Arrays redimensionáveis, análise amortizada

Fernando Lobo

Algoritmos e Estrutura de Dados

Arrays redimensionáveis

- Nesta aula vamos ver como podemos ter arrays cuja capacidade máxima expande ou encolhe automaticamente à medida das necessidades.
- Denominam-se por *arrays redimensionáveis* ou *tabelas dinâmicas*.
- Várias colecções do Java (e de outras linguagens de programação) são implementadas à base destes arrays/tabelas.

Problema

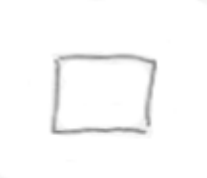
- Queremos ter um array mas não sabemos à partida quantos elementos é que o array poderá ter.
- O array deve ser o mais pequeno possível (para não desperdiçar memória), mas também suficientemente grande para não haver overflow.
- Como resolver esta questão?

Uma solução possível

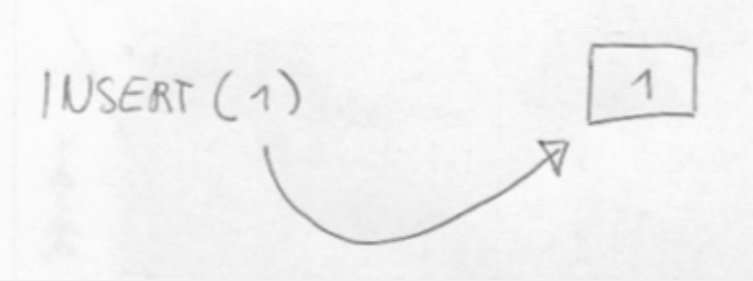
- Começamos com um array pequeno.
- Se encher, alocamos espaço para um novo array de maior dimensão.
  - copiam-se os elementos do array antigo para o array novo.
  - deitamos fora o array antigo.
    (em C, com *free*. Em Java, deixando para o *garbage collector*.)

Exemplo

- Começamos com um array com apenas uma posição.
- Vamos inserindo elementos.
  - Para já, vamos assumir que inserimos sempre ao final.
- Se o array encher, duplicamos o tamanho.

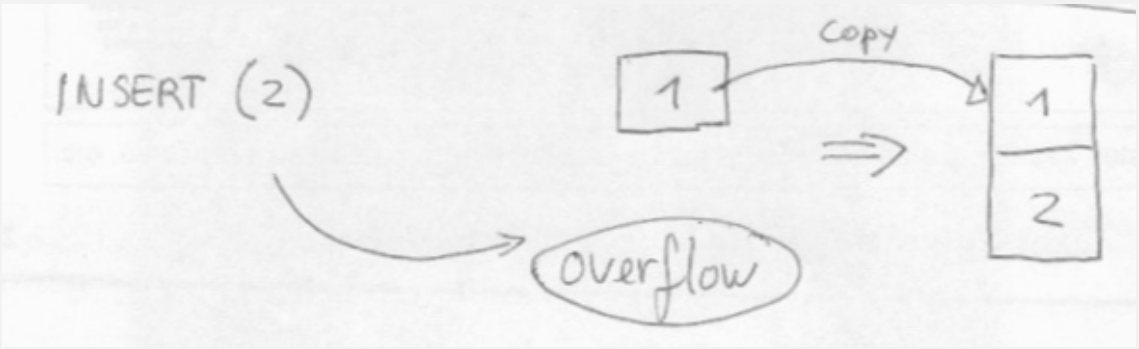


INSERT(1)



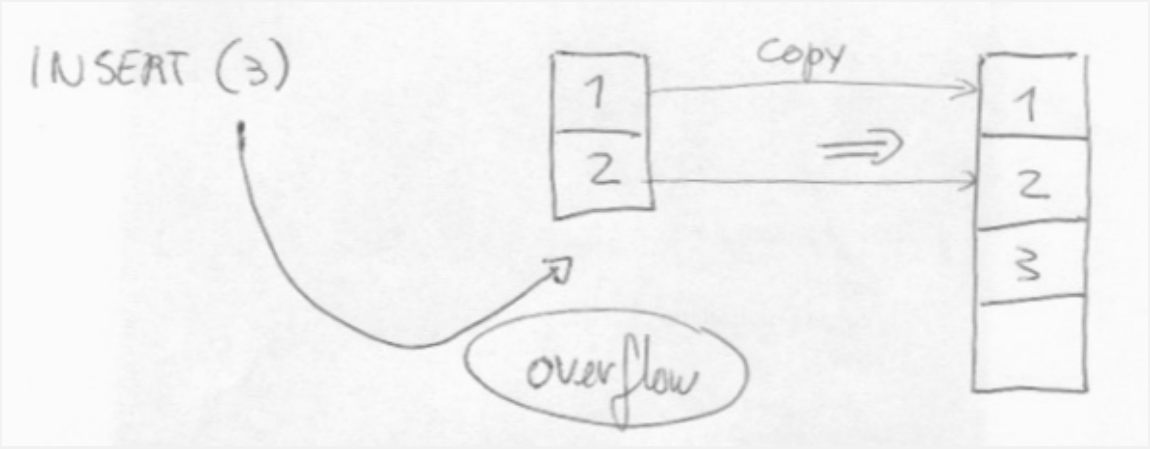
- O array está cheio. O próximo INSERT não vai caber.

INSERT(2)



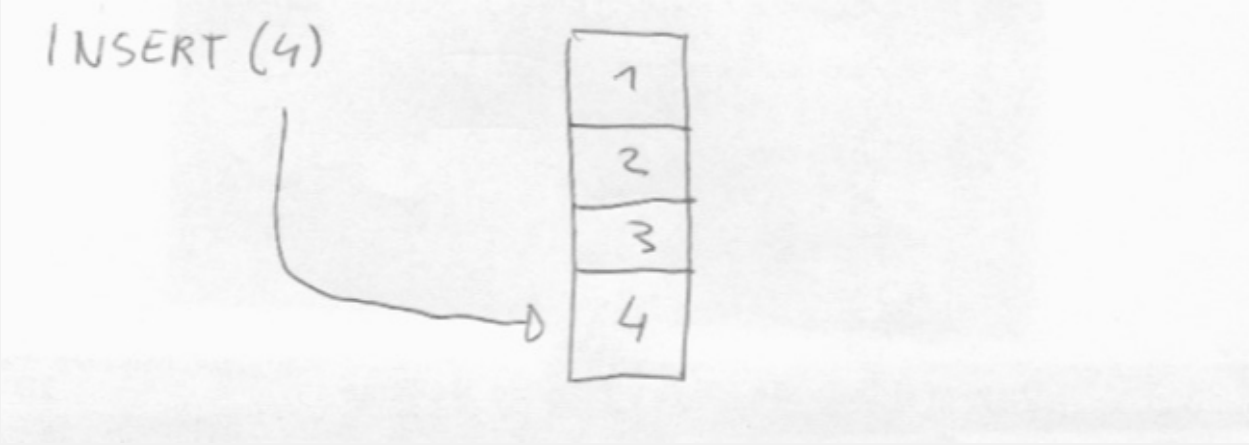
- Deu overflow.
- Cria-se um array novo com o dobro do tamanho, copia-se os elementos do array antigo, insere-se o novo elemento, e apaga-se o array antigo.
- O novo array passa a ter dimensão 2 e contém 2 elementos: 1 e 2. Está cheio novamente.

INSERT(3)



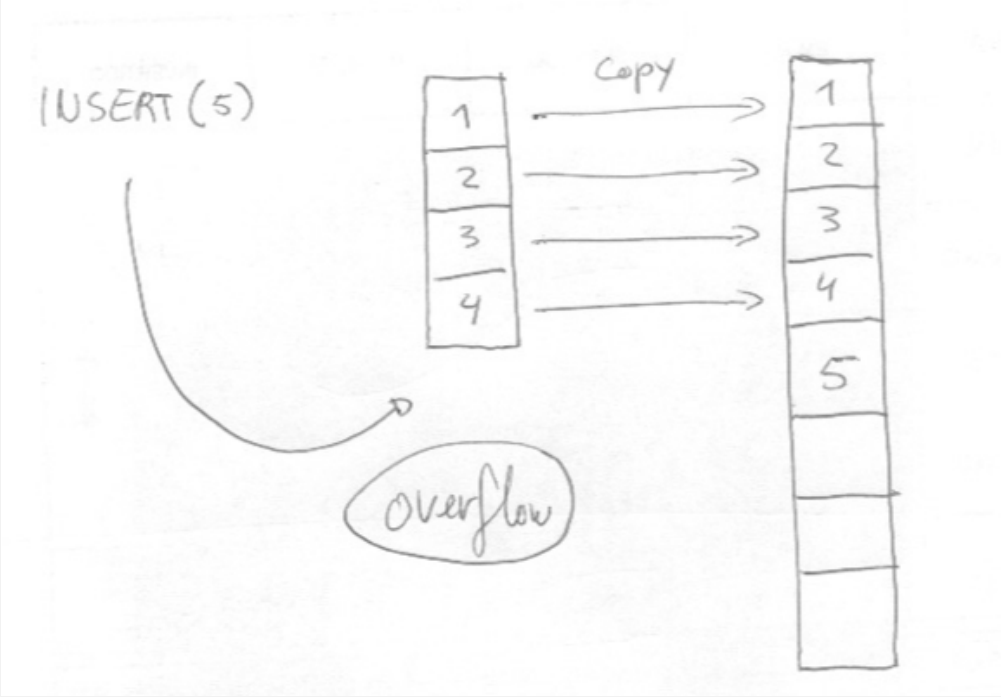
- Deu overflow.
- Cria-se um array novo com o dobro do tamanho, copia-se os elementos do array antigo, insere-se o novo elemento, e apaga-se o array antigo.
- O novo array passa a ter dimensão 4 e contém 3 elementos: 1, 2, 3.

INSERT(4)



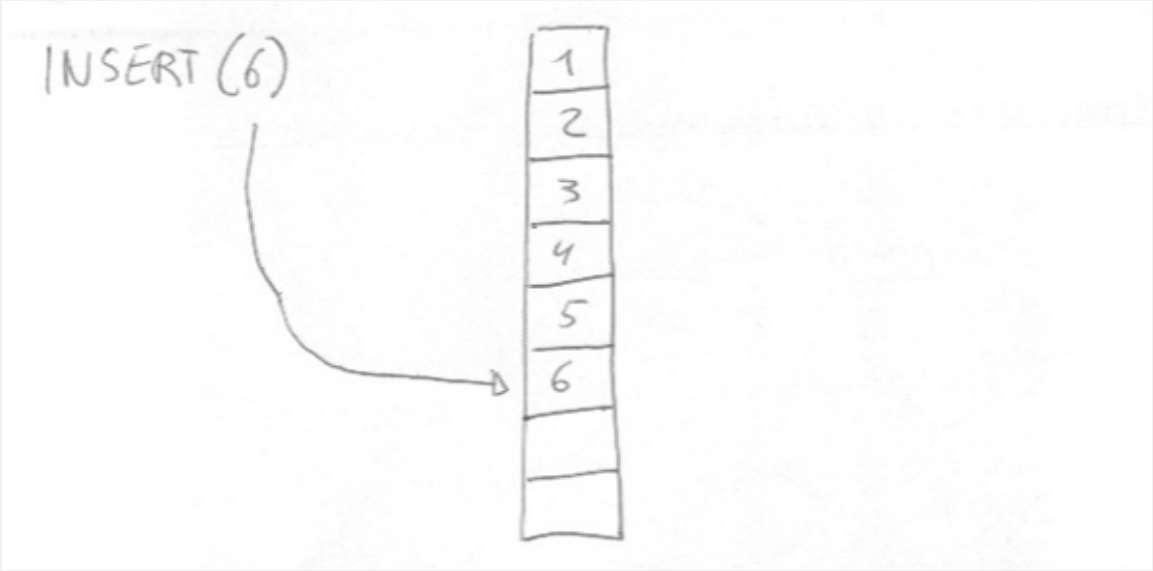
- Não há problema. Insere-se 4.
- O array passa a ter 4 elementos e está novamente cheio.

INSERT(5)



- Overflow. Cria-se um array com o dobro do tamanho, ...
- O novo array passa a ter dimensão 8 e contém 5 elementos.

INSERT(6)



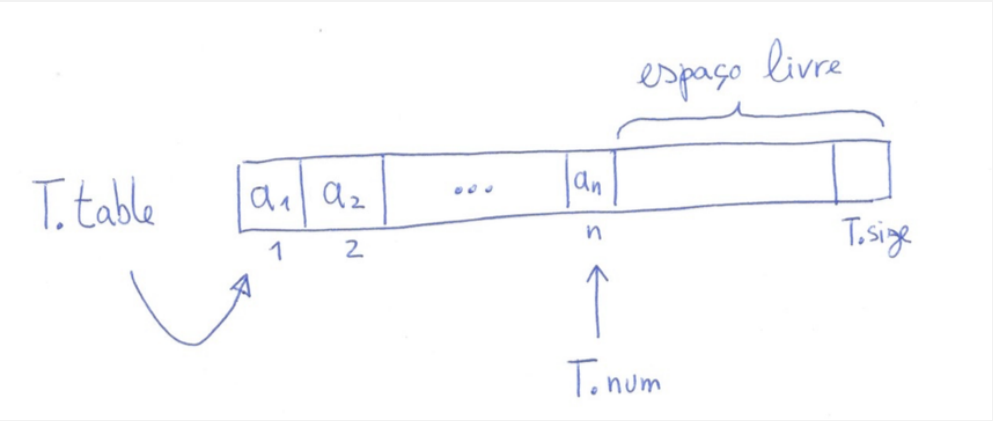
- Não há problema. Insere-se 6.
- O array passa a ter 6 elementos.

INSERT(7), INSERT(8), . . .

- Estão a ver a ideia. Não é necessário dizer mais nada . . .

Implementação

- T é um objecto que representa um array redimensionável.
  - $T.table$ é o array propriamente dito
  - $T.size$ é a capacidade máxima do array
  - $T.num$ é o número de elementos contidos no array
- Inicialmente $T.num = T.size = 0$



Pseudocódigo

TABLE-INSERT(T, x)

```
 1  if T.size == 0
 2       allocate T.table with 1 slot
 3       T.size = 1
 4  if T.num == T.size
 5       allocate new-table with 2 · T.size slots
 6       insert all items in T.table into new-table
 7       free T.table
 8       T.table = new-table
 9       T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

Factor de carga (load factor)

- O *load factor* $\alpha(T)$ de uma tabela não vazia é a percentagem de ocupação da tabela.
  - $\alpha(T) = 0 \quad \rightarrow$ array está vazio
  - $\alpha(T) = 1 \quad \rightarrow$ array está cheio
- No caso de apenas termos INSERTs, $\alpha(T) \geq 1/2$ (não se desperdiça demasiado espaço).

Análise do pior caso

- Consideremos uma sequência de n INSERTs.
- No pior caso, um INSERT é $\Theta(n)$.
- Logo, no pior caso n INSERTs é $n \cdot \Theta(n) = \Theta(n^2)$
- Certo? ERRADO!
- No pior caso, n INSERTs é apenas $\Theta(n)$.

Intuição

- Apesar de uma operação individual poder ser dispendiosa, o custo médio por operação é pequeno.
- Porquê?
  - Porque há poucas operações dispendiosas e muitas operações baratas.

Análise detalhada

Seja c_i = custo do i-ésimo INSERT.

$$c_i = \begin{cases} i & \text{, se } i-1 = 2^k \text{ , } k \in \mathbb{N} \\ 1 & \text{, caso contrário} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10	...
$size_i$	1	2	4	4	8	8	8	8	16	16	...
c_i	1	2	3	1	5	1	1	1	9	1	...

$size_i$ é o tamanho da tabela imediatamente após o i-ésimo INSERT.

Mais fácil de ver um padrão se reescrevermos c_i

i	1	2	3	4	5	6	7	8	9	10	...
$size_i$	1	2	4	4	8	8	8	8	16	16	...
c_i	1	2	3	1	5	1	1	1	9	1	...
c_i	1	1+1	1+2	1	1+4	1	1	1	1+8	1	...

$$\begin{aligned} \text{Custo de } n \text{ INSERTs} &= \sum_{i=1}^{n} c_i \\ &= ? \end{aligned}$$

Custo de n INSERTs

$$\begin{aligned} \text{Custo de } n \text{ INSERTs} &= \sum_{i=1}^{n} c_i \\ &= n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\leq n + 2n \\ &= 3n \\ &= \Theta(n). \end{aligned}$$

- Logo, o custo médio de cada operação é $\frac{\Theta(n)}{n} = \Theta(1)$.
- Ao custo médio chamamos custo amortizado.

Análise Amortizada

- Resulta da análise de uma sequência de operações numa estrutura de dados.
- Permite obter a complexidade no pior caso para uma qualquer sequência de operações.

Análise Amortizada

Apesar de estarmos a fazer médias, isto não tem nada que ver com a análise de complexidade no caso médio.

- A análise não envolve probabilidades/distribuição de inputs.
- Análise amortizada dá a performance no pior caso para uma sequência de operações, o que é igual à performance média de cada operação no pior caso.

Contração do array

- Até agora apenas consideramos a inserção de elementos.
- E se quisermos remover elementos?
- Faz talvez sentido contrair o array quando o load factor ficar abaixo de um determinado valor, de modo a não se desperdiçar muito espaço.
- O procedimento é análogo ao da expansão: alloca-se um novo array, copia-se os elementos do antigo para o novo, liberta-se o array antigo.

Contração do array

- Uma possível estratégia:
  - Se o array estiver cheio duplicamos o tamanho.
  - Se estiver menos de metade cheio, diminuímos o tamanho do array para metade.
- Com esta abordagem, $\alpha(T)$ é sempre $\geq 1/2$

Contração do array

- A abordagem anterior parece boa, mas infelizmente não é!
- Para ver que não, consideremos uma sequência de n operações em que n é uma potência de 2.
  - As primeiras $n/2$ operações são INSERTs.
  - As restantes $n/2$ operações são: I, D, D, I, I, D, D, I, I, ... (I = INSERT, D = DELETE)
- Vamos estar num ping-pong constante de expansão, contração, expansão, contração, ...

Análise

- A sequência dos primeiros $n/2$ INSERTs' tem complexidade $\Theta(n)$.
- Após essa sequência de operações o array estará cheio e o INSERT seguinte irá provocar uma expansão (com complexidade $\Theta(n)$.)
- Segue-se dois DELETEs que provocam uma contração do array.
- Segue-se dois INSERTs que provocam uma expansão.
- ... e assim sucessivamente.

Análise (cont.)

- Cada expansão e contração tem complexidade $\Theta(n)$.
- Teremos $n/4 = \Theta(n)$ expansões e contrações. Logo o custo total desta sequência de operações é de $\Theta(n^2)$.
- Ou seja, o custo amortizado por operação seria $\Theta(n)$.

Uma melhor estratégia

- Temos de permitir que o load factor baixe um bocado mais de que $1/2$.
  - Se o array estiver cheio duplicamos o tamanho.
  - Se estiver menos de $1/4$ cheio, diminuímos o tamanho do array para metade.
- Com esta abordagem, $\alpha(T)$ é sempre $\geq 1/4$
- Vou omitir a análise, mas é possível provar que esta abordagem garante que qualquer sequência de n operações (INSERTs ou DELETEs) tenha complexidade $\Theta(n)$.
  - O seja, o custo amortizado por operação é $\Theta(1)$.

Nota final

- Estes arrays redimensionáveis podem obviamente ser usados para implementar o conceito de lista que vimos na aula passada.
  - Deixamos de ter a desvantagem de estarmos limitado a um tamanho máximo de elementos.
  - Continuamos a ter as vantagens que a implementação do array tinha: acesso directo, localidade da memória.
  - Inserções e remoções ao final da lista continuam a ser feitas de modo eficiente, tal como antes.
  - Inserções e remoções em posições arbitrárias continuam a requerer tempo $\Theta(n)$.
    - ⋆ porque é necessário "abrir espaço" a meio do array e "empurrar" todos os elementos que estão à sua direita.
- java.util.ArrayList está implementado de forma análoga.