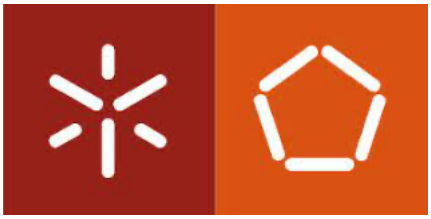

Universidade do Minho

Departamento De Informática

Licenciatura em Engenharia Informática
Laboratórios de Informática III

Trabalho Prático – Fase 2

Grupo 30

Diogo Andrade Fernandes (a100746)
Jéssica Cristina Lima da Cunha (a100901)
Samuel Macieira Ferreira (a100654)

Índice

Introdução

Este relatório apresenta a segunda fase do projeto desenvolvido no âmbito da unidade curricular de Laboratórios de Informática III. O propósito fundamental deste projeto é conceber uma solução eficiente para o processamento e armazenamento de grandes volumes de dados, permitindo um acesso otimizado a essas informações. Além disso, o projeto visa fortalecer os conceitos de modularidade, encapsulamento e abstração de código, visando torná-lo mais compreensível e fácil de manter.

Na fase inicial, fomos desafiados a criar uma aplicação com funcionalidades especificas, centradas em dados relacionados a utilizadores, passageiros, reservas e voos. Esta serviu para nos focarmos no desenvolvimento inicial da aplicação.

Neste relatório, concentramos a nossa atenção nos métodos selecionados para resolver consultas especificas, nas estruturas de dados adotadas para armazenar informações cruciais, na implementação de um modo iterativo e nos testes realizados para avaliar tanto a correção como o desempenho da aplicação. Esta fase não apenas solidifica os conhecimentos adquiridos, mas também amplia a compreensão da importância da eficiência no processamento de dados e na organização do código.

Ao explorar as secções subsequentes, será possível compreender em detalhes as escolhas e implementações realizadas, bem como os desafios superados no caminho para a construção de uma solução robusta e eficaz.

1. Encapsulamento e Modularidade

Nesta segunda fase, conscientes da importância da modularidade e do encapsulamento, promovemos alterações na estrutura do código. A transferência das *structs*, originalmente nos arquivos .h, para os .c, foi uma medida para garantir encapsulamento adequado dos dados e facilitar a modularidade nas funcionalidades.

Essas práticas visam promover uma estrutura mais coesa, alinhada com boas práticas de programação, proporcionando organização eficiente do código. A modularidade facilita a manutenção e compreensão, permitindo a modificação de partes específicas sem afetar o todo, enquanto o encapsulamento restringe o acesso direto a dados, promovendo uma abstração controlada e segura.

2. Menu iterativo

Na segunda fase, atendendo ao requisitado, ampliamos a funcionalidade do programa para além do modo *batch*, definido na primeira fase, implementando um modo iterativo.

O menu interativo implementado no código permite aos utilizadores interagirem com um sistema de gestão de catálogo, fornecendo diversas opções de consulta. O utilizador pode selecionar consultas numeradas de 1 a 10, cada uma oferecendo funcionalidades específicas relacionadas com o catálogo de dados. Além disso, existe a opção de sair do menu, indicando "0".

Ao escolher uma consulta, o utilizador é orientado a fornecer informações relevantes, como ids, datas, ou prefixos, dependendo da natureza da consulta selecionada.

O menu é projetado para ser intuitivo e interativo, guiando os utilizadores através das opções disponíveis e solicitando dados necessários para a execução das consultas. A estrutura do menu favorece uma interação fácil e direta, contribuindo para uma experiência eficiente do utilizador.

Além disso, o menu é sensível a erros, fornecendo feedback quando o utilizador fornece alguma entrada inválida e garantindo a integridade das consultas executadas. No final de cada operação, o programa libera os recursos utilizados e fornece a opção de sair ou realizar consultas adicionais.

Em resumo, o menu interativo oferece uma interface eficaz para interagir com o sistema de gestão de catálogo, permitindo aos utilizadores explorar e obter informações relevantes de forma personalizada.

3. Modulação

3.1. Estruturas de dados

3.1.1. Estruturas principais

Seguindo a divisão dos dados do *dataset* fornecido pela equipa docente, formam criadas as entidades principais:

- ***User***: contém a informação essenciais de um utilizador, resultante do processamento do *dataset* efetuado na criação dos catálogos;

- ***Flight***: contém a informação essenciais de um voo, resultante do processamento do *dataset* efetuado na criação dos catálogos;
- ***Reservation***: contém a informação essenciais de uma reserva, resultante do processamento do *dataset* efetuado na criação dos catálogos;
- ***Passenger*****:** contém a informação essenciais de um passageiro, resultante do processamento do *dataset* efetuado na criação dos catálogos.

3.1.2. Catálogos

Os catálogos representam estruturas de dados fundamentais no sistema implementado, organizando e armazenando informações relevantes de maneira eficiente para facilitar a execução das diversas *queries*. Cada catálogo possui uma finalidade específica e é projetado para armazenar dados relacionados a diferentes entidades do sistema.

CATALOG_FLIGHTS:

Utiliza uma tabela de *hash* **(*flights_hash*)** para mapear os voos com base em identificadores únicos.

Contém outra tabela de *hash* **(*airports_flights*)** para associar aeroportos aos respetivos voos.

CATALOG_PASSENGERS:

Utiliza uma tabela de *hash* **(*passengers_hash*)** para mapear informações sobre passageiros com base em identificadores únicos.

CATALOG_RESERVATIONS:

Utiliza uma tabela de *hash* **(*reservations_hash*)** para mapear as reservas de voos. Contém outra tabela de *hash* **(*hotel_reservations*)** para associar hotéis às respetivas reservas.

CATALOG_USERS:

Utiliza uma tabela de *hash* **(*users_hash*)** para mapear informações sobre usuários com base em identificadores únicos.

A escolha de tabelas de *hash* permite acesso rápido aos dados, proporcionando uma busca eficiente por identificadores únicos. Além disso, a segmentação em catálogos específicos permite uma organização clara e modularização do sistema, facilitando a manutenção e expansão do código.

4. Queries

Como na primeira fase já tínhamos 6 das *queries* feitas e não foram feitas alterações nas estruturas de dados que justificassem a sua alteração, a implementação das *queries* 1, 2, 3, 4, 8 e 9 permanece igual.

4.1. *Query* 1

Implementada na primeira fase, resolução inalterada.

Descrição: Listar o resumo de um utilizador, voo, ou reserva, consoante o identificador recebido por argumento.

Resposta: Para resolver esta *query,* começamos por verificar se o identificador é uma *flight*, *reservation* ou *user*, analisando se este é constituído apenas por números, se começa por "*Book*" ou outro, respetivamente. Depois deste processo, usamos a função da *glib*, *g_hash_table_lookup*, encontramos a entidade correspondente na *hashtable* do catálogo correspondente através do seu identificador e guardamos todas as informações necessárias no formato definido e é feita a impressão dessas informações no ficheiro de output.

4.2. *Query* 2

Implementada na primeira fase, resolução inalterada.

Descrição: Listar os voos ou reservas de um utilizador, se o segundo argumento for *fights* ou *reservations*, respetivamente, ordenados por data (da mais recente para a mais antiga). Caso não seja fornecido um segundo argumento, apresentar voos e reservas, juntamente com o tipo (*flight* ou *reservation*).

Resposta: Para a resolução da Query 2 a função Q2 é chamada com um catálogo e um ID como argumentos. Esta função, por sua vez, chama a função *flights_reservations_id_date_type*. A função *flights_reservations_id_date_type* inicialmente verifica se o usuário está inativo. Se estiver, retorna NULL. Caso contrário, a função com a ajuda de duas funções auxiliares (*get_flights_with_passenger* e *get_reservations_with_id*) cria duas listas, uma delas com os voos que incluem o passageiro com o ID fornecido e outra de reservas que também incluem o ID. Em seguida, a função aloca memória para um *array* de estruturas ID_DATE_TYPE, que guardará as informações necessárias para a *query,* um ID, uma data e um tipo (*reservation ou flight*). A função percorre a lista de voos, aloca memória para a nova estrutura ID_DATE_TYPE para cada voo, preenche os campos da estrutura e adiciona-a a um *array*. O mesmo processo é repetido para a lista de reservas. Finalmente, a função ordena o *array* de estruturas ID_DATE_TYPE por data e retorna o *array* como resultado. A função printQ2 é então chamada para imprimir as informações do *array*.

4.3. *Query* 3

Implementada na primeira fase, resolução inalterada.

Descrição: Apresentar a classificação média de um. hotel, a partir do seu identificador.

Resposta: Para resolver esta query verificamos inicialmente se o id do hotel era válido, em seguida utilizamos a *g_hash_table_lookup* para obtermos a estrutura de reservas associada ao hotel específico é então calculada a média das avaliações do hotel. Posteriormente, é usada a *snprintf* para converter o valor de *int* para *char* para depois, ser dado o output corretamente.

4.4. *Query* 4

Implementada na primeira fase, resolução inalterada.

Descrição: Listar as reservas de um hotel, ordenadas por data de início (da mais recente para a mais antiga). Caso duas reservas tenham a mesma data, deve ser usado o identificador da reserva como critério de desempate (de forma crescente).

Resposta: Para resolver esta *query*, primeiramente verificamos se o hotel especificado existe e se possui pelo menos uma reserva, em seguida ordenamos as reservas utilizando a função *qsort*, cada reserva é formatada e armazenada em uma *string*, que é então atribuída a um *array* de *strings* (*result*), quando terminadas todas as reservas o *array result* é retornado para depois ser dado o devido output.

4.5. *Query* 5

Implementada na segunda fase, mantendo a estratégia original.

Descrição: A *Query 5* tem como objetivo fornecer uma lista de voos partindo de uma determinada origem, dentro de um intervalo de datas especificado, ordenada por data de partida, da mais recente para a mais antiga. Caso dois voos tenham a mesma data de partida, o critério de desempate é o identificador do voo, seguindo uma ordem crescente.

Resposta: Na resolução da *Query 5*, inicialmente, todos os voos do catálogo são recuperados usando a função *get_all_flights*. Em seguida, é aplicado um filtro para manter apenas os voos que correspondem à origem fornecida e que estão dentro do intervalo de datas especificado. O *array* resultante é ordenado com base nas datas de partida. Em caso de datas iguais, o critério de desempate é o identificador do voo, assegurando uma ordenação crescente. Por fim, os voos filtrados e ordenados são formatados em *string* e armazenados em um a*rray* de *string* denominado *result.* Este *array* é então retornado como o resultado, representando a lista desejada de voos.

4.6. *Query* 6

Querie não implementada.

4.7. *Query* 7

Implementada na segunda fase.

Descrição: A *Query 7* proporciona uma lista dos principais aeroportos com base na mediana dos atrasos nos voos. A função *topNAirportsMedianDelays* é responsável por realizar a *query*, utilizando estruturas de dados e funções auxiliares.

Resposta: Na resolução da *Query 7*, a função *topNAirportsMedianDelays* itera sobre os aeroportos, calculando a mediana dos atrasos nos voos de cada um. Os resultados são armazenados em uma *GArray* chamada *airport*s, contendo estruturas *AirportDelays* com o nome do aeroporto e sua mediana de atrasos. A ordenação da *GArray* é realizada pela função *compare_airports*, que compara as medianas de atrasos entre dois aeroportos. Em caso de empate, o critério de desempate é o nome do aeroporto. Após a ordenação, os resultados são formatados em *string* e armazenados no *array result.* Esse *array* é então retornado como o resultado da *Query 7*, apresentando a lista dos principais aeroportos com base na mediana dos atrasos nos voos.

4.8. *Query* 8

Implementada na primeira fase, resolução inalterada.

Descrição: Apresentar a receita total de um hotel entre duas datas (inclusive), a partir do seu identificador. As receitas de um hotel devem considerar apenas o preço por noite (*price_per_night*) de todas as reservas com noites entre as duas datas.

Resposta: Para resolver esta *query* começamos por converter as *string* vinda do input para *DATE* usando a função *convertStringToDate*, em seguida, ela itera sobre as reservas associadas ao hotel especificando e calcula o total de receitas considerando apenas as reservas que se enquadram no intervalo de datas fornecido. O total de receitas é acumulado e é convertido para uma *string* usando *sprintf*, a *string* acaba então por ser retornada para depois ser dado o devido output.

4.9. *Query* 9

Implementada na primeira fase, resolução inalterada.

Descrição: Listar todos os utilizadores cujo nome começa com o prefixo passado por argumento, ordenados por nome (de forma crescente). Caso dois utilizadores tenham o mesmo nome, deverá ser usado o seu identificador como critério de desempate (de forma crescente). Utilizadores inativos não deverão ser considerados pela pesquisa.

Resposta: Para resolver esta *query*, primeiramente verificamos se o *prefix* é nulo, caso não seja prosseguimos com a função, usando a função *g_hash_table_iter_init* e verifica se o nome de cada usuário tem o prefixo especificado e se o status da conta é ativo, a lista é depois ordenada usando a função compareUsers é então preenchida uma string com a lista de resultados que é então retornada para ser dado o devido output.

4.10. *Query* 10

Querie não implementada.

5. Testes funcionais

Na fase de testes do programa, foi implementado um "programa-testes" para avaliar o funcionamento de cada *query* descrita na Secção 4. Os testes realizam a comparação entre o resultado obtido e o resultado esperado, indicando se o teste passou ou não.

Adicionalmente, o "programa-testes" fornece informações sobre o tempo de execução de cada *query*, o tempo de execução total e a memória usada pelo programa durante a execução.

5.1. Tempos de execução

Foram realizados testes em diferentes máquinas, variando o hardware e o número de repetições para obter uma análise abrangente do desempenho do programa. As máquinas utilizadas para os testes incluem.

	Sistema Operativo	**CPU**	**RAM**
Máquina A	Ubuntu 23.04 LTS 64-bit	I7-12700H 2.70Ghz	16 GB
Máquina B	Ubuntu 23.04 LTS 64-bit	Ryzen 5 7600X	32 GB
Máquina C	MacOs	Apple M1	16GB

	Queries	**Memória**
Máquina A	0.022182 seg	35072 KB
Máquina B	0.023983 seg	35072 KB
Máquina C	0.034542 seg	32063488 KB

5.2. Resultados e discussão

Os resultados obtidos indicam que o programa atende às expectativas, produzindo resultados consistentes com as *queries* implementadas. A análise do desempenho não revelou grandes variações nos tempos de execução em diferentes máquinas.

No entanto, eficiência na execução das *queries* não depende apenas do tempo de execução, mas também da capacidade de gestão de memória. Uma máquina pode apresentar um tempo de execução mais rápido, mas se o consumo de memória for excessivo, pode levar a problemas de desempenho em cenários de uso mais amplo.

A variação nos resultados destaca a importância de considerar a arquitetura do processador ao otimizar o desempenho do programa. A execução das *queries* pode ser afetada por características específicas do hardware, influenciando diretamente os tempos de execução e o uso de memória.

Concluímos então que os testes funcionais e de desempenho são cruciais para assegurar a robustez e eficiência do programa implementado. Este processo de validação contínua é essencial para garantir que o programa atenda eficazmente aos requisitos do utilizador em diversas condições.

Conclusão

Em resumo, consideramos que atingimos a maioria dos objetivos propostos no projeto pela equipa docente, implementando completamente as metas na primeira fase e enfrentando desafios na segunda devido a uma gestão de tempo menos eficiente, nomeadamente a não implementação de duas *queries*.

Destacamos a valiosa experiência com a biblioteca *glib* e a dedicação aos princípios de modularidade e encapsulamento de dados.

Para o futuro, planejamos concluir toda a implementação das *queries* que faltam e otimizar o código e as estruturas de dados, considerando a incorporação de estruturas auxiliares para reduzir tempos de execução e evitar procuras repetidas.