# Relatório da Fase 1 – Grupo 25

Hugo António Gomes Ramos (A100644)
Martim Quintelas Pinto Félix (A100647)
João Silva Loureiro (A100832)

18 de Novembro de 2023

## 1. Introdução

O presente relatório tem como objetivo abordar a primeira fase do projeto da UC Laboratórios de Informática III do ano letivo 2023/2024. Nesta fase, como objetivos, o trabalho necessitava de fazer o parsing dos dados de entrada, funcionar através do modo de operação *batch,* realizar 6 das 10 queries propostas e validar o dataset fornecido pelos docentes. Para além disto, temos de ter em conta, durante todo o projeto, o uso de encapsulamento e modularidade.
Este relatório irá abranger as decisões tomadas pelo grupo, como os métodos de raciocínio para o desenvolvimento do mesmo.
Dito isso, é importante destacar que nas estruturas que exemplificaremos posteriormente, consideramos a natureza dos dados envolvidos e as questões que precisavam de ser resolvidas dentro de um tempo útil. Portanto, se considerássemos outros fatores, a abordagem adotada certamente seria diferente.

## 2. Catálogos

Primeiramente, a fase inicial do projeto aborda o armazenamento dos dados, ou seja, a necessidade de preservar todas as informações relevantes numa estrutura específica.

No desenvolvimento do nosso projeto, adotamos uma estratégia consistente ao utilizar HashTables para todas as estruturas principais assim como utilizar também listas ligadas. A implementação das HashTables foram, a nosso ver, uma das melhores formas de realizar o projeto já que contribuem para a eficiência e facilidade de gestão dos dados.

Vantagens de utilizar HashTables:

- Acesso Rápido por *Key*:

As HashTables proporcionam acesso direto aos dados através de uma chave única associada a cada entidade. Este método de acesso constante é especialmente valioso ao lidar com grandes conjuntos de dados como é o caso, pois não depende do tamanho total do ficheiro.

- Eficiência em Termos de Tempo:

As operações de inserção, pesquisa e remoção numa HashTable têm uma complexidade média de tempo O(1) (constante) para operações bem-sucedidas. Isto resulta em tempos de resposta rápidos, essenciais para otimizar o desempenho em cenários onde a velocidade de acesso aos dados é crucial.

- Identificação Única por Chave (ID):

A natureza única das chaves nas HashTables corresponde perfeitamente à necessidade de identificadores distintos (ID) para cada entidade (*Users*, *Flights*, *Reservations*, *Passengers*). Isto garante a unicidade em todo o sistema.

- Facilidade de Implementação:

A estrutura das HashTables facilita a implementação e gestão do código. A utilização de funções Hash permite a distribuição eficiente dos dados, mantendo ao mesmo tempo a acessibilidade.

Desvantagens de utilizar HashTables:

- Colisões:
Uma possível desvantagem das HashTables é a possibilidade de colisões, onde duas chaves diferentes são mapeadas para a mesma posição na tabela. Existem métodos de resolução de colisões, mas a escolha do método certo é crucial para evitar atrasos e problemas no desempenho.

- Consumo de Memória:
HashTables podem consumir mais memória em comparação com estruturas de dados lineares, principalmente devido à necessidade de manter uma tabela Hash adicional.

- Dificuldade na Ordenação:
As HashTables não mantêm uma ordem específica nos elementos, o que pode ser uma desvantagem se a ordenação dos dados for uma consideração importante no contexto do projeto. Visto que este é um ponto importante no trabalho tivemos de arranjar uma solução. Solução esta que iremos explicar mais à frente quando chegarmos à parte do relatório referente à resolução das queries.

Em suma, a decisão de utilizar HashTables para todas as entidades foi baseada na busca por um equilíbrio entre eficiência operacional e facilidade de implementação. Ao reconhecer as vantagens e desvantagens associadas às HashTables, conseguimos criar uma estrutura de dados coesa para atender às necessidades específicas do nosso projeto, onde a identificação única por *Key* desempenha um papel central.

# Utilizadores

```c
#define HASHSIZE 10050 // Tamanho da tabela hash

typedef char KeyType[300];


typedef struct Q2 {
    char *id;
    char *data;
    int tipo; //1-flight, 2-reserva
    double total_gasto;
    struct Q2 *next;
} Q2;

typedef struct User {
    char *id;
    char *nome;
    char *email;
    char *phone;
    char *birth;
    char *sex;
    char *passport;
    char *country;
    char *address;
    char *account_creation;
    char *pay_method;
    char *account_status;
    int total_reservas;
    int total_voos;
    double total_gasto;
    struct User *next;
    struct Q2 *q2;
} User;

// Tabela hash
typedef User *hash_user[HASHSIZE];
```

*Figura 1: Estrutura de dados dos Utilizadores (user.h)*

Tal como podemos observar na figura 1, a struct User contém todos os dados referentes a si, mais o total de reservas, o total de voos, o total gasto (os parâmetros necessários para realizar a query 1). Para além disto tem também um apontador para si mesmo e outro para a struct Q2. Esta última struct foi criada já com a finalidade de tornar a realização da query 2 mais fácil. A lista ligada Q2 serve para armazenar os voos e reservas de um utilizador, apresentando o ID, a data, no caso dos voos é referente ao 'schedule_departure_date' e no caso das reservas refere-se à 'begin_date', apresenta o tipo, podendo este ser um voo(identificado por 1) ou uma reserva(identificado por 2). e o total gasto que é calculado através dos dias de viagem (dias*preco+dias*(preco/100)*taxa da

cidade). Os utilizadores serão guardados numa tabela Hash onde é previamente alocado um tamanho de 10050 e onde cada utilizador é mapeado para a sua posição através do seu ID.
Apesar de fazer com a que implementação das queries seja mais fácil, a utilização de listas ligadas também tem desvantagens como, não tira proveito de localidade espacial, visto que os apontadores para os próximos elementos estão dispersos na memória e como tal a miss rate aumenta, demorando mais tempo a aceder ao elemento seguinte.

# Reservas

```c
#define HASHSIZE 10050 // Tamanho da tabela hash

typedef char KeyType[300];


typedef struct Reserva {
    char *id;
    char *user_id;
    char *hotel_id;
    char *hotel_name;
    char *hotel_stars;
    char *city_tax;
    char *address;
    char *begin_date;
    char *end_date;
    char *price_per_night;
    char *includes_breakfast;
    char *room_details;
    char *rating;
    char *comment;
    int total_noites;
    double total_gasto;
    struct Reserva *next_reserva;
} Reserva;

typedef struct ReservaResumo {
    char *id;
    char *begin_date;
    char *end_date;
    char *user_id;
    double total_price;
    double rating;
    struct ReservaResumo *next_resumo;
} ReservaResumo;

typedef struct Hotel {
    char *hotel_id;
    struct Hotel *next;
    struct ReservaResumo *next_resumo;
} Hotel;

typedef Hotel *hash_hoteis[HASHSIZE];
typedef Reserva *hash_reservas[HASHSIZE];
```

*Figura 2: Estrutura de dados das Reservas (reserva.h)*

O raciocínio seguido para fazer a estrutura de dados das reservas foi bastante semelhante, no entanto é importante fazer notar que fizemos 3 estruturas diferentes. A 'Reserva' que contém todas as informações que os ficheiros .csv têm, assim como o total de noites (informação necessária para implementar a query 4) e o total gasto (utiliza a mesma formúla que nos utilizadores e é nos pedida essa informação também na query 4). Temos também 'ReservaResumo', criamos esta estrutura já que, como iremos mencionar à frente, a estrutura 'ReservaResumo' não é uma HashTable como a 'Reserva', mas sim uma lista ligada para colocarmos no HashTable 'Hotel'. A struct 'ReservaResumo' guarda todas as informações necessárias pedidas na query 4. Como mencionado, adicionamos também no mesmo ficheiro uma struct Hotel que guarda o id de um hotel e tem 2 apontadores, um para si mesmo e outro para a struct 'ReservaResumo'. Tivemos também a necessidade de criar esta estrutura específica também como forma de facilitar nas queries (nomeadamente a query 1). Assim como os utilizadores a struct 'Reserva' e a struct 'Hotel' são guardados numa tabela Hash, também com tamanho previamente definido 10050 e onde são mapeados para a sua posição na tabela através do ID (id e hotel_id, respetivamente).

# Voos

```c
#define HASHSIZE 10050 // Tamanho da tabela hash

typedef char KeyType[300];


typedef struct Voo {
    char *id;
    char *airline;
    char *plane_model;
    char *total_seats;
    char *origin;
    char *destination;
    char *schedule_departure_date;
    char *schedule_arrival_date;
    char *real_departure_date;
    char *real_arrival_date;
    char *pilot;
    char *copilot;
    char *notes;
    int total_passengers;
    int delay;
    struct Voo *next_voo;
} Voo;

typedef struct VooResumo {
    char *id;
    char *schedule_departure_date;
    char *destination;
    char *airplane;
    char *plane_model;
    struct VooResumo *next_voo;
} VooResumo;

typedef struct Aeroporto {
    char *name;
    struct Aeroporto *next;
    struct VooResumo *next_resumo;
} Aeroporto;

typedef Aeroporto *hash_aeroportos[HASHSIZE];
typedef Voo *hash_voos[HASHSIZE];
```

*Figura 3: Estrutura de dados sobre Voos (voo.h)*

A estrutura de dados referente aos voos segue o mesmo raciocínio que a estrutura das reservas. A struct 'Voo' contêm todas as informações que estão no ficheiro csv (flights.csv), assim como o total de passageiros (informação que nos é pedida na query 6) e o delay (informação necessária para fazer a query 7), onde utilizamos uma função auxiliar 'CalcularDiferençaSegundos' para descobrir esse valor. Identificamos também a lista ligada que iremos colocar na struct 'Aeroporto', 'VooResumo'. Colocamos na struct 'VooResumo' todos os parâmetros necessários para fazer a query 5. Depois temos também a struct Aeroporto que guarda o nome (cidade) e com dois apontadores assim como a struct 'Hotel'. Um apontador para si mesmo e outro para a struct 'VooResumo'.
Como disse, tanto as informações da struct 'Aeroporto', como a de 'Voo' são guardadas numa HashTable onde a chave para mapear é o nome e o id, respetivamente.

# 3. Queries

Para a primeira fase do trabalho decidimos implementar as bases do projeto, como o parser e definir as estruturas dos dados que utilizaremos ao longo do projeto (Como o catálogo para guardar os dados e os *Utilizadores*, *Voos* e Reservas). Foram implementadas as queries 1, 2, 3, 4, 5, 9.

## Query 1 – Listar o resumo de um utilizador, voo ou reserva, consoante o identificador recebido por argumento.

Para possibilitar o acesso eficiente à informação dos utilizadores, voos e reservas através dos respetivos identificadores, foram utilizadas três HashTables: hash_user, hash_voos, e hash_reservas.
Cada uma dessas HashTables tem a função de mapear o identificador (id) para a estrutura de dados correspondente: User, Voo, ou Reserva.
A função q1 recebe um identificador (arg) e uma flag (f), onde o identificador pode ser associado a um utilizador, a um voo ou a uma reserva. A flag f determina o formato de saída (1 para detalhes, 0 para resumo).
Para cada tipo de entidade (utilizador, voo, reserva), são utilizadas funções auxiliares (RetrieveUser, RetrieveVoo, RetrieveReserva) que realizam a procura na HashTable correspondente.
Se o identificador pertence a um utilizador, vamos buscar e formatar as informações relevantes, como nome, sexo, idade, código do país, passaporte, número de voos, número de reservas e total gasto.
No caso de um voo, são obtidos detalhes como a companhia aérea, modelo do avião, origem, destino, data de partida estimada, data de chegada estimada, número de passageiros e tempo de atraso.
Para uma reserva, são recuperados dados como id do hotel, nome do hotel, estrelas do hotel, data de início, data de fim, inclusão de pequeno-almoço, número de noites e preço total.
A condição (strcasecmp(user->account_status, "active") == 0) garante que apenas são processados utilizadores com o estado de conta ativo. Utilizadores inativos são ignorados.
A complexidade da consulta é influenciada pelo número de viagens realizadas por um condutor ou utilizador (N). A análise de complexidade é $\theta(N)$.

## Query 2 - Listar os voos ou reservas de um utilizador, se o segundo argumento for flights ou reservations, respetivamente, ordenados por data.

A função q2 recebe a HashTable de utilizadores h_users, um array de argumentos argv, o número de argumentos argc, a flag f e um apontador para um arquivo fp_output.
O utilizador associado ao identificador fornecido é recuperado usando a função (RetrieveUser). Se o utilizador existir e a conta estiver ativa, o processamento continua.
A função remover_horas é utilizada para remover a componente de horas/minutos/segundos de uma data e obter apenas a parte da data.
O número de argumentos (argc) determina se a query deve apresentar apenas as reservas ou os voos, ou ambos. Se o argc for igual a 3, é fornecido um segundo argumento, indicando se se trata de "reservations" ou "flights".
A estrutura Q2 associada ao utilizador contém informações sobre reservas e voos. A função percorre esta estrutura e imprime as informações formatadas para o arquivo de output, considerando a flag `f` para conseguir decidir se usa um formato detalhado ou resumido.
As informações são ordenadas por data (com a componente das horas removida) e, em caso de empate, pelo identificador de forma crescente.
Utilizadores com o estado de conta "inactive" são ignorados, garantindo assim que apenas informações de utilizadores “active” são processadas.

## Query 3 - Apresentar a classificação média de um hotel, a partir do seu identificador.

A função q3 recebe a HashTable de hotéis h_hoteis, um identificador de hotel argv, a flag f, e um apontador para um arquivo fp_output, semelhante à query anterior.
A estrutura Hotel é utilizada para armazenar informações sobre um hotel.
A estrutura ReservaResumo é utilizada para resumir informações sobre reservas, mas
Incluindo o parâmetro da classificação.
Utilizamos a função RetrieveHotel para obter as informações do hotel associado ao identificador fornecido.
A função GetRatingByHotel percorre a lista de hotéis para ir buscar a classificação de um hotel e ao mesmo tempo calcular a classificação média do hotel com base nas reservas associadas.
Percorre a lista de reservas associadas ao hotel, acumulando a classificação total e contando o número de avaliações válidas (classificação entre 1 e 5).
Se só houver avaliações válidas, calcula a média da classificação total pelo número de avaliações.
Imprime a classificação média no arquivo de output, utilizando o formato detalhado ou resumido com base na flag f, como nas queries anteriores.
Se o hotel não for encontrado ou não tiver avaliações válidas, retorna -1 para indicar que a classificação não está disponível.

## Query 4 - Listar as reservas de um hotel, ordenadas por data de início

A função q4 recebe a HashTable de hotéis h_hoteis, um identificador de hotel, a flag f, e um apontador para um arquivo fp_output.
Utiliza a função `RetrieveHotel` para obter as informações do hotel associado ao identificador fornecido (argv).
Percorre a lista de resumos de reservas associadas ao hotel dado.
As reservas são ordenadas por data de início (da mais recente para a mais antiga).
Em caso de empate na data, utiliza o identificador da reserva como critério de desempate (de forma crescente).
Como nas queries anteriores, imprime as informações detalhadas ou resumidas das reservas no arquivo de output, utilizando o formato apropriado com base na flag f.
Se o hotel não for encontrado, não há reservas associadas ou outra condição especial, o código trata adequadamente esses casos.

## Query 5 - Listar os voos com origem num dado aeroporto, entre duas datas, ordenados por data de partida estimada.

A função q5 recebe a HashTable de aeroportos h_aeroportos, a origem do voo origin, as datas de início e fim begin_date e end_date, a flag f e um apontador para um arquivo fp_output.
A estrutura VooResumo é utilizada para resumir informações sobre os voos.
Utilizamos a função GetVoosAeroportoEntreDatas para obter os voos do aeroporto de origem entre as datas fornecidas, como é pedido na query.
Percorremos a lista de resumos de voos.
Os voos são ordenados por data de partida estimada (da mais antiga para a mais recente).
Em caso de empate na data, utiliza o identificador do voo como critério de desempate (de forma crescente como requisitado).
Assim como nas outras queries, imprime as informações detalhadas ou resumidas dos voos no arquivo de output, utilizando o formato apropriado com base na flag `f`.
Se o aeroporto não for encontrado, não houver voos associados ou outra condição especial, o código trata adequadamente esses casos.
O output segue o formato especificado, incluindo detalhes como o identificador, data de partida estimada, destino, companhia aérea e modelo de avião.

### Query 9 - Listar todos os utilizadores cujo nome começa com o prefixo passado por argumento, ordenados por nome (de forma crescente)

Utilizamos uma abordagem eficiente para encontrar utilizadores cujo nome começa por um determinado prefixo. A função GetUserPrefix percorre a hash de utilizadores, identificando os utilizadores ativos que possuem o prefixo desejado, como pedido na query.
Os resultados são ordenados por nome de forma crescente. Em casos de nomes iguais, o identificador do utilizador é utilizado como critério de desempate, garantindo uma ordenação consistente.
Utilizadores inativos são excluídos da pesquisa, mantendo a lista de resultados relevante no contexto.
A função addUserToList permite adicionar novos utilizadores à lista de resultados de forma ordenada e eficiente. Isto contribui para um tempo de execução otimizado, especialmente quando e se começarmos a trabalhar em conjuntos de dados maiores.
Nós optamos por copiar utilizadores para a lista de resultados usando a função copyUser para garantir que os dados originais não são modificados, mantendo a integridade das estruturas de dados.

# 4. Análise de Desempenho

Para avaliar o desempenho do nosso programa executamos as 6 queries nos 3 computadores (dos 3 elementos do grupo) e chegamos então aos resultados presentes na tabela 2 e as especificações de cada computador na tabela 1.
Os valores apresentados na tabela 2 são uma média de 3 execuções, após de uma primeira de aquecimento.

| | PC 1 | PC 2 | PC 3 |
|---|---|---|---|
| CPU | Intel i7-1165G7 4-core | Intel i5 1,8GHz Núcleo Duplo | Intel i9-9880H 8-core |
| RAM | 16 GB DDR4 3200MHz | 8 GB DDR3 1600 MHz | 16 GB DDR4 2667 MHz |
| Disco | 1 TB SSD | 128 GB SSD | 1 TB SSD |
| Sistema Operativo | Windows 11 WSL 1.0 | macOS Monterey 12.6.6 | Ventura 13.4.1 |
| Compilador | GCC 9.2.0 (Ubuntu 20.04) | Clang 14.0.0 | Clang 14.0.3 |

Tabela 1: Especificação dos PCs

| | PC 1 | PC 2 | PC 3 |
|---|---|---|---|
| Processor time taken | 0.171875 sec | 0.953453 sec | 0.558141 sec |

Tabela 2: Tempos de execução em diferentes PCs

# 5. Futuras limitações e possíveis melhorias

Quanto a futuras limitações destacamos os desafios de manutenção do código, ou seja.
Conforme o código evolui, desafios na manutenção podem surgir. Para isso uma boa documentação é essencial para garantir a continuidade e a compreensão eficaz do sistema ao longo do tempo.
Apesar de utilizarmos o valgrind para detetar memory leaks e não ser detetado nenhum como é possível ver pela imagem, podemos aferir pelos testes automáticos que temos de corrigir o problema dos leaks. 26.705 MB de leaks é demasiado e temos de averiguar como diminuir este valor e assim melhorar para a próxima fase do trabalho.

```
leak-check=full  ./programa-principal
==141== Memcheck, a memory error detector
==141== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==141== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==141== Command: ./programa-principal
==141==
==141== error calling PR_SET_PTRACER, vgdb might block
==141==
==141== HEAP SUMMARY:
==141==     in use at exit: 0 bytes in 0 blocks
==141==   total heap usage: 17 allocs, 17 frees, 4,048 bytes allocated
==141==
==141== All heap blocks were freed -- no leaks are possible
==141==
==141== For lists of detected and suppressed errors, rerun with: -s
==141== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No entanto ficamos satisfeitos com a escolha de estrutura que desenvolvemos já que nos permitiu realizar as queries de forma mais rápida e fácil.

# 6. Conclusão

Para concluir, acreditamos que ,apesar de tudo, o desenvolvimento da primeira fase do projeto de Laboratórios de Informática III foi muito satisfatória. Consolidamos os nossos conhecimentos relativamente à linguagem C e aprimoramos conhecimentos adquiridos nos anos anteriores.
O uso de HashTables foi o mais complicado, considerando a inexperiência no que toca ao uso desta estrutura em específico, no entanto ficamos agradados com o trabalho elaborado.
Como em tudo, ainda existe espaço para melhorias e pretendemos desenvolvê-las para a próxima fase do trabalho.