Universidade do Minho
Relatório Laboratórios de Informática III

A100535, Gonçalo Nuno da Silva Loureiro
A100538, Ivan Sérgio Rocha Ribeiro
A100709, Pedro Miguel Meruge Ferreira

Índice

1. Introdução

No âmbito da disciplina de LI3, desenvolvemos um projeto no qual atribuímos prioridade aos conceitos de encapsulamento e modularidade. Este projeto consiste em carregar três ficheiros de dados para memória, para responder a diversas questões sobre esses dados. Para tal, usamos diferentes estruturas de dados e estratégias de encapsulamento a descrever.

2. Estruturas de Dados

2.1. Users

Visto que os utilizadores são normalmente identificados pelo *username* (no formato *string*) decidimos utilizar uma *GHashTable*, (encapsulada com o nome *UserData*), que faz corresponder um *username* a uma *struct* com as informações de um utilizador. Cada struct (encapsulada com o nome *UserStruct*) guarda um nome, data de nascimento e data de criação de conta no formato string, tal como género, método de pagamento e estado.

Parece-nos uma estratégia viável, considerando o facto de haver um número relativamente reduzido de utilizadores (100.000) e de tal estrutura tornar o tempo de acesso à informação de um utilizador reduzido. O compromisso no tempo inicial de construção da *GHashTable* é depois superado pelo tempo constante de acesso à *struct* de cada utilizador, em cada chamada.

Os dados são carregados/destruídos com `getUserData`/`freeUserData`, chamadas na *main* apenas.

2.2. Drivers

Como os condutores estão categorizados por ID, ordenado de 1 a 1.000.000 de modo contíguo, utilizamos um *array* (encapsulado com o nome *DriverData*) para guardar structs com as informações de cada driver. Simplificadamente, o ID do driver 'i' corresponde à posição 'i-1' no array. Cada struct (encapsulada com o nome DriverStruct) guarda um nome, data de nascimento, matrícula, cidade e data de criação de conta no formato string, tal como género, classe de carro e estado.

Em maior pormenor, utilizamos um array primário de arrays secundários de 1000 drivers cada, pois julgamos que arranjar espaço contíguo na *heap* para todos os drivers (1.000.000) seria menos eficiente que esta solução.

Para além disso, arrays são estruturas simples e extremamente rápidas de utilizar, visto que, neste caso, procurar um driver por ID corresponde apenas a aceder

a um índice em tempo constante. Os dados são carregados/destruídos com `getDriverData`/`freeDriverData`, chamadas na *main* apenas.

2.3. Rides

Tal como para os drivers (ver 2.2), foi utilizado um array primário de arrays secundários de 1000 viagens cada, por razões análogas. Cada viagem tem uma *struct* (encapsulada com o nome *RidesStruct*), com informação de data, ID do condutor, nome do utilizador, cidade, distância percorrida, avaliação do utilizador e condutor, gorjeta e comentário. Esta escolha tem a desvantagem óbvia de ocupar uma grande quantidade de memória (são 1.000.000 de viagens).

Em paralelo, foram construídas outras estruturas de dados para organizar as viagens de forma acessível às *queries*. Entendemos que o tempo perdido a construir estas estruturas no início do programa é compensado pelo acesso rápido e independente das *queries* a eles, em fases posteriores. Soluções como construir estas estruturas nas *queries* repetidamente, ou recuperar cada estrutura após ser criada numa *query* pareceram-nos soluções menos eficientes e robustas.

Todas as estruturas relativas a viagens mencionadas (encapsulados em conjunto com o nome *RidesData*) são, de momento, carregadas/destruídas em simultâneo, com `getRidesData`/`freeRidesData`, chamadas na *main* apenas. Nomeia-se, em seguida, as estruturas criadas "em paralelo".

2.3.1. Rides ordenadas com base em cidades

Cria-se uma *GHashTable* que usa como *keys* os nomes das cidades e lhes faz corresponder um *GPtrArray* com pointers para rides de uma cidade, através do tipo *CityRides* e das funções que lhe estão associadas.

Para além disso, cada *GPtrArray* das cidades está ordenado por data, utilizando multithreading para tornar o processo mais rápido, o que diminuiu o tempo real de loading de dados em cerca de 30%, mas aumentou o de CPU em cerca de 45%.

Ainda assim, o *sort* destes arrays é muito custoso em termos de CPU, mas isto é compensado pela utilidade desta estrutura , por exemplo, na Q4, Q5 e Q6.

2.3.2. Rides ordenadas com base em condutores

Cria-se um *GPtrArray* (encapsulado com o nome *RidesByDriver*) de 10.000 posições (número de condutores). Acede-se com o ID do condutor de uma ride (pela mesma razão que em 2.2) à posição no array 'ID-1' em tempo constante (mesma razão que em 2.2). Incrementa-se sucessivamente os parâmetros na struct, conforme os valores obtidos em cada viagem, para ocupar espaço reduzido. Nomeadamente, Avaliações são guardadas num array de 5 posições, correspondentes a cada tipo de avaliação, o que permitirá calcular a avaliação média e o total de viagens em simultâneo. Cada struct do condutor (*driverRatingInfo*) guarda, no final, informação

relativa à avaliação média total, total de gorjetas, data de viagem mais recente, número de viagens e número do condutor. Existem *structs* sem informação para eventuais drivers que não tenham qualquer viagem associada, o que previne problemas ao usar a função `sort_byRatings` sobre este array.

É uma solução que ocupa pouca memória e com versatilidade para expansão (com ajuda da Glib). Em comparação com um *GArray*, funções de *sort* aplicadas a esta estrutura são mais rápidas. Esta estrutura é-nos útil na Q1 e Q2.

3. Parsing

3.1. Parsing dos dados

Os ficheiros de input são todos abertos no módulo "files.h" com a função `open_cmdfiles`. Com recurso a `strcpy` e `strcat` constrói-se o caminho correto dos ficheiros a abrir.

De seguida, os ficheiros "*users.csv*", "*drivers.csv*" e "*rides.csv*" são carregados, respetivamente, nos módulos "*userData*", "*driversData*" e "*RidesData*", de forma independente e antes de qualquer query ser chamada . Os três módulos seguem a estratégia de ler em ciclo linhas dos ficheiros ".csv" com a função `fgetc`, guardando os caracteres num *buffer* até atingir caracteres ';' ou '\n'. Cada buffer é depois copiado para um elemento de uma struct *UserStruct*, *DriverStruct* ou *RidesStruct*.

3.2. Interpretação dos comandos

O ficheiro de input das queries é carregado no módulo "query_requests" com a função `queryRequests`. Cada linha do ficheiro é lida com `getline` e separada em segmentos de *string* com *`strsep`*. Para acelerar e economizar o processo de atribuição dos inputs às respectivas queries, criamos uma *dispatch table*, e atribuímos quaisquer segmentos de string obtidos, menos o primeiro, a uma *query* (o primeiro caractér de cada linha de "input.txt" é o número da query a executar, que é utilizado para escolher a query de destino da dispatch table). Deste modo, evitamos o uso de funções pesadas como `sscanf` e de múltiplos *ifs* ou uma *switch* que atrasariam a atribuição às queries, numa secção central do programa. Cada query recebe o número de inputs que precisa e, ao ser implementada no seu módulo, ignoram-se os restantes inputs NULL recebidos.

3.3. Handling do output

Ainda no módulo "*query_requests.h*", qualquer *query* retorna uma string à função `queryRequests`. Se uma query precisar de retornar várias linhas, as várias strings são concatenadas na própria query, antes de retornar o resultado. A função `writeResults` recebe essa *string* e um *int*, que controla o número do comando no ficheiro "*input.txt*" a que a *string* corresponde. Através da função `snprintf`, obtém-se

o destino do ficheiro de output ("*Resultados/command%d_output.txt*") e cria-se um novo "*.txt*" com o output, dentro da pasta *Resultados*.

4. Queries

4.1. Query 1

Esta *query* requer informações sobre *drivers* (nome, género, idade, avaliação média, número de viagens e total auferido) e *users* (nome, género, idade, avaliação média, número de viagens e total gasto). Para obter estas informações, precisamos de aceder às respectivas *structs DriverData* e *UserData* e também à *struct* das viagens *RidesData*. Depois deste acesso, usamos as funções de *get* para conseguirmos obter as informações que nos são pedidas, algumas destas são, por exemplo, `getDriverPtrByID`, `getUserPtrByUsername` e `getRidesByDriver` que nos vão possibilitar usar de usar outras como, por exemplo, `getDriverName`, `getDriverGender`, `getUserName e getUserGender`.

4.2. Query 2

Esta query requer os N condutores com maior avaliação média. Usamos a estrutura *ridesByDriver*, referidas nas rides (ver 2.3.2). Chamando a função `getRidesByDriver`, obtém-se um *GPtrarray* com o resumo da informação de todas as viagens em que cada condutor participa. Organiza-se o *GPtrArray* referido com base em avaliações médias, viagem mais recente e ID dos condutores (por esta ordem de importância). Para tal, usa-se a função `qSortArray`, a receber como argumento a função `sort_byRatings`. De seguida, na função *strResults* lê-se N elementos dos maiores índices da estrutura (correspondentes aos drivers com melhor avaliação média), saltando eventuais condutores que tenham estado inativo.

4.3. Query 4

Esta query requer preço médio das viagens de uma cidade. Para tal, usamos a estrutura `CityRides` sobre cidades, referida nas rides (ver 2.3). Basta usar `getRidesByCity` para obter as rides associadas a uma cidade, e depois funções como `getCityRidesByIndex` para iterar sobre elas, somando o número de viagens por classe de carro e as suas distâncias. Para recolher os dados, visto que as structs usadas são opacas, utilizam-se várias funções de get, tais como `getNumberOfCityRides` para iterar no loop, `getRideDriver`, `getDriverCar` e `getRideDistance`.

4.4. Query 5

Esta query requer o preço médio das viagens entre duas datas. Decidimos implementar, por simplicidade e pelo facto de os arrays das cidades estarem ordenados por data, a mesma estrutura da Q4. Porém, visto poderem haver um número arbitrário de cidades a percorrer, foi criada a função `iterateOverCities` para aplicar uma função e um pointer para dados quaisquer ao array de uma cidade, usando um `GHashTableIter`. Neste caso, foi criada uma struct com os dados necessários (acumuladores e as datas), e uma função que recebe um pointer para essa struct e o array de uma cidade, iterando sobre os elementos da cidade que estão entre as datas que queremos. No final, cada iteração da função sobre cada cidade terá alterado os campos da struct de modo a ficar com os dados que precisamos para calcular o preço médio, tal como na Q4.

4.5. Query 6

Esta query requer distância média percorrida por cidade, num intervalo de datas. Para tal, usamos uma estratégia semelhante à da Q4, visto, novamente, que os arrays das cidades estão ordenados por datas, e iteramos sobre as rides recolhendo o número de viagens e as suas distâncias, até ultrapassar a data máxima.

5. Modularidade e encapsulamento

Dividimos o projeto em vários módulos. No módulo "*main*" consta apenas a função main, que chama todas as outras funções do projeto. No módulo "*files*" os ficheiros de input são abertos e testam-se erros na abertura. Cada função que processa uma query tem o seu módulo "*query_x*". No módulo "*query_requests*" os vários comandos de *input* são redirecionados para os respetivos módulos de queries. Também neste módulo os resultados são escritos para ficheiros de output. Nos módulos "*userdata*", "*driverdata*" e "*ridesdata*", são transferidos os dados dos ficheiros de input "*users.csv*", "*drivers.csv*" e "*rides.csv*", respetivamente, para as estruturas de dados referidas acima (ver secção 2). No módulo "*commonParsing*" constam funções usadas por esses módulos no *parsing* dos ficheiros de input referidos na linha anterior.

Nas várias queries, para consultar os dados sobre os ficheiros, são disponibilizadas structs opacas pelos módulos "*userdata*", "*driverdata*" e "*ridesdata*". Com funções de *"get",* é possível extrair os seus dados ou aceder a estruturas auxiliares, que têm as suas próprias funções. Se os dados em questão forem pointers (como por exemplo a string de uma data), é retornada uma cópia da mesma para que não haja interferência com a base de dados original.

6. Por implementar

Com vista a partilhar a estrutura que planeamos para o trabalho, apresentamos de seguida algumas ideias sobre estrutura e desempenho que pensamos implementar na próxima fase.

Pretendemos implementar um módulo que recupere novas estruturas de dados produzidas em queries. Deste modo acelera-se o tempo de resposta a essas queries no futuro, o que será essencial aquando da implementação do menu de input interativo. Por exemplo, as estruturas ordenadas resultantes das queries Q2,Q3 e Q7 podem ser recuperadas para poupar tempo e recursos em próximas chamadas destas queries, bastando apenas ler um número diferente de elementos para output. Relativamente à query 2, pretendemos modularizar a função `strResults` para devolver N elementos de eventuais diferentes estruturas de dados implementadas na Q2, Q3 e Q7. Também pretendemos modificar a função `sort_byRatings` para receber arrays de diferentes tipos, não apenas do tipo *ridesByDriver*.

Os condutores e as viagens são, neste momento, arrays estáticos, mas para a segunda fase os arrays "primários" serão alterados para *GPtrArrays*, para permitir ficheiros de qualquer tamanho.