Chapter 17: Amortized Analysis

With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.

Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

- **Amortized analysis:**
  - Guarantees the avg. performance of each operation in the worst case.
  - Aggregate method
  - Accounting method
  - Potential method
- **Aggregate method**
  - A sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the amortized cost (average cost) per operation is $T(n)/n$

- Eg.1 [Stack operation]

  PUSH(S,x): pushes object x onto stack S

  POP(S): pops the top of stack S and return the popped object

  Each runs in O(1) time (假設只需一單位時間)

  故 n 個 PUSH 和 POP operations 需 $\Theta(n)$ time

  考慮 MULTIPOP(S, k): pops the top k objects of stack S

```
MULTIPOP(S, k)
    While not STACK-EMPTY(S) and $k \neq 0$
        do { POP(S)
             $k \leftarrow k-1$ }
```

  故執行一 MULTIPOP 需 min{s, k} 個步驟

- 問題: 執行上述 PUSH, POP, MULTIPOP n次

  假設 initial stack 為 empty

  試問每一 operation 在 worst-case 之 amortized cost?

The worst-case cost of a MULTIPOP operation in the sequence is O(n), since the stack size is at most n. The worst-case time of any stack operation is therefore O(n), and hence a sequence of n operations costs $O(n^2)$, since we may have O(n) MULTIPOP operations costing O(n) each. (not tight)

- Each object can be popped at most once for each time it is pushed.
- 上述問題只需 O(n) time

  O(n)/n = O(1)   每個 operation 平均所需的時間

- Eg.2 [Incrementing a binary counter]

  A k-bit binary counter, counts upward from 0

  A[k-1] A[k-2]…A[1] A[0]

  highest order bit ↗ ↖ lowest order bit

```
Increment(A)
  { i ← 0;
    while i<A.length and A[i]=1
        do A[i] ← 0;
           i ← i+1;
    if i<A.length
       then A[i] ← 1;
  }
```

A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	1
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	1
0	0	0	0	1	0	1	0
0	0	0	0	1	0	1	1
0	0	0	0	1	1	0	0

- 問題: 執行 n 次(依序) Increment operations.
  在 worst-case 每一operation 之 amortized cost 為何?

A single execution of INCREMENT takes time Θ(k) in the worst case, in which array A contains all 1s. Thus, a sequence of n INCREMENT operations on an initially zero counter takes time O(nk) in the worst case. (not tight)

- A[0] 每 1 次改變一次
  A[1] 每 2 次改變一次
  A[i] 每 2^i 次改變一次

故 amortized cost of each operation
為 O(n)/n = O(1)

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

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

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 17.1 starts with aggregate analysis, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 17.2 covers the accounting method, in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

- **Accounting method:**
  - Assign differing ***charges*** to different operations

    ↖close to actual charge (amortized cost)

    If the amortized cost > actual cost, the difference is treated as ***credit***

    (在aggregate method所有operation有相同的amortized cost)

  - 選擇(定義)每一operation之適當 amortized cost 需注意:

    1. 若要展示每一 op 在 worst-case 之平均 cost小, 則一串 operations 之 total amortized cost必須 ≥ total actual cost (as an upper bound)

    $$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

    2. Total credit 必須是非負

- Eg. 1 [Stack operation]

  Amortized cost: PUSH 2 , POP 0 , MULTIPOP 0

  Actual cost: PUSH 1 , POP 1 , MULTIPOP min(k,S)

  在 PUSH 之 amortized cost 中預支了 POP 時所需之 cost

  故在 n 個運作之後, total amortized cost = O(n)

- Eg. 2 [Increment a binary counter]

  Set a bit to 1: amortized cost 2

  Reset a bit to 0: amortized cost 0

  在 Increment procedure 中每次最多只 set 一個 bit 為 1

  故每次最多 charge 2,

  執行 n 次 Increment 其 total amortized cost = O(n)

Potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with data structure D_i. The ***amortized cost*** $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \qquad (17.2)$$

- **Potential method**
  - D_0 : initial data structure

    c_i : actual cost of the ith operation

    D_i : data structure after applying the ith operation to D_{i-1}

    Φ : potential function

    $\Phi(D_i)$ is a real number

    $\hat{c}_i$: the amortized cost of the ith operation

    w.r.t potential function is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

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

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

若 $\Phi(D_n) \geq \Phi(D_0)$, 則 total amortized cost 為 total actual cost 之 upper bound

- Eg. 1 [Stack operation]

  定義 potential function Φ: stack size

  $\Phi(D_0)=0$, 亦知 $\Phi(D_i) \geq 0$

  i th op.

  PUSH: $$\boxed{\begin{aligned}&\Phi(D_i)-\Phi(D_{i-1})=1\\ &\hat{c}_i = c_i + \Phi(D_i)-\Phi(D_{i-1})\\ &\quad = c_i + 1 = 2\end{aligned}}$$

  MULTIPOP(S, k): 令 k'=min{k,S} $$\boxed{\begin{aligned}&\Phi(D_i)-\Phi(D_{i-1})=-k'\\ &\hat{c}_i = c_i + \Phi(D_i)-\Phi(D_{i-1})\\ &\quad = k'-k' = 0\end{aligned}}$$

  POP: 同理 $\boxed{\hat{c}_i = 0}$

Since we have already argue

that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

- Eg. 2 [Incrementing a binary counter]
  對 counter 而言其 potential function 值為在 i-th op 後被設定為 1 的 bit 個數
  b_i = # of 1's in the counter after the i-th operation
- Suppose that the i-th Increment op. resets t_i bits
  Actual cost $c_i \le t_i+1$
- # of 1's in the counter after the i-th op.:
  if $b_i=0$, then $b_{i-1}= t_i=k$ (all k bits were reset)
  if $b_i>0$, then $b_i = b_{i-1} - t_i + 1$

$$\Phi(D_i)-\Phi(D_{i-1}) \le (b_{i-1}-t_i+1)-b_{i-1}$$
$$=1-t_i$$
$$\hat{c}_i = c_i + \Phi(D_i)-\Phi(D_{i-1})$$
$$\le (t_i+1)+(1-t_i)=2$$

$$\boxed{\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n}\hat{c}_i - \Phi(D_n)+\Phi(D_0) \\ &= 2n - b_n + b_0\end{aligned}}$$

Dynamic tables

- 考慮某一問題，在其過程中，我們要維持一個dynamic table並要作a sequence of n operations: insert及deletion

- 假設每insert一個item之cost是O(1)
- 假設每delete一個item之cost是O(1)
- 假設每move一個item之cost是O(1)

- 名詞介紹: load factor $\alpha(T)$ of a table T=(# of items stored in the table)/(size of the table)

Dynamic tables:

Operations: **Table-Delete, Table-Insert**

Load factor: $\alpha(T)$ **= (number of items)/(table size)**

Define the load factor of an empty Table as 1.

Table T:

i	**100**	**11**	**7**			

$\alpha(T) = ?$

問題敘述

- 假設目前reserve一個固定size之table T(例如size k)，當作了一陣insert、delete的動作後，可能table T已經裝滿item，若下一個動作是insert，為避免error發生，假設我們的處理方式是expand the table by allocating a new table。

- Ex: 我們allocate a new table而size是原來的2倍，並假設我們還需付出代價，要將舊table的item一個一個move到new table

- 在另一方面，因為delete operation也穿插其間，有時會使得table的load factor降到某一程度以下，例如α(T)<1/4。我們就要釋放出來一些space

- 例如目前之table其size是k，而α(T)<1/4，則重新allocate一個new table，其size降為k/2，叫construct the table

- 同樣這樣作要付出代價，此次之delete operation它的cost是1+(目前存放在table中item數目)

問題

- 在上述情況下，作a sequence of n operations: insert & delete其total cost是多少? 平均一次operation其cost是多少?

- 以上問題當然可以考慮得更general一點，例如load factor α(T)限制在β≤α(T)≤α。一旦α(T)超過α，就要expand the table； α(T)低過β，就要construct the table，或考慮其他operation(例如search)，其他方式expand或construct the table

- 但是單只就前述之情況就已經有點複雜，不容易分析其total cost之tight bound。所以先考慮一些簡單情況作為初步探討

- 先看一個最簡單的情況: 只有insert operation，load factor不限，table滿了才double其size
- Initial: table size=0，第一次insert後table size=1
- 作insert時，一旦發現是full table就立即將size "double"
- 看課本464頁，TABLE-INSERT(T,x)
- 鬆散粗略的估計: 令c_i=作第i次operation之cost，若table尚未滿→ $c_i=1$；若full table→ $c_i=i$

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;   ← Expansion
        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;}
```

- Worst case: c_i=i (i+i-1)
- 在作了n次operations，table cost $\sum c_i=O(1+2+\ldots+n)$
- 因此table cost $O(n^2)$ worst case
- Each operation O(n) worst case
- Too coarse, let’s get into more details in order to use amortized analysis.
- Calculate space_b (s_b) and cost for each i.

Aggregate analysis

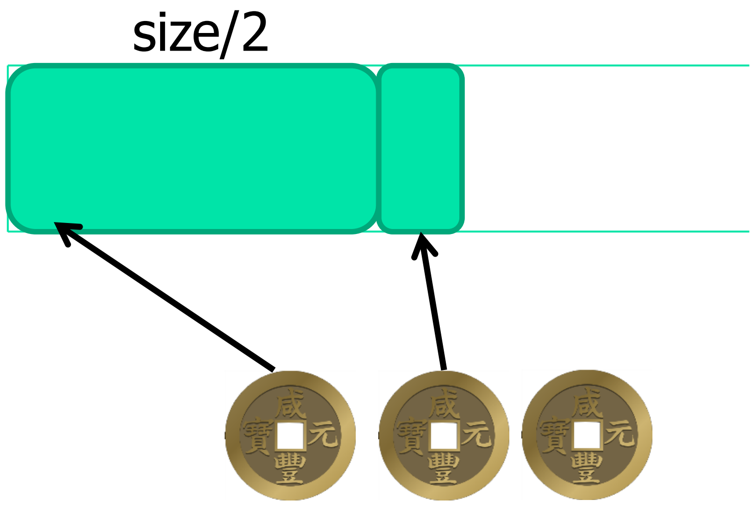
- $c_i = \begin{cases} i & \text{, if i-1 is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$
- $\sum_{i=1}^{n} c_i \leq \mathrm{n} + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + 2n = 3n$
- 所以n個operation的cost=3n=O(n)
- 平均每個operation的cost=3=O(1)

i	s_b	c
0	1	0
1	1	1
2	2	2
3	4	3
4	4	1
5	8	5
6	8	1
7	8	1
8	8	1
9	16	9

Accounting method

我們設計每個table-insert的amortized cost是3, 假設現在table剛expand完, size=m, number=m/2, credit=0

Claim: insert的時候付3塊錢



一塊錢把最新的item insert到table的時候花掉

一塊錢給之前已經在裡面的其中一個item (準備當之後滿了需要移動的時候可以花掉)

一塊錢給自己 (準備當之後滿了需要移動的時候可以花掉)

當全滿的時候, 則每一個item都有預存了一塊錢可以拿來搬移(insert到新的table)

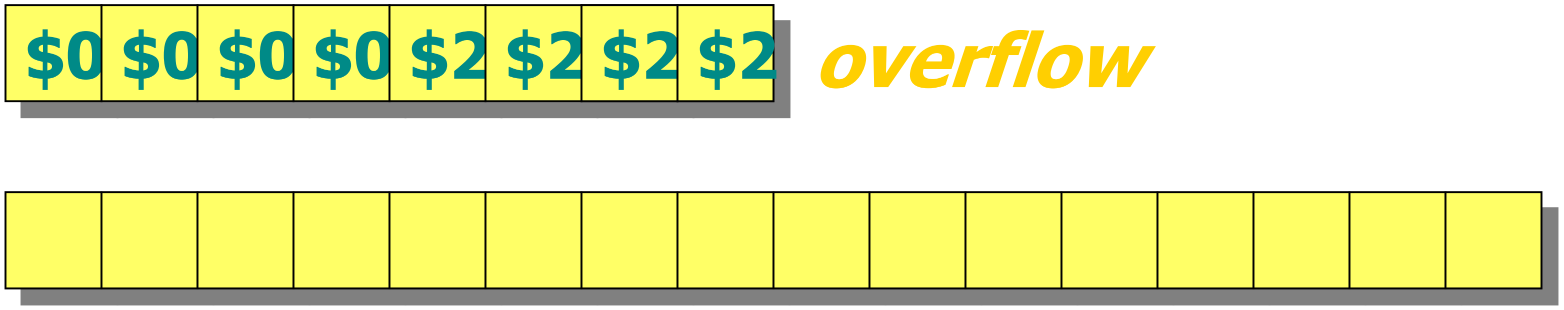
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- **\$1 pays for the immediate insertion.**
- **\$2 is stored for later table doubling.**

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:



Define potential function: $\Phi(T) = 2num[T] - size[T]$

function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential.

Immediately after an expansion, we have $T.num = T.size/2$, and thus $\Phi(T) = 0$, as desired. Immediately before an expansion, we have $T.num = T.size$, and thus $\Phi(T) = T.num$, as desired. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

Define potential function: $\Phi(T) = 2num[T] - size[T]$

function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential.

No expansion after the ith op: $(\mathrm{size_i} = \mathrm{size_{i-1}})$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$
$$= 1 + (2num_i - size_i) - (2(num_i - 1) - size_{i-1})$$
$$= 3$$

With expansion after the ith op: $(\mathrm{size_i}/2 = \mathrm{size_{i-1}} = \mathrm{num_i} - 1)$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$
$$= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$
$$= num_i + (2num_i - (2num_i - 2)) - (2(num_i - 1) - (num_i - 1))$$
$$= num_i + 2 - (num_i - 1)$$
$$= 3$$

Table expansion and contraction:

I, D, I, I, D, D, I, I, I………. : A sequence of n operations.

To implement a TABLE-DELETE operation, We want to preserve two properties:

. Load factor is bounded below by a constant.

. The amortized cost of a table op is bounded above by a const.

Strategy:

.Double the table size when an item is inserted into a full table.

.Halve the table size when a deletion causes the table to become less than **¼** full.

Define potential function:

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

If the i-th operation is Table-Insert:

If the load factor >= ½, then the analysis is the same as before.

$$(1)\quad if\quad \alpha_{i-1} < 1/2\ and\ \alpha_i < 1/2:$$

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

Define potential function:

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

$$(2)\quad if\quad \alpha_{i-1} < 1/2\ \ but\ \alpha_i \geq 1/2:$$

$$\begin{aligned}
\hat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1}+1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3num_{i-1} - (3/2)size_{i-1} + 3 \\
&= 3\alpha_{i-1}size_{i-1} - (3/2)size_{i-1} + 3 \\
&< 3
\end{aligned}$$

Define potential function:

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

If the i-th operation is Table-Delete:

$$(1)\quad if\quad \alpha_{i-1} < 1/2: \qquad (without\ contraction)$$

$$\begin{aligned} \overset{\wedge}{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 \end{aligned}$$

Define potential function:

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

$$(2)\quad if\quad \alpha_{i-1} < 1/2: \qquad (with\ \ contraction)$$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$

$$= (num_i + 1) + (size_i / 2 - num_i) - (size_{i-1} / 2 - num_{i-1})$$

$$= (num_i + 1) + ((num_i + 1) - num_i) - (2(num_i + 1) - (num_i + 1))$$

$$= 1$$

$$\because size_i / 2 = size_{i-1} / 4 = num_i + 1$$

$$(3)\quad if\quad \alpha_{i-1} \geq 1/2:$$

Exercise 17.4-2

HW 6 (6/4)
17.1-3, 17.2-2, 17.3-2, 17.4-2