# PROJETO ALGORITMOS E TIPOS ABSTRATOS DE DADOS

IPS – LEI 19/20

Francisco Zacarias nº190221105

João Rosete nº190221109

Turma: 1ºL_EI-01

Docente Laboratório: Patrícia Macedo

# Contents

Introdução ..... 2
Tipos Abstratos Utilizados ..... 2
List ..... 2
Map ..... 2
Queue ..... 2
*ArrayList vs LinkedList* ..... 3
Módulos e Makefile ..... 4
Módulos ..... 4
*Makefile* ..... 4
Complexidades Algorítmicas ..... 5
Implementação em pseudo-código ..... 6
Funcionalidade 1 ..... 6
Funcionalidade 2 ..... 7
Funcionalidade 3 ..... 8
Limitações ..... 9
Conclusão ..... 9

# Glossário

TAD – Tipo abstrato de dados

*FIFO – First in, first out (*primeiro a entrar, primeiro a sair*)*

# Introdução

Relatório relativo ao projeto da cadeira de Algoritmos e Tipos Abstratos de dados, do curso de Engenharia Informática do Instituto Politécnico de Setúbal. Neste relatório vão constar todos os pontos pedidos no enunciado do projeto, assim como a nossa reflexão após ter realizado o projeto.

Este projeto tem com base dois ficheiros, em formato *.csv,* que devem ser exportados para os tipos abstratos designados, gerir toda a memória utilizada pelos *TAD's* e realizar diversas operações sobre os dados. Nesta entrega do projeto, acreditamos ter cumprido todos os objetivos propostos para o projeto, estes que serão desenvolvidos ao longo deste relatório.

# Tipos Abstratos Utilizados

Tipos abstratos de dados especificam um comportamento de um tipo de dados, não primitivo. Definem as suas operações, os seus códigos de erro, e expõem a sua interface para o utilizador. A sua implementação é independente da sua utilização, isto é, o utilizador tem acesso à assinatura da função, sabe o que faz, mas não tem acesso (ou não é o objetivo) à sua implementação.

Iremos falar de cada tipo abstrato de dados utilizado no projeto, e apenas no fim do capítulo, explicar qual a implementação que utilizamos, e justificar as nossas conclusões, tanto com a teoria da utilização dos TADs, como uma demonstração concreta do impacto no nosso projeto.

## List

O tipo abstrato de dados *List* (lista), é uma coleção cuja política de acesso é baseada em *rank/índex,* permite repetição de elementos e cuja ordem destes, é relevante para o seu funcionamento.

Para o contexto do projeto, este tipo abstrato de dados foi utilizado para conter a lista de todos os pacientes importados do ficheiro fornecido no enunciado. Esta lista guarda uma estrutura *Patient* que representa cada paciente importado.

## Map

O tipo abstrado de dados *Map* (mapa), é uma coleção cuja política de acesso é baseada em chave/valor, isto é, ao dar uma chave, o mapa devolve o valor associado à chave dada. Não permite repetição de elementos, visto um elemento possuir apenas uma chave e a ordem dos mesmos não interessa.

Para o contexto do projeto, este tipo abstrato de dados foi utilizado para conter toda a informação relativa ás regiões importadas do ficheiro fornecido no enunciado. Como chave, foi guardada uma *string* que representa a região, e como valor foi guardada a respetiva estrutura que representa essa região.

## Queue

O tipo abstrado de dados *Queue* (fila) é uma coleção cuja política de acesso é designada por *FIFO,* ou seja, primeiro a entrar, primeiro a sair, que permite a repetição dos seus elementos, e cuja ordem interessa para o seu funcionamento.

Este tipo abstrato de dados não foi pedido no enunciado especificamente, mas nós utilizamos para desenvolver o comando *oldest,* pedido no projeto.

## *ArrayList vs LinkedList*

Para a implementação final do projeto, decidimos que a implementação do mapa e da lista deviam ter como base uma *ArrayList.* Após analisar o enunciado, percebemos que a maior parte (se não todos) os comandos para implementar, apenas precisavam de aceder aos elementos (*listGet, mapGet*). Estas operações podem ser feitas em tempo constante, utilizando uma *arraylist.* Apenas consideraríamos uma *LinkedList* caso os comandos tivessem uma forte ênfase na inserção e remoção de elementos.

Com esta teoria, implementámos o projeto utilizando os tipos abstratos de dados com implementação de uma *ArrayList.* Contudo, quando concluímos o projeto, decidimos testar o tempo que o projeto demora a correr com as diferentes implementações. Para isto, criamos um ficheiro que se encontra em **test/testProject.c** que, essencialmente, corre todas os comandos do projeto (inclusivamente o *load* dos pacientes e regiões) e testamos o tempo com as diferentes implementações. Correrms 5 vezes e calculamos a média dos tempos (para aumentar a precisão):

| | MAPA (*ARRAYLIST)* | MAPA (*LINKEDLIST)* |
|---|---|---|
| LISTA (*ARRAYLIST*) | 0.0198 segundos | 0.0204 segundos |
| LISTA (*LINKEDLIST)* | 1.3841 segundos | 1.4200 segundos |

Como podemos observar na tabela anterior, as implementações dos TAD's, beneficia bastante da utilização de *ArrayLists,* face às *LinkedLists*. Podemos observar que a diferença entre usar uma *ArrayList* ou *LinkedList* no mapa não faz grande diferença nos tempos. Acreditamos que isto é devido à dimensão do projeto e às poucas operações que são feitas com o mapa. Caso as operações de leitura fossem superiores, iriamos ver tempos mais elevados na implementação do mapa com *LinkedList.* Isto fez com que optássemos por utilizar, exclusivamente, implementações com *ArrayLists,* para o contexto deste projeto.

# Módulos e Makefile

Neste capítulo vamos abordar a organização do projeto, nomeadamente os módulos criados, e as pastas do projeto.

## Módulos

O projeto tem todos os módulos e implementações na pasta principal. Existem duas pastas dentro do projeto. A pasta *lib* que contém os TAD's utilizados no projeto e a pasta *test* que contém testes para cada módulo e um teste para o projeto inteiro. Achamos por bem incluir isto no projeto, pois foi uma boa ferramenta de auxílio para garantir que implementamos corretamente o projeto, e testar o mesmo por completo (inclusivamente fazer as comparações entre implementações de *ArrayLists e LinkedLists*).

Em cada módulo, existem as implementações das funções especificas a cada módulo, assim como funções estáticas de auxílio imediato para qualquer função de um dado módulo. Por função estática, compreendemos ser uma função que só existe no escopo do ficheiro onde é implementado

Todos os módulos contêm um ficheiro com a sua interface (*module.h*) com a sua assinatura e a documentação *doxygen,* e um ficheiro com a implementação da interface (*module.c*):

- *Command*: Contém a implementação de todos os comandos pedidos no enunciado do projeto.
- *Date*: Define um tipo de dados com o objetivo de guardar datas e definir operações associadas.
- *Patient*: Define um tipo de dados com o objetivo de guardar todos os pacientes importados e definir operações associadas.
- *Region*: Define um tipo de dados com o objetivo de guardar todas as regiões importadas e definir as operações associadas
- *Utils*: Contém a implementação de operações que são usadas para auxílio do projeto, mas que não pertencem a um escopo específico.

## *Makefile*

O projeto também contém um *makefile* com várias *flags* que permitem compilar o projeto na sua integra (*all),* assim como *flags* que permitem correr os vários testes aos módulos do projeto.

As *flags* seguintes permitem compilar os testes que fizemos aos módulos: *testDate, testRegion, testPatient* e *testUtils*. Temos também duas *flags* que compilam o programa que testa todos os comandos do projeto (*testProject*) e uma flag que compila o teste, mas utilizando implementações de *LinkedLists (testProjectLinked).* Para compilar o projeto como ele é suposto ser utilizado, utilizamos a *flag all.*

# Complexidades Algorítmicas

Neste capítulo vamos dizer, e justificar, quais são as complexidades algorítmicas de todos os comandos que implementamos para o projeto:

| Função | Complexidade Algorítmica | Justificação |
|---|---|---|
| loadp | O(n) | Chama a função *loadPatientsIntoList* que vai executar *n* instruções, sendo que *n* é o número de linhas que o ficheiro *.csv* contém. |
| loadr | O(n) | Chama a função *loadRegionsIntoMap* que vai executar *n* instruções, sendo que *n* é o número de linhas que o ficheiro *.csv* contém. |
| matrix | O(n) | Executa *n* instruções, em que *n* é o tamanho da lista. Os restantes ciclos são constantes, e as funções chamadas, tem complexidade constante. |
| show | O(n^2) | No pior dos casos, para a função calcular o número de dias de um paciente doente, terá que encadear dois ciclos, onde ambos dependem dos dados do paciente. |
| average | O(n^2) | Executa *n* instruções, em que *n* é o tamanho da lista. Dentro do ciclo, é chamada a função *strcmp,* cuja complexidade, no pior dos casos, é linear. |
| sex | O(n^2) | Executa *n* instruções, em que *n* é o tamanho da lista. Dentro do ciclo, é chamada a função *strcmp,* cuja complexidade, no pior dos casos, é linear. |
| top | O(n^3) | Executa *n* instruções em que *n* é o tamanho da lista. Dentro deste ciclo, é feita uma ordenação quadrática em cada iteração, logo O (n * n^2) |
| growth | O(n) | Executa *n* instruções em que *n* é o tamanho da lista. |
| follow | O(n^2) | Executa *n* instruções, em que *n* é o número de infeções encadeadas entre cada paciente. Dentro de cada iteração, é também iterado o tamanho da lista. |
| oldest | O(n^2) | Executa *n* instruções, em que *n* é o tamanho da lista. Dentro do ciclo, é chamada a função *strcmp,* cuja complexidade, no pior dos casos, é linear. |
| regions | O(n^3) | Executa *n* instruções, em que *n* é o tamanho do mapa. Dentro do ciclo, é chamada a função *isRegionInfected* que, no pior dos casos, itera todos os elementos do mapa. Dentro deste ciclo, é chamada a função *strcmp.* Logo O(n*n*n) |
| report | O(n^3) | Executa *n* instruções, em que *n* é o tamanho da lista. Dentro do ciclo, é chamada a função *stats,* que itera todos os elementos do mapa. Dentro deste ciclo, é chamada a função *strcmp.* Logo O(n*n*n) |

## Implementação em pseudo-código

### Funcionalidade 1

Tipo B – Função “sex”:

```
Algorithm sex
        input: list - Pointer to list
        output: void
BEGIN
        FOR i <- 0 TO list_size DO
                patient <- listGet(patient, i)
                IF patient->name == "male" THEN
                        male += 1
                ELSE IF patient->name == "female" THEN
                        female += 1
                ELSE
                        unknown += 1
                END IF
        END FOR
        print(male, female, unknown)
END
```

## Funcionalidade 2

Tipo B – Função "follow":

```
Algorithm follow
        input: list - Pointer to list, id - long int
        output: void
BEGIN
        patient <- listGet(patient, i)
        infectedBy <- patient->infectedBy
        DO
                FOR i <- 0 TO list_size DO
                        patient <- listGet(patient, i)
                        IF patient->id == infectedBy THEN
                                print(patient)
                                infectedBy == patient->infectedBy
                                BREAK
                        END IF
                        patient <- NULL
                END FOR
        WHILE(patient != NULL)
END
```

## Funcionalidade 3

Tipo C – Função "regions"

```
Algorithm follow
        input: list - Pointer to List, map - Pointer to map
        output: void
BEGIN
        keys <- mapKeys()
        index <- 0
        FOR i <- 0 TO map_size DO
                region <- mapGet(keys[i])
                IF isRegionInfected(region->name, list) THEN
                        infected_regions[index++] <- region
                END IF
        END FOR
        orderRegionsList(infected_regions, index)
        FOR i <- 0 TO index DO
                print(infected_regions[i])
        END FOR
END
```

## Limitações

Acreditamos não haver limitações nenhumas que vão contra qualquer parâmetro do enunciado.

- Todos os comandos aparentam estar de acordo com o ficheiro *results.txt* fornecido, e os comandos cujo output não pudemos confirmar, também acreditamos terem sido implementado corretamente.
- Não aparenta existir qualquer tipo de *memory leak* no projeto. A memória é libertada, independentemente se é utilizado o comando *clear* ou não. Caso o utilizador não utilize o comando *clear,* o código verifica se as estruturas de dados não estão vazias e caso haja memória por libertar, esta é libertada.
- Temos duas possíveis limitações, no que está relacionado com o contexto deste projeto:
    - A implementação dos comandos não ter a complexidade algorítmica mais baixa possível. Isto foi algo que tivemos consciência enquanto os implementávamos e mantém-se a única incógnita para nós, no que trata a limitações.
    - Os comandos que são executados com parâmetros adicionais (*show, follow, growth*), não recebem o parâmetro como especificado no projeto E.g. show {ID}. Neste projeto, para passar um parâmetro é preciso introduzir o comando, clicar no enter, e só depois é que é solicitada a inserção do parâmetro, isto é, são necessários 2 passos, em vez de só um.

## Conclusão

Foi um projeto que nos educou bastante e deu-nos uma sensibilidade superior para tudo o que envolve gerir a memória de um programa, manuseamento de ponteiros e a sua importância e para qualquer linguagem, às quais estes conceitos são abstratos. Concluímos este projeto com uma nota pessoal positiva, pois acreditamos ter atingido os objetivos da cadeira, independentemente da nota do projeto.