# Universidade do Minho
# Licenciatura em Engenharia Informática

## Laboratório de Informática III
## 1º Fase


Grupo - 25:

Diogo Costa(a95460) Ivo Ribeiro(a96726)
Martim Ribeiro (a96113)


10 novembro 2022

# Índice

# 1. Introdução

No âmbito da Unidade Curricular de Laboratórios de Informática III, foi-nos pedida, nesta primeira fase do trabalho, a implementação de uma aplicação que obedecesse à seguinte arquitetura:

- um parsing dos dados de entrada, que trata da leitura de 3 ficheiros ("users.csv", "rides.csv" e "drivers.csv");
- um modo de operação batch, responsável por ler um ficheiro de comandos, interpretar cada um e executar a respetiva query;
- 1/3 das queries descritas na Secção 4;
- 3 catálogo de dados para os três ficheiros de entrada;

# 2. Organização do projeto

Nesta primeira fase, o projeto foi organizado de forma a respeitar as boas práticas de modularidade e encapsulamento do código, de modo a melhorar a organização e a facilitar o eventual trabalho de manutenção futuro. O código foi assim dividido em 4 partes: parsing dos dados, criação dos catálogos (users, drivers e rides), interpretação de comandos e queries.

A secção de parsing lê os dados para a memória e guarda-os nos respetivos catálogos de cada ficheiro. Não há qualquer tipo de validação de dados pois apenas é pedido para a segunda fase.

A secção de interpretação de comandos lê cada linha do ficheiro de comandos e chama a respetiva query com os dados disponibilizados nesse ficheiro e ainda os catálogos necessários para o funcionamento de determinada query.

A secção das queries recebe a informação passada pelo interpretador de comandos e excuta o conjunto de funções necessários para obter os resultados pretendidos pelas mesmas.

Todos estes módulos e as suas funções encontram-se devidamente documentados, aumentado a organização e contribuindo para uma mais fácil interpretação do código.

# 3. Estrutura de dados

Depois de alguma ponderação relativamente a como iríamos abordar a questão dos catálogos, chegamos à conclusão de que seria importante guardar os dados numa estrutura que fosse eficiente em termos de acesso (inserção e procura) com base nos ids/usernames.

A opção que escolhemos foi recorrer ao uso de hashtables. Para a implementação desta estrutura de dados nos diferentes catálogos, recorremos à biblioteca "glib", onde já se encontram definidas grande parte das funções que iríamos utilizar ao longo do trabalho.

Os 3 catálogos seguem a mesma estrutura, organizados por ids/usernames, como será melhor explicado de seguida.

Estes 3 módulos foram implementados de forma a funcionarem de forma independente, tendo em vista os objetivos de abstração, modularidade e encapsulamento dos dados e do código. Como tal, cada um deles fornece a respetiva API, com funções básicas de alocação de estruturas, pesquisa e inserção. De modo a guardar informação necessária lida a partir de cada ficheiro, utilizamos as seguintes structs, guardadas nas hashtables dos seus catálogos correspondentes:

- User;
- Driver;
- Ride;

### 3.1 Struct user

Um user é composto pelo seu username e pelos restantes campos, tal como vem no ficheiro de entrada.

```
struct user {
    char *username;
    char *name;
    char *gender;
    char *birth_date; // dd/mm/aaaa
    char *account_creation;
    char *pay_method;
    char *account_status; // active ou inactive
    double distanciapercorrida;
};
```

### 3.2 Struct driver

Um driver é composto pelo seu id e pelos restantes campos, tal como vem no ficheiro de entrada.

```
struct driver {
    char *id;
    char *name;
    char *birth_date;
    char *gender;
    char *car_class;
    char *license_plate;
    char *city;
    char *account_creation;
    char *account_status;
    int viagens;
    double allavs;
};
```

### 3.3 Struct ride

Um ride é composto pelo seu id e pelos restantes campos, tal como vem no ficheiro de entrada.

```
struct ride {
    char *id;
    char *date;
    char *driver;
    char *user;
    char *city;
    char *distance;
    char *score_user;
    char *score_driver;
    char *tip;
    char *comment;
};
```

# 4. Parsing

No parsing, a nossa abordagem foi criar 3 funções, 1 para cada ficheiro de entrada. Em cada 1 dessas funções o ficheiro era aberto, lido linha a linha e inserido no respetivo catálogo.

# 5.Queries

De maneira a resolver as três primeiras queries como pretendido nesta primeira fase verificamos que seria necessário percorrer as hashtables onde tínhamos a informação armazenada.

Para tal o primeiro método que nos surgiu foi o “g_hash_table_foreach” porem este método tal como verificamos na libraria do mesmo recebe uma hashtable e uma função que não recebe nenhum argumento predefinido, como tal, este método criou-nos alguns problemas quando se tratava de passar informações à função que executava de forma recursiva, uma vez que pretendíamos que essa função que executaria de forma recursiva incrementa-se uma variável.

Estes problemas levaram nos a optar por outro método para a execução das funções a todos os elementos da hashtable ao que nos surgiu a ideia de implementarmos uma lista que contivesse a informação presente na hashtable mas fosse facilmente manuseada então decidimos implementar a “GList” pois verificamos que para alem de ser possível manusear de maneira fácil, e conseguirmos garantir que toda a informação presente na hashable era consultada, verificamos ainda que este tipo de estrutura aceitava como “data” um apontador para qualquer tipo de estrutura o que funcionaria para o nosso caso de manusear a informação presente nas hashtables.

Apos definirmos a maneira de como iriamos manusear a informação das hashtables tudo o resto se tornou bastante intuitivo e pratico, dando apenas alguns erros que ao executar os testes através da nossa makefile e visualizar os erros e os warnings apresentados no terminal facilmente os corrigíamos.

Deixar apenas uma nota que é a seguinte, tivemos de fazer alguns ajustes sim nas estruturas iniciais do nosso código, ao que passo a citar, na estrutura do driver adicionamos uma variável com a informação do numero de voltas efetuadas por cada condutor e ainda uma variável com um somatório de todas as avaliações que esse mesmo condutor recebeu, e ainda na estrutura dos users acrescentamos uma variável que corresponde ao somatório de todas as distancias percorridas por esse utilizador na avaliação.

Estas alterações visam um maior desempenho do nosso código uma vez que faz com que apenas tenhamos de percorrer uma vez, e apenas se necessário, toda a lista que contem a informação das viagens efetuadas na aplicação ao invés de percorrer essa mesma lista para cada utilizador ou para cada condutor, o que torna por sua vez o nosso código bem mais eficiente.