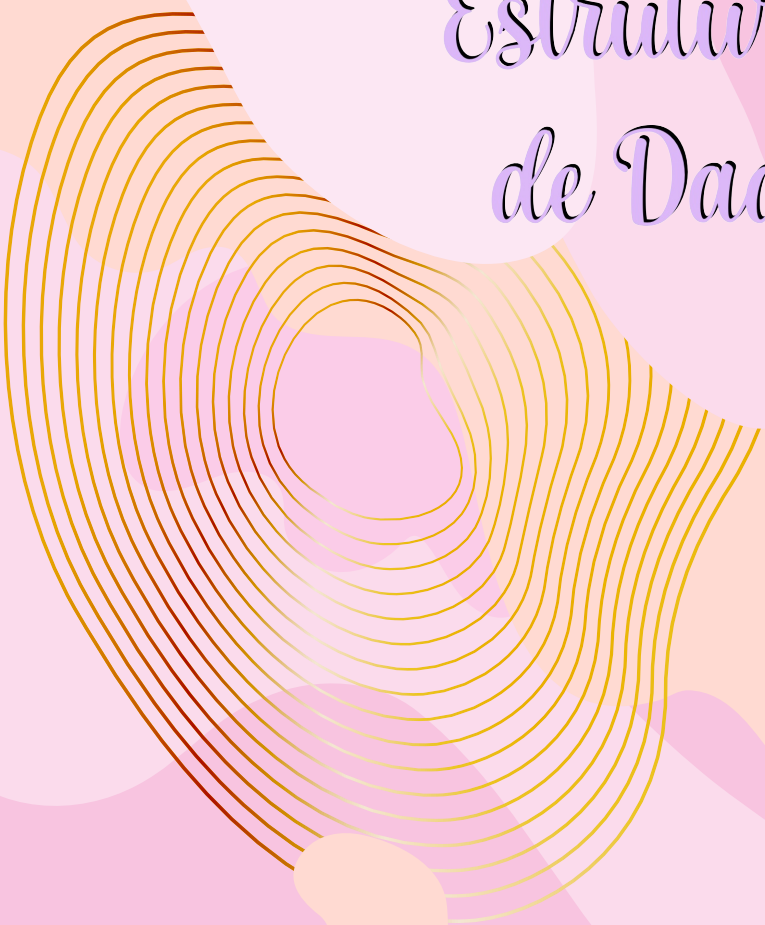
Algoritmos e Estruturas de Dados



Algoritmo: método para resolver um problema computacional
↳ caracterizado pela descrição do seu input e output

Correção: Tem de resolver corretamente todas as instâncias do problema
\+
Eficiência: A performance (de tempo e memória) tem de ser adequada

Loop Invariant

- Capta o significado semântico dos loops (lógica e intuição)
- Condição que é necessariamente verdadeira imediatamente antes e depois de cada iteração do loop

Passos Prova ↓

Inicialização: Invariante é ✓ antes da 1ª iteração

Maintenance: Se é ✓ antes de uma iteração, continua ✓ antes da próx. iteração

Terminação: Quando loop termina, o invariante fornece uma propriedade que nos ajuda a mostrar que o algoritmo está correto

Progresso: Cada iteração aproxima-nos do final, até terminar

Algoritmo melhor num Computador + lento **>** Algoritmo pior num Computador + Rápido

O que se pode fazer com a análise do tempo de execução?

Previsão - Tempo/espaço necessário, Como escala? (ou memória)
\+
Comparação - Existe algum algoritmo melhor para resolver o problema?

Random Access Machine → modelo genérico e independente da linguagem e da máquina

$O(n)$ - Limite Superior

$\Omega(n)$ - Limite inferior

$\Theta(n)$ - Limitado superior e inferiormente

Estimativa de tempo de execução

$$time(n_2) = \frac{f(n_2)}{f(n_1)} \times time(n_1)$$

Nota:

$n! \geq 2^n \geq n^3 \geq n^2 \geq n\log n \geq n \geq \log n \geq 1$

- $\sqrt{n}$ pior que $\log_2 n$

Dividir: o problema em instâncias + pequenas

Conquistar: os sub-problemas com recursão

Combinar: as soluções dos sub-problemas numa solução para o original

Templates
- forma de criar código genérico
- são extendidos no tempo de compilação
- praticamente não têm diferenças semânticas com classes

Algoritmos de Pesquisa

- Pesquisa Sequencial
  - Complexidade Temporal: $O(n)$
  - Complexidade Espacial: $O(1)$
- Pesquisa Binária
  - Complexidade Temporal: $O(\log n)$
  - Complexidade Espacial: $O(1)$
    - Como vetores são passados por referência, o espaço das variáveis locais é constante

Nota. Para comparar classes
- overload operador == (etc.)

Sorting

- Algoritmos Comparativos

		Complexidade Temporal
InsertionSort, SelectionSort, BubbleSort, ShellSort	$O(n^2)$	
MergeSort	$O(n \log n)$	$O(n)$
QuickSort	$O(n \log n)$	$O(\log n)$ pior: $O(n^2)$
HeapSort	$O(n \log n)$	

+ eficiente exceto vetores pequenos: InsertionSort

- Algoritmos não comparativos

CountixSort	$O(n+K)$	K = máx
RadixSort	$O(d \times (n+K))$	d = máx. nº dígitos

! Algoritmo de sort é estável se deixa elementos com = chave pela ordem original

Algoritmo ordenação em sort() = IntroSort = (Quick + Heap) + Insertion

Tipos de Dados Abstratos (ADTs)

ABSTRAÇÃO

- **Processual**
  - Abstração dos detalhes dos procedimentos
  - Especificação é a abstração
    - Satisfaz-se com a implementação
  - Uso do procedimento depende do seu propósito → ~~implementação~~
- **Data**
  - Abstração dos detalhes da representação de dados
  - Mecanismo de especificação

ADT

- Abstrai de organização → significado e estrutura → uso
- Representação não deve importar ao utilizador → esconder
- Suporta abstração, encapsulação e ocultação de informação

Operações

- **Creators**: Criar novos objetos → + comuns em ADT's imutáveis
- **Producers**: Criar novos objetos a partir de outros do mesmo tipo
- **Observers**: Recebe abstract type e retorna objetos de tipo diferente
- **Mutators**: Modificar Objetos

Nota: Guardar um objeto mutável numa coleção imutável pode expor a representação

Lista

- Sequência de elementos do mesmo tipo
- Array-based: pesquisa, inserção e remoção: $O(n)$
- Linked list: inserção e remoção: $O(1)$ | pesquisa: $O(n)$
- Pode usar o find() mas não o sort() de STL

Iterador

Iterador: objeto que referencia um elemento de um certo ADT

~~apontador~~

Exemplo:

```
vector<int> :: iterator it;
list<int> :: iterator lit;
```

:: → namespace

≠ implementações

Nota: it -- não é um requisito da classe iterador!

Stack LIFO ≠ **Queue** FIFO

- Caso particular das listas
- Usa-se por motivos de eficiência
- Não é requisito ter um iterador associado (nas listas/vetores é)

Nota: Deque = double-ended queue

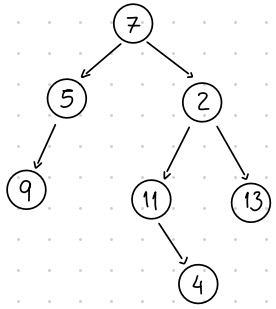
Árvores Binárias

- Depth of a node: Comprimento da raíz ao node (depth root = 0)
- Height of a node: Comprimento do node à folha + profunda
- Size of a node: Número de descendentes (descendente)

Propriedades:

- Tree with depth h has: Min → h+1 nodes | Máx → 2^{h+1} nodes
- Depth of tree with n elements: Min → $\log_2 n$ | Máx → n-1
- Depth média de uma árvore de n nodes é $\sqrt{n}$

Traversals:



by Level: 7, 5, 2, 9, 11, 13, 4

Nó, esq, dir (preorder): 7, 5, 9, 2, 11, 4, 13

Esq, dir, nó (posorder): 9, 5, 4, 11, 13, 2, 7

Esq, nó, dir (inorder): 9, 5, 7, 11, 4, 2, 13

Árvore Binária de Pesquisa

- Árvore binária, sem duplicados
- Dado um node: todos os seus filhos esquerdos são inferiores a ele; todos os seus filhos direitos são superiores a ele
- Todas as operações podem ser realizadas em O(n)
  - se árvore balanceada → Todas as folhas têm de estar ao mesmo nível ± 1

Remoção: Substitui-se:

- ou pelo **maior** dos filhos **esquerdos**
- ou pelo **menor** dos filhos **direitos**

! Se usarmos **inorder** (esq, nó, dir) os elementos ficam ordenados por ordem crescente

A classe que estamos a usar (**set**) é uma

Vermelha - Preta bool em vez de int (↓ AVL) e gasta - espaço

- Raiz é preta
- Todo o nó vermelho tem filhos pretos
- Nº de nós de cor preta é igual em todos os ramos

Nota:

Sempre que usamos estruturas de dados ordenadas temos de ter o operador **<** implementado ou função de comparação