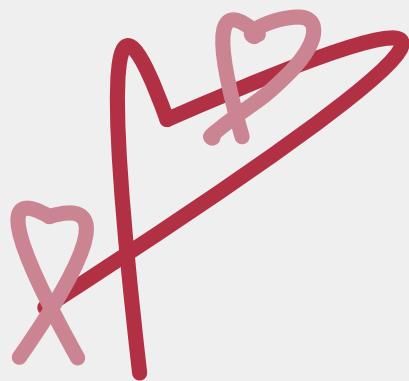


AED



Mariama Perna ✓

# Algoritmos e estruturas de dados

Um bom programa tem uma complexidade computacional reduzida (tanto quanto possível), ou seja, corre em pouco tempo e espaço (de memória), sendo que por vezes temos de trocar um pelo outro.

**ALGORITMOS**, são descrições detalhadas da resolução de um problema



um **Bom Algoritmo**, deve ser correto, inequívoco, finito, eficiente (velocidade/recursos), e determinístico ou aleatório (separa além das entradas, depender de valores aleatórios).

## • Algoritmos de ordenação

um algoritmo de ordenação que em caso de repetição de elementos os mantenha na ordem original diz-se **estável**.

## Dados abstratos

o programador pode decidir que tendo em conta o nº elevado de acesso às variáveis de um determinado tipo, estas devem priorizar a velocidade de acesso em detrimento da memória ocupada. No entanto, durante o desenvolvimento aparece-se que a memória é mais importante e decide mudar a implementação. Os dados abstratos são a solução para evitar uma completa reestruturação do programa.

Diferentes algoritmos trabalham c/ os dados de diferentes formas, sendo esta forma volátil durante o processo de desenvolvimento de algoritmo. Para evitar problemas de compatibilidade ao mudar a forma como um tipo de dados é implementado, existem os dados abstratos.

Estes permitem criar uma abstração entre o programa e os dados, cujos processos internos podem ser alterados, mas nunca afetando as suas funcionalidades básicas (como que uma interface), utilizadas pelo programa.

## COMPLEXIDADE COMPUTACIONAL

Os modelos de computação permitem avaliar a complexidade computacional de um algoritmo, ou seja, o trabalho necessário para o executar. Um dos mais simples é o modelo de computação RAM, segundo o qual o trabalho é dividido em passos terminais (time steps).

Operações elementares (+, -, /, \*) , comparações , atribuições de valores a variáveis , a chamada de uma subrotina , avaliação de operações numéricas transcendentais (sin), saltos condicionais e o acesso à memória (r/w) custam um passo temporal.

Ciclos e subrotinas são estudados tendo em conta os seus componentes individuais.

O espaço em memória utilizado para armazenar um nº com x bits (determinados) utiliza uma unidade de espaço.

Apesar de não nos dar um valor exato do custo , aproxima-se deste apenas pela multiplicação de um valor constante.

A complexidade computacional é então dada pelo número de passos temporais e de unidades de espaço necessárias à execução do algoritmo.

## NOTAÇÃO ASSINTÓTICA

quando  $n$  tende para  $+\infty$

, esta notação permite-nos omitir detalhes irrelevantes sobre a velocidade de crescimento de uma função . Caracteriza a complexidade em função do tamanho da entrada ,  $n$ .

pode ter várias formas

- BIG OH NOTATION is useful to deal with upper bounds.

$f(n) = O(g(n))$  means that there exists an  $n_0$  and a constant  $C$  such that , for all  $n \geq n_0$  ,  $f(n) \leq C g(n)$ .

- BIG OMEGA NOTATION is useful to deal with lower bounds.

$f(n) = \Omega(g(n))$  means that exists an  $n_0$  and a constant  $C$  such that , for all  $n \geq n_0$  ,  $f(n) \geq C g(n)$

- BIG THETA NOTATION is useful to deal with upper and lower bounds that have the same form .

$f(n) = \Theta(g(n))$  means that there exists an  $n_0$  and two constants  $C_1$  and  $C_2$  that for all  $n \geq n_0$  one has  $C_1 g(n) \leq f(n) \leq C_2 g(n)$  .

## • SMALL OH NOTATION

$$f(n) = \tilde{O}(g(n))$$

means that

$$f(n) = O(g(n) \log^k g(n)) \text{ for some } k > 0$$

A notação  $O$  é mais utilizada que a teta, pois muitas vezes os algoritmos têm casos particulares em que o crescimento não acompanha o padrão, tornando-se mais difícil majorar e minorar simultaneamente do que fazer apenas um destes dois.

Estas notações são válidas para  $n > n_0$ , sendo  $n_0$  suficientemente grande.

## Notação da SOMA de funções:

$\tilde{O}(f+g) \rightarrow$  a função que cresce mais rápido "ganha"

- $f(n) = \Theta(g(n))$ , which implies that  $g(n) = \Theta(f(n))$ .

In this case  $\tilde{O}(f(n)+g(n)) = \tilde{O}(f(n))$

- $f(n) = O(g(n))$ .

In this case  $\tilde{O}(f(n)+g(n)) = O(g(n))$

- $g(n) = O(f(n))$ .

In this case  $\tilde{O}(f(n)+g(n)) = O(f(n))$

$\tilde{\Omega}(f+g) \rightarrow$  a função que cresce mais devagar "ganha"

## Big theta

$$f(n) = \Theta(g(n)) \text{ then } \Theta(f(n)+g(n)) = \Theta(f(n))$$

## Notação na MULTIPLICAÇÃO de escalares

(mesmo que a multiplicação do escalar pela notação da função)

- $\tilde{O}(c f(n)) = \tilde{O}(f(n))$ ,  $\tilde{\Omega}(c f(n)) = \tilde{\Omega}(f(n))$

and  $\Theta(c f(n)) = \Theta(f(n))$

## Notação de MULTIPLICAÇÃO de funções

(mesmo que a multiplicação da notação das funções)

- $\tilde{O}(f(n)g(n)) = \tilde{O}(f(n))\tilde{O}(g(n))$

- $\tilde{\Omega}(f(n)g(n)) = \tilde{\Omega}(f(n))\tilde{\Omega}(g(n))$  and  $\Theta(f(n))\Theta(g(n))$

$\sim$

# Determinar a notação

$$\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = \begin{cases} 0, & \Rightarrow f(n) = o(g(n)) \\ > 0, & \Rightarrow f(n) = O(g(n)) \\ \infty, & \Rightarrow f(n) = \Omega(g(n)) \end{cases}$$

$f(n) = \Theta(g(n))$     $f(n) = \underline{\Omega}(g(n))$   
 $f(n) = \underline{\Omega}(g(n))$

## Tempos de execução

- $\frac{1}{n} < 100 < n \log(n) < n^* n^{1/2} < n^2$
- $n! < n^n$

## Estruturas de dados elementares

### → DATA CONTAINER:

É uma estrutura de dados elementar que especifica como a informação é organizada e armazenada, definindo também uma interface de acesso e modificação da informação contida. (*trata-se de uma classe*)

Existem vários tipos de data containers dependendo da finalidade que podem ser utilizados e da sua complexidade computacional. Podem variar no tipo de acesso (sequencial, aleatório, ...), pesquisa, inserção e remoção e as suas eficiências.

### • ARRAYS

Estas estruturas de dados pode ser utilizada para implementar um data container, assumindo que os elementos são armazenados consecutivamente na memória. É no entanto necessária atenção a alguns aspetos, que quando tidos em conta diminuem os recursos necessários para a sua manipulação.

*complexidade computacional:*

$O(1)$	$O(\log n)$	$O(n)$
Acesso aleatório a um índice	Procurar por elemento quando array está ordenado	Redimensionando
Adicionar/eliminar elemento ao fim		Adicionar/diminar a índice específico, criando/removendo espaço para que era (d)este
Alterar elemento num índice		

O acesso sequencial é rápido

## • CIRCULAR BUFFER

INFORMAÇÃO ADICIONAL: número de elementos(n)

Consiste num array que tem um tamanho fixo e que se comporta como que é um círculo, em que o último elemento está "ligado ao primeiro".

Um array com n elementos, terá assim índices entre 0 e n, sendo que o último corresponde ao primeiro ( $a[0] = a[n]$ ).

↳ complexidade computacional, as operações de manipulação são semelhantes às do array, com exceção de que podemos aceder a mais um índice.

## • LINKED LISTS

INFORMAÇÃO OPCIONAL: último elemento

INFORMAÇÃO ADICIONAL NECESSÁRIA: primeiro elemento da lista

Listas ligadas são estruturas dinâmicas, pois podem crescer à medida que são inseridos elementos. Isto deve-se ao facto de um elemento conter a sua informação e a informação do elemento seguinte na lista (na forma de ponteiros), podendo até conter para o elemento anterior (nas listas ligadas duplas).

Nos casos mais comuns o último elemento tem o ponteiro do próximo elemento no último com o valor NULL e nas listas duplas o ponteiro para o anterior do primeiro segue a mesma regra. No entanto, podemos guardar os endereços do primeiro e do último nos dois casos anteriores, respetivamente.

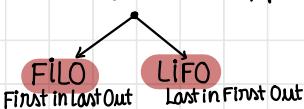
complexidade computacional:

$O(1)$	$O(n)$
Adicionar/eliminar informação no inicio	Acesso aleatório a um índice
Adicionar informação ao final, sabendo qual o último	Adicionar informação ao final, não sabendo qual o último
Apagar um nó numa lista dupla	Apagar um nó numa lista singular
	Procurar por informação (mesmo que a lista esteja ordenada)

O acesso sequencial é rápido do início para o fim em ambos os tipos e do fim para o início apenas na lista ligada dupla

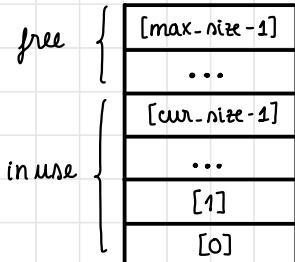
## STACKS

, permite fazer as seguintes operações

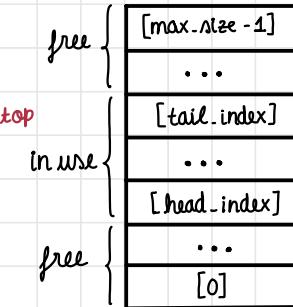


- Push Adicionar elemento no topo da stack
- POP Eliminar elemento no topo
- top Mostrar elemento no topo  
Devolver tamanho atual

Esta estrutura de dados pode ser implementada utilizando arrays (com a limitação do tamanho), listas ligadas ou dequeues.



exemplo de implementação c/ array



exemplo de implementação c/ buffer circular



exemplo de implementação c/ lista ligada dupla

## • QUEUES

Filas podem seguir as mesmas duas filosofias que os stacks, devendo permitir as seguintes operações:

- enqueue , adicionar elemento ao final
- dequeue , eliminar primeiro elemento (avançando um índice)

Podem ser implementadas c/ circular buffers, listas ligadas ou dequeues.

## • DEQUES

Consistem em filas c/ dois fins, sendo possível realizar operações (adicionar/remover elementos) nos dois finais. Suporta as seguintes operações habituais, com exceção que o final da fila não é o final do array.

Pode ser implementado utilizando um buffer circular ou uma lista ligada dupla (para remoções serem eficientes).

## • HEAPS

Um heap (pilha/monte) é um array com uma estrutura interna, que torna a forma de uma relação de ordem.

### → binary max-heap

Consiste numa heap (de comprimento n), cuja estrutura interna caracteriza-se pelos elementos c/ índice i não poderem ser superiores que os elementos de índice  $\lfloor \frac{i-1}{2} \rfloor$ , para valores no intervalo  $[2, n]$ .

Parte interna  $\lfloor \frac{i-1}{2} \rfloor = i - \text{fix}$

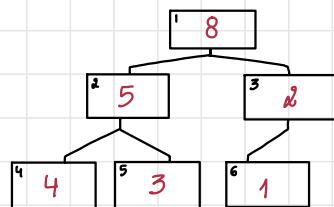
Estas estruturas de dados podem ser representadas numa árvore binária.

index	value
0	-
1	8
2	5
3	2
4	4
5	3
6	1

not used



implicit  
binary tree  
organization



### → binary min-heap

Segue a mesma filosofia que o anterior, mas os elementos de índice  $i$  não podem ser inferiores aos de índice  $\lfloor i/2 \rfloor$ .

### → M-way heap

Os índices pai e filho num heap, dependem do índice do elemento raiz, podendo ser para índices raiz 0 ou para  $\frac{i+m-2}{m}$  índices raiz 1.

Para diferentes valores de  $m$ , os arrays são implicitamente divididos em conjuntos de  $m$  elementos, a partir do índice raiz, não inclusivo.

Exemplo: subvisão de um heap c/ índice raiz 1 e  $m=3$

0	not used	1	2	3	4	5	6	7	8	9	10	11	12	13
---	----------	---	---	---	---	---	---	---	---	---	----	----	----	----

O caminho para a raiz a partir de um índice  $i$  pode ser determinado.

Exemplo: cálculo do caminho para a raiz para um heap de índice raiz 1 e  $m=2$  e/2 n elementos. O caminho é longo é  $1 + \log_2 n$  que tem complexidade computacional  $O(\log n)$ .

$$\left[ \frac{i}{2^2} \right], \left[ \frac{i}{2^3} \right], \dots, \left[ \frac{i}{2^K} \right], \text{ onde } K = \lceil \log_2 i \rceil$$

### Operações

Estas estruturas de dados, suportam as operações de criação/destruição, inserção do elemento maior (raiz) e inserção/remoção do elemento.

A **inserção** de um elemento obriga à verificação de que os seus índices "pai" cumprem os critérios da heap, trocando de posição/c se estes caso contrário. Este processo tem complexidade computacional  $O(\log n)$ .

A **remoção** de um elemento, obriga à substituição deste pelo seu "filho" de valor superior de forma recursiva, até não dar mais, quando substituirmos pelo último elemento. Tem complexidade computacional  $O(\log n)$ .

## • PRIORITY QUEUES

Deve implementar, para além das opções de criação e destruição, os seguintes métodos:

- **peek**, descoberta do maior elemento
- **enqueue**, inserção do elemento
- **dequeue**, remoção de elemento

Segue uma filosofia diferente das queues, pois numa vez de se remover o mais antigo/recente, **remove-se o maior**. Pode ser implementada à max\_heap.

## • ÁRVORES BINÁRIAS

São uma estrutura de dados dinâmica, composta por nós, cada um c/ a seguinte informação:

- ponteiro para o nó esquerdo/direito
- um ponteiro para o nó "pai"

cujo valor será menor/maior - se não existir nó de um dos lados, o ponteiro será NULL