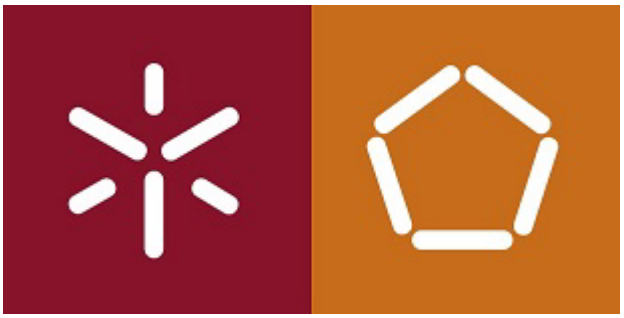


Universidade do Minho
Departamento de Informática

Sistema Estatístico de Viagens

Laboratórios De Informática III

Grupo 06

João Paulo Campelo Gomes, a100747
Rodrigo Casal Novo, a100534
Mariana Antunes Silva, a100702

Índice

Índice **2**

Introdução **3**

1.1 Descrição do Problema 3

1.2 Análise da Solução 3

2.1 Drivers **4**

2.2 Users **5**

2.3 Rides **7**

2.4 Utilities **8**

2.5.1 Query Handler 8

2.5.2 Query 1 9

2. Ordenação 9

3. Função principal 10

4. Output-file 10

2.5.4 Query 3 10

Estatísticas de Execução **11**

3.1 Tempo necessário para processar os três ficheiros csv 11

3.2 Tempo necessário para a execução do programa nos nossos dispositivos 11

3.3 Tempo necessário para processar e executar cada query (s) 11

Conclusão **12**

Introdução

O presente relatório tem como objetivo apresentar o projeto realizado no âmbito da unidade curricular de Laboratórios de Informática III, ao longo do primeiro semestre, do segundo ano, da Licenciatura em Engenharia Informática da Universidade do Minho. Este consiste na criação de um programa capaz de ler , armazenar e gerir toda a informação válida contida em ficheiros *csv*(s).

1.1 Descrição do Problema

Serão 3 os ficheiros *csv*(s) que deverão ser processados: *drivers.csv, users.csv e rides.csv*, cada linha nesses ficheiros deverá conter um registo com a informação de um condutor, utilizador e viagem, respetivamente.
Um dos objetivos deste trabalho é armazenar toda a informação contida nesses ficheiros, nas estruturas de dados mais adequadas para cada fim. Posteriormente a isso recebe-se um ficheiro *txt* onde estarão pedidos.

1.2 Análise da Solução

Numa primeira fase averiguamos quais as estruturas de dados que melhor se adequavam à resolução dos problemas que enfrentávamos. A nossa escolha de organização teve sempre em vista a rapidez das queries e a obtenção de dados de forma rápida, sendo assim para armazenar o ficheiro de entrada drivers.cvs optamos por criar uma HashTable, para o users.csv decidimos elaborar de raíz uma HashTable dinâmica já no caso do rides.cvs nós decidimos não o armazenar diretamente, mas utilizá-lo para fazer as estatísticas necessárias para as queries.

No que toca a modularidade decidimos dividir o drivers, o users, o rides, a utilities e as queries todas em módulos (*.c) separados para conseguirmos reutilizar o máximo de código possível e assim evitar ter de reescrever código já definido noutros módulos.

Com o objetivo de visar a questão do encapsulamento decidimos utilizar uma data de funções get para aceder aos valores das suas que se encontram no drivers.c e user.c de forma a não dar a conhecer as estruturas definidas nos módulos.

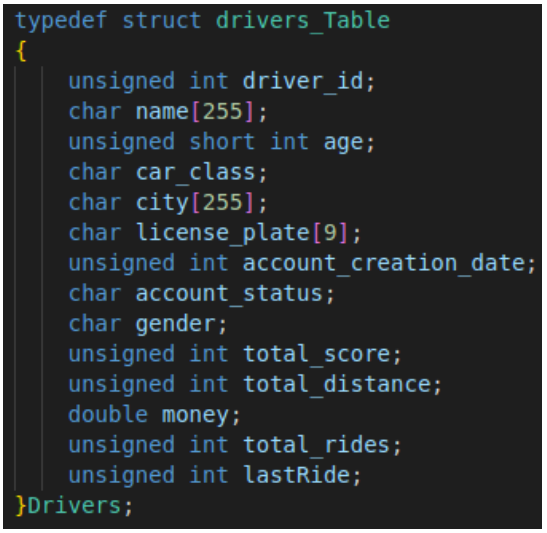
Módulos

A finalidade deste capítulo é expor detalhadamente, e com a devida fundamentação, a nossa abordagem para o armazenamento, gestão de dados e resolução das queries. Esta componente constitui toda a parte lógica do nosso programa.

2.1 Drivers

2.1.1 Struct drivers

O ficheiro drivers.csv tem 9 características diferentes de um condutor (id, nome, data de nascimento, car class, cidade, matrícula, gênero, data de criação de conta e estado da conta), por isso criamos variáveis na struct para cada uma delas, no entanto algumas são processadas antes de serem guardadas para poupar espaço. O id é convertido para unsigned int (driver_id); a data de nascimento é convertida para a idade para poder ser armazenada como unsigned short int (ag)e; o gênero (gender) é guardado como char porque ; a car class é guardado como char uma vez que as 3 classes possíveis começam com letras diferentes; semelhante à car class o estado da conta também é armazenado como um char, uma vez que os dois estados possíveis são diferenciáveis pelo primeiro caracter, e por fim a data de criação de conta é convertida para dias e armazenada como unsigned int. O nome e a cidade não podem ser processados, sendo por isso armazenados como strings de tamanho 255 valor que nos foi dito ser o máximo possível. Os cinco parâmetros finais da struct foram aqui colocados para se poder armazenar dados de estatística necessários para as queries.



```
typedef struct drivers_Table
{
    unsigned int driver_id;
    char name[255];
    unsigned short int age;
    char car_class;
    char city[255];
    char license_plate[9];
    unsigned int account_creation_date;
    char account_status;
    char gender;
    unsigned int total_score;
    unsigned int total_distance;
    double money;
    unsigned int total_rides;
    unsigned int lastRide;
}Drivers;
```

2.2.2 Método de organização

O ficheiro drivers.csv tem a particularidade de estar ordenado por id e destes ids serem números naturais com diferença de uma unidade entre cada id sucessivo. Graças a isto é possível percorrer em loop o ficheiro e gravar os dados de cada linha em elementos sucessivos de um array de drivers. Ao fazer isto desta forma, quando acabarmos de processar o ficheiro, o array obtido será uma hash table, cuja função de hash function é id-1. Devido à simplicidade desta função e de não haver ids repetidos, podemos gravar qualquer número de elementos neste array contando que façamos realloc quando o número de espaços ocupados estiver próximo da capacidade.

```c
Drivers *d_createTable(FILE *file){
    int current_table_size = 10010;
    int iterations = -1;
      char *line = (char*)malloc(850 * sizeof(char));
   char *word = (char*)malloc(255 * sizeof(char));
    size_t j = 850;
    int aux;
    aux = getline(&line, &j, file);
    if (aux == -1)printf("getline do drivers falhou\n");
    Drivers *table = (Drivers*)calloc(current_table_size, sizeof(Drivers));
    while(!feof(file)){
                iterations++;
     createNode(file, table, iterations, line, word);
      if(iterations >= current_table_size - 5){
                current_table_size *=10;
        memoryExpansion(&table, current_table_size);
      }
    }
    free(line);
    free(word);
  return table;
}
```

```c
void createNode(FILE *file, Drivers* drivers, int iteration, char*line, char *word){
    char **aux = Parser(file, line, word);
     if(!feof(file)){
     drivers[iteration].driver_id = atoi(aux[0]);
     strcpy(drivers[iteration].name, aux[1]);
     drivers[iteration].age =calculateAge(calculateDate(aux[2]));
     drivers[iteration].gender = aux[3][0];
     drivers[iteration].car_class = aux[4][0];
     strcpy(drivers[iteration].license_plate, aux[5]);
     strcpy(drivers[iteration].city, aux[6]);
     drivers[iteration].account_creation_date = calculateDate(aux[7]);
     drivers[iteration].account_status = aux[8][0];
     for (int i = 0; i < 9; i++){
       free(aux[i]);
     }
     }
         free(aux);
}
```

2.2.3 Set functions e get functions

Em 2.2.1 referimos que na struct drivers existem 5 parâmetros que são dedicados a guardar dados estatísticos, no entanto devido ao encapsulamento e modularidade não podemos aceder diretamente à tabela dos drivers, fora do ficheiro drivers.c, por isso criamos uma data de funções começados por get, que tal como o nome indica, servem para obter cópias de um determinado parâmetro da struct fora do ficheiro drivers.c, e também criamos uma função chamada d_addValue que serve para modificar o valor de todos os parâmetros de estatística da struct simultâneamente.

```c
void d_addValue(int id, int distance, int score, double tip, int lastride, Drivers* currentTable){
  currentTable[id-1].total_distance += distance;
  currentTable[id-1].total_score += score;
  currentTable[id-1].total_rides++;
  double arroz = calculate_price(distance, currentTable[id-1].car_class, tip);
  currentTable[id-1].money += arroz;
  currentTable[id-1].lastRide = get_most_recent_date( currentTable[id-1].lastRide, lastride);
}
```

2.2 Users

2.2.1 Struct users

A struct users é bastante semelhante à struct drivers, pelo que aqui vamos falar apenas dos parâmetros que não são comuns. O pay method é uma string de tamanho 3 porque com os primeiros 2 chars já se consegue identificar qual método é, e o username é uma string de tamanho 255, porque esse foi nos dito ser o máximo de caracteres possível e à semelhança do nome este não pode ser comprimido ou expresso de outra forma.

O total_rides do users é um unsigned short int em vez de unsigned int como o drivers, porque como há um número maior de users o risco de haver milhares de viagens para um

```c
typedef struct users_Table
{
    unsigned short int identifier;
    unsigned short int size;
    char username[255];
    char name[255];
    unsigned short int age;
    unsigned int account_creation_date;
    unsigned int lastRide;
    char pay_method[3];
    double money_wasted;
    unsigned short int total_rides;
    int total_distance;
    int total_avaliation;
    char gender;
    char account_status;

    struct users_Table* copycats;
}Users;
```

mesmo utilizador é muito menor. O identifier, size e copycats são variáveis que são necessárias para o bom funcionamento de uma hash table dinâmica.

2.2.2 Método de Organização

Ao contrario da drivers, o ficheiro users.csv não está organizado de nenhuma forma útil e não contém nenhum parâmetro que permita uma indexação direta num array, pelo que optamos por criar uma hash table dinâmica usando como função de hash a função djb2 criada por Dan Bernstein. Esta função é muito útil para criar valores inteiros únicos para uma dada string, o que nos dá uma forma eficiente de transformar os usernames (que é o único parâmetro que é garantido não ser repetido) em índices para o array. No entanto, a função não é perfeita e temos espaço limitado, pelo que precisamos de ter um failsafe para o caso de dois usernames darem a mesma chave e temos de garantir que a chave seja um índice dentro dos limites do array.

Para resolver o problema da colisão, na Users existe um parâmetro chamado copycats que é um apontador para um Users. Graças a este parâmetro, podemos criar um array paralelo à tabela para cada índice e colocar os valores com a mesma chave. É por causa disto que a struct contém os parâmetros identifier (armazena o primeiro índice vazio do copycats, o que reduz o número de comparações necessárias para introduzir um elemento na copycats) e size (guarda o tamanho máximo do copycats), resolvendo assim o problema das colisões.

Para resolver o problema de ter chaves out of bonds, na função que usamos para calcular as chaves, invés de retornar o valor que a função djb2 calculou, retornamos o resto da divisão inteira desse valor por 271127 (este é um número primo que é mais de 2 vezes maior que o número de users, o que garante a funcionalidade da tabela como hash table, e que ao criar o copycats com tamanho 3 permite armazenar até 1 milhão de users sem necessitar teoricamente de fazer realloc no copycats), valor este que corresponde ao tamanho do users, garantindo assim que todas as chaves são um valor entre 0 e 271126.(A função que calcula as chaves está definida no ficheiro utilities.c com o nome calculateIdentifier).

(A certo ponto do nosso trabalho trocamos temporariamente para uma hash table estática, que usava a mesma função de hash, era ligeiramente mais rápida e

```
void *u_addNode(Users *table, FILE *file,  char *line, char *word){
    char **aux = Parser(file, line, word);
    if(!feof(file)){
  int key = calculateIdentifier(aux[0]);
  if (table[key].username[0] == '\0'){
    strcpy(table[key].username, aux[0]);
    strcpy(table[key].name, aux[1]);
    table[key].gender = aux[2][0];
    table[key].age = calculateAge(calculateDate(aux[3]));
    table[key].account_creation_date = calculateDate(aux[4]);
    table[key].pay_method[0] = aux[5][0];  table[key].pay_method[1] = aux[5][1];
    table[key].account_status = aux[6][0];
    table[key].identifier = 0;
  }else{
    if(table[key].size == 0){
    table[key].size = 2;
    table[key].copycats = (Users*)calloc(table[key].size, sizeof(Users));
    }
    else if(table[key].identifier == table[key].size) {
      table[key].size = table[key].size << 1;
      int aux =  table[key].size;
      u_memoryExpansion(&table[key].copycats, table[key].size);
                for(int i = aux/2; i < aux; i++){
      table[key].copycats[i].username[0] = '\0';
    }
    }
    strcpy( table[key].copycats[table[key].identifier].username, aux[0]);
    strcpy( table[key].copycats[table[key].identifier].name, aux[1]);
    table[key].copycats[table[key].identifier].gender = aux[2][0];
    table[key].copycats[table[key].identifier].age = calculateAge(calculateDate(aux[3]));
    table[key].copycats[table[key].identifier].account_creation_date = calculateDate(aux[4]);
    table[key].copycats[table[key].identifier].pay_method[0] = aux[5][0];
    table[key].copycats[table[key].identifier].pay_method[1] = aux[5][1];
    table[key].copycats[table[key].identifier].pay_method[2] = '\0';
    table[key].copycats[table[key].identifier].account_status = aux[6][0];
    table[key].identifier++;
  }
    for (int i = 0; i < 7; i++){
    free(aux[i]);
    }
  }
    free(aux);
}
```

substancialmente mais eficiente em termos de memória, diferença de 50 MB, mas decidimos trocar um pouco de eficiência para ter uma aplicação mais flexível).

2.2.3 Set Functions e Get Functions

Semelhante à drivers, users possui parâmetros dedicados ao armazenamento de dados de estatística, pelo que tal como na drivers recorremos ao uso de get functions para obter cópias dos parâmetros de um users fora do ficheiro users.c e usamos uma função chamada u_addValue para modificar os parâmetros estatístico fora deste ficheiro. No entanto, apesar de a ideia ser a mesma a forma de ir buscar os valores é fundamentalmente diferente, uma vez que aqui a função de hash não é trivial e pode haver colisões, o que resulta em que dentro de cada função get exista uma chamada da função getUser (que tal como o nome indica retorna o User que tem o username pretendido), pelo que os acessos são ligeiramente mais lentos do que na drivers.

```c
void u_addvalue(Users* table, char*username, double cash, int avaliation, int distance, unsigned int lastride){
    int key = calculateIdentifier(username);
  if (strcmp(table[key].username, username) == 0){
    table[key].money_wasted += cash;
    table[key].total_avaliation += avaliation;
    table[key].total_distance += distance;
    table[key].total_rides++;
      if (compareDates(table[key].lastRide, lastride) == 1)
  table[key].lastRide = lastride;
  }else{
    int copycatkey = findCopycats(table[key].copycats, username);
      table[key].copycats[copycatkey].money_wasted += cash;
     table[key].copycats[copycatkey].total_avaliation += avaliation;
     table[key].copycats[copycatkey].total_rides++;
          table[key].copycats[copycatkey].total_distance += distance;
                if (compareDates(table[key].copycats[copycatkey].lastRide, lastride) == 1)
  table[key].copycats[copycatkey].lastRide = lastride;
  }
}
```

```c
Users getUser(Users *table, char* username){
  int key = calculateIdentifier(username);
  if (strcmp(table[key].username, username) == 0){
    return table[key];
  }else{
    int copycatkey = findCopycats(table[key].copycats, username);
    return table[key].copycats[copycatkey];
  }
}

int findCopycats(Users *copycats, char* username){
  int i = 0;
  for(; copycats[i].username[0] != '\0'; i++){
    if(strcmp(copycats[i].username, username) == 0) break;
  }
  return i;
}
```

2.3 Rides

2.3.1 Cálculos estatísticos

Rides.csv, ao contrário dos outros ficheiros, não tem de ser armazenado, contando que usemos os seus dados imediatamente para processar todos os dados estatísticos necessários para realizar as queries posteriormente. Por isso, o ficheiro rides.c é o mais curto dos 3 ficheiros principais, tendo apenas duas funções updateTables e addValues, a updateTables é uma função muito simples que apenas chama em loop até acabar de percorrer o ficheiro a função addValues que usa a função parser para decompor as linhas e depois chama as funções d_addValue e u_addValue com os parâmetros adequados, para alterar os valores estatísticos.

```c
void addValues(FILE *file, Drivers *drivers, int iterations, Users *users, char *line, char *word){
  char **aux = Parser(file, line, word);
  if(!feof(file)){
    int distance = atoi(aux[5]);
    double tip = strtod(aux[8], NULL);
    unsigned int lastRide = calculateDate(aux[1]);
    d_addValue(atoi(aux[2]),distance, aux[7][0] - '0', tip, lastRide, drivers);
    u_addvalue(users, aux[3], calculate_price(distance, getD_car_class(drivers, atoi(aux[2])-1), tip), aux[6][0] - '0',distance, lastRide);
    for (int i = 0; i < 10; i++) free(aux[i]);
  }
  free(aux);
}

void updateTables(FILE *file, Drivers *drivers, Users * users){
  int tableSize = 1000001;
  int iterations = 0;
  int row = 0;
  char *line = (char*)malloc(850 * sizeof(char));
  char *word = (char*)malloc(255 * sizeof(char));
  size_t j = 850;
  int aux;
  aux = getline(&line, &j, file);
  if (aux == -1)printf("getline do rides falhou\n");
  while(!feof(file))
    addValues(file, drivers, iterations, users, line, word);
  free(line);
  free(word);
}
```

2.4 Utilities

O ficheiro utilities.c, tal como o nome indica contém funções que são usadas por muitas outras funções de outros ficheiros, mas as mais notáveis são as funções:
Parser -> esta função recebe um ficheiro e duas strings (estas são usadas para armazenamento de valores intermédios e são recebidas como argumento em vez criadas internamente para reduzir o número de frees e mallocs feitos pela aplicação reduzindo assim o tempo de execução), para ler uma linha do ficheiro dado e separá-la em várias strings com strtok, usando como separador o " ; ". Esta função é chamada pelas 3 funções de processamento de ficheiros.

CalculateDate-> esta função recebe uma string em formato data e retorna o seu valor em dias desde o ano 0.
CalculateAge -> esta função recebe um unsigned int e retorna a idade em anos daquela data de nascimento em dias.
CompareDates -> esta função compara duas datas e retorna -1 se forem iguais, 0 se a primeira for maior e 1 caso contrário.
CalculateIdentifier -> esta é a função responsável por calcular a chave para o users, o motivo de estar aqui e não no users é que esta função também poderá ser usada para a tabela cidades posteriormente, uma vez que ela é genérica.
CalculatePrice -> esta é a função usada para calcular o valor de uma viagem dada car class, distância e gorjeta. Ela está neste ficheiro, porque é usada pelo users, pela drivers e poderá ser usada no futuro para as cidades.

2.5 Queries

As queries, nesta primeira fase, são passadas ao programa, através do fullpath para um ficheiro que as contém, neste ficheiro, cada linha é uma query, linha esta que é separada em 2 partes pelo QueryHandler.

2.5.1 Query Handler

Este módulo está dividido em duas funções principais, a função q_createquerylist que visa ler o ficheiro das queries e inserir as instruções numa estrutura de dados chamada querys.
Esta estrutura é constituída pelo número da linha em que a instrução se encontrava no ficheiro (id), pelo tipo da query (nquery), e por uma string com os argumentos necessários para execução da mesma.

```
typedef struct querys {
    int id;
    char nquery;
    char arguments[300];
} Querys;
```

Optamos por usar um único caractere no número da query dado que este está compreendido entre um e nove, e, nos argumentos, usamos uma string de tamanho máximo 280 pois reparámos que a query máxima, é de uma cidade e duas datas (query 6).

A outra função deste módulo, execute_query, começa por verificar o valor máximo para o top da query2 e da query3, decidimos que fazer isto seria a melhor estratégia, uma vez que tendo o pedido máximo ordenado, também teremos ordenado para pedidos menores. De seguida são executadas todas as queries de forma quase instantânea.

2.5.2 Query 1

Aqui adotamos uma estratégia simplificada, começando por determinar se o ID é um condutor(driver) ou um utilizador(user) através de um loop que percorre a string recebida como argumento, verificamos se um elemento da string é um caractere, se o for aux toma o valor de -1 e se não for o aux não se altera, assim sendo, se o aux for igual a -1 é um utilizador e se for igual a 0 estamos perante um condutor, de seguida é verificado se o ID contém uma conta ativa, se assim for, o conteúdo que nos é pedido é copiado e guardado, através de funções get, em variáveis, e logo de seguida, escrevemos num ficheiro criado. Caso contrário, apenas é criado um ficheiro vazio.

2.5.3 Query 2

A estratégia pensada para esta query teve como objetivo conseguir ordenar de forma simples e rápida, para tal optamos por um lista ligada pois é muito fácil manipular a sequência de structs que compõem a lista ligada e ordená-la corretamente.

1. Criação do newNode

Na função q_createNode introduzimos no newNode a variável unsigned int query2_id (guarda o id do condutor) que é necessário para conseguirmos, posteriormente, obter o nome do condutor para escrever no output-file, a double query2_mscore (armazena a avaliação média do condutor) e a unsigned int query2_lastRide (conserva a viagem mais recente do condutor) será utilizada caso a avaliação média dos condutores empate. É importante referir que todos os valores destas variáveis vão ser obtidos por funções get definidas na drivers.c respeitando portanto o encapsulamento.

```c
/*Estrutura usada para a query2*/
typedef struct query2_Table
{
    unsigned int query2_id;
    unsigned int query2_lastRide;
    double query2_mscore;
    struct query2_Table *next;
}Query2;
```

2. Ordenação

Ordenamos a lista utilizando a função recursiva Table2 que recebe como armento a lista, um n que representa o número de sequências de structs que a lista têm de ter, um i que começa por ser igual a zero que vai nos ajudar a determinar em que posição estamos na lista, um ant, um topo e um worts_score, esta função que é composta por uma data de if(s), no primeiro if começamos por identificar o caso de paragem da recursividade retornando o topo da lista caso o n seja igual a 0, no

segundo temos código que indica se a lista é NULL ou se a última viagem da lista for igual a zero, caso se verifique introduzimos o newNode na lista utilizamos a variável i, já no terceiro if verificamos se a avaliação média do newNode é maior que a da lista se sim utilizamos a variável i referida acima para inserir o newNode de forma correta, passando ao último if vemos o caso em que a avaliação média entre o newNode e a lista são iguais e caso isto aconteça passamos a verificar se a última viagem do newNode é mais recente que a da lista se for inserimos o newNode na lista conforme a variável i nos indicar, neste processo se o n for igual a 1 que indica que estamos na última struct da lista indicamos qual é o valor da variável worst_score, útil na função principal, na recursividade passamos todos os argumentos iguais exceto o i que aumenta em 1 para termos noção em que struct estamos e o n que subtraimos-lhe 1.

3. Função principal

Na função TopN_classificacao introduzimos um loop que vai percorrer todos os condutores, para os inserir na lista dentro deste loop fizemos algumas otimizações introduzindo a variável atividade, elementos e worst_score, na atividade armazenamos o estado da conta do condutor para assim podemos selecionar apenas os condutores que têm contas ativas, já na elementos guardamos quantas sequências de structs tem a lista ligada para que caso a lista ainda não tem N condutores criamos um newNode que vai ser introduzido na função Table2 que retornará a lista ordenada, no entanto se a lista já tem N condutores inserimos a variável worst_score que conserva a avaliação média do último condutor da lista, utilizando-a para podermos comparar a avaliação média de um condutor que ainda não foi inserido com a worst_score, caso a avaliação média do condutor for maior ou igual que a worst_score quer dizer que esta é maior que o último condutor da lista já ordenada então vamos inserir este newNode na função Table2 já no caso contrário concluímos que o condutor não tem maior avaliação média dos restantes que já se encontram na lista sendo assim é descartado. Com estas variáveis conseguimos reduzir o número de vezes que temos de ordenar a lista e reduzir as interações aumentando assim o tempo de execução.

4. Output-file

Para escrever a lista resultante da TopN_classificacao no output-file optamos por usar um loop para percorrer a lista e escrever as variáveis pedidas, decidimos determinar qual o nome do condutor apenas quando o fôssemos escrevê-lo no ficheiro, a partir do id, conseguindo assim diminuir a quantidade de malloc(s) realizados.

2.5.4 Query 3

Neste módulo seguimos um procedimento bastante parecido com o explicado na query 2, difere, no entanto, na forma como percorre a *hashtable* dos utilizadores, uma vez que esta está organizada de forma bastante diferente. Nesta função percorre todos os valores entre zero e o *RPrime* (uma vez que é o index máximo possível), e para cada um, averigua se deve inserir esse utilizador na tabela, vendo primeiramente se se encontra ativo e, caso esteja, se a sua distância percorrida é maior que a mais pequena do top daquele momento (*worst_distance*). Depois deste nodo ser verificado, percorre todos os usernames que estejam no *copycats* repetindo o procedimento de verificação. Como explicado

acima, o processo para percorrer os *copycats* é facilitado, pois, no primeiro utilizador deste índice, são guardados o número de elementos deste conjunto.

A estrutura de dados ao lado é onde é guardada a informação necessária para a ordenação, feita pela função recursiva *ordTable* (similar à função Table2) e para a eventual escrita do resultado, com a função *printFile*.

Para esta query decidimos criar funções get diferentes daquelas que foram criadas na query 1, apesar de ir buscar algumas informações iguais. Para a query 1, apenas temos conhecimento do username, no entanto, nesta query, sabemos a posição exata da memória onde queremos copiar a informação. Seguindo esta estratégia, diminuímos bastante o número de instruções e também o tempo de execução, respeitando o encapsulamento.

```
typedef struct query3_Table
{
    char query3_username[255];
    unsigned int query3_lastRide;
    int query3_distance;
    unsigned int U_ids;
    struct query3_Table *next;
}Query3;
```

Estatísticas de Execução

Para as estatísticas a seguir apresentadas, foram executados 10 testes no computador de cada elemento do grupo. Como estes tempos dependem do computador no qual são executados, apresentamos valores de três processadores diferentes.

3.1 Tempo necessário para processar os três ficheiros *csv*

	Mariana	João Paulo	Rodrigo
Tempo compilação médio (s)	1,18	0,78	1,10

3.2 Tempo necessário para a execução do programa nos nossos dispositivos

	Mariana	João Paulo	Rodrigo
Texec médio teste 1 (s)	1,20	0,78	1,18
Texec médio teste 2 (s)	1,20	0,79	1,18

3.3 Tempo necessário para processar e executar cada query (s)

Rodrigo	Query 2	Query 3
top 10	0,017	0,032
top 900	0,020	0,073
top 10 (10 vezes)	0,022	0,036

Conclusão

Acreditamos que a nossa primeira fase do projeto correu bastante bem, uma vez que a nossa aplicação no PC do João Paulo está a executar dentro dos valores de referência, que os professores nos disseram nas aulas (tempo de execução abaixo de um segundo e as queries na casa dos milissegundos) e não temos memory leaks como se pode haver abaixo pelo relatório do memcheck do valgrind (a memória que aparece como still reachable é por causa do fprintf).
No entanto, a nossa aplicação devido às suas otimizações para realizar um grande número de queries no menor número de travessias pelas tabelas possíveis, como por exemplo percorrer primeiro o ficheiro inteiro dos inputs para determinar qual o maior top pedido de forma a calcular os tops apenas uma vez, acaba por ser menos eficiente a calcular pedidos únicos de uma query. Para além disto nós usamos uma hash table para o users, o que significa que para ela ser eficiente a tabela tem de ter pelo menos o dobro do tamanho do users, o que leva a que num caso em que não haja colisões metade do espaço ocupado na ram não seja usado. Para além disso como optamos por usar uma tabela dinâmica em vez de uma estática, o tempo de execução é ligeiramente maior (no PC do João Paulo não se nota diferença, mas nos outros dois há uma diferença de 1ms) e ocupa mais espaço uma vez que as colisões são resolvidas alocando mais espaço. Por causa disto, esta versão do trabalho ocupa aproximadamente mais 50MB que a versão estática, Optamos, no entanto, por esta versão por uma questão de flexibilidade.
Também acreditamos que respeitamos bem o encapsulamento e modularização, uma vez que todos os nossos ficheiros trata apenas de um tipo de dados ou de uma query e que para aceder a qualquer elemento das nossas tabelas fora do ficheiro de origem é necessário recorrer às funções get onde só se obtêm cópias dos valores e que para os alterar é necessário recorrer às funções addValue definidas nos seus ficheiros de origem.


```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 5,664 bytes in 12 blocks
        suppressed: 0 bytes in 0 blocks
```