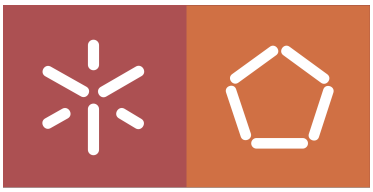

Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Relatório Trabalho Prático - Fase 2
LEI - 2º Ano - 1º Semestre

A98695 Lucas Oliveira
A89292 Mike Pinto
A96208 Rafael Gomes

Braga,
5 de fevereiro de 2023

Conteúdo

1 Introdução

Na Fase 2 do Trabalho Prático da Unidade Curricular de Laboratórios de Informática III, foi-nos proposto desenvolver uma aplicação realizada na *Linguagem C* dando continuidade da Fase 1 do projeto. Foram realizadas nesta fase as seguintes alterações/adições:

- *M*elhorias sugeridas pelos docentes à primeira fase do projeto (generalização do módulo de estatísticas e do módulo de *parser*, implementação de estruturas dinâmicas e melhoria do encapsulamento);
- A implementação das restantes *Queries*;
- Realização do modo interativo usando o terminal;
- Módulo de testes.

2 Abordagem

Nesta fase do trabalho prático foi realizada a criação de vários novos módulos:

- *Módulos Testes*: Modulo referente à realização de testes da execução do programa;
- *Módulos Interativo/Navegador*: Modulo referente à implementação de um navegador interativo;
- *Módulos de ligação Catalogo/Hash Table*: Modulos referentes a funções do módulo de *Hash Tables* e do catálogo de utilizadores/condutores;

Para além dos novos módulos, as diferentes estruturas de dados utilizadas foram alteradas, tornando o seu comportamento de alocação de memória dinâmico, generalizaram-se os módulos de estatística e de *parsing* como também a implementação de funções de verificação e validação de *strings*.

3 Desenvolvimento

3.1 Queries

3.1.1 Query 5

Na *querie 5* era pedido para calcular o preço médio das viagens num determinado intervalo de datas. Inicialmente percorremos o catálogo das *Rides* verificando as datas de cada viagem, se estas se encontrarem dentro do intervalo dado. Se a data da viagem estiver nesse intervalo, é então calculado o custo dessa viagem usando as informações de distância e *car class* associada ao condutor dessa viagem. Caso a viagem não se encontre dentro do intervalo pedido, é lido a próxima viagem do catálogo.

3.1.2 Query 6

Na querie 6 era pedido para calcular a distância média numa cidade num determinado intervalo de datas. Inicialmente percorremos o catálogo das rides verificando as datas e a respetiva cidade. Se a viagem tiver ocorrido na cidade pedida no intervalo de datas é então guardado a distancia dessa viagem e incrementado o número de viagens. No fim é dividido a distância final pelo número de viagens totais encontradas, retorna a distância média da cidade pretendida. No caso onde a viagem não se encontra dentro do intervalo dado ou não ser a cidade pretendida, avança para a viagem seguinte.

3.1.3 Query 7

Na querie 7 era pedido os tops N condutores numa determinada cidade, onde primeiro são ordenados pela *avaliação média* dos condutores e em caso de empate, ordena pelos *ids* dos mesmos em ordem decrescente. Inicialmente percorremos o catálogo dos drivers e verificamos se este existe ou não, caso exista verificamos se tem estatísticas associadas a ele, caso tenha verificamos se o driver está ativo ou inativo e também se o score do driver existe. Ao final das verificações ele adiciona o driver à lista, onde reorganizamos com os parâmetros pedidos. Se falhar nas verificações, avançamos para o próximo driver. Por fim, imprimi-mos os tops N da lista criada.

3.1.4 Query 8

Na querie 8 era pedido para listar todas as viagens dos utilizadores e condutores do género pedido e que tivessem contas criadas com igual, ou mais anos do tempo de conta pedido, onde primeiro são ordenadas pelas contas mais antigas do condutos, depois se necessário, ordenar pelas contas mais antigas dos utilizadores, por fim se persistir o empate, ordenar pelo o id da viagem (por ordem crescente). Inicialmente percorremos o catálogo das rides e verificamos em cada viagem se o utilizador ou condutor têm conta ativa, caso sejam, verificamos se o género de ambos é o pedido, se sim, adiciona-mos essa viagem à lista, onde reorganizamos com os parâmetros pedidos. Caso falhe nas verificações, avançamos para a próxima viagem. Por fim, imprimi-mos todas as viagens da lista criada.

3.1.5 Query 9

Na querie 9 era pedido para listar as viagens, onde o utilizador deu gorjeta num determinado intervalo de tempo, onde primeiro são ordenados por ordem de distância percorrida (em ordem decrescente), caso existem empates, as viagens mais recentes aparecem primeiro, por fim se persistir o empate, ordenar pelo id da viagem (por ordem decrescente). Inicialmente percorremos o catálogo das rides e verificamos se o utilizador deu gorjeta nessa viagem, e também verificamos se a data da viagem está entre o intervalo de tempo pedido, se sim, adicionamos à lista, onde reorganizamos com os parâmetros pedidos. Caso falhe nas verificações, avançamos para a próxima viagem. Por fim, imprimi-mos as viagens da lista criada.

4 Estruturas

4.1 Estruturas Utilizadas

Em semelhança à fase 1 realizada neste trabalho prático foram usadas estruturas como listas ligadas, *Hash Tables*, *Arrays* e estruturas de "catalogo"(catalogo de utilizadores, condutores, viagens e estatística).

Destas estruturas, em contraste à fase 1, foi tornado as estruturas de *Hash Tables* e *Arrays* em estruturas genéricas e dinâmicas em relação ao seu tamanho.

4.1.1 *Hash Tables*

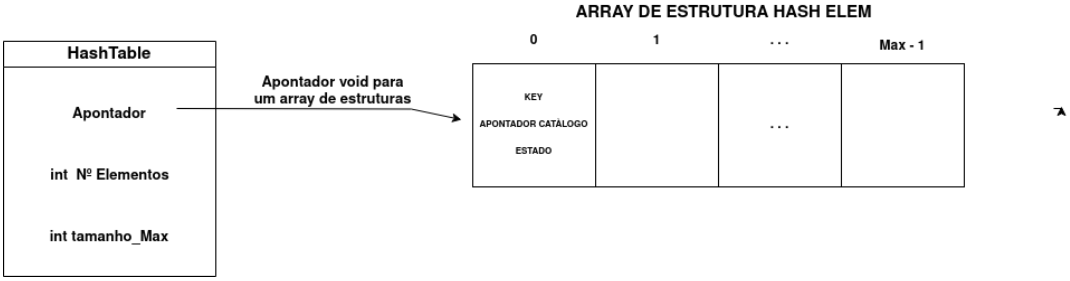

Figura 1: Diagrama a exemplificar uma HashTable.

A estrutura das *Hash Tables* contem um apontador *void* para um array de uma estrutura secundaria *Hash_Elem* que contem respetivamente uma Chave, um apontador *void* utilizado para apontar para os diversos catálogos(utilizadores, condutores e viagens) e uma variável "estado"usada para sinalizar se o índice da *Hash Table* está ocupado ou não. Com esta generalização esta estrutura ganhou o seu próprio modulo onde com a implementação de módulos adicionais de ligação para os catálogos de utilizadores, condutores e de estatística de forma poder respeitar o encapsulamento. Quando esta estrutura está com 70% da capacidade é então alocado outro array da estrutura secundaria com o dobro da capacidade máxima, feito o *reHash* dos elementos contidos no *anterior* e atualizando o apontador na estrutura principal..

No catálogo de Estatísticas foi ainda adicionado um apontador para uma lista ligada utilizada para guardar as estatísticas das viagens de cada condutor nas diferentes cidades.

4.1.2 *Array*

De forma semelhante à estrutura das *Hash Tables*, o comportamento desta estrutura é dinâmica, tento uma estrutura principal que contem um apontador para um *Array* de apontadores *void*, uma variavel para guardar a capacidade máxima da estrutura e outra para indicar qual o proximo indice a ser utilizado. Quando esta estrutura possui elemento igual à capacidade máxima menos dois é feito então a realocação de memória para o dobro do tamanho do array de apontadores.

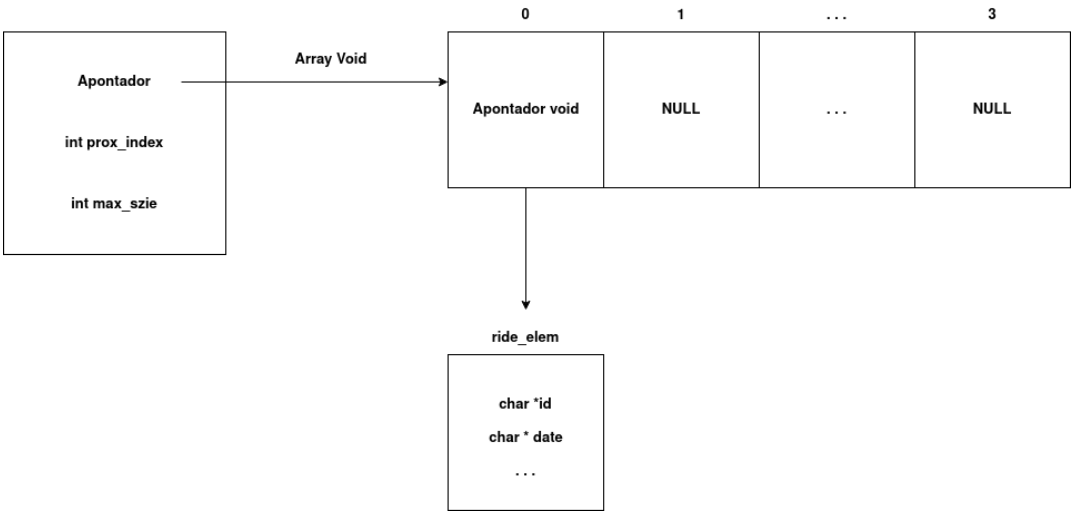

Figura 2: Diagrama a exemplificar um Array.

5 Modo de Operação Interativo

Este modo de operação interativo consiste numa ligação entre o utilizador e o programa. Este pode ser acedido caso o utilizador não forneça os ficheiros necessários para executar o programa de modo automático.

Inicialmente, através da função switch-menu() é apresentado o menu principal com a possibilidade de avançar para o menu das queries ou encerrar o programa. Caso optemos pelo menu das queries este passa a ser gerido pela função switch-queries, permitindo executar de 1 a 9 Queries individualmente, conseguirmos retornar para o menu principal e até mesmo encerrar o programa. Cada uma das Queries tem uma função que recebe um respetivo comando que imprime o resultado desejado, sendo essa função denominada por nav-queries-(nºquerie)(). Caso a querie selecionada tenha sido a 2, 3, 7, 8 ou 9 o output desta será feito pela função pagination() que imprime uma lista de resultados, tendo ainda a possibilidade de escolher o número de linhas que queremos por página. No final da execução de cada Query podemos escolher entre voltar a executar essa Query novamente, retornar para o menu das queries e retornar para o menu principal.

6 Módulo testes

Para a realização do módulo testes, utilizamos a biblioteca *time.h* onde nos ajudou a saber o tempo que demorava a executar o programa. Começamos por as definir as variáveis start e end onde iríamos guardar o tempo de execução de cada querie e inicialização de estruturas, e definimos as variáveis startP e endP para guardar o tempo de execução do programa. Em primeiro lugar, inicializamos a variável startP e utilizamos a função *clock()* onde retorna o horário em que foi chamada e nisso guardamos na variável startP, de seguida inicializamos o start e utilizamos o *clock()* para sabermos o tempo que ia começar antes do parser das estruturas dos catálogos, depois de realizar o parser, voltamos a utilizar a função *clock()* e guardamos na variável end, depois realizamos o cálculo de tempo que demorou, ou seja, fizemos (end - start) e dividimos pelo CLOCKS PER SEC (que representa o número de clocks do processador por segundos) e devolve o tempo que demorou em segundos. De seguida repetimos o mesmo processo para cada querie, mas verificamos duas condições, primeiro vemos se os resultados das queries executadas são iguais aos resultados dos comandos esperados pelo utilizador, e caso sejam iguais, verificamos se cada teste executado demorou menos de 10 segundos, se sim imprime o tempo que cada teste demorou a realizar, caso contrário, imprime que o teste passou mas que demorou mais que o tempo determinado, e caso os resultados sejam diferentes, imprime que o teste falhou e diz em que comando falhou. Por fim, voltamos a utilizar a função *clock()* e guardamos no endP, e realizamos outra vez o mesmo cálculo mas agora para saber o tempo de execução do programa todo ((endP - startP) dividindo por (CLOCKS PER SEC)).

6.1 Dados estatísticos dos testes

Para realizarmos os dados estatísticos dos testes temos estes 3 computadores com as seguintes:

- Computador (Lucas) -
  - Sistema Operativo: Ubuntu 20.04.5 LTS 64-bit;
  - Hardware:
    - Processador: Intel® Core™ i7-4510U CPU @ 2.00GHz × 4;
    - Memória RAM: 12GB;
- Computador (Rafael) -
  - Sistema Operativo: Ubuntu 21.04 64-bit;
  - Hardware:
    - Processador: Intel® Core™ i7-10510U CPU @ 1.80GHz × 8;
    - Memória RAM: 16GB;
- Computador (Mike) -
  - Sistema Operativo: Manjaro Linux x86_64
  - Hardware:
    - Processador: Intel i5-7200U (4) @ 3.100GHz

* Memória RAM: 8 GB + 4 GB SWAP

Para a verificação do desempenho do programa utilizamos os seguintes datasets, *Dataset-regular*, *Dataset-regular-errors*. Em que cada Dataset utilizamos dois ficheiros de input, um ficheiro contém as 9 queries (sendo que nesse ficheiro contém 2 queries-1 para verificar o user e o driver e 2 queries-8 para verificar o género masculino e feminino), o outro ficheiro contém 50 linhas com queries aleatórias.

6.1.1 Desempenho das Queries

Queries	Computadores	Tempos Médios
1	Lucas	0.000167 s
	Rafael	0.000215 s
	Mike	0.000153 s
2	Lucas	0.026516 s
	Rafael	0.033741 s
	Mike	0.087200 s
3	Lucas	0.136991 s
	Rafael	0.225701 s
	Mike	0.118039 s
4	Lucas	0.000080 s
	Rafael	0.000120 s
	Mike	0.000071 s
5	Lucas	0.695401 s
	Rafael	0.800352 s
	Mike	0.775318 s
6	Lucas	0.502674 s
	Rafael	0.629323 s
	Mike	0.454054 s
7	Lucas	0.515666 s
	Rafael	0.854331 s
	Mike	0.462176 s
8	Lucas	2.073465 s
	Rafael	2.951129 s
	Mike	1.843101 s
9	Lucas	1.028267 s
	Rafael	1.759218 s
	Mike	0.977861 s
Tempo médio a executar o programa	Lucas	13.645079 s
	Rafael	18.6836470 s
	Mike	16.617666 s

6.2 Explicação de desempenho

Relativamente ao desempenho com o Datalarge não conseguimos cumpri-lo nos requisitos pedidos, isto é, tanto em tempo execução e em termos de memória alocada. Ao qual, a nossa memória ronda entre os 400 e 500 megabytes. Um dos motivos pelo qual não consegue executar no tempo pedido é devido à implementação usada para listar as *querie 2, 3, 7, 8 e 9* pois acabou por não ser

a melhor estratégia.

7 Conclusão

A partir dos conhecimentos obtidos ao longo do semestre, achamos que o trabalho mostra respeitar os conceitos de modularidade e encapsulamento, sem danificar a performance das queries. Muito pelo contrário, nós conseguimos melhorar a eficiência das funções, resolvendo múltiplos problemas de memória e corrigindo ineficiências encontradas pelo código. Implementamos um menu simples, bonito e eficaz. Realizamos testes para observar a performance deste trabalho. E acima de tudo conseguimos implementar tanto um parser genérico e estruturas dinâmicas, Em conclusão, acreditamos ter realizado as tarefas propostas respeitando os princípios de modularidade e encapsulamento.