Amortized Analysis

交通大學 資訊工程系

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*.**
- **Three methods:**
  - ***Aggregate analysis*, *accounting method*, and *potential method*.**

Aggregate Analysis[1]

- **In aggregate analysis**
  - **For all *n*, a sequence of *n* operations takes worst-case time *T*(*n*) in total.**
  - **In the worst case, the average cost, or *amortized cost*, per operation is therefore *T*(*n*)/*n*.**
- **Stack operations**
  - **PUSH(*S*, *x*): *O*(1) each ⇒ *O*(*n*) for any sequence of *n* operations.**
  - **POP(*S*): *O*(1) each ⇒ *O*(*n*) for any sequence of *n* operations.**

Aggregate Analysis[2]

- **MULTIPOP*(S, k)***

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```

☞ **Running time of MULTIPOP:**

- **Linear in # of POP operations.**
- **Let each PUSH/POP cost 1.**
- **# of iterations of while loop is min*(s, k)*, where *s* = # of objects on stack.**
- **Therefore, total cost = min*(s, k)*.**

Aggregate Analysis[3]

- **Sequence of *n* PUSH, POP, MULTIPOP operations:**
  - **Worst-case cost of MULTIPOP is *O(n)*.**
  - **Have *n* operations.**
  - **Therefore, the worst-case cost of a 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)*.**

Aggregate Analysis[4]

- Average over the n operations ⇒ $O(1)$ per operation on average.

☞ **Incrementing a 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 x

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

- Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$.
- To increment, add 1 (mod 2^k):

Aggregate Analysis[5]

- Procedure INCREMENT

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

- Cost of INCREMENT = (# of bits flipped) .

Aggregate Analysis[6]

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Aggregate Analysis[7]

- **Running time**

Not every bit flips every time.

[Show costs from above.]

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	$\lfloor n/2 \rfloor$
2	1/4 the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

Aggregate Analysis[8]

- The total number of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

  - The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$.
  - The average cost of each operation, the amortized cost per operation, is $O(n)/n = O(1)$.

Accounting Method[1]

- Assign different charges to different operations.
  - Some are charged more than actual cost.
  - Some are charged less.
- ***Amortized cost*** = the amount we charge an operation.
  - 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.
- Differs from aggregate analysis:
  - In the accounting method, different operations can have different amortized costs.
  - In aggregate analysis, all operations have same amortized cost.

Accounting Method^2

- Need total credit never become 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 = actual cost of *i*th operation ,
  - $\hat{c}_i$ = amortized cost of *i*th operation .

Then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of *n* operations.

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

Accounting Method[3]

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

- Stack operations: 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.

Accounting Method[4]

- **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 ≥ 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[1]

- 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.

Potential Method^2

☞ ***Potential function*** Φ: $D_i \rightarrow$ R

Let D_i = data structure after ith operation ,
D_0 = initial data structure ,
c_i = actual cost of ith operation ,
$\widehat{c_i}$ = amortized cost of ith operation .

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

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .\end{aligned}$$

Potential Method[3]

$$
\begin{aligned}
\text{Total amortized cost} &= \sum_{i=1}^{n} \hat{c}_i \\
&= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\
&\quad \text{(telescoping sum: every term other than } D_0 \text{ and } D_n \\
&\qquad \text{is added once and subtracted once)} \\
&= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \ .
\end{aligned}
$$

- 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 .

Potential Method[4]

- Stack operations
  - Φ = # of objects in stack

    (= # of \$1 bills in accounting method)
  - D_0 = empty stack $\Rightarrow$ $\Phi(D_0) = 0$.
  - Since # of objects in stack is always ≥ 0, $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all i.

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1	$(s+1) - s = 1$ where s = # of objects initially	$1 + 1 = 2$
POP	1	$(s-1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

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

Potential Method[5]

- Binary counter
  - $\Phi(D_i) = b_i$ = # of 1's after ith INCREMENT.
  - Suppose ith operation resets t_i bits to 0.
  - $c_i \le t_i + 1$ (resets t_i bits, sets ≤ 1 bit to 1)
  - If $b_i = 0$, the ith operation reset all k bits and didn't set one, so $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i$.
  - If $b_i > 0$, the ith operation reset t_i bits, set one, so
    $b_i = b_{i-1} - t_i + 1$.
  - Either way, $b_i \le b_{i-1} - t_i + 1$.

Potential Method[6]

- Therefore, the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\le (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \ .\end{aligned}$$

- The amortized cost is therefore

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (t_i + 1) + (1 - t_i) \\ &= 2 \ .\end{aligned}$$

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

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

Potential Method[7]

- When the counter does not start at zero
  - The counter starts with b_0 1s, and after n INCREMENT operations it has b_n 1s, where $0 \le b_0, b_n \le k$. (k is the number of bits in the counter.)

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

  - We have $\hat{c}_i \le 2$ for all $1 \le i \le n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

Dynamic Tables[1]

☞ ***Dynamic table***

- We do not always know in advance how many objects some applications will store in a table.
- Dynamically expanding and contracting a table.
- How to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.
- Support operations TABLE-INSERT and TABLE-DELETE.

☞ ***Load factor $\alpha(T)$***

- The number of items stored in the table T divided by the size (number of slots) of the table.

Dynamic Tables[2]

- Define the *load factor of an empty table* to be *1*.
- If the load factor of a dynamic table is bounded below by a constant
  - The unused space in the table is never more than a constant fraction of the total amount of space.

☞ **Table expansion**

- A table is allocated as an array of slots.
- *Full table*: load factor = 1.
- Expand the table upon inserting an item into a full table.
- Allocate a new array for the larger table and then copy items from the old table into the new table.

Dynamic Tables[3]

- **Size of the new table**
  - ***Twice*** **as many slots as the old one.**
  - **If the only table operations are insertions, then the load factor of the table is always at least 1/2.**
- **Algorithm TABLE-INSERTION**
  - $T.table$ **is a pointer to the block of storage.**
  - $T.num$ **contains the number of items in the table.**
  - $T.size$ **gives the total number of slots in the table.**
  - **Initially the table is empty:** $T.num = T.size = 0$.
  - ***Elementary insertion*****: Lines 10.**
    - **Assigning a cost of 1.**

Dynamic Tables[4]

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2       allocate T.table with 1 slot
 3       T.size = 1
 4  if T.num == T.size
 5       allocate new-table with 2 · T.size slots
 6       insert all items in T.table into new-table
 7       free T.table
 8       T.table = new-table
 9       T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

Dynamic Tables[5]

- What is the cost c_i of the ith operation?
  - $c_i = 1$, if the current table has room for the new item.
  - $c_i = i$, if the current table is full and an expansion occurs.
  - If we perform n operations, the worst-case cost of an operation is $O(n)$.
- A not tight upper bound: $O(n^2)$.
- Aggregate analysis
  - The amortized cost of an operation is in fact $O(1)$.

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

Dynamic Tables[6]

- The total cost of n TABLE-INSERT operations

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \ ,\end{aligned}$$

- The amortized cost of a single operation is at most 3.

☞ **Accounting method**

- Each item pays for 3 elementary insertions:
  - Inserting itself into the current table.
  - Moving itself when the table expands.

Dynamic Tables[7]

- Moving another item that has already been moved once when the table expands.

- Charge 3 dollars for each insertion

☞ **Potential method**

- Defining a potential function Φ that is 0 immediately after an expansion.

$$\Phi(T) = 2 \cdot T.num - T.size$$

- Immediately after an expansion, $T.num = T.size/2$ and thus $\Phi(T) = 0$.
- Immediately before an expansion, $T.num = T.size$, thus $\Phi(T) = T.num$.

Dynamic Tables[8]

- $\Phi(T)$ is always nonnegative, since
  - The initial value of the potential is 0.
  - The table is always at least half full, $T.num \geq T.size/2$.
- If the ith TABLE_INSERT operation does not trigger an expansion, then $size_i = size_{i-1}$ and the amortized cost is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3 .
\end{aligned}$$

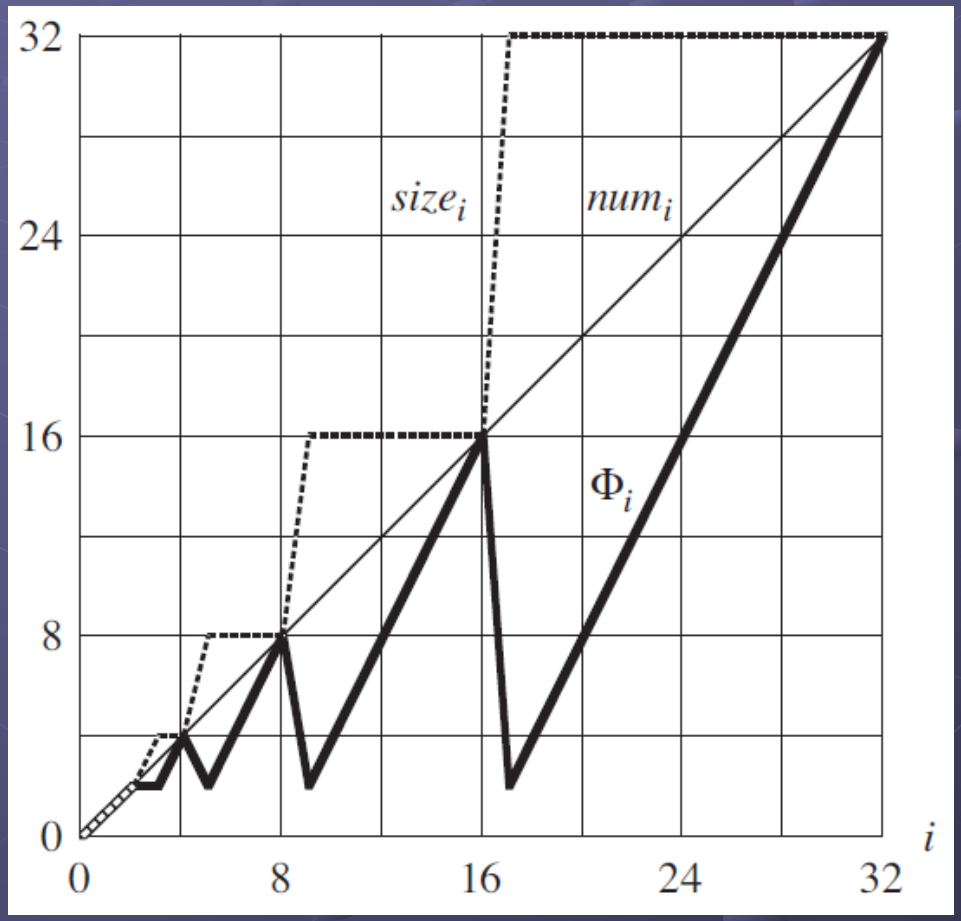
Dynamic Tables[9]

- If the ith operation does trigger an expansion, then $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, implies $size_i = 2 \cdot (num_i - 1)$. Thus

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}
$$

Dynamic Tables[10]

- The values of num_i, $size_i$, and Φ_i against i.



Dynamic Tables[11]

- Table contraction
  - When the number of items in the table drops too low
  - Allocate a new, smaller table and then copy the items from the old table into the new one.
  - Ideally, to preserve two properties:
    - The load factor of the dynamic table is bounded below by a positive constant.
    - The amortized cost of a table operation is bounded above by a constant.
  - Halve the size when deleting an item would cause the table to become less than half full.

Dynamic Tables[12]

- 1/2 size contraction is not good, consider a sequence of n operations:
  - The first $n/2$ operations are insertions.
  - The second $n/2$ operations are

    insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

  - The cost of each expansion and contraction is $\Theta(n)$.
  - The total cost of the n operations is $\Theta(n^2)$.
  - Improvement:
    - Contract when the table becomes less than 1/4 full.

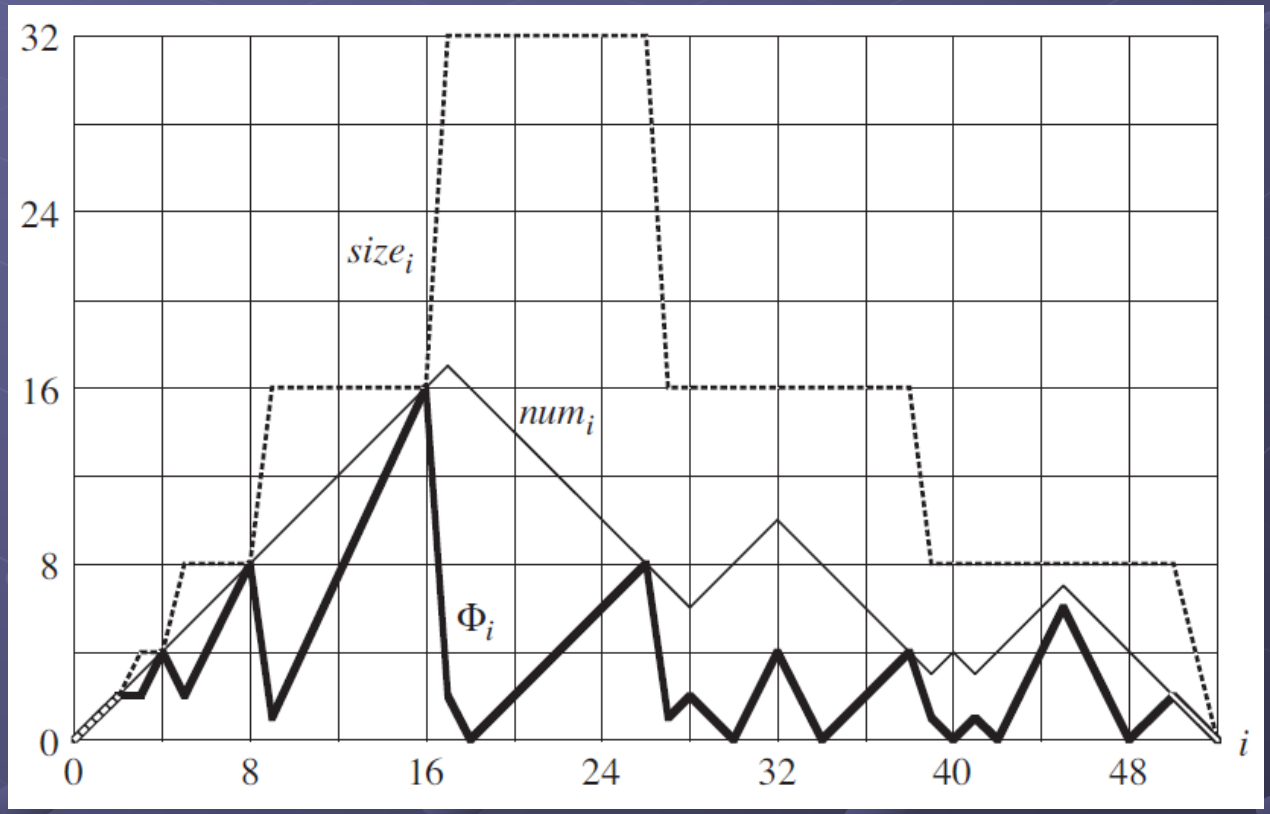
Dynamic Tables[13]

- Use potential method to analyze
  - Define a potential function Φ that is 0 immediately after an expansion or contraction.
  - Build as the load factor increases to 1 or decreases to 1/4.

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

Dynamic Tables[14]

- The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations.



Dynamic Tables[15]

- Initially, $num_0 = 0, size_0 = 0, \alpha_0 = 1$, and $\Phi_0 = 0$.
- Case 1: The ith operation is TABLE-INSERT.
  - If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$, then

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0 .
\end{aligned}$$

  - If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3
\end{aligned}$$

Dynamic Tables[16]

$$
\begin{aligned}
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3 .
\end{aligned}
$$

- Thus, the amortized cost of a TABLE-INSERT operation is at most 3.

☞ Case 2: The ith operation is TABLE-DELETE.

- If $\alpha_{i-1} < 1/2$ and it does not contract, then $size_i = size_{i-1}$

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\
&= 2 .
\end{aligned}
$$

Dynamic Tables[17]

- If $\alpha_{i-1} < 1/2$ and the i th operation does trigger a contraction, the actual cost $c_i = num_i + 1$. We have $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$,

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\
&= 1 .
\end{aligned}$$

- When $\alpha_{i-1} \geq 1/2$, the amortized cost is also bounded above by a constant.