Accounting method

- تخصیص charge های متفاوت به operation های متفاوت
- برخی بیشتر از هزینه واقعی شارژ می‌شوند.
- برخی کمتر

Amortized cost = amount we charge

هنگامی که amortized cost > actual cost، تفاوت را به‌عنوان credit بر روی DS و روی specific objects ذخیره کن.

از این credit بعداً برای operation هایی که actual cost > amortized cost استفاده می‌شود

تفاوت این روش از روش aggregate analysis :

- در accounting method، operation های مختلف هزینه‌های مختلفی دارند.
- در aggregate analysis، همه operation ها هزینه‌ی یکسانی دارند.

از credit استفاده می‌شود تا در هر زمانی منفی نشود.

- می‌خواهیم در هر زمانی دنباله‌ای از operation ها انجام شود که amortized cost یک upperbound بر روی actual cost باشد.
- اگر credit منفی شود Amortized cost هیچ معنوی را نخواهد رساند.

c_i = actual cost of ith operation

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

و نیاز است که :

برای هر دنباله n تایی از n تا operation داشته باشیم که:

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

مجموع credit های ذخیره شده $= \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$

Staek

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

روش کار: هنگامی که یک objeet را PUSH می‌کنیم، 2$ می‌پردازیم.

- 1$ برای انجام عمل PUSH
- 1$ پیش‌پرداخت وقتی که آن objeet بخواهد توسط POP یا MULTIPOP، pop شود
- هر object در stack، 1$ روی خودش دارد که credit است، پس credit هیچ وقت صفر نمی‌شود.
- بنابراین total amortized cost $= O(n)$ که یک حد بالا برای actual cost است.

برای تبدیل هر بیت به 1 ، 2$ شارژ می دهیم.

- 1$ برای تبدیل خود بیت به 1.
- 1$ پیش پرداخت برای زمانی که 1 را غلیب بک کردیم به 0.
- بروی هر رقم 1 روی counter 1$ داریم.
- بنابراین، credit منفی نمی شود.

هزینه سرشمن برای INCREMENT :

- هزینه reset شدن بیت به 0 با credit پرداخت شده است.
- حداکثر 1 بیت به 1 ست می شود.
- بنابراین amortized cost ≤ 2$
- برای n تا operation ، amortized cost = O(n)

Potential method

این روش شبیه روش accounting method است ولی به جای آنکه credit بروی هر object ذخیره شود. credit به عنوان potential برای کل DS ذخیره می شود.

accounting method_ ، credit ها را به همراه object خاصی ذخیره می کرد.

Potential method_ ، Potential ها را در ساختمان داده ذخیره می کند.

- می توان از Potential برای پرداخت operation های آتی استفاده کرد.

• نسبت به سایر روش‌های سرشکن انعطاف بیشتری دارد.

D_i = دیتا استراکچر بعد از i امین عملیات

D_0 = دیتا استراکچر اولیه

c_i = هزینه واقعی i امین عملیات

$\hat{c}_i$ = هزینه سرشکن برای i امین عملیات

Potential function $\phi : D_i \rightarrow R$

$\phi(D_i)$ تخصیص یافته به دیتا استراکچر D_i potential

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

$$= c_i + \underbrace{\Delta\phi(D_i)}$$

میزان افزایش potential بر اساس i امین operation

$$\text{Total amortized cost} = \sum_{i=1}^{n} \hat{c}_i$$

$$= \sum_{i=1}^{n} \left(c_i + \phi(D_i) - \phi(D_{i-1})\right)$$

مجموع تلسکوپی

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

به شرط آنکه $\phi(D_i) \geq \phi(D_0)$ برای تمام i ها، آن گاه amortized cost می‌تواند یک upper bound برای actual cost باشد.

In practice: $\phi(D_0) = 0$, $\phi(D_i) \geq 0$ for all i

stack

ϕ = # of object in stack

(= # of \$1 bills in accounting method)

D_0 = empty stack $\Rightarrow \phi(D_0) = 0$

همیشه تعداد object ها در stack بزرگتر یا مساوی 0 است؛

$\phi(D_i) \geq 0 = \phi(D_0)$

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

بنابراین هزینه سرشکن برای n تا عملیات برابر $O(n)$ است.

Binary Counter

$\phi = b_i = \#$ of 1's after ith INCREMENT.

فرض کنید که در این عملیات i امین بیت را به صفر برمی‌گرداند.

$c_i \leq t_i + 1$ (reset t_i bits, sets ≤ 1 bit to 1)

- اگر $b_i = 0$ ، در این عملیات تمام k بیت را reset می‌کند و set کردن به یک وجود ندارد

بنابراین، $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i$

- اگر $b_i > 0$ ، در این عملیات t_i بیت را reset می‌کند و یکی را به یک set می‌کند

$$b_i = b_{i-1} - t_i + 1$$

- به هر حال:

$$b_i \leq b_{i-1} - t_i + 1$$

- بنابراین:

$$\Delta\phi(D_i) \leq (b_{i-1} - t_i + 1) - b_{i-1}$$

$$= 1 - t_i$$

$$\hat{c}_i = c_i + \Delta\phi(D_i)$$

$$\leq (t_i + 1) + (1 - t_i)$$

$$= 2$$

اگر شمارنده از صفر شروع کند.

$\phi(D_0) = 0$

$\Rightarrow$ amortized cost of n operations $= O(n)$

Dynamic tables:

یک استفاده خوب از تحلیل سرشکن

سناریو:

- یک جدول داریم - شاید که یک hash table باشد.
- نمی دانیم از قبل که چه تعدادی object در آن ذخیره شده است.
- هرگاه که پر شود باید به یک سایز بزرگتر تبدیل شود (reallocate) و تمام عناصر آن به جدول بزرگتر جدید کپی شود.
- وقتی به قدر کافی خالی و کوچک شود ممکن است به یک جدول با سایز کوچکتر reallocate شود.

جزئیات سازماندهی این جدول مهم نیست

اهداف:

زمان سرشکن هر operation برابر $O(1)$ باشد.

فضای آزاد بدون استفاده کوچکتر یا مساوی یک کسر ثابت از کل فضای تخصیص یافته باشد

$$\text{Load factor} = \alpha = \frac{num}{size}$$

num = # items stored

size = allocated size

اگر size=0، آنگاه num=0 ← call $\alpha = 1$

هیچگاه $\alpha > 1$ نمی‌شود، همیشه $\alpha >$ a constant fraction

Table expansion

در نظر بگیرید که فقط ورودی داریم

- وقتی که جدول پر می‌شود، سایز آن را دو برابر کن و تمام item ها را مجدداً وارد کن
- تضمین می‌شود که $\alpha \geq \frac{1}{2}$
- هر زمان یک item جدید به جدول اضافه می‌کنیم یک elementary insertion است.

```
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

مقدمتاً: num[T] = size[T] = 0

زمان اجرا به ازای هر elementary insertion یک واحد شارژ می‌کنیم. فقط elementary insertion را می‌شماریم چون سایر هزینه‌های دیگر به ازای هر call ثابت است.

c_i = actual cost of ith operation

- اگر پر نیست، $c_i = 1$
- اگر پر است، $i-1$ آیتم داریم در جدول در ابتدای اجرای عملیات i ام که همه‌ی $i-1$ تا باید کپی و i امین insert شود. $c_i = i$

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

همیشه expand می‌کنیم:

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

$$\text{Total cost} = \sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1}$$

$$< n + 2n = 3n$$

بنابراین تحلیل aggregate analy. می‌گوید که amortized cost برابر 3 است.