

---

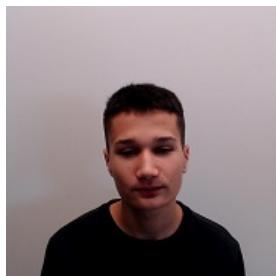
# Laboratórios de Informática III

---

TRABALHO REALIZADO POR:

BENJAMIM MELEIRO RODRIGUES  
XAVIER SANTOS MOTA  
PEDRO DANTAS DA CUNHA PEREIRA

GRUPO 46



A93323  
Benjamim Rodrigues



A88220  
Xavier Mota



A97396  
Pedro Pereira

---

# Índice

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Estruturação</b>	<b>3</b>
2.1	<i>Catálogo de Users</i>	3
2.2	<i>Catálogo de Drivers</i>	3
2.3	<i>Catálogo de Rides</i>	4
<b>3</b>	<b>Queries</b>	<b>6</b>
3.1	<i>Query I</i>	6
3.2	<i>Query II</i>	6
3.3	<i>Query III</i>	6
3.4	<i>Query IV</i>	7
3.5	<i>Query V</i>	7
3.6	<i>Query VI</i>	7
3.7	<i>Query VII</i>	7
3.8	<i>Query VIII</i>	8
3.9	<i>Query IX</i>	8
<b>4</b>	<b>Menu Interativo</b>	<b>9</b>
<b>5</b>	<b>Testes e Desempenho do Programa</b>	<b>9</b>
5.1	Tempos de Execução nas Diferentes Máquinas	9
5.2	Discussão de Resultados	10
<b>6</b>	<b>Encapsulamento e Modularidade</b>	<b>10</b>
<b>7</b>	<b>Conclusão</b>	<b>10</b>

---

## 1 Introdução

No âmbito da unidade curricular de Laboratórios de Informática III foi-nos proposto, o desenvolvimento de uma aplicação com determinadas propriedades e funcionalidades descritas no enunciado do próprio trabalho, disponibilizado pela equipa docente da UC. Estas funcionalidades giram em torno de ficheiros de dados que contém diferentes informações relacionadas a *users*, *drivers* e *rides*.

Após a primeira fase, na qual desenvolvemos os catálogos necessários para a realização do trabalho, o método de *parsing* para os dados de entrada e 3 das 9 *queries* estipuladas para o trabalho, passamos para a realização das tarefas da segunda fase, sendo estas as seguintes:

- Implementação das 9 *queries* na sua totalidade
- Implementação de um modo interativo para o nosso programa
- Validação do formato dos ficheiros de dados conforme as estipulações feitas no enunciado
- Análise do desempenho da solução desenvolvida

Tal como nos foi inicialmente dito, para esta fase houve um aumento na ordem de grandeza dos ficheiros de entrada, pelo que efetuamos alterações ao código de modo a mantê-lo eficiente e viável face aos novos ficheiros de entrada.

Tendo em conta o propósito da UC, bem como o grande peso das componentes relacionadas à modularização e encapsulamento de dados, estes foram aspectos aos quais demos foco ao longo da realização de todo o trabalho, respeitando-os sempre e encontrando soluções que estivessem em sintonia com estes conceitos.

---

## 2 Estruturação

De forma a responder aos *Datasets* de tamanho muito superior desta fase decidimos mudar as estruturas já existentes usadas para guardar os catálogos de *Drivers*, *Users* e *Rides*.

Dividimos também o validator.c, de forma a melhorar a modularização do trabalho, criando o date.c, responsável pelas funções relativas a datas.

Em relação às datas deixamos de usar a *struct tm* do header time.c e passamos a usar um int, com o objetivo de diminuir a memória usada por cada data. Para isso criamos uma função que guarda cada elemento dumha data (dia, mês e ano) em bytes diferentes do int.

Removemos do catalog.c as funções usadas para imprimir as respostas das *queries* para o terminal ou para um ficheiro, dependendo do modo de execução do programa, para print\_outputs.c de forma a modularizar ainda mais o nosso trabalho.

### 2.1 Catálogo de *Users*

```
struct catalog_users {
    GHashTable *users_hash;
    User** biggestDistance_array;
    int size, used;
};
```

Fig. 1 - Estrutura do Catálogo de *Users*

O catálogo de *Users* é composto por uma *Hashtable*, onde são guardados todos os *Users* e onde a *Key* é o *username* do *User* e por um *Array* dinâmico onde são inseridos apenas os *Users* cujo estado da conta é *Active*.

Este *Array* é ordenado depois do *parsing* das *Rides* por ordem de decrescente de distância total percorrida por cada *User* e permite-nos responder à *query 3* da forma mais eficiente possível.

### 2.2 Catálogo de *Drivers*

```
struct catalog_drivers {
    GHashTable *drivers_hash;
    Driver** biggestRating_array;
    int size, used;
};
```

Fig. 2 - Estrutura do Catálogo de *Users*

O catálogo de *Drivers*, tal como o catálogo de *Users*, é composto por uma *Hashtable* e por um *Array* dinâmico mas neste caso guardam *Drivers* e a *Key* é o ID do *Driver*.

O *Array* é ordenado também de maneira diferente depois do *parsing* das *Rides*, sendo este ordenado por ordem de decrescente da avaliação média de cada *Driver* e permite-nos responder à *query 2* da forma mais eficiente possível.

---

### 2.3 Catálogo de *Rides*

```
struct catalog_rides {
    GHashTable *city_rides;
    GHashTable *date_rides;

    GSList *list_feminino;
    GSList *list_masculino;
};
```

Fig. 3 - Estrutura do Catálogo de *Rides*

O catálogo de *Rides* é composto por duas *Hashtables*, onde são guardadas todas as *Rides*, sendo a *Key* duma *Hashtable* a cidade onde a *Ride* aconteceu e da outra a data associada à *Ride*.

```
typedef struct drivers_in_city {
    double total_rating;
    int total_rides;
    int id;
} DriversInCity;

typedef struct city_rides {
    double price_notip_total, price_total;
    int rides_total;
    int size;
    int flag_ridesSorted;
    Ride **rides;
    GHashTable *drivers;
    GList *drivers_sorted;
} CityRides;
```

Fig. 4 - Estrutura das *CityRides*

Na *Hashtable* *city\_rides* os *values* são guardados na estrutura *city\_rides*, que por sua vez guarda o preço e número total de *Rides* na cidade, um *Array* dinâmico das *Rides* na cidade, e uma *Hashtable* de *Drivers* que vão sendo atualizados à medida que dá-mos *parsing* ao ficheiro de *Rides*.

Guardámos também uma *List* de *Drivers* na cidade.

---

```
typedef struct date_rides {
    double price_notip_total;
    int rides_total;
    int size;
    int rides_tip;
    Ride **rides;
} DateRides;
```

Fig. 5 - Estrutura das *DateRides*

Na *Hashtable date\_rides* os *values* são guardados na estrutura *date\_rides*, que por sua vez guarda o preço e número total de *Rides* numa data e um *Array* dinâmico das *Rides* nessa data onde o passageiro deu gorjeta que, tal como na *Hashtable city\_rides*, vão sendo atualizados à medida que damos *parsing* do ficheiro das *Rides*.

O catálogo tem também dois *Arrays* dinâmicos onde são inseridos apenas as *Rides* cujo estado da conta é *Active*, sendo um dos *Arrays* apenas constituído por *Rides* em que ambos o *User* e o *Driver* são femininos e outro *Array* em que ambos são masculinos.

Estes *Arrays* são ordenados depois do *parsing* das *Rides* por ordem de decrescente de idade das contas associadas à *Ride* permitindo-nos responder à *query 8* da forma mais eficiente possível.

---

### 3 Queries

Após a alteração das estruturas de forma a responder ao aumento do tamanho dos *Datasets*, alteramos as *queries* desenvolvidas na primeira fase do trabalho e desenvolvemos as restantes *queries* propostas.

#### 3.1 Query I

**Descrição:** listar o resumo de um perfil registado no serviço através do seu identificador, representado por  $\langle ID \rangle$ . Note que o identificador poderá corresponder a um utilizador (i.e., *username* no ficheiro *users.csv*) ou a um condutor (i.e., *id* no ficheiro *drivers.csv*). Em cada caso, o *output* será diferente, mais especificamente:

- Utilizador

nome;genero;idade;avaliacao\_media;numero\_vagens;total\_gasto

- Condutor

nome;genero;idade;avaliacao\_media;numero\_vagens;total\_aufrido

Caso o utilizador/condutor não tenha nenhuma viagens associadas, considerar a *avaliacao\_media*, *numero\_vagens*, e *total\_gasto* como sendo 0.000, 0, e 0.000, respectivamente.

**Resolução:** a resolução desta *query* manteve-se igual ao que foi inicialmente implementado na primeira fase do projeto.

#### 3.2 Query II

**Descrição:** listar os N condutores com maior avaliação média. Em caso de empate, o resultado deverá ser ordenado de forma a que os condutores com a viagem mais recente surjam primeiro. Caso haja novo empate, deverá ser usado o *id* do condutor para desempatar (por ordem crescente). O parâmetro N é representado por  $\langle N \rangle$ .

**Resolução:** para a resolução desta *query*, começamos por analisar o comando dado para a inicializar e, a partir dele, retiramos o valor de N necessário para a execução da *query*. Após isto, sendo que no catálogo de *Drivers* já temos um array ordenado de forma correta para responder a esta *query* apenas temos de percorrer os N primeiros elementos do array e guardar as informações necessárias no formato definido num array de strings para depois as imprimir-mos no ficheiro de output.

#### 3.3 Query III

**Descrição:** listar os N utilizadores com maior distância viajada. Em caso de empate, o resultado deverá ser ordenado de forma a que os utilizadores com a viagem mais recente surjam primeiro. Caso haja novo empate, deverá ser usado o *username* do utilizador para desempatar (por ordem crescente). O parâmetro N é representado por  $\langle N \rangle$ .

**Resolução:** tendo em conta a semelhança desta *query* e da anterior, o método utilizado para a resolver foi também idêntico, sendo as únicas diferenças que o top N é agora um top de *Users*, logo vamos percorrer os N primeiros elementos do array do

---

catálogo de *Users* ao invés do catálogo de *Drivers*, sendo que a ordem pela qual este array está ordenado já corresponde à ordem pedida por esta *query*.

### 3.4 *Query IV*

**Descrição:** preço médio das viagens (sem considerar gorjetas) numa determinada cidade, representada por <city>.

**Resolução:** a resolução desta *query* passa apenas pelo cálculo do quociente do preço total das *Rides* (sem considerar as gorjetas) numa certa cidade pelo número total de *Rides* nessa mesma cidade. Estas informações são ambas guardadas na *Hashtable* de *city\_rides* no catálogo das *Rides*, usando como *key* a cidade, como foi anteriormente explicado. Desta forma temos apenas de aceder à *city\_rides* associada à cidade desejada e obter o resultado final a partir da divisão dos valores presentes na mesma e apresentar o resultado.

### 3.5 *Query V*

**Descrição:** preço médio das viagens (sem considerar gorjetas) num dado intervalo de tempo, sendo esse intervalo representado por <data A> e <data B>.

**Resolução:** para esta *query* começamos por verificar qual das datas fornecidas é a data inicial. A partir daqui podemos aceder à *Hashtable* de *date\_rides* guardado no catálogo das *Rides* que usa como *Key* uma data e contém os valores do preço total das viagens (sem gorjetas) e o número de viagens associados a essa mesma data. Assim, aumentando a data em 1 dia por iteração até alcançarmos a data final a partir da data inicial, vamos acedendo aos *values* da *date\_rides* correspondentes a cada dia entre a data inicial e a data final e fazemos um somatório dos valores necessários. Para terminar, resta-nos apenas efetuar a operação de divisão e apresentar o resultado.

### 3.6 *Query VI*

**Descrição:** distância média percorrida, numa determinada cidade, representada por <city>, num dado intervalo de tempo, sendo esse intervalo representado por <data A> e <data B>.

**Resolução:** para esta *query* começamos por aceder ao *city\_rides* do catálogo das *Rides* associado à cidade fornecida e verificar se o array de *rides* se encontra ordenado por data e, caso isso não se verifique, passamos a ordená-la. A partir daqui fazemos uma procura binária para encontrar uma posição no array cuja *ride* seja no intervalo de tempo dado e a partir desta posição acedemos a todas as viagens no intervalo de tempo correto e conseguimos assim calcular assim a média da distância percorrida.

### 3.7 *Query VII*

**Descrição:** top N condutores numa determinada cidade, representada por <city> (no ficheiro *rides.csv*), ordenado pela avaliação média do condutor. Em caso de empate, o resultado deverá ser ordenado através do id do condutor, de forma decrescente. A avaliação média de um condutor numa cidade é referente às suas viagens nessa cidade,

---

e não na cidade que está no seu perfil (ou seja, o mesmo condutor poderá ter médias diferentes dependendo da cidade).

**Resolução:** para a resolução desta *query*, começamos por analisar o comando dado para a inicializar e, a partir dele, retiramos o valor de N necessário para a execução da *query*. Depois só temos de aceder ao *city\_rides* no catálogo das *Rides* associado à cidade fornecida e verificar se a lista de condutores é nula ou não, caso seja nula criamos a lista a partir da *Hashtable* de condutores e ordenamos-a. A partir daqui resta-nos apenas aceder aos N primeiros elementos desta lista para obtermos a nossa resposta, sendo que a ordem pela qual a lista é ordenada corresponde à ordem pedida por esta *query*.

### 3.8 *Query VIII*

**Descrição:** listar todas as viagens nas quais o utilizador e o condutor são do género passado como parâmetro, representado por <gender> e têm perfis com X ou mais anos, sendo X representado por <X>. O output deverá ser ordenado de forma que as contas mais antigas apareçam primeiro, mas especificamente, ordenar por conta mais antiga de condutor e, se necessário, pela conta do utilizador. Se persistirem empates, ordenar por id da viagem (em ordem crescente).

**Resolução:** para a resolução desta *query*, sendo que no catálogo de *Rides* já temos dois arrays ordenados de forma correta para responder a esta *query*, apenas temos de aceder ao array correspondente, ou feminino ou masculino, e percorrer os elementos do array até que a idade do *Driver* seja menor que a idade dada como input, guardando estas *Rides* numa lista de forma a podermos depois imprimir no ficheiro de output as informações necessárias para responder corretamente a esta *query*.

### 3.9 *Query IX*

**Descrição:** Listar as viagens nas quais o passageiro deu gorjeta, no intervalo de tempo (data A, data B), sendo esse intervalo representado pelos parâmetros <data A> e <data B>, ordenadas por ordem de distância percorrida (em ordem decrescente). Em caso de empate, as viagens mais recentes deverão aparecer primeiro. Se persistirem empates, ordenar pelo id da viagem (em ordem decrescente).

**Resolução:** para esta *query* começamos por verificar qual das datas fornecidas é a data inicial. A partir daqui podemos aceder à *Hashtable* de *date\_rides* guardado no catálogo das *Rides* que usa como *Key* uma data e as *rides* efetuadas nessa data onde o passageiro deu gorjeto. Assim, aumentando a data em 1 dia por iteração até alcançarmos a data final a partir da data inicial, vamos acedendo aos *values* da *date\_rides* correspondentes a cada dia entre a data inicial e a data final e guardando as *rides* numa lista. Após percorrermos todas as datas só nos resta ordenar a lista de acordo com o pedido na *query* e imprimir o output.

---

## 4 Menu Interativo

Tal como foi pedido para esta segunda fase, para além do modo *batch*, no qual o programa é executado com dois argumentos (sendo o primeiro o caminho para a pasta onde estão os ficheiros de entrada e o segundo o caminho para um ficheiro de texto que contém uma lista de comandos a serem executados), implementamos também um modo interativo que recebe input do utilizador e apresenta as respostas às *queries* pedidas no terminal, usando um método de paginação.

## 5 Testes e Desempenho do Programa

De modo a testar as soluções implementadas e a analisar o desempenho do programa, corremos uma série de testes nos diferentes computadores que temos ao nosso dispor.

Desta forma, é-nos possível ter uma melhor percepção dos tempos de execução das diferentes *queries* e processos realizados ao longo de todo o programa e a forma como estes se relacionam com as especificações da máquina em que estão a ser testados.

### 5.1 Tempos de Execução nas Diferentes Máquinas

Em relação aos diferentes computadores em que fizemos os testes, apresentamos as especificações dos mesmos abaixo:

	Sistema Operativo	CPU	RAM
Máquina A	Ubuntu 22.04 LTS 64-bit	i7-8300H 2.2Ghz x 12	16 Gb
Máquina B	Mint 21 Vanessa 64-bit	i74720HQ 2.6Ghz x 8	7.7 Gb
Máquina C	Ubuntu 22.04 LTS 64-bit	i5-8300H 2.6Ghz x 8	8 Gb

Fig. 6 - Especificações das Máquinas

Para começar, testamos o tempo que o programa demora a inicializar, ou seja, o tempo que demora a fazer o *parse* dos dados para os diferentes ficheiros de entrada juntamente com a ordenação dos diferentes *arrays* que são criados aquando da criação dos catálogos. Para isto, o programa foi inicializado 12 vezes por cada máquina, nas quais se descartaram o melhor e pior tempos e os restantes foram utilizados para calcular a média. Os resultados obtidos (em segundos) foram os seguintes:

	Drivers parsed	Users parsed	Rides parsed	Drivers sorted	Users sorted	M/F sorted
Máquina A	0.178377	1.534314	23.497896	0.082561	0.851050	29.577559
Máquina B	0.145279	1.520632	23.919474	0.059313	0.792853	31.043724
Máquina C	0.115019	1.180842	18.644126	0.042190	0.626828	21.043724

Fig. 7 - Tempos de Execução da Inicialização do Programa

Depois disto, passamos à testagem dos tempos de execução das diferentes *queries*. Em cada pc e para cada *query* foram utilizados os mesmos *inputs* e, no fim, foi feita novamente a média para cada uma, de forma semelhante ao que foi feito para a inicialização do programa. Os resultados (em segundos) foram os seguintes:

	query 1	query 2	query 3	query 4	query 5	query 6	query 7	query 8	query 9
Máquina A	0.000254	0.002557	0.001853	0.000085	0.000250	0.852611	0.002584	0.080564	0.000175
Máquina B	0.000363	0.002641	0.001922	0.000025	0.000430	1.003210	0.002632	0.083204	0.000341
Máquina C	0.000341	0.001945	0.001399	0.000020	0.003268	0.273873	0.021423	0.072228	0.005281

Fig. 8 - Tempos de Execução das Queries

---

## 5.2 Discussão de Resultados

Após a testagem local das nossas soluções, e com uma noção dos testes automáticos aos quais a equipa docente nos deu acesso, sentimos que os tempos de execução do programa são, no geral, bastante satisfatórios.

Em relação às *queries*, acreditamos que os tempos de execução são bons para qualquer uma das 9 que foram implementadas, sendo que, dadas as diferenças de complexidade entre elas, existem sempre umas que serão mais rápidas que outras.

Quanto ao processo de inicialização do programa, consideramos também que é feito num tempo satisfatório. Admitimos, no entanto, a possibilidade de ter criado as estruturas de maneira diferente de modo a melhorar o desempenho do mesmo. Apesar disto, a forma como foram implementadas foi a melhor que o grupo conseguiu conceber, pelo que não sabemos ao certo que alterações poderiam ser feitas para melhorar os resultados obtidos.

## 6 Encapsulamento e Modularidade

Para esta fase, mantivemos as diversas estruturas encapsuladas, bem como, com o intuito de manter a modularidade e facilitar a manutenção do código, decidimos separar a validação de dados(*validator.c*), manipulação de dados e as funções relativas à data(*date.c*) em módulos distintos. Além disso, também decidimos separar o catálogo de informações(*catalog.c*) e as funções de impressão de consultas(*print\_outputs.c*).

Tudo isto permitiu uma melhor a organização, a modularidade e a facilidade de manutenção do código.

## 7 Conclusão

Concluindo, o grupo acredita que todos os resultados esperados com este projeto foram, no geral, obtidos com sucesso. Tanto na primeira como na segunda fase pensamos ter cumprido todos os objetivos estipulados. Mais especificamente, foram implementadas todas as *queries* com tempos de execução satisfatórios, foi implementado o menu interativo com paginação e, com o progredir do projeto, as soluções foram sendo adaptadas conforme o necessário (por exemplo com o aumento do tamanho dos ficheiros de entrada entre fases).

Adicionalmente, destaca-se a atenção que foi dada aos princípios de modularidade e encapsulamento de dados, que consideramos ter respeitado ao longo de todo o trabalho.

Os conceitos que aprendemos nesta UC e com a realização deste projeto são, para além de interessantes, extremamente úteis para uma melhor organização, segurança, reutilização de código, etc. pelo que esperamos poder vir a usá-los e, se possível, aprofundá-los ainda mais no futuro.