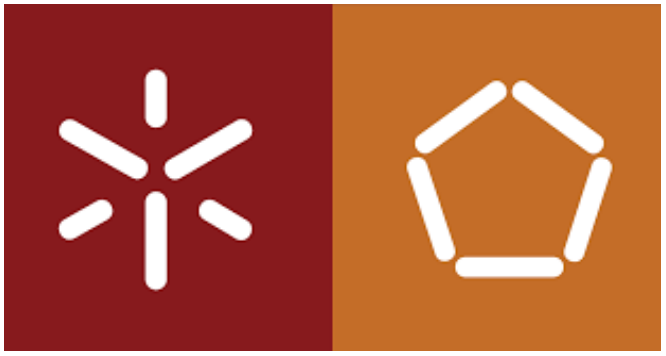


Universidade Do Minho
Engenharia Informática

Trabalho Prático

Laboratórios de Informática III 2023/2024

Relatório Grupo17 data de entrega: **25/01/2024**

Composição do Grupo 17

Tiago Pinheiro Silva
a93285

Tiago Alexandre Ferreira Silva
a93182

Carlos Alberto Fernandes Dias Da Silva
a93199

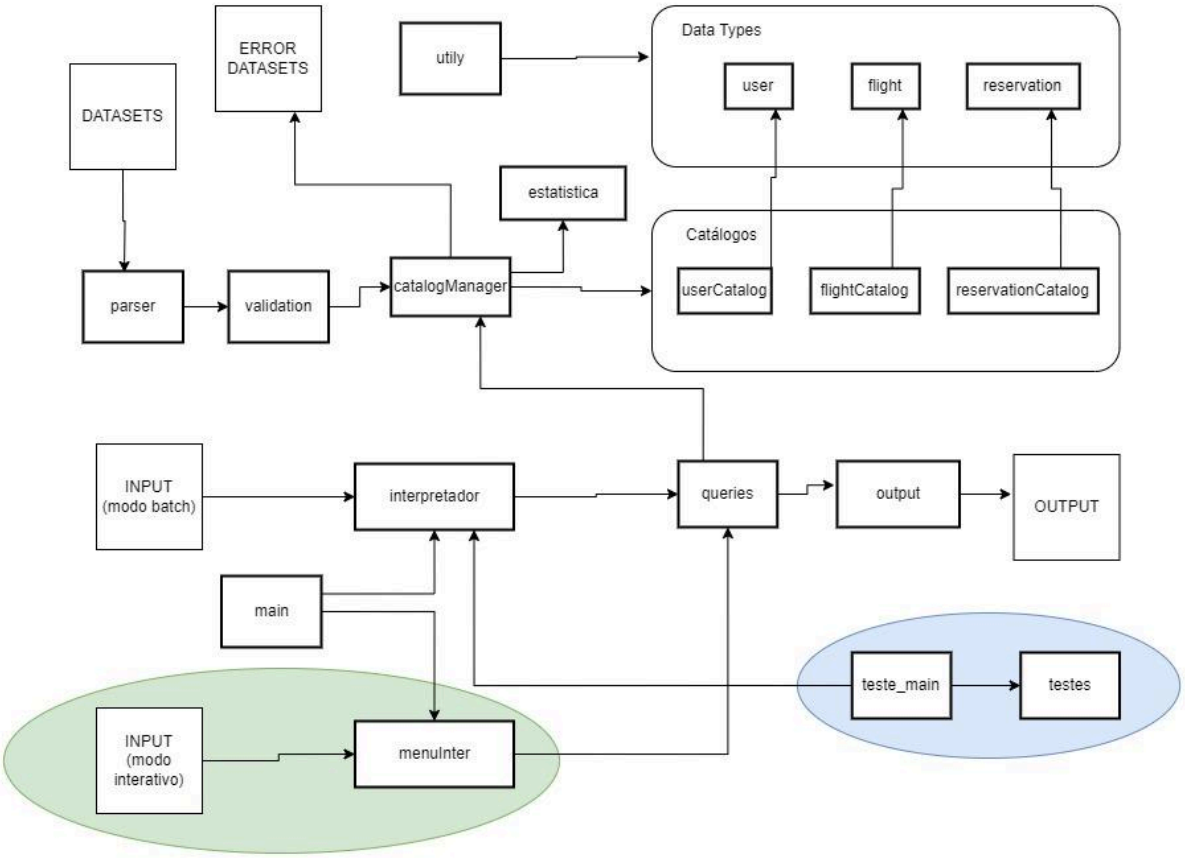
Índice

Introdução

Este trabalho prático da Unidade Curricular de Laboratórios de Informática III tem como fim consolidar e aprofundar competências de Engenharia Software e da linguagem de programação C.
Encontramo-nos então perante os objetivos de cimentar conhecimentos relativos à modularidade e encapsulamento, a estruturas dinâmicas de dados, validação de ficheiros e medir desempenho.

Paralelamente, este conjunto de objetivos ajudar-nos-á a solidificar o nosso domínio em C, junto da gestão eficiente de um repositório.

Desenho e Implementação



O nosso projeto foi estruturado em torno de uma arquitetura com três entidades que sobressaltam. USER, FLIGHT e RESERVATION, cada uma representada por structs correspondentes com os atributos que achamos necessários para o desenvolvimento deste sistema. Estas entidades encapsulam informações essenciais acerca de utilizadores, voos e reservas.

Cada uma dessas entidades é agrupada em diversos catálogos e a organização dos mesmos foi pensada de modo a facilitar a resolução das queries que nos foram propostas. Cada um dos módulos *xCatalog* gere as estruturas de dados no qual uma struct *x* está inserida. Todos esses *xCatalog*'s são por sua vez parte de um *catalogManager* que é a representação da totalidade da informação, sendo o objeto que irá ser alvo das queries no momento da sua execução.

```
struct userCatalog {
    GHashTable * user_by_id;
    GHashTable * user_by_initial;
    GHashTable * user_by_data;
};
```

```
struct flightCatalog {
    GHashTable * flight_by_id;
    GHashTable * flight_by_origin;
    GHashTable * flight_by_origin_by_data;
};
```

```
struct reservationCatalog {
    GHashTable * reservation_by_id;
    GHashTable * reservation_by_hotel;
    GHashTable * reservation_by_hotel_by_data;
};
```

Há ainda uma struct Estatística que foi idealizada com o intuito de potenciar um tempo médio de execução mais baixo como é o caso da Q7 onde apenas precisamos de fazer o cálculo das medianas uma vez.

```
struct catalogManager {
    FLIGHT_CATALOG flightCatalog;
    RESERVATION_CATALOG reservationCatalog;
    USER_CATALOG userCatalog;
    ESTATISTICA e;
};
```

Destacado a verde no esquema acima apresentado está o *menuInter* que é um módulo exclusivo do modo interativo do programa e que tem um papel semelhante ao interpretador na medida que chama as funções das queries quando assim é necessário.

Nesse mesmo esquema a azul, a *teste_main*, que contém a definição de main do programa-testes, e o módulo *testes* têm funções que permitem não só avaliar o desempenho do programa no que diz respeito a tempo e memória, como também detetar erros nos outputs gerados pelo programa.

Consoante a imagem acima apresentada, ainda temos também um interpretador que processa o ficheiro input.txt, e invoca as queries correspondentes a cada linha de input.

O módulo *queries*, como o próprio nome indica, é responsável responder às queries com base na informação presente no *catalogManager*.

Em *utily* estão funções auxiliares variadas que são usadas por vários módulos. Em retrospectiva, consideramos que este ficheiro poderia estar melhor organizado e talvez dividido em vários.

Estratégia de Queries

Q1 : Uso de hashtables em que a chave é o id da struct, permitindo acessos rápidos à informação;

Q2: Uso da mesma hashtable que a de cima, mas cada user tem uma GSList com o id das suas reservas e outra com os ids das suas reservas, depois e consultar as respetivas entradas nas respetivas hashtables, guardando a informação num array que depois vai ser ordenado por data;

Q3: Temos uma hashtable em que a chave é um hotel_id e possui um GList com todas as suas reservas, apenas é necessário percorrer todas essas reservas e calcular o rating médio.

Q4: Vamos buscar a entrada do hotel na hash, vemos qual é o tamanho da GList e criamos um array onde guardamos todas as reservas temporariamente, para depois ordená-las por data. Decidimos não ordenar no momento da inserção na hashtable pois é um processo demorado, visto que usamos GLists, além de que assim só é ordenado quando é necessário, diminuindo o tempo geral que demora a correr o programa, caso não tivéssemos em conta o tempo de inicialização do programa (fazer as estruturas iniciais e ler os ficheiros) seria mais eficiente já introduzir as reservas e voos pela ordem necessária.

Q5: Temos uma hashtable em que a sua chave é o aeroporto e o seu value é uma outra hashtable que tem como chave uma data, e valor uma lista com os respectivos voos que saíram desse aeroporto, tendo assim a data de início vamos ver todas as datas entre a data de início e fim dadas no input e passando os voos para um array que vai ser ordenado, caso seja a chave da hash seja igual à data de início e fim tem ainda o cuidado de ver se a hora é válida, caso não seja este voo não é escrito no documento.

Q6: Aqui usamos a hashtable que foi usada acima para percorrer todos os voos referentes àquele ano de todos os aeroportos, para guardar os valores criamos uma hashtable que tinha como chave o aeroporto e como valor o número de passageiros, quando percorremos um voo víamos a quantidade de passageiros presentes nesse voo e somava mos nos respectivos aeroportos de origem e destino.

Q7: devido à complexidade desta query a nível de tempo e como esta funciona criamos uma estrutura auxiliar de maneira a só a fazermos uma vez, guardando o seu primeiro resultado de forma persistente que pode depois ser consultado caso a query seja pedida novamente. A estrutura auxiliar nada mais é do que um par aeroporto+mediana que é guardado num GArray de forma persistente até o programa desligar. Começamos por usar a hashtable voo por aeroporto, criando um array de tamanho igual aos voos do respectivo aeroporto e a guardar todos os valores de delay neste. Depois ordená-mo-lo e calculamos a mediana, criando um novo par aeroporto+mediana, depois de o fazer para todos os aeroportos, ordenamos o GArray por delay e agr so é necessario consultar este GArray sempre que esta query for pedida.

Q8: De maneira a percorrer só as reservas necessárias, decidimos criar uma hashtable de res_por_hotel_por_data, em que cada reserva que estivesse ativa naquela data seria lá inserida (exemplo: uma reserva entre dia 5/10/2022 a 10/10/2022 irá aparecer em todas as datas que estão entre a data de início e fim exclusive). Mesmo sendo uma solução má a nível de memória, permite uma procura mais fácil visto que só precisamos de consultar a hashtable nas datas que constituem o intervalo passado como argumento.

Q9: Nesta query decidimos implementar uma hashtable em que a chave seria a inicial do nome do user e a chave uma gslist. Sabemos que esta não é a melhor solução, sendo que podíamos usar uma Tree para organizar os users em vez duma gslist. No entanto, a complexidade de implementação era muito superior e acabamos por optar por esta solução para resolver a query.

Q10: Nesta query só usamos uma nova hashtable de user, sendo esta ordenada por data de criação. Desta forma temos os 3 tipos de dados (users, voos e reservas) organizados por datas. De maneira a guardar a informação necessária, criamos uma struct auxiliar que irá guardar todos os campos importantes nesta query e uma data em que o seu tipo irá mudar dependendo do que for pedido na query. Caso não sejam dados argumentos a data será um ano, caso seja dado um ano a data será um mês, e caso seja dado um ano e mês a data será um dia. Esta data também será usada como

chave de uma hashtable que posteriormente será usada para guardar estas structs para termos acesso fácil a estas, visto que as vamos ter de alterar várias vezes. Além disso, a estrutura auxiliar também possui uma hashtable que serve de apoio para ignorar entidades repetidas.

Menu Interativo

```
superfake@MSI:~/grupo-17/trabalho-pratico$ ./programa-principal
Por favor indique qual o caminho do dataset a processar!
Path:
../../data
O ficheiro de Users foi processado!!!
O ficheiro de Reservas foi processado!!!
O ficheiro de Voos foi processado!!!
O ficheiro de Passageiros foi processado!!!
*********************************************
*                                           *
*              Welcome to LI3               *
*              Menu Interativo              *
*                                           *
*********************************************
1. Query 1
2. Query 2
3. Query 3
4. Query 4
5. Query 5
6. Query 6
7. Query 7
8. Query 8
9. Query 9
10. Query 10
0. Sair
Escolha uma opção:
```

Desenvolvemos um menu interativo, porém não implementamos paginação. O resultado de uma query é apresentado da mesma maneira que um comando com a flag F é escrito num ficheiro no modo batch.

Análise de desempenho

Com alguns memory leaks à mistura, de maneira geral o nosso programa usa demasiada memória. No entanto, desde o início do projeto, tencionamos privilegiar o tempo de execução mesmo que isso envolvesse estruturas de dados extra e outros sacrifícios de memória.

Não conseguimos otimizar o programa e prepará-lo para receber um dataset grande. Mesmo testando localmente, o programa acaba por esgotar toda a memória durante o processamento de ficheiros/criação e preenchimento de catálogos. Por essa razão, apenas conseguimos testar o com o dataset regular.

```
$ ./programa-testes ~/dataset/data ~/dataset/input.txt  ~/dataset/outputs/
O ficheiro de Users foi processado!!!
O ficheiro de Reservas foi processado!!!
O ficheiro de Voos foi processado!!!
O ficheiro de Passageiros foi processado!!!


Queries corretas: 100
Queries incorretas: 0
Percentagem de acerto: 100.000000
Memory usage: 55436 KB

Tempo médio de execução de cada query

Query 1: 0.000025
Query 2: 0.000031
Query 3: 0.000056
Query 4: 0.000947
Query 5: 0.009584
Query 6: 0.096279
Query 7: 0.000072
Query 8: 0.002159
Query 9: 0.000435
Query 10: 0.208150
```

```
superfake@MSI:~/grupo-17/trabalho-pratico$ ./programa-testes ../..
O ficheiro de Users foi processado!!!
O ficheiro de Reservas foi processado!!!
O ficheiro de Voos foi processado!!!
O ficheiro de Passageiros foi processado!!!


Queries corretas: 100
Queries incorretas: 0
Percentagem de acerto: 100.000000
Memory usage: 54968 KB

Tempo médio de execução de cada query

Query 1: 0.000013
Query 2: 0.000017
Query 3: 0.000042
Query 4: 0.000440
Query 5: 0.002454
Query 6: 0.026377
Query 7: 0.000046
Query 8: 0.000640
Query 9: 0.000275
Query 10: 0.067046
```

Calculamos a taxa de acerto face ao output esperado, bem como o tempo médio da execução de cada query e a memória máxima utilizada. Quando ainda não tínhamos todas as queries bem implementadas, foi muito útil o programa-testes, pois quando o nosso output não corresponde ao esperado, é imprimido o nome do ficheiro que não está correto juntamente com a linha esperada e a linha errada.

As estatísticas geradas não serviram como métrica para continuar a aprimorar o desempenho, visto que durante o desenvolvimento do trabalho sempre estivemos atrás de conseguir atingir a correção total do programa, não havendo uma fase que pudéssemos parar para pensar em como reformular ou abordar problemas de maneiras diferentes em prol da memória e do tempo de execução.

Limitações e aspectos a melhorar no futuro

Memory Leaks

Não conseguimos tratar de todas as memory leaks. É um aspeto que seria importante resolver, pois é provável que, com a resolução dessas leaks, outros erros relacionados com o mau desempenho fossem identificados e igualmente tratados. Sabemos que ocorrem no momento de inserção nos catálogos e que estão relacionados com a duplicação de chaves sempre que algo é inserido numa das GHashTables.

Algoritmos e estratégias novas

Tal como referido na explicação de algumas queries, várias abordagens estão longe do ideal. Depois dos memory leaks, algo a trabalhar no futuro seria o desenvolvimento de novas estratégias e algoritmos que permitissem melhorar tanto a nível de tempo de execução como de memória utilizada. Estratégias essas que passariam pelo uso de estruturas de dados mais indicadas, porque, embora algumas decisões de

estruturas tenham sido boas, outras foram medíocres e ficaram mais distantes de uma solução ótima do que deviam.

Conclusão

Em suma, apesar de termos ficado aquém das nossas próprias expectativas, consideramos que o desenvolvimento deste trabalho foi positivo, na medida que respeitamos, de uma forma geral, os conceitos de modularidade e encapsulamento que eram dois dos tópicos mais importantes a aprender e consolidar desta Unidade Curricular.