Algorithms
Chapter 17
Amortized Analysis

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

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

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.
- We show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total.

Example 1: Stack operations

- Stack operations: PUSH(S, x), POP(S), and MULTIPOP(S,k).
- PUSH(S, x): push object x onto stack S.
  - Each runs in $O(1)$ time.
  - A sequence of n PUSH operations takes $O(n)$ time.
- POP(S): pop the top of stack S and returns the popped object.
  - Each runs in $O(1)$ time.
  - A sequence of n POP operations takes $O(n)$ time.
- MULTIPOP(S,k)

```
1.  while not STACK-EMPTY(S) and k > 0
2.      POP(S)
3.      k ← k−1
```

Running time analysis$_{1/2}$

- Running time of MULTIPOP(S,*k*):
  - Let each PUSH/POP cost $O(1)$.
  - The number of iterations of while loop is min(s, k), where s = number of objects on stack.
  - Therefore, total cost = min(s, k).

- The running time of a sequence of n PUSH, POP, MULTIPOP operations?

- **Analysis(I):**
  - Worst-case cost of MULTIPOP is $O(n)$.
  - Have n operations.
  - Therefore, worst-case cost of sequence is $O(n^2)$.

Running time analysis$_{2/2}$

- **Analysis(II):**
  - Each object can be popped only once per time that it's pushed.
  - At most n objects are pushed into S.
  - Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPs, including those in MULTIPOP.
  - Therefore, total cost = $O(n)$.
  - Average cost of an operation = $O(1)$.

- Emphasize again, no probabilistic reasoning was involved.
  - Showed worst-case $O(n)$ cost for sequence.
  - Therefore, $O(1)$ per operation on average.

Example 2: Incrementing a binary counter

- *k*-bit binary counter $A[0\ ..\ k-1]$ of bits
  - $A[0]$ is the least significant bit.
  - $A[k-1]$ is the most significant bit.
  - Value of counter is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, counter value is 0.
- To add 1 (modulo 2^k), we use the following procedure.

INCREMENT(*A*)

```
i ← 0
while i < k and A[i] = 1
      A[i] ← 0
      i ← i + 1
if i < k
      then A[i] ← 1
```

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

Running time analysis

- The running time of a sequence of n INCREMENT operations ?
- **Analysis(I) :**
  - A single execution of INCREMENT takes time $O(k)$ in the worst case.
  - Have n operations.
  - Therefore, worst-case cost of sequence is $O(nK)$.
  - Average cost of an operation = $O(k)$.
- **Analysis(II):**
  - $A[0]$ flips every time, $A[1]$ flips only every other time, $A[2]$ flips only every fourth time, and so on.
  - Total number of flips is $T(n) = n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots$
    $\leq 2n$
  - Average cost of an operation = $O(1)$.

Outline

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

The accounting method$_{1/2}$

- Assign different charges to different operations.
  - Some are charged more than actual cost.
  - Some are charged less.
- The amount we charge an operation is called its **amortized cost**.
- When amortized cost > actual cost, store (amortized cost − actual cost) 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.

The accounting method$_{2/2}$

- 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 = actual cost of ith operation,
  $\hat{c}_i$ = amortized cost of ith operation.
- Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations.
- Total credit stored in the data structure = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$.

Example 1: Stack operations

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

- **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 ≥ 0.
  - Therefore, total amortized cost ≤ $2n$, is an upper bound on total actual cost.
  - Average cost of an operation = $O(1)$.

Example 2: Incrementing a 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 = $O(n)$.
  - Average cost of an operation = $O(1)$.

Outline

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

The Potential method$_{1/2}$

- Like the accounting method, but think of the credit as **potential** stored with the **entire data structure**.
  - Can release potential to pay for future operations.
  - Most flexible of the amortized analysis methods.
- Let D_i = data structure after ith operation,
  D_0 = initial data structure,
  c_i = actual cost of ith operation,
  $\hat{c}_i$ = amortized cost of ith operation.
- **Potential function** $\Phi : D_i \to R$
  - $\Phi(D_i)$ is the potential associated with data structure D_i.

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

The Potential method$_{2/2}$

- $$\text{Total amortized cost} = \sum_{i=1}^{n} \hat{c}_i$$
$$= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

- 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.

Example 1: Stack operations

- Φ = # of objects in stack.
- D_0 = empty stack $\Rightarrow \Phi(D_0) = 0$.
- Since # of objects in stack ≥ 0, $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all i.

operation	actual cost	$\Phi(D_i) - \Phi(D_{i-1})$	amortized cost
PUSH	1	$(s + 1) - s = 1$	$1 + 1 = 2$
POP	1	$(s - 1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

s = # of objects initially.

- Therefore, amortized cost of a sequence of n operations $= \sum_{i=1}^{n} \hat{c}_i = O(n)$.

Example 2: Incrementing a binary counter$_{1/2}$

- $\Phi = b_i$ = # of 1's after ith INCREMENT.
- D_0 = all bits are set to zero $\Rightarrow \Phi(D_0) = 0$.
- Suppose ith operation resets t_i bits to 0.
  - $c_i \leq t_i + 1$. (resets t_i bits, sets at most one 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 $b_i = b_{i-1} - t_i + 1$.
  - In either case, $b_i \leq b_{i-1} - t_i + 1$.

Example 2: Incrementing a binary counter$_{2/2}$

- Therefore, $\Phi(D_i)-\Phi(D_{i-1}) = b_i - b_{i-1}$
$$\leq (b_{i-1} - t_i + 1) - b_{i-1}$$
$$= 1 - t_i.$$

- The amortized cost is therefore $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
$$\leq (t_i + 1) + (1 - t_i)$$
$$= 2.$$

- Thus, amortized cost of *n* operations = $\sum_{i=1}^{n} \hat{c}_i$ = $O(n)$.

Outline

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

Dynamic tables$_{1/2}$

- **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.
  - Initially, *T* is a table of size 0.
  - Perform a sequence of *n* operations on *T*, each of which is either Insert or Delete.
- **Goals**
  - $O(1)$ amortized time per operation.
  - Unused space always ≤ constant fraction of allocated space.

Dynamic tables$_{2/2}$

- **Load factor $\alpha(T)$**
  - $\alpha(T)$ = num[T]/size[T]
    - num[T] = # items stored, size[T] = allocated size.
  - If size[T] = 0, then num[T] = 0. Define $\alpha = 1$.
  - Never allow $\alpha > 1$.
  - Keep $\alpha >$ a constant fraction.

Table expansion

```
TABLE-INSERT (T, x)
    if size[T] = 0
        then  allocate table[T] with 1 slot
              size[T] ← 1
    if num[T] = size[T]
        then  allocate new-table with 2 · size[T] slots
              insert all items in table[T] into new-table
              free table[T]
              table[T] ← new-table
              size[T] ← 2 · size[T]
    insert x into table[T]
    num[T] ← num[T] + 1
```

Notice : If only insertions are performed, the load factor of a table is always at least 1/2.

- When the table becomes full, double its size and reinsert all existing items.
- Each time we actually insert an item into the table, it's an **elementary insertion**.

Running time analysis

- The running time of a sequence of n TABLE-INSERT operations on an initially empty table ?
- **Analysis(I) :**
  - c_i = actual cost of ith operation.
  - If not full, $c_i = 1$.
  - If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item $\Rightarrow c_i = i$.
  - n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations.

Aggregate analysis

- **Analysis(II):**
  - Expand only when *i* – 1 is an exact power of 2.

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

  - $$\text{Totol cost} = \sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2-1} < n + 2n = 3n$$

  - 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 that the size of the table is *m* immediately after an 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 \$2*m* of credit by next expansion, when there are 2*m* items to move. Enough to pay for the expansion, with no credit left over!

Potential method$_{1/3}$

- $\Phi(T) = 2 \cdot \text{num}[T] - \text{size}[T]$
  - Initially, $\text{num}[T] = \text{size}[T] = 0 \Rightarrow \Phi(T) = 0$.
  - Just after expansion, $\text{size}[T] = 2 \cdot \text{num}[T] \Rightarrow \Phi(T) = 0$.
  - Just before expansion, $\text{size}[T] = \text{num}[T] \Rightarrow \Phi(T) = \text{num}[T]$

    $\Rightarrow$ have enough potential to pay for moving all items.
  - Always have $\Phi(T) \geq 0$.

    $\text{num}[T] \geq 1/2 \cdot \text{size}[T]$

    $\Rightarrow 2 \cdot \text{num}[T] \geq \text{size}[T]$

    $\Rightarrow \Phi(T) \geq 0$.

- **Amortized cost of ith operation:**

  $num_i = \text{num}[T]$ after ith operation,

  $size_i = \text{size}[T]$ after ith operation ,

  $\Phi_i = \Phi$ after ith operation.

Potential method$_{2/3}$

- If no expansion:

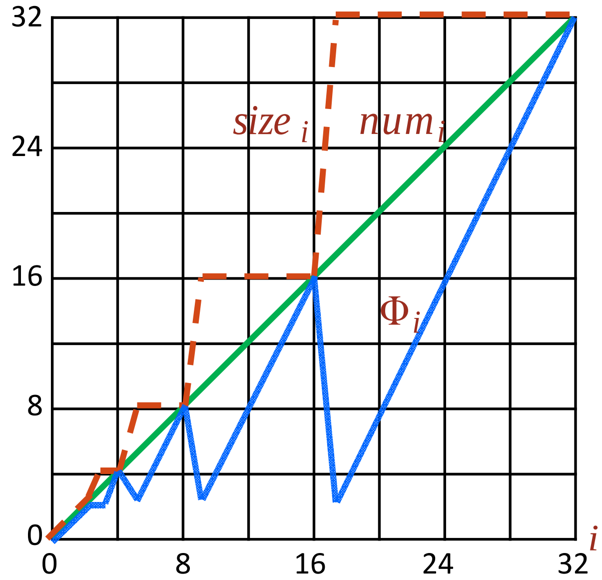
$$\begin{aligned} size_i &= size_{i\text{-}1}, \\ num_i &= num_{i\text{-}1} + 1, \\ c_i &= 1. \end{aligned}$$

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

- If expansion:

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

Potential method$_{3/3}$



- The above Figure plots the values of num_i, $size_i$, and Φ_i against i.
- Notice how the potential builds to pay for the expansion of the table.

Expansion and contraction$_{1/2}$

- When α drops too low, contract the table.
  - Allocate a new, smaller one.
  - Copy all items.
- Preserve two properties
  - load factor α bounded below by a positive constant, and
  - amortized cost per operation bounded above by a constant.
- **Obvious strategy**
  - Double size when inserting into a full table.
  - Halve size when deletion would make table less than half full.
  - Then always have $1/2 \le \alpha \le 1$.

Expansion and contraction$_{2/2}$

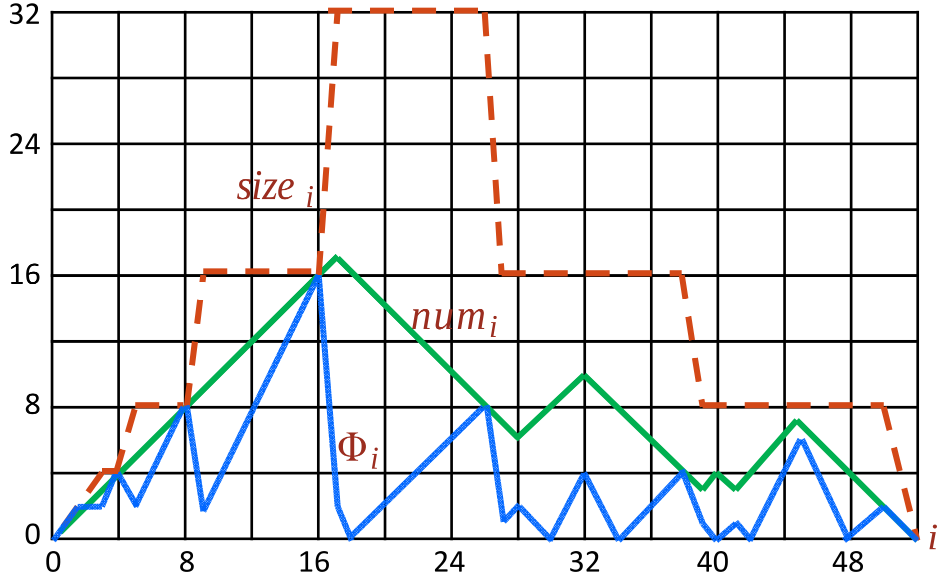
- **Consider the following scenario**
  - The first $n/2$ operations are insertions,
  - For the second $n/2$ operations, we perform the following sequence: insert, delete, delete, insert, insert, delete, delete, insert, insert,…

    double   halve   double   halve   double
  - The cost of each expansion and contraction is $\Theta(n)$.
  - The total cost of the n operations is $\Theta(n^2)$.
- **Simple solution:**
  - Double size when inserting into a full table.
  - Halve size when deleting from a 1/4 full table.
  - After either expansion or contraction, $\alpha = 1/2$.
  - Always have $1/4 \le \alpha \le 1$.

Some properties$_{1/3}$

- **Observation 1:**
  - Need to delete half the items before contraction.
  - Need to double number of items before expansion.
- Let $\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha \geq 1/2, \\ size[T]/2 - num[T] & \text{if } \alpha < 1/2. \end{cases}$
  - T empty $\Rightarrow \Phi = 0$.
  - $\alpha \geq 1/2 \Rightarrow$ num $\geq 1/2 \cdot$ size $\Rightarrow 2 \cdot$ num $\geq$ size $\Rightarrow \Phi \geq 0$.
  - $\alpha < 1/2 \Rightarrow$ num $< 1/2 \cdot$ size $\Rightarrow \Phi \geq 0$.

Some properties$_{2/3}$



- The potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ is an upper bound on the actual cost of the sequence.

Some properties$_{3/3}$

- **Observation 2:**
  - $\alpha = 1/2 \quad \Rightarrow \quad \Phi = 2 \cdot \text{num} - 2 \cdot \text{num} = 0.$
  - $\alpha = 1 \quad \Rightarrow \quad \Phi = 2 \cdot \text{num} - \text{num} = \text{num}.$
  - $\alpha = 1/4 \quad \Rightarrow \quad \Phi = \text{size}/2 - \text{num} = 4 \cdot \text{num}/2 - \text{num} = \text{num}.$
  - Therefore, when we double or halve, have enough potential to pay for moving all num items.

- Let c_i = actual cost of *i*th operation,
  $\hat{c}_i$ = amortized cost of *i*th operation,
  num_i = the number of items after the *i*th operation,
  $size_i$ = the size of the table after the *i*th operation,
  α_i = the load factor of the table after the *i*th operation,
  Φ_i = the potential after the *i*th operation.

Analysis: insert operation$_{1/2}$

- Case 1: $\alpha_{i-1} \geq 1/2$
  - The same analysis as before.
  - The amortized cost $\hat{c}_i = 3$.

- Case 2: $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$ (no expansion)

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

Analysis: insert operation$_{2/2}$

- Case 3: **$\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2$** (no expansion)

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

- Therefore, amortized cost of insert is at most 3.

Analysis: delete operation$_{1/3}$

- Case 1: $\alpha_{i-1} < 1/2$
  - This implies $\alpha_i < 1/2$.
  - 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:

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

Analysis: delete operation$_{1/2}$

- Case 2: $\alpha_{i-1} \geq 1/2$ (No contraction happens)
  - Case 2a: $\alpha_i \geq 1/2$:

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

  - Case 2b: $\alpha_i < 1/2$:

    Since $\alpha_{i-1} \geq 1/2$, we have

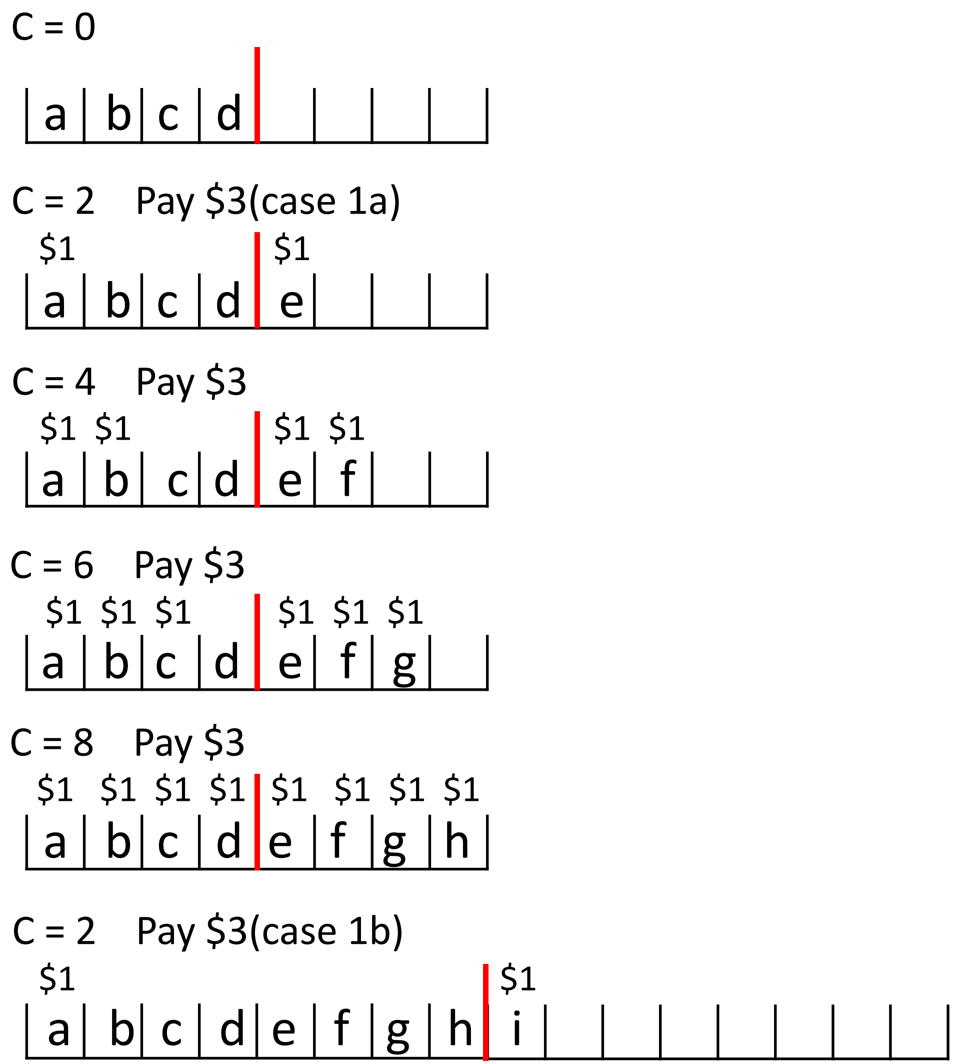
    $num_i = num_{i-1} - 1 \geq ½ \cdot size_{i-1} - 1 = ½ \cdot size_i - 1$.

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

Summary

- Therefore, amortized cost of delete is at most 2.
- The amortized cost of each operation is bounded above by a constant .
- The actual time for any sequence of n operations on a dynamic table is $O(n)$.

Insertion Only ($\alpha \geq 1/2$)



Delection Only ($\alpha \leq 1/2$)