IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal


# Projeto de ATAD

## Importação, Normalização e Classificação de dados

Engenharia Informática

Alexandre Coelho, nº190221093

Sérgio Veríssimo, nº190221128

# Índice

## a) ADTs utilizados

Os ADT utilizados neste projeto foram ADT Map e ADT List. O ADT List foi utilizado para conter os dados relativos a pacientes e o ADT Map foi utilizado para conter os dados relativos a região. Esta implementação trata-se de uma obrigatoriedade na execução do projeto, e até foi discutido por um membro do nosso grupo com o professor sobre a troca de implementações (ADT List para regiões e ADT Map para pacientes). Esta proposta de troca de implementações, não faria muito sentido devido a diversos fatores, como a necessidade de ligação nos dados de regiões (principalmente nas chaves) e também devido à complexidade algorítmica que ambos os ADT tem (anexado abaixo).

| Implementação | listAdd | listRemove | listGet | listSet |
|---|---|---|---|---|
| Array List | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |

**Figura 1- Complexidade algorítmica na ADT List utilizada**

| Implementação | mapPut | mapRemove | mapGet | mapContains |
|---|---|---|---|---|
| Array List | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |

**Figura 2 - Complexidade algorítmica no ADT Map utilizado**

Sendo a complexidade algorítmica de todas operações de ADT Map (com Array List) lineares, não daria muito jeito aplicar nos registos de pacientes, que são e serão também de tamanho linear (pois pode ser acrescentado um número muito maior de pacientes com o decorrer desta epidemia). Este argumento prevalece o facto de também existir um identificador na lista de pacientes que certamente serviria como uma chave sólida (não existe repetição, por isso, única). Os métodos utilizados estão todos presentes nos "*Abstract Data Types*" presentes no repositório do GitHub, do docente Bruno Silva. (https://github.com/brunomnsilva/AbstractDataTypesInC).

# b) Complexidade algorítmica dos comandos implementados

Como já fora documentado a complexidade algorítmica dos tipos abstratos de dados anteriormente, agora passaremos à demonstração das complexidades algorítmicas nos algoritmos que utilizamos para interpretar os comandos requisitados.

Começando pelo mais extenso, os **comandos B**:

## 1. AVERAGE

O algoritmo abordado para a conceção deste comando, possui uma complexidade linear (***O(n)***), composta por um **ciclo for** está apenas condicionado pelo tamanho da lista de pacientes. Dentro deste **ciclo for** encontra-se um **listGet**, que contem uma complexidade algorítmica constate (como constatado na figura 1), não alterando a complexidade linear do comando (pondera-se a pior complexidade).

## 2. FOLLOW

Quanto ao comando "***Follow***" foi utilizada recursividade não linear (pois encontra-se dentro de um ciclo) e também o uso obrigatório do ADT List e dos seus métodos (principalmente listSize e listGet). A complexidade algorítmica apenas depende do tamanho da lista, pois não é utilizado enésimo na invocação da função recursivamente, ou seja, obtemos uma complexidade linear (***O(n)***).

## 3. SEX

Este comando, também apenas contém um **ciclo for** que percorre a lista e obtém informação sobre os pacientes através do **listGet**. Da mesma forma que os comandos anteriores, este comando vai depender do tamanho da lista e mantém a complexidade linear (***O(n)***), mesmo possuindo a complexidade constante no **listGet** (novamente, pondera-se a pior complexidade).

## 4. SHOW

Neste comando, contemos uma complexidade quadrática (***O(n²)***), devido a ser constituído por um ciclo for principal, que possui outro **ciclo for** encadeado (presente no método DateDiff dos métodos auxiliares), ambos condicionados por variantes distintas lineares. Também possui um listGet (***O(1)***), mas este não influencia, pois pondera-se a pior complexidade.

## 5. TOP5

O algoritmo abordado para a realização deste comando, representa talvez um dos que possivelmente possui uma complexidade algorítmica pior. Contém um duplo selection sort (um dependente do tamanho da lista e outro com número constante, 5 (top **five**)). Esta abordagem tem em consideração uma ordenação da lista toda, e posteriormente uma ordenação do nosso top five. Para a impressão dos valores, também é utilizado um ciclo for para imprimir os resultados, mas com um tamanho constante (como no segundo selection sort). A complexidade algorítmica deste comando é quadrática (***O(log n)***), devido a existir a existir uma pesquisa na lista de pacientes, baseada num campo especifico. Ou seja, não é necessário percorrer a lista toda para verificar o nosso top **five**, pois existem diversos campos que aumentam a exclusão e também ainda existe uma ordem especifica (como referido anteriormente), através de selection sorts.

## 6. OLDEST

O comando "***Oldest***" constitui-se de diversos ciclos **for** (sem encadeamento), estando dependentes apenas do tamanho da lista (listSize). Existem também listGet (complexidade algorítmica constante), mas não influenciam a complexidade algorítmica deste comando, que é uma complexidade algorítmica linear (***O(n)***).

7. GROWTH

Este comando, constitui-se de uma complexidade algorítmica linear ***O(n)***, pois apenas contém um ciclo que está dependente do tamanho da lista. Aqui também é invocado o método **listGet**, mas não influencia a complexidade algorítmica.

8. MATRIX

O algoritmo abordado para este comando possui uma complexidade algorítmica linear (***O(n³)***), devido a ser um produto de matrizes. É composto também por listGet, mas este não influencia a complexidade algorítmica, pois considera-se a pior complexidade.

Seguidamente, os **comandos A** (exceto QUIT e CLEAR):

1. LOADP

Este comando, que tem o intuito de criar pacientes com informação presente num ficheiro CSV, contém apenas um ciclo que está dependente das linhas de conteúdo que o ficheiro em questão possui. Dentro deste ciclo também é executado listAdd, que é composto por uma complexidade linear. Aqui temos duas complexidades lineares que embora estejam encadeadas, não estão dependentes uma da outra (por exemplo o listAdd é efetuado com a mesma complexidade, independentemente do tamanho da lista). A complexidade deste algoritmo é linear ***O(n).***

2. LOADR

Semelhante ao outro comando, dos comandos A, o algoritmo implementado, também contém um ciclo que está dependente das linhas de conteúdo que o ficheiro CSV (regions.csv) tem. Dentro deste ciclo também existe a operação MapPut que contém uma complexidade linear, mas da mesma forma que o listAdd, é realizada independentemente da linearidade do conteúdo do ficheiro. A complexidade deste algoritmo também é linear ***O(n).***

Por fim, possuímos os **comandos C**:

1. REGIONS

Este comando, tem o intuito de mostrar uma lista de regiões por ordem alfabética, que tem pessoas ainda doentes. Os primeiros dois ciclos for são encadeados, mas não possuem uma dependência de complexidade entre eles, sendo cada um ***O(n).*** O segundo conjunto de ciclos for encadeados (bubbleSort), já possuem uma dependência entre eles elevando a complexidade algorítmica deste comando para ***O(n²).*** Como escolhemos o pior cenário a complexidade algorítmica deste comando é ***O(n²).***

2. REPORT

Este comando, tem o intuito de criar um ficheiro report.txt e guardar neste a taxa de mortalidade e taxa de incidência, total e por região. Este possui vários ciclos for encadeados, mas que não possuem uma dependência de complexidade entre eles, sendo cada um ***O(n).*** Como escolhemos o pior cenário a complexidade algorítmica deste comando é ***O(n).***

## c) Pseudocódigo de 3 funcionalidades

### 1. AVERAGE

**Algorithm Average**

**Input**: listPatients – PtList pointer

**BEGIN**

Year <- getCurrentDate().year

patient <- NULL

ptSize <- 0

sizeErrorCode <- listSize(*listPatients,&ptSize)

**IF** sizeErrorCode = LIST_FULL **OR** sizeErrorCode = LIST_INVALID_RANK **OR** sizeErrorCode = LIST_NO_MEMORY **OR** sizeErrorCode = LIST_NULL **THEN**

    **PRINT** "An error occurred… Please try again…"

    **RETURN**

**END IF**

sumDeceasedPatients <- 0.0

sumReleasedPatients <- 0.0

sumIsolatedPatients <- 0.0

counterIsolated <- 0

counterDeceased <- 0

counterReleased <- 0

**FOR** i **IN** 0 **TO** ptSize **DO**

    listGet(*listPatients,i,&patient)

    **IF** patient.status = "isolated" **AND** patient.birthYear > 0

    sumIsolatedPatients <- sumIsolatedPatients + year – patient.birthYear

    counterIsolated <- counterIsolated + 1

**ELSE IF** patient.status = “deceased” **AND** patient.birthYear > 0

sumDeceasedPatients <- sumDeceasedPatients + year – patient.birthYear

counterDeceased <- counterDeceased + 1

**END IF**

**IF** patient.status = “released” **AND** patient.birthYear > 0

sumReleasedPatients <- sumReleasedPatients + year – patient.birthYear

counterReleased <- counterReleased + 1

**END IF**

**ENDFOR**

**PRINT** “Average age for deceased patients: $sumDeceasedPatients/$counterDeceased”

**PRINT** “Average age for released patients: $sumReleasedPatients/$counterReleased”

**PRINT** “Average age for isolated patients: $sumIsolatedPatients/$counterIsolated”

**END**

2. SEX

**Algorithm Sex**

**Input**: listPatients – PtList pointer

patient <- NULL

ptSize <- 0

sizeErrorCode <- listSize(*listPatients,&ptSize)

**IF** sizeErrorCode = LIST_FULL **OR** sizeErrorCode = LIST_INVALID_RANK **OR** sizeErrorCode = LIST_NO_MEMORY **OR** sizeErrorCode = LIST_NULL **THEN**

**PRINT** “An error occurred… Please try again…”

**RETURN**

**END IF**

```
counterFemales <- 0

counterMales <- 0

counterUnknown <- 0

FOR i IN 0 ptSize TO ptSize DO

        listGet(*listPatients,I,&patient)

        IF patient.sex = “male”

                counterMales = counterMales + 1

        ELSE IF patient.sex = “female”

                counterFemales = counterFemales + 1

        ELSE

                counterUnknown = counterUnknown + 1

END IF

END FOR

percentageMale <- 0.0

percentageFemale <- 0.0

percentageUnknown <- 0.0

percentageMale <- counterMales/ptSize * 100

percentageFemale <- counterFemales/ptSize * 100

percentageUnknown <- counterUnknown/ptSize * 100

PRINT “Percentage of Females: $percentageFemale”

PRINT “Percentage of Males: $percentageMale”

PRINT “Percentage of Unknown: $percentageUnknown”

PRINT “Total of patients $ptSize”
```

3. REGIONS

**Algorithm regions**

**Input:** [*mapRegions – PtMap, *listPatients – PtList]

**BEGIN**

```
size <- 0;
tempReg <- null;
patient <- null;
*mapKeyArray = mapKeys(*mapRegions);
ptListSize <- 0;
listSizeErrorCode <- listSize(*listPatients, &ptListSize);
regionArray[size];
FOR i <- 0 TO (i < size) DO
        mapGet(*mapRegions, mapKeyArray[i], &tempReg);
        FOR j <- 0 TO (j < ptListSize) DO
                listGet(*listPatients, j, &patient);
                IF (strcmp(patient.region, mapKeyArray[i].region) = 0 AND
                strlen(patient.region) > 0) THEN
                        IF (isIso(patient)) THEN
                                regionArray[i] <- tempReg;
                        END IF
                END IF
        END FOR
END FOR
FOR j <- 0 TO (j < size) DO
        FOR i <- j + 1 TO (i < size) DO
                IF (strcmp(regionArray[j].name, regionArray[i].name) > 0) THEN
                        tempReg <- regionArray[j];
                        regionArray[j] <- regionArray[i];
                        regionArray[i] <- tempReg;
                END IF
        END FOR
END FOR
```

```
    FOR i <- 0 TO (i < size-1) DO
        PRINT "$regionArray[i]"
    END FOR
    free(mapKeyArray)
END
```

# d) Limitações

Durante o processo de desenvolvimento do projeto existiram limitações que potencialmente obstruíram a obtenção de uma nota melhor. Estas limitações compõem-se principalmente pela falta de pensamento critico no desenvolvimento inicial dos algoritmos dos comandos. Este desenvolvimento seguiu uma linha de pensamento diferente do que a seria expectável (pseudocódigo -> complexidade algorítmica -> código c).

A falta de resultados de exemplo nos comandos c não permitiram uma validação correta dos comandos assertiva dos comandos em si.

# e) Conclusões

Certamente o pensamento critico no desenvolvimento de algoritmos relacionados com problemas reais teria de ser outro, e é algo que este projeto fortaleceu. A maneira de conseguir pegar num problema e arranjar a melhor solução, deixa de ser um complemento, e começa a ser mais uma obrigatoriedade. Ambos os membros do grupo somos provenientes de um curso anterior que era baseado em *Project Based Learning* e que consistia num "ataque de olhos vendados" ao problema, com base numa programação direta, sem pensar muito na lógica do projeto em si.

A abstração e a sua implementação também foram bastante retidas durante o desenvolvimento deste projeto e formaram alicerces (claramente, em conjunto com a componente teórica pratica) para uma utilização futura, numa situação profissional, tanto na linguagem abordada nesta Unidade Curricular, como noutra que nos seja pedida.