

Topic 6:
Amortized Analysis



Amortized (分期償還) 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. Instead,
  - ***No probability is involved***.
  - We're talking about
    - ***Average performance of each operation in the worst case***.
    - *The time required to perform a sequence of data structure operations in average over all the operations performed.*

For all n, a sequence of n operations takes worst time $T(n)$ in total. The amortize cost of each operation is $T(n)/n$.



Outline

- Aggregate analysis
- The accounting method
- The potential method
- Dynamic tables

Aggregate Analysis

Stack Operation

- Stack operation
  - PUSH(*S*, *x*)
  - POP(S)
  - MULTIPOP(*S*, *k*)

```
MULTIPOP(S, k)
while S is not empty and k > 0
    POP(S)
    k = k − 1
```

top → 23
17
6
39
10
47
———
initial

top → 10
47
———
MULTIPOP(S,4)

———
MULTIPOP(S,7)



Stack Operation (Cont.)

- 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).***
- 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. → 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 $\pmod{2^k}$:

```
INCREMENT(A, k)
i = 0
while i < k and A[i] == 1
    A[i] = 0
    i = i + 1
if i < k
    A[i] = 1
```



Binary Counter (Cont.)

- Each call could flip k bits, so n INCREMENTs takes ***O(nk)*** time.
- **Observation**
  - Not every bit flips every time.

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

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



Binary Counter (Cont.)

- Analysis
  - n INCREMENTs costs *O(n)*.
  - Average cost per operation = *O(1)*.

$$
\begin{aligned}
\text{total \# of flips} &= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor \\
&< n \sum_{i=0}^{\infty} 1/2^i \\
&= n \left(\frac{1}{1-1/2} \right) \\
&= 2n \ .
\end{aligned}
$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$



The Accounting Method



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***.
- Differs 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,
  - We have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
  - Amortized cost would tell us *nothing*.



Accounting Method (Cont.)

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

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

$$\text{Total credit stored} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0\,.$$



Stack Operation

- ***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.
    - ***Total amortized cost = O(n)*** is an upper bound on total actual cost.

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



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.
  - We 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 $2.
  - For n operations, amortized cost = ***O(n)***.



The Potential Method



Potential Method

- Like the accounting method, but think of the credit as *potential (位能、勢能)* stored with the entire data structure.
  - Accounting method stores credit with specific objects.
  - Potential method stores potential in the data structure as a whole.
  - We can release potential to pay for future operations.
  - Most flexible of the amortized analysis methods.

Potential Method (Cont.)

$$
\begin{aligned}
\text{Let } D_i &= \text{data structure after } i\text{th operation ,} \\
D_0 &= \text{initial data structure ,} \\
c_i &= \text{actual cost of } i\text{th operation ,} \\
\hat{c}_i &= \text{amortized cost of } i\text{th operation .}
\end{aligned}
$$

Potential function $\Phi : D_i \to \mathbb{R}$

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

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



Potential Method (Cont.)

- 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 Operation

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

$$\begin{aligned}\Phi &= \text{\# of objects in stack} \\ (&= \text{\# of \$1 bills in accounting method})\end{aligned}$$

$D_0 =$ 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 c_i	$\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1})$	$\hat{c}_i$ 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	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

The 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.
- $c_i \le t_i + 1$ (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, $b_i \le b_{i-1} - t_i + 1$.
  - Therefore,

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

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

If counter starts at 0, $\Phi(D_0) = 0$. Therefore, amortized cost of n operations = O(n).

Dynamic Tables



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, we 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.



Dynamic Tables (Cont.)

- ***Goals***
  - O(1) amortized time per operation.
  - Unused space always ≤ constant fraction of allocated space.
- ***Load factor*** α = *num*/*size*, *where*
  - *num* = # items stored,
  - *size* = allocated size.
  - If *size* = 0, then *num* = 0. Call $\alpha = 1$.
  - Never allow $\alpha > 1$
  - Keep α > a constant fraction ⇒ 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 Expansion (Cont.)

Initially, $T.num = T.size = 0$.

TABLE-INSERT(T, x)

```
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size                                   // expand?
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table      // T.num elem insertions
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table                                // 1 elem insertion
T.num = T.num + 1
```



Running Time – Aggregate Analysis

- 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, we have *i-1* items in the table at the start of the *i*th operation.
    - We have to copy all *i-1* existing items, then insert *i*th item $\Rightarrow c_i = i$.

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

Not tight



Running Time - Aggregate Analysis (Cont.)

- Actual cost of ith operation (c_i):

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

- Total cost:

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

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.
  - It 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.
  - It will 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

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

- Initially, $num = size = 0 \Rightarrow \Phi = 0$.
- Just after expansion, $size = 2 \cdot num \Rightarrow \Phi = 0$.
- Just before expansion, $size = num \Rightarrow \Phi = num \Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.

  Always have

$$\begin{aligned} size \geq \quad & num && \geq size/2 \Rightarrow \\ & 2 \cdot num && \geq size \Rightarrow \\ & \Phi && \geq 0 . \end{aligned}$$

Potential Method (Cont.)

$$\begin{aligned} num_i &= num \text{ after } i \text{th operation} , \\ size_i &= size \text{ after } i \text{th operation} , \\ \Phi_i &= \Phi \text{ after } i \text{th operation} . \end{aligned}$$

- Amortized Cost of ***i*** th Operation $\hat{c}_i$
  - If no expansion:

$$\begin{aligned} size_i &= size_{i-1} , \\ num_i &= num_{i-1} + 1 , \\ c_i &= 1 . \end{aligned}$$

Then we have

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



Potential Method (Cont.)

$$
\begin{aligned}
num_i &= num \text{ after } i\text{th operation} ,\\
size_i &= size \text{ after } i\text{th operation} ,\\
\Phi_i &= \Phi \text{ after } i\text{th operation} .
\end{aligned}
$$

- If expansion:

$$
\begin{aligned}
size_i &= 2 \cdot size_{i-1} ,\\
size_{i-1} &= num_{i-1} = num_i - 1 ,\\
c_i &= num_{i-1} + 1 = num_i .
\end{aligned}
$$

Then we have

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

Φ_i

Potential Method (Cont.)

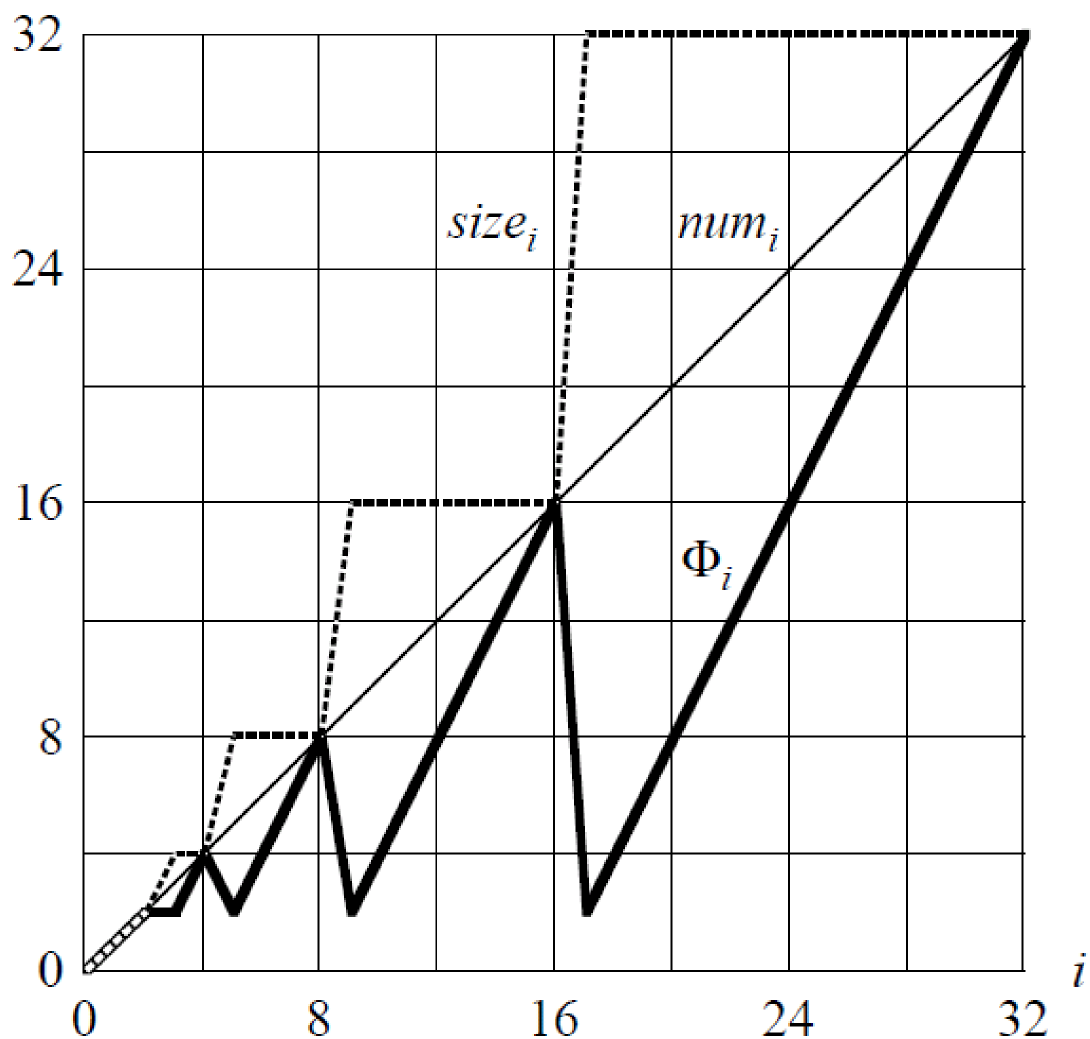




Table Expansion and Contraction

- When α drops too low, contract the table.
  - Allocate a new, smaller one.
  - Copy all items.
- Still want
  - α 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 α would become > 1.
- Halve size when deletion would make table less than half full
  - When α 1/2, so that after deletion α would become < ½.
- Then always have $½ \leq \alpha \leq 1$.

Suppose we fill table.

Then insert	$\Rightarrow$	double
2 deletes	$\Rightarrow$	halve
2 inserts	$\Rightarrow$	double
2 deletes	$\Rightarrow$	halve

…

Not performing enough operations after expansion or contraction to pay for the next one.
The cost of each expansion and contraction is $\Theta(n)$ and there are Q(n) operations. →The total cost of the n operations is $\Theta(n^2)$.



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 $1/4 \le \alpha \le 1$.

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

T empty $\Rightarrow \Phi = 0$.

$\alpha \ge 1/2 \Rightarrow num \ge size/2 \Rightarrow 2 \cdot num \ge size \Rightarrow \Phi \ge 0$.

$\alpha < 1/2 \Rightarrow num < size/2 \Rightarrow \Phi \ge 0$.

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.



Further Intuition

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

Φ measures how far from $\alpha = 1/2$ we are.

- $\alpha = 1/2 \Rightarrow \Phi = 2 \cdot num - 2 \cdot num = 0$.
- $\alpha = 1 \Rightarrow \Phi = 2 \cdot num - num = num$.
- $\alpha = 1/4 \Rightarrow \Phi = size/2 - num = 4 \cdot num/2 - num = num$.
- Therefore, when we double or halve, have enough potential to pay for moving all *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 α has different distances to go to get to 1 or $1/4$, starting from $1/2$, rate of increase of Φ differs.
  - For α to go from $1/2$ to 1, *num* increases from $size/2$ to *size*, for a total increase of $size/2$. Φ increases from 0 to *size*. Thus, Φ needs to increase by 2 for each item inserted. That's why there's a coefficient of 2 on the $T.num$ term in the formula for Φ when $\alpha \geq 1/2$.
  - For α to go from $1/2$ to $1/4$, *num* decreases from $size/2$ to $size/4$, for a total decrease of $size/4$. Φ increases from 0 to $size/4$. Thus, Φ needs to increase by 1 for each item deleted. That's why there's a coefficient of -1 on the $T.num$ term in the formula for Φ when $\alpha < 1/2$.



Amortized Costs: More Cases

Insert

- $\alpha_{i-1} \geq 1/2$, same analysis as before $\Rightarrow \hat{c}_i = 3$.
- $\alpha_{i-1} < 1/2 \Rightarrow$ *no expansion* (only occurs when $\alpha_{i-1} = 1$).
  - If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$:

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

- If $\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2$:

$$\begin{aligned}
\hat{c}_i &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&= 3 \cdot \alpha_{i-1} size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&< \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&= 3\,.
\end{aligned}$$

amortized cost of insert is < 3.

Amortized Costs: More Cases (Cont.)

Delete

- If $\alpha_{i-1} < 1/2$, then $\alpha_i < 1/2$.
  - If no contraction:

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

  - If contraction:

$$\begin{aligned}\hat{c}_i &= \underbrace{(num_i + 1)}_{\text{move + delete}} + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &\quad [size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1] \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1\,.\end{aligned}$$



Amortized Costs: More Cases (Cont.)

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

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

- If $\alpha_i < 1/2$, since $\alpha_{i-1} \geq 1/2$, have

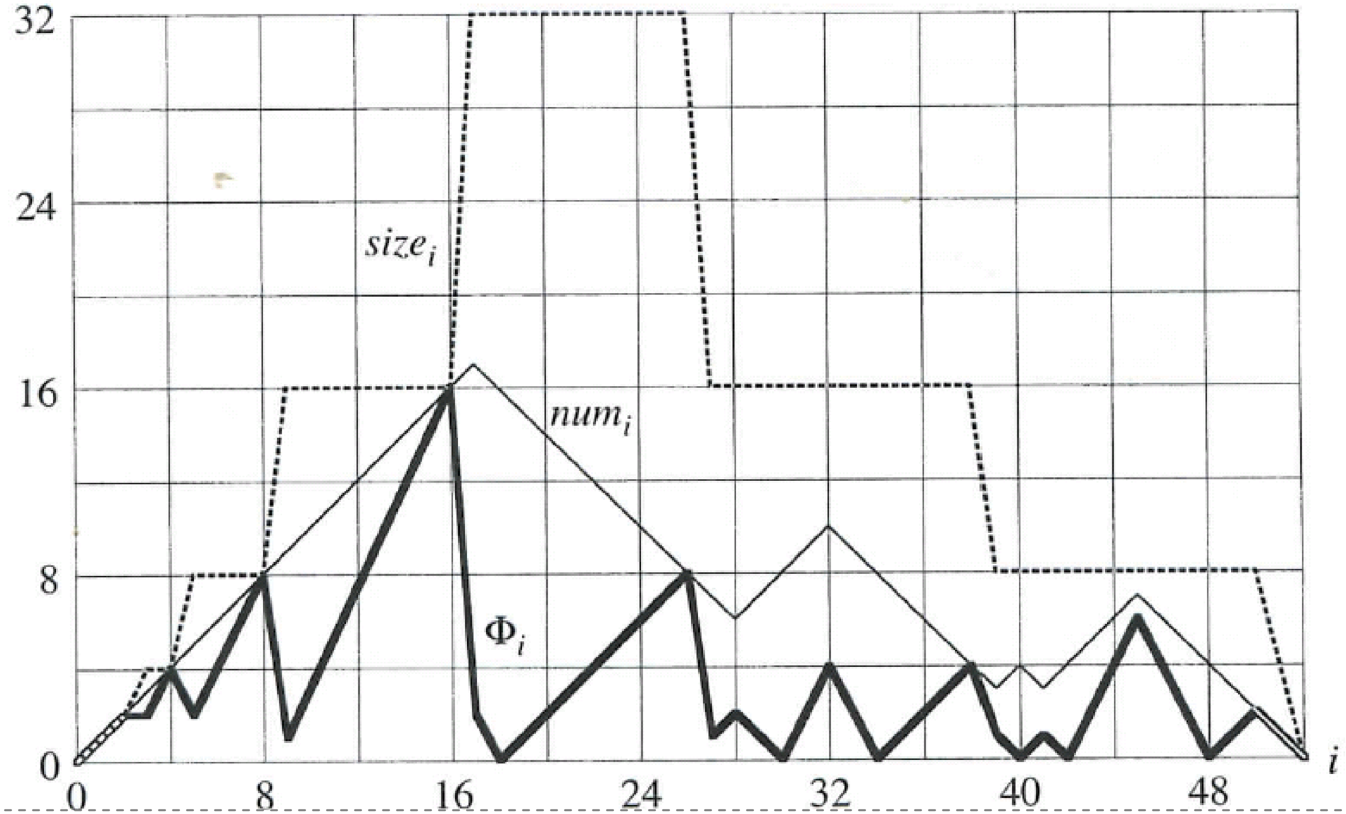
$$num_i = num_{i-1} - 1 \geq \frac{1}{2} \cdot size_{i-1} - 1 = \frac{1}{2} \cdot size_i - 1 \,.$$

Thus,

$$\begin{aligned} \hat{c}_i &= 1 + (size_i/2 - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (2 \cdot num_i + 2 - size_i) \\ &= -1 + \frac{3}{2} \cdot size_i - 3 \cdot num_i \\ &\leq -1 + \frac{3}{2} \cdot size_i - 3\left(\frac{1}{2} \cdot size_i - 1\right) \\ &= 2 \,. \end{aligned}$$

amortized cost of delete is ≤ 2.

Example



The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$