

# T5. Análise Amortizada de Algoritmos

Uma nova ferramenta de análise, que permite estudar o tempo necessário para se efectuar uma **sequência de operações sobre uma estrutura de dados**.

- Frequentemente, ao longo da vida de uma estrutura de dados, as operações que é possível realizar sobre ela são sujeitas a restrições que são ignoradas por uma análise de pior caso tradicional.
- A ideia chave é considerar o *pior caso da sequência* de N operações, em situações em que este é claramente mais baixo do que a soma dos tempos de pior caso das N operações singulares.
- Trata-se do estudo do custo médio (relativamente à sequência) de cada operação no pior caso, e não de uma análise de caso médio!
- Ao contrário da análise de caso médio, não envolve ferramentas probabilísticas nem suposições sobre os inputs.

## 3 técnicas:

1. Análise *agregada*
2. Método *contabilístico*
3. Método do *potencial*

## Exemplo: Arrays Dinâmicos

Os vectores ou *arrays* são estruturas de dados com capacidade fixa e por isso limitada. Podem no entanto ser alocadas quer *estaticamente* quer *dinamicamente*. Por exemplo em C:

```
int u[1000]; // estático
int *v = calloc(1000, sizeof(int)); // dinâmico
```

Uma solução comum para o problema do crescimento de um vector para além da sua capacidade é a utilização de vectores dinâmicos com *realocação*: quando se pretende inserir um elemento que já não cabe no vector, cria-se um novo vector, tipicamente com o *dobro* da capacidade do primeiro, copiando-se todos os elementos do primeiro para o segundo, antes de se inserir o novo elemento.

```
int* myrealloc(int* v, int n) {
    int* new = calloc(2*n, sizeof(int));
    for (i=0 ; i<n ; i++) new[i] = v[i];
    return new;
}
```

Considere-se agora uma *stack* implementada com um *array* dinâmico, e a sua operação *push*:

```
typedef struct stack {
    int *vec;
    int n;        // n. de elementos inseridos
    int cap;      // capacidade máxima
} Stack;

Stack push (Stack s, int x) {
    if (s.n == s.cap) {
        s.vec = myrealloc(s.vec, s.cap);
        s.cap *= 2;
    }
    (s.vec)[s.n] = x;
    (s.n)++;
    return s;
}
```

Como analisar uma sequência de operações *push*?

A análise de pior caso clássica da função *push* leva-nos a  $T(S) = O(S)$ , com  $S$  o tamanho actual da *stack* (no caso em que há lugar a realocação).

Note-se que de acordo com esta análise uma sequência de  $N$  operações *push* executa no pior caso em tempo  $T(N) = O(N^2)$ . No entanto, se examinarmos passo a passo a sequência em questão, vemos que a realocação será um evento raro, e por esta razão a análise acima é na verdade demasiado pessimista.

A partir de uma *stack* de capacidade  $c$  inicialmente vazia, teremos:

1.  $c$  inserções de custo (1),
2. seguidas de uma inserção de custo  $(c + 1)$ ,
3. novamente seguidas de  $c - 1$  inserções de custo (1),
4. seguidas de uma inserção de custo  $(2 * c + 1)$ ,
5. seguidas de  $2 * c - 1$  inserções de custo (1),
6. seguidas de uma inserção de custo  $(4 * c + 1)$ ,
7. seguidas de  $4 * c - 1$  inserções de custo (1),
8. ...

### Análise Agregada

A ideia desta forma de análise é simplesmente:

- Mostrar que para qualquer  $N$ , uma sequência de  $N$  operações tem no pior caso um *custo agregado*  $O(P(N))$ , que é inferior a  $N$  vezes o custo de pior caso de uma operação executada isoladamente;
- Então o *custo amortizado* por operação é  $O(P(N))/N = O(P(N)/N)$ .

Examinemos então com detalhe a sequência de operações *push*, calculando o custo de cada operação:

$i$	1	2	3	4	5	6	7	8	9	10	...
$cap_i$	1	2	4	4	8	8	8	8	16	16	...
$t_i$	1	<b>1+1</b>	<b>2+1</b>	1	<b>4+1</b>	1	1	1	<b>8+1</b>	1	...
$\log(i-1)$		0	1		2				3		...

Pretendemos agora calcular o *custo agregado*  $\sum_{i=1}^n t_i$ .

Note-se que  $1 + 2 + 4 + 8 + \dots = \sum_{k=0}^{\log(n-1)} 2^k$ , logo

$$\sum_{i=1}^n t_i = n + \sum_{k=0}^{\log(n-1)} 2^k$$

$$= n + (2^{\log(n-1)+1} - 1) = n + 2(n-1) - 1 = 3n - 3$$

O custo agregado da sequência é na verdade linear!

Consequentemente, o custo amortizado da operação *push* será constante:

$$T(n) = \frac{3n-3}{n} \leq 3 = O(1).$$

Com esta técnica (análise agregada),

**Havendo diferentes operações sobre a estrutura de dados, considera-se que todas têm o mesmo custo amortizado**

(as outras técnicas de análise amortizada diferem neste aspecto).

### Método Contabilístico

Na análise agregada calculámos o custo amortizado a partir do custo agregado de uma sequência de operações. No método contabilístico vamos

- Arbitrar um custo amortizado **fixo**  $c$  para a operação, que não corresponderá ao real — será superior para algumas execuções da operação (acumulando *crédito*) e inferior para outras (gastando crédito acumulado)

- Em seguida teremos que argumentar que o custo amortizado individual que arbitrarmos para a operação pode ser considerado válido para efeitos de análise de pior caso, ou seja

*o custo amortizado total de uma qualquer sequência de  $N$  operações é um limite superior para o custo real dessa sequência no pior caso:*

$$\sum_i t_i \leq Nc$$

- Para isso basta calcular o *saldo* entre custo amortizado acumulado e custo real acumulado:

$$bal_{i+1} = bal_i - t_{i+1} + c_{i+1}$$

partindo de um qualquer saldo inicial  $bal_0 = K \geq 0$ , e mostrar que para qualquer  $i > 0$  se tem  $bal_i \geq 0 \implies bal_{i+1} \geq 0$ .

Arbitremos o valor  $c = 2$  para custo amortizado da operação *push* e vejamos a evolução do saldo ao longo da sequência, com  $bal_0 = 0$  e  $bal_{i+1} = bal_i - t_{i+1} + c$ .

$i$	1	2	3	4	5	6	7	8	9	10	...
$cap_i$	1	2	4	4	8	8	8	8	16	16	...
$t_i$	1	2	3	1	5	1	1	1	9	1	...
$c_i$	2	2	2	2	2	2	2	2	2	2	...
$bal_i$	1	1	0	1	-2						...

O custo amortizado escolhido não garante a condição fundamental de *saldo acumulado não-negativo* ao longo de toda a execução.

Façamos nova tentativa, arbitrando o valor  $c = 3$  para o custo amortizado:

$i$	1	2	3	4	5	6	7	8	9	10	...
$cap_i$	1	2	4	4	8	8	8	8	16	16	...
$t_i$	1	2	3	1	5	1	1	1	9	1	...
$c_i$	3	3	3	3	3	3	3	3	3	3	...
$bal_i$	2	3	3	5	3	5	7	9	3	4	...

O custo amortizado de 3 é adequado para mostrar que a operação *push* é, em termos amortizados, de tempo constante no pior caso.

Note-se que a tabela acima constitui um argumento, mas **não uma prova formal**. É habitual, quando se aplica este método, apresentar também uma **interpretação intuitiva** para este custo arbitrado.

Intuitivamente, o custo amortizado 3 para a operação *push* corresponde a:

1. Inserção de um elemento na pilha, havendo espaço; note-se que este elemento ficará na *metade superior* da pilha.
2. Cópia posterior desse mesmo elemento para uma nova pilha, quando a capacidade da actual é excedida e há lugar a realocação; neste passo o elemento passa para a *metade inferior* da nova pilha
3. cópia de um outro elemento, da *metade inferior* da pilha antiga para a nova pilha.

Este terceiro custo tem a ver com a necessidade de cada elemento da metade superior da pilha actual (que foram colocados pela primeira vez depois da última realocação) “suportar” a cópia de um elemento na metade inferior, que já foi copiado para a actual na última realização. Cada elemento suporta a sua própria cópia quando está na metade superior da pilha; quando passar para a metade inferior as cópias passarão a ser pagas pela 3a. moeda de elementos da actual metade superior.

## NOTA

Observe-se que na análise amortizada calculámos o custo  $3n - 3 \leq 3n$  para uma sequência de  $n$  operações, o que reforça a escolha do custo amortizado  $c = 3$ .

## Método do Potencial

Recapitulemos como foi calculado o custo amortizado:

- Na análise agregada, calculou-se o custo total da sequência de operações, e definiu-se o custo amortizado como sendo a média desse custo agregado;
- No método contabilístico, *arbitrou-se* o custo amortizado.

No método do potencial define-se uma *função de potencial* sobre o estado da estrutura de dados, e *calcula-se o custo amortizado* a partir desta função.

A ideia é que o potencial da estrutura deve *aumentar com operações de baixo custo*, e *diminuir com operações de alto custo*, tal como o saldo no método contabilístico. Mas enquanto naquele método o saldo é calculado a partir do custo amortizado, o potencial é definido à partida.

Seja  $\Phi_i$  o potencial da estrutura de dados no estado  $i$ , ou seja depois de  $i$  operações da sequência. Esta função deve obedecer às condições

- $\Phi_0 = 0$
- $\Phi_i \geq 0$  para  $i > 0$ .

O **custo amortizado da operação  $i$**  será dado por  $c_i = t_i - \Phi_{i-1} + \Phi_i$ .

| Compare-se com  $bal_{i+1} = bal_i - t_{i+1} + c_{i+1}$  no método contabilístico.

O cálculo do **custo amortizado total** é *telescópico*:

$$\begin{aligned}\sum_{i=1}^n c_i &= \sum_{i=1}^n t_i - \Phi_{i-1} + \Phi_i \\ &= (t_1 - \Phi_0 + \Phi_1) + (t_2 - \Phi_1 + \Phi_2) + (t_3 - \Phi_2 + \Phi_3) + \dots + (t_n - \Phi_{n-1} + \Phi_n)\end{aligned}$$

Simplificando:

$$\sum_{i=1}^n c_i = \Phi_n - \Phi_0 + \sum_{i=1}^n t_i$$

Ora, uma vez que  $\Phi_n \geq 0$ , é necessariamente verdade que  $\sum_i c_i \geq \sum_i t_i$ , ou seja, o custo total amortizado constitui um limite superior para o tempo de execução da sequência de operações, como desejado.

Voltando ao nosso exemplo, tentemos sintetizar uma função de potencial a partir das propriedades da estrutura de dados.

Uma vez que já obtivemos, pelo método contabilístico, um bom candidato (3) para o custo amortizado, conhecemos à partida os valores de uma função de potencial possível — são os valores que o saldo toma ao longo da execução. Basta então encontrar a expressão simbólica correspondente a esta função.

Nesta perspectiva, o método do potencial é usado depois do método contabilístico, no sentido de *justificar* os custos amortizados arbitrados naquele método.

Observemos então de novo a tabela anterior, substituindo o saldo pelo potencial  $\Phi_i$  no estado actual, e acrescentando uma coluna para o estado inicial, em que temos capacidade inicial 1, com ocupação 0.

$i = n_i$	0	1	2	3	4	5	6	7	8	9	10	...
$cap_i$	1	1	2	4	4	8	8	8	8	16	16	...
$t_i$		1	2	3	1	5	1	1	1	9	1	...
$\Phi_i$	0	2	3	3	5	3	5	7	9	3	5	...

Tentemos chegar à definição de uma função de potencial adequada, esquecendo por momentos que conhecemos os saldos resultantes da aplicação do método contabilístico.

Procuramos uma expressão para  $\Phi_i$  tal que:

- $\Phi_0 = 0$  e  $\Phi_i \geq 0$  para  $i > 0$
- que dependa apenas de  $n_i$  e de  $cap_i$
- que aumente, a partir do valor inicial 0, à medida que a ocupação da pilha aumenta, diminuindo nos passos em que o array é duplicado.

Como este valor aumenta mesmo quando  $cap_i$  não varia, tem de crescer com  $n_i$ . Por outro lado, diminui quando  $cap_i$  aumenta, por isso deve decrescer com  $cap_i$ .

Observe-se que, com a excepção das duas primeiras configurações, em qualquer momento se tem mais de metade da capacidade sempre seguramente preenchida, uma vez que só nesse momento é feita a realocação, ou seja  $n_i \geq cap_i/2 + 1$ , logo  $2 * n_i - cap_i \geq 2$ .

Para  $i = 0$  tem-se  $2 * n_i - cap_i = -1$ , e  
para  $i = 1$  tem-se  $2 * n_i - cap_i = 1$



Em todas as configurações temos então  $2 * n_i - \text{cap}_i \geq -1$ ,  
 logo  $2 * n_i - \text{cap}_i + 1 \geq 0$ .

Será esta expressão um bom candidato para definir a função de potencial? De facto sim, a função definida como

$$\Phi_i = 2 * n_i - \text{cap}_i + 1$$

produz os valores contidos na tabela acima (correspondentes ao saldo no método contabilístico).

Podemos confirmar isto, *calculando* o custo amortizado de *push* que resulta desta função de potencial.

Relembremos que  $c_i = t_i - \Phi_{i-1} + \Phi_i$ , logo

$$c_i = t_i - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * n_i - \text{cap}_i + 1)$$

$$c_i = t_i - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * (n_{i-1} + 1) - \text{cap}_i + 1)$$

Temos dois casos distintos:

- Se  $n_{i-1} < \text{cap}_{i-1}$  ( $\text{cap}_i = \text{cap}_{i-1}$ , não há realocação)  
 $c_i = 1 - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * (n_{i-1} + 1) - \text{cap}_{i-1} + 1) = 3$
- Se  $n_{i-1} = \text{cap}_{i-1}$  ( $\text{cap}_i = 2 * \text{cap}_{i-1}$ , há realocação)  
 $c_i = (n_{i-1} + 1) - (2 * n_{i-1} - n_{i-1} + 1) + (2 * (n_{i-1} + 1) - 2 * n_{i-1} + 1) = 3$

Obtemos o mesmo custo amortizado que tínhamos obtido pelo método contabilístico.

*Temos agora certeza de que, em qualquer sequência de operações push, o tempo amortizado de execução da operação ao longo da sequência é constante.*

Note-se que é possível utilizar o método do potencial de forma isolada, sem nenhuma ideia de quais os custos amortizados que serão obtidos, e nesse caso a síntese da função de potencial será mais desafiadora.

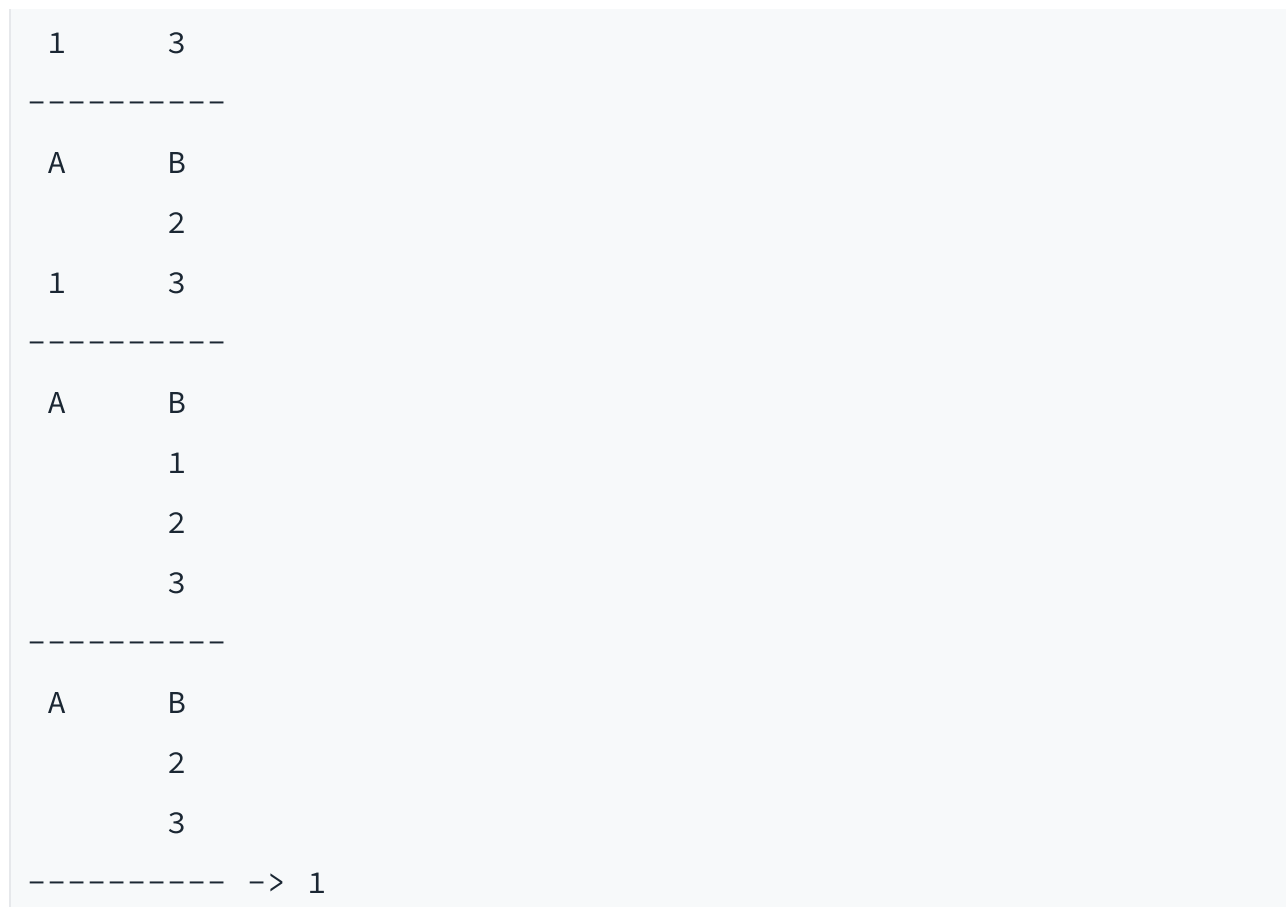
## Exemplo: Fila de espera (*queue*) implementada com duas pilhas (*stacks*)

Uma implementação possível de uma fila de espera (*Queue*) utiliza duas pilhas A e B, por exemplo:

```
typedef struct queue {  
    Stack a;  
    Stack b;  
} Queue;
```

- A inserção (`enqueue`) de elementos é sempre realizada na pilha A
- para a saída de elementos (`dequeue`), se a pilha B não estiver vazia, é efectuado um `pop` nessa pilha;
- caso contrário, para todos os elementos de A, faz-se sucessivamente `pop` e `push` na pilha B. Faz-se depois `pop` da pilha B, devolvendo-se como resultado o elemento extraído.

```
enqueue 1; enqueue 2; enqueue 3  
  
A      B  
3  
2  
1  
-----  
dequeue  
A      B  
2
```



Pela análise tradicional de pior caso,

- `enqueue` executa em tempo  $O(1)$
- `dequeue` executa em tempo  $O(N)$

Veremos no entanto que **em termos amortizados, ambas as operações executam em tempo constante  $O(1)$**

A análise amortizada é aqui mais desafiante, uma vez que uma sequência típica de operações será agora **heterogênea**, contendo ocorrências de `enqueue` e de `dequeue`. Na análise agregada e pelo método contabilístico, apenas conseguiremos analisar as operações no contexto de **sequências concretas** que fixamos à partida.

Veremos que neste caso o método do potencial se revela superior, uma vez que é independente de qualquer sequência particular de operações que se considere.

### Análise Agregada

Consideremos a seguinte **sequência concreta** de operações:

- $n$  operações `enqueue` seguidas de  $n$  operações `dequeue`

$i$	0	1	2	...	$n$	$n+1$	$n+2$	...	$2n$
$n_A$	0	1	2	...	$n$	0	0	...	0
$n_B$	0	0	0	...	0	$n-1$	$n-2$	...	0
$t_i$		1	1	...	1	<b><math>2n+1</math></b>	1	...	1

Temos  $\sum_{i=1}^{2n} t_i = n + (2n + 1) + (n - 1) = 4n$

Globalmente a sequência executa em tempo  $4n$ , logo cada uma das  $2n$  operações executa em tempo constante. Intuitivamente:

*uma operação `dequeue` de custo linear  $\Theta(n)$  só pode ser executada depois de  $n$  operações `enqueue` de tempo constante, que coloquem  $n$  elementos na pilha  $B$ . Os `enqueues` amortizam o `dequeue` mais custoso.*

O custo amortizado que resulta desta análise agregada é  $\frac{4n}{2n} = 2$ . Note-se que este valor resulta da análise feita para a sequência específica de operações considerada em cima, e é igual para ambas as operações. Veremos em seguida que os outros métodos permitirão chegar a custos diferentes, válidos para outras sequências.

## Método Contabilístico

Considerando a mesma sequência concreta de operações, comecemos por considerar um custo amortizado  $c_i = 2$  para todas elas.

$i$	0	1	2	...	$n$	$n+1$	$n+2$	...	$2n$
$n_A$	0	1	2	...	$n$	0	0	...	0
$n_B$	0	0	0	...	0	$n-1$	$n-2$	...	0
$t_i$		1	1	...	1	$2n+1$	1	...	1
$c_i$		2	2	...	2	2	2	...	2
$bal_i$		1	2	...	$n$	1-n			

O custo amortizado considerado para a operação `enqueue` é insuficiente para cobrir o custo real da operação `dequeue` no pior caso. Consideremos então o custo amortizado  $c_i = 3$ :

$i$	0	1	2	...	$n$	$n+1$	$n+2$	...	$2n$
$n_A$	0	1	2	...	$n$	0	0	...	0
$n_B$	0	0	0	...	0	$n-1$	$n-2$	...	0
$t_i$		1	1	...	1	$2n+1$	1	...	1
$c_i$		3	3	...	3	3	3	...	3
$bal_i$		2	4	...	$2n$	2	4	...	$2n$

Este valor para o custo amortizado é suficiente.

Vemos no entanto que este custo é excessivo no caso da operação `dequeue`: basta para esta operação considerar um custo  $c_i = 1$ :

$i$	0	1	2	...	$n$	$n+1$	$n+2$	...	$2n$
$t_i$		1	1	...	1	$2n+1$	1	...	1
$c_i$		3	3	...	3	1	1	...	1
$bal_i$		2	4	...	$2n$	0	0	...	0

Intuitivamente, o custo de 3 corresponde no caso do `enqueue` ao custo de efectuar:

1. `push` de um elemento na pilha A
2. `pop` desse elemento da pilha A
3. `push` do mesmo elemento na pilha B

## Método do Potencial

Nas análises anteriores, agregada e pelo método contabilístico, escolhemos uma sequência que intuitivamente parecia levar ao pior caso de execução de `dequeue`.

No entanto as análises **não provaram** que os custos amortizados calculados são adequados para uma `qualquer` sequência das operações. Com o método do potencial isto será possível.

Para a aplicação do método consideremos a função de potencial seguinte:

$$\Phi_i = 2n_{Ai}.$$

A intuição para esta escolha vem da observação da tabela acima: a única operação custosa (custo  $> 1$ ) é o primeiro `dequeue`, que tem um custo  $2n + 1$ , sendo  $n$  o número de elementos da primeira pilha; o potencial armazenado permite cobrir este custo (e é fácil de ver que o potencial corresponde ao saldo na tabela acima).

Claramente temos  $\Phi_i \geq 0$  para  $i \geq 0$  e  $\Phi_0 = 0$  (desde que se inicie a sequência com a stack A vazia). Simulemos a evolução do potencial ao longo da execução da sequência que temos vindo a considerar:

$i$	0	1	2	...	$n$	$n+1$	$n+2$	...	$2n$
$n_A$	0	1	2	...	$n$	0	0	...	0
$n_B$	0	0	0	...	0	$n-1$	$n-2$	...	0
$t_i$		1	1	...	1	<b><math>2n+1</math></b>	1	...	1
$\Phi_i$	0	2	4	...	$2n$	0	0	...	0

É visível que o potencial cresce nas operações “leves” e decresce na operação pesada `dequeue`, de tempo linear.

Calculemos o custo amortizado de ambas as operações

- `enqueue`:

$$t_i - \Phi_{i-1} + \Phi_i = 1 - 2n_A + 2(n_A + 1) = 3$$

- `dequeue`. Teremos que considerar dois cenários:

- Se  $n_B \neq 0$ :  $t_i - \Phi_{i-1} + \Phi_i = 1 - 2n_A + 2n_A = 1$
- Se  $n_B = 0$ :  $t_i - \Phi_{i-1} + \Phi_i = (2n_A + 1) - 2n_A + 0 = 1$

É de realçar que

- Mais uma vez, o método de potencial vem confirmar e provar os custos amortizados a que se tinha chegado de forma intuitiva pela análise agregada e pelo método contabilístico.  
(uma vez que a função de potencial cumpre os requisitos necessários)
- Os custos amortizados calculados com o método do potencial são garantidamente **válidos para qualquer sequência de operações**, ao contrário das análises anteriores.

### EXERCÍCIO: Pilha com operação `Multipop` [A resolver nas aulas TP]

Considere-se uma estrutura de dados do tipo *stack* com a habitual operação `push`, mas em que a operação `pop` é substituída por uma operação `multipop`, uma generalização que remove os  $k$  primeiros elementos, deixando a pilha vazia caso contenha menos de  $k$  elementos.

Uma implementação possível será

```
multiPop(S,k) {  
    while (!IsEmpty(S) && k != 0) {  
        pop(S);  
        k -= 1;  
    }  
}
```

Pela análise tradicional de pior caso,

- `push` executa em tempo  $O(1)$
- `multiPop` executa em tempo  $O(N)$

Utilize os 3 diferente métodos estudados, para mostrar que em termos amortizados a operação `multiPop` executa também ela em tempo constante  $O(1)$ .