RELATÓRIO DO PROJETO LI3

1ºFASE

João Henrique da Silva Gomes Peres Coelho - a100596

Duarte Soares Pinto Oliveira Araújo - a100750

José Filipe Ribeiro Rodrigues - a100692

ÍNDICE

Introdução

No âmbito da cadeira de Laboratórios de Informática III foi nos pedido que realizássemos um projeto utilizando a linguagem de programação C que visa a manipulação de 3 ficheiros, com tamanhos variados, e a utilização dos mesmos na obtenção de informações que consideramos importantes para responder às questões presentes no guião.

Ao longo da realização do projeto necessitámos de conhecimento sobre a manipulação de memória, encapsulamento, modularização e, por fim, estruturas de dados, todos estes conceitos permitiram a realização de um programa mais rápido e eficaz.

Estruturas

O primeiro desafio quando se inicia a realização deste projeto é que estruturas pretendemos utilizar para guardar os dados dos ficheiros que nos foram fornecidos e como as manipular a nosso favor.

Esta primeira questão demonstrou ser uma das mais problemáticas e difíceis de resolver, visto que nenhuma das estruturas que nos tinham sido apresentadas até o momento pareciam adequadas para resolver tal desafio.

Primeiramente pensámos em utilizar *arrays*, mas isto logo se viu ser pouco eficaz, já que necessitaríamos de um *array* massivo para conseguir guardar todos os dados necessários e, devido à necessidade de percorrer a estrutura por completo sempre que quiséssemos chegar aos dados presentes nas últimas posições da mesma, haveria um aumento exponencial do tempo de execução do programa.

De seguida surgiram na discussão as *listas ligadas,* mas, tal como a estrutura anteriormente referida, logo se verificou que estas também não seriam a melhor escolha devido aos mesmos motivos da estrutura anterior.

Por fim, discutimos a estrutura que mais se adequava das três que nos eram conhecidas, a *árvore ordenada.* Já que a mesma nos permitia "ignorar" grande parte dos valores durante a pesquisa, devido à sua organização em torno de determinado valor, esta pareceu ser a melhor opção, no entanto, depois de analisarmos o guião com atenção, verificámos que dependendo das *queries*

teríamos de ordenar a árvore em torno de valores diferentes, ora isto criou um problema fatal, teríamos de criar uma árvore em cada *query* o que causaria, mais uma vez, um elevado tempo de execução.

Depois de alguns dias a contemplar as nossas opções descobrimos as *hash tables, e*sta estrutura demonstrou ser a mais adequada para a tarefa que tínhamos em mãos devido ao elevado número de dados que precisamos de manipular. As *hash tables* funcionam de uma maneira peculiar, elas são constituídas por pares (key,data), ou seja, cada membro da estrutura possui os dados que se pretendem manipular associados a uma chave única que os distingue de todos os outros presentes na *hash table*, no nosso projeto estas keys são os ids nas rides e nos drivers e o username nos users.

Devido a essa estruturação o acesso ao conteúdo da estrutura, sabendo a chave, é possível em tempo linear, sendo assim muito mais rápido que qualquer outra estrutura que conhecêssemos até o momento. Contudo, implementar uma *hash table* do zero não é uma tarefa fácil, e perdemos um tempo considerável a tentar entender como tal poderia ser possível tentando, posteriormente, aplicar aquilo que aprendemos no nosso projeto. Foi nesta altura que descobrimos a existência de uma biblioteca que fazia grande parte do trabalho por nós, tornando a criação de uma *hash table* possível a partir da chamada de uma única função, tal biblioteca é denominada *glib*. Descobrimos assim a estrutura que iriamos utilizar para a base do nosso projeto.

De seguida, de maneira a guardar as informações nas *hashs tables*, decidimos criar uma estrutura para cada ficheiro recebido, nomeadamente, User, DRIVERS e RIDES, cada uma delas vai guardar os dados do ficheiro que lhe está associado. Para além disso, criámos também outra para colocar as três *hash tables* e assim facilitar o acesso à base de dados ao longo do programa.
Com as estruturas planeadas podemos avançar para o parse dos ficheiros para conseguir assim dar uso às mesmas.

Parse

Para podermos usufruir das estruturas referidas anteriormente precisamos de encontrar uma forma de separar a informação presente nos ficheiros que nos foram dados e passá-la para as estruturas, este processo de separar os valores de um ficheiro utilizando um determinado delimitador é conhecido como *parsing*.

Para realizar o *parse* dos ficheiros baseamo-nos bastante nos exemplos dados pelos professores nas aulas. Desta forma, depois de abrir os ficheiros com a função *fopen*, percorremos os mesmos com *getLine* e de seguida separamos os valores dos ficheiros pelo delimitador ";" que separa cada coluna dos ficheiros tipo *.csv*. À medida que é feita esta separação vamos adicionando essas informações ao respetivo campo das estruturas criadas. Além disso, os campos das mesmas que serão usados posteriormente para cálculos adicionais são inicializados como 0 ou NULL.

Por fim, visto que a estrutura está completa, inserimos a mesma na respetiva *hash table*. Em relação ao ficheiro que nos é passado como argumento e que contém os comandos a serem executados e respetivos argumentos, o parse é realizado de forma semelhante aos outros ficheiros. Neste caso, as linhas são divididas utilizando um espaço como delimitador e as palavras resultantes são colocadas num *array* de *strings* que vai ser passado à função responsável por tratar destes comandos.

Queries

De seguida, com as estruturas base do projeto e o parse feito, podemos, por fim, iniciar a realização das *queries.* Para a entrega da 1º fase do projeto precisamos de ter, no mínimo, 3 das 9 *queries* feitas e, depois de uma leitura atenta do guião, decidimos que iríamos concluir 3 *queries,* sendo elas as primeiras 3 que nos são apresentadas.

Query 1

A primeira *query* tem como objetivo calcular / apresentar informações de um determinado *driver* ou *user,* tais como avaliação média da pessoa em questão e total gasto ou auferido, sabendo o id ou username, respetivamente.

Numa fase inicial do projeto não eram feitos cálculos ao longo do parse e, por isso, fomos obrigados a percorrer novamente a *hash table* dos *rides* para assim conseguir obter as informações necessárias para responder ao problema proposto. Esta estratégia rapidamente se mostrou pouco eficiente tanto para esta *query* como para *queries* futuras. Logo, para facilitar os cálculos e aumentar a eficiência do programa, como referido brevemente no capítulo do parse, foram adicionados

às estruturas campos que nos indicam quantas viagens foram efetuadas, o total da avaliação dada em cada viagem e o total gasto ou auferido.

Assim sendo, só é necessário aceder à *hash table* associada à estrutura a que se pretende retirar os resultados da query e fazer os *prints*. De salientar que antes de realizar qualquer instrução referida anteriormente, é feita uma verificação do estado do *driver* ou *user* que pode ser ativo ou inativo, pois, no caso de corresponder à segunda opção, o processo anteriormente referido não é realizado.

Query 2

De seguida, a segunda *query,* ao contrário da primeira, exige que seja devolvida uma lista de drivers como resultado da mesma. Esta lista tem ainda de estar ordenada consoante o valor da avaliação média associado com cada um dos membros do conjunto.

Tal como foi referido anteriormente na *query* 1, a técnica usada, previamente, para responder às *queries* rapidamente demonstrou ser pouco eficaz e, depois da mesma ter sido alterada, a performance do programa melhorou significativamente, passando a realizar a *query* 2 num segundo, em vez dos 20 que se verificavam anteriormente.

A resposta a esta query começa com a verificação do estado do driver, ou seja, se o mesmo se encontra ativo ou não, caso se verifique a segunda opção então este driver já não poderá ser considerado como um membro da lista.

Depois desta verificação ter sido completada pode ser iniciado o enchimento da lista, no entanto, caso haja um empata no valor da avaliação média, é necessário que haja um desempate. Tal processo possui duas fases, a primeira corresponde à comparação da viagem mais recente que o *driver* realizou, aquele que tiver a viagem mais recente encontrar-se-á à frente na lista. Finalmente, a segunda é iniciada quando a viagem mais recente dos *drivers* é igual, caso isto aconteça então o id dos mesmos terá de ser verificado e aquele que tiver o maior id ficará à frente na lista.

Concluindo, com estes passos todos realizados, pode ser retornada a lista que irá responder à *query* 2.

Query 3

A terceira *query* é bastante semelhante à anterior, por isso, seguimos o mesmo raciocínio para a resolver. Os mesmos problemas referidos anteriormente também se aplicam a esta parte do projeto, no entanto, depois das alterações no parse, tudo se tornou mais simples.

Os elementos da *hash table* dos *users* são todos inseridos na estrutura *ARRAY_USERS*, que posteriormente é ordenada recorrendo ao algoritmo *Quick Sort*, visto que este método de ordenação se prova mais eficiente em *arrays* de tamanhos elevados. À medida que o *array* é ordenado pelos valores da distância total percorrida, temos em conta o processo de desempate, que implica comparar a data da última viagem dos *users*, e, no caso de novo empate, os seus usernames.

Com base no resultado dessas comparações podemos decidir quem deve ser apresentado primeiro. Quando chega a altura de dar print aos N *users* com maior distância percorrida (N últimos elementos do *array*), é feita uma verificação para que se saiba se a conta está ativa ou não. No caso de ser inativa, o *user* é ignorado.

Memory Leaks

Após a conclusão das *queries* podemos iniciar um dos passos mais importantes do projeto que é retificar o bom funcionamento do código, o que implica verificar a existência de *memory leaks* e eliminá-los caso existam.

Com efeito, a primeira tarefa a ser realizada foi criar funções que dessem *free* às estruturas criadas e mudar a função que inicializava as *hash tables* para assim poder realizar free a todas as estruturas acumuladas nas mesmas.

Concluindo este passo, passamos a procurar erros de memória mais facilmente ignorados no código, tais como não fazer free a *strings* que foram criadas com o uso da função *strdup*. Houve um caso que nos desafiou mais ao longo deste processo de resolução de *memory leaks*, a função *compareDates*. Passamos um tempo considerável a tentar descobrir o porquê deste erro e era crucial corrigi-lo visto que esta função era chamada diversas vezes. Depois de várias tentativas, decidi-mos colocar um *pointer* para o início das *strings* criadas na função, que serviriam para truncar as datas e assim obter os valores inteiros dos dias, meses e anos. Assim já nos foi possível fazer free das *strings* criadas, pois, no final de

separar várias vezes cada uma delas, as mesmas se tornavam um *pointer NULL*, pelo que o free não funcionava.

Encapsulamento e modularização

O encapsulamento foi nos introduzido no âmbito desta cadeira pelo que era um conceito novo e uma nova preocupação.

No início encontramos obstáculos na forma de implementar as funções pelo facto de não podermos aceder a estruturas que não fossem declaradas no próprio ficheiro, pelo que nos causou alguma confusão. Todavia, habituamo-nos rapidamente a este novo modo de programar e superamos as dificuldades previamente estabelecidas.

Devemos salientar as dificuldades que surgiram no que toca ao envio de dados como argumento para funções com o objetivo de respeitar o encapsulamento. Este obstáculo revelou-se mais presente quando pretendíamos enviar a estrutura da própria *hash table*, visto que precisávamos alterar valores dessa estrutura e não simplesmente aceder às informações.

Para além disso temos noção de que o encapsulamento não está totalmente bem implementado especialmente no que toca às estatísticas dos *users* e *drivers*, visto que estas informações deveriam estar em estruturas próprias e não nas estruturas dos dados fornecidos.

Todas as funções e estruturas foram divididas em módulos diferentes para facilitar o trabalho em grupo e melhorar a organização. Para além disso, esta separação ajuda na identificação de erros.

Conclusão

Concluindo, ao longo da resolução do projeto foram surgindo dificuldades, no entanto, fomos capazes de as superar, umas mais facilmente que outras. Esta primeira fase permitiu desenvolver os conhecimentos necessários para a realização de um programa desta dimensão e forneceu uma base sólida para a continuação deste projeto.