alpha = 1 \), so that after insertion \( \alpha \) would become > 1).
- Halve size when deletion would make table less than half full (when \( \alpha = 1/2 \), so that after deletion \( \alpha \) would become < 1/2).
- Then always have \( 1/2 \leq \alpha \leq 1 \).
- Suppose we fill table.
  - Then insert \( \Rightarrow \) double
  - 2 deletes \( \Rightarrow \) halve
  - 2 inserts \( \Rightarrow \) double
  - 2 deletes \( \Rightarrow \) halve

\ldots

Not performing enough operations after expansion or contraction to pay for the next one.

Simple solution:

- Double as before: when inserting with \( \alpha = 1 \) \( \Rightarrow \) after doubling, \( \alpha = 1/2 \).
- Halve size when deleting with \( \alpha = 1/4 \) \( \Rightarrow \) after halving, \( \alpha = 1/2 \).
- Thus, immediately after either expansion or contraction, have \( \alpha = 1/2 \).
- Always have \( 1/4 \leq \alpha \leq 1 \).
Intuition:

- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

\[ \Phi(T) = \begin{cases} 
2 \cdot \text{num}[T] - \text{size}[T] & \text{if } \alpha \geq 1/2, \\
\text{size}[T]/2 - \text{num}[T] & \text{if } \alpha < 1/2. 
\end{cases} \]

\( T \) empty \( \Rightarrow \) \( \Phi = 0 \).

\( \alpha \geq 1/2 \Rightarrow \text{num} \geq \frac{1}{2} \cdot \text{size} \Rightarrow 2 \cdot \text{num} \geq \text{size} \Rightarrow \Phi \geq 0. \)

\( \alpha < 1/2 \Rightarrow \text{num} < \frac{1}{2} \cdot \text{size} \Rightarrow \Phi \geq 0. \)

Intuition: \( \Phi \) measures how far from \( \alpha = 1/2 \) we are.

- \( \alpha = 1/2 \Rightarrow \Phi = 2 \cdot \text{num} - 2 \cdot \text{num} = 0. \)
- \( \alpha = 1 \Rightarrow \Phi = 2 \cdot \text{num} - \text{num} = \text{num}. \)
- \( \alpha = 1/4 \Rightarrow \Phi = \text{size}/2 - \text{num} = 4 \cdot \text{num}/2 - \text{num} = \text{num}. \)
- Therefore, when we double or halve, have enough potential to pay for moving all \( \text{num} \) items.
- Potential increases linearly between \( \alpha = 1/2 \) and \( \alpha = 1 \), and it also increases linearly between \( \alpha = 1/2 \) and \( \alpha = 1/4 \).
- Since \( \alpha \) has different distances to go to get to 1 or 1/4, starting from 1/2, rate of increase of \( \Phi \) differs.

- For \( \alpha \) to go from 1/2 to 1, \( \text{num} \) increases from \( \text{size}/2 \) to \( \text{size} \), for a total increase of \( \text{size}/2 \). \( \Phi \) increases from 0 to \( \text{size} \). Thus, \( \Phi \) needs to increase by 2 for each item inserted. That’s why there’s a coefficient of 2 on the \( \text{num}[T] \) term in the formula for \( \Phi \) when \( \alpha \geq 1/2 \).
- For \( \alpha \) to go from 1/2 to 1/4, \( \text{num} \) decreases from \( \text{size}/2 \) to \( \text{size}/4 \), for a total decrease of \( \text{size}/4 \). \( \Phi \) increases from 0 to \( \text{size}/4 \). Thus, \( \Phi \) needs to increase by 1 for each item deleted. That’s why there’s a coefficient of \(-1\) on the \( \text{num}[T] \) term in the formula for \( \Phi \) when \( \alpha < 1/2 \).

Amortized costs: more cases

- insert, delete
- \( \alpha \geq 1/2, \alpha < 1/2 \) (use \( \alpha_i \), since \( \alpha \) can vary a lot)
- \( \text{size} \) does/doesn’t change

Insert:

- \( \alpha_{i-1} \geq 1/2 \), same analysis as before \( \Rightarrow \hat{c}_i = 3. \)
- \( \alpha_{i-1} < 1/2 \Rightarrow \text{no expansion} \) (only occurs when \( \alpha_{i-1} = 1 \)).
• If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$:
  \[
  \tilde{c}_i = c_i + \Phi_i + \Phi_{i-1}
  = 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})
  = 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_i / 2 - (\text{num}_i - 1))
  = 0.
  \]

• If $\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2$:
  \[
  \tilde{c}_i = 1 + (2 \cdot \text{num}_i - \text{size}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})
  = 1 + (2(\text{num}_{i-1} + 1) - \text{size}_{i-1}) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})
  = 3 \cdot \text{num}_{i-1} - \frac{3}{2} \cdot \text{size}_{i-1} + 3
  < \frac{3}{2} \cdot \text{size}_{i-1} - \frac{3}{2} \cdot \text{size}_{i-1} + 3
  = 3.
  \]

Therefore, amortized cost of insert is $< 3$.

**Delete:**

• If $\alpha_{i-1} < 1/2$, then $\alpha_i < 1/2$.
  
  • If no contraction:
    \[
    \tilde{c}_i = 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})
    = 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_i / 2 - (\text{num}_i + 1))
    = 2.
    \]
  
  • If contraction:
    \[
    \tilde{c}_i = \lfloor \text{move + delete} \rfloor
    \begin{align*}
      \text{size}_i / 2 &= \text{size}_{i-1} / 4 = \text{num}_{i-1} = \text{num}_i + 1
      \end{align*}
    \]
    \[
    = (\text{num}_i + 1) + ((\text{num}_i + 1) - \text{num}_i) - ((2 \cdot \text{num}_i + 2) - (\text{num}_i + 1))
    = 1.
    \]

• If $\alpha_{i-1} \geq 1/2$, then no contraction.

  • If $\alpha_i \geq 1/2$:
    \[
    \tilde{c}_i = 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1})
    = 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_i + 2 - \text{size}_i)
    = -1.
    \]

  • If $\alpha_i < 1/2$, since $\alpha_{i-1} \geq 1/2$, have
    \[
    \text{num}_i = \text{num}_{i-1} - 1 \geq \frac{1}{2} \cdot \text{size}_{i-1} - 1 = \frac{1}{2} \cdot \text{size}_i - 1.
    \]
Thus,
\[
\hat{c}_i = 1 + \left( \frac{\text{size}_i}{2} - \text{num}_i \right) - \left( 2 \cdot \text{num}_{i-1} - \text{size}_{i-1} \right)
\]
\[
= 1 + \left( \frac{\text{size}_i}{2} - \text{num}_i \right) - \left( 2 \cdot \text{num}_i + 2 - \text{size}_i \right)
\]
\[
= -1 + \frac{3}{2} \cdot \text{size}_i - 3 \cdot \text{num}_i
\]
\[
\leq -1 + \frac{3}{2} \cdot \text{size}_i - 3 \left( \frac{1}{2} \cdot \text{size}_i - 1 \right)
\]
\[
= 2.
\]
Therefore, amortized cost of delete is \( \leq 2 \).