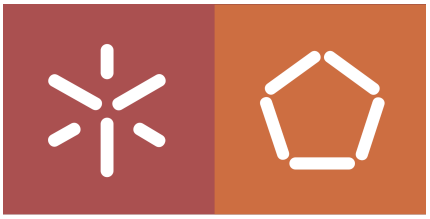


ESCOLA DE ENGENHARIA DA UNIVERSIDADE DO MINHO
LICENCIATURA EM ENGENHARIA INFORMÁTICA

Laboratórios de Informática III

Fase 2

A84675, António L. Mendes
Lu1sm3ndes

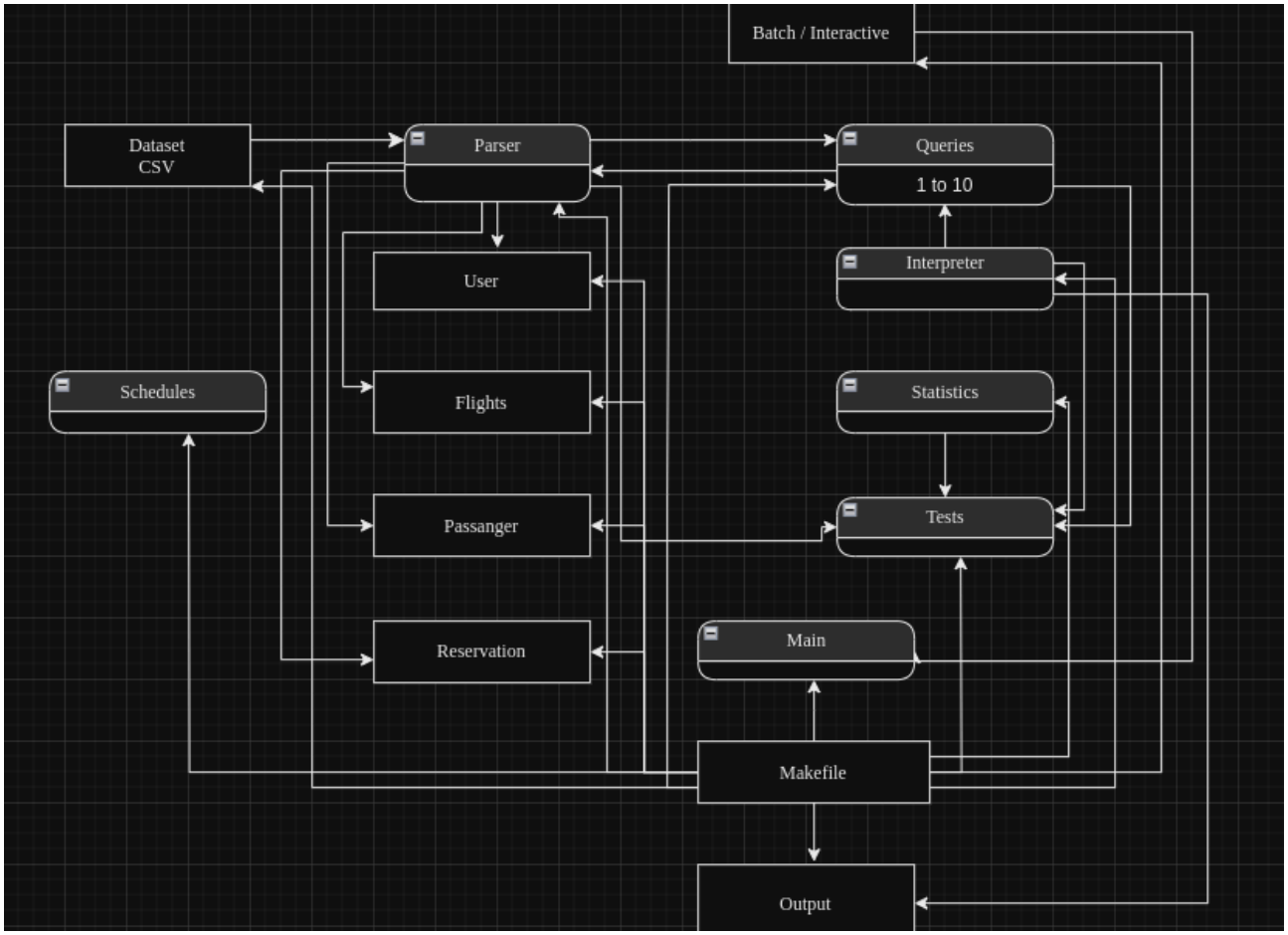
A104611, João M. da Cunha
JoaoCunha50

A104612, João P. de Sá
joaosa24

ÍNDICE

1. Modularidade e Encapsulamento

No caso dos users, flights, passengers e reservations mantivemos uma estrutura primária idêntica à da primeira fase. Usamos arrays dinâmicos para armazenar a memória dos outputs. Inicializamos o array com valor nulo, depois calculamos o valor de memória necessária para o output que vai ser devolvido, otimizando a quantidade de memória que cada uma das querry está a requerer no momento que foi devolvido o output desta mesma querry. Isto permite uma maior dinâmica tanto na query como na alocação de memória para o output. No caso do parser, a memória é alocada para o primeiro elemento, e nas próximas iterações, ela é realocada dinamicamente usando *realloc*. Isso permite que a alocação de memória seja ajustada dinamicamente para evitar atribuir muito ou pouco espaço.



2. Queries implementadas na fase 2

2.1 Query 2

Esta query tem como objetivo listar os voos e/ou as reservas de um utilizador. Estas, por sua vez, devem estar ordenadas por data (da mais recente para a mais antiga). Caso seja especificado o tipo, deve retornar apenas os outputs correspondentes (i.e, se o segundo argumento for “reservations”, apenas deverão ser listadas as reservas do utilizador)

Para a implementação da mesma, começamos por “dividir” o input recebido em campos, tais como o User ID e se for o caso , o tipo de output esperado (reservations e/ou flights) . No caso do tipo de output esperado ser “flights”, fazemos travessias pelo array de passageiros, e comparamos os respectivos ID’s de utilizador com o input , utilizando os getters.

Guardamos os ID’s de voo dos passageiros com o ID igual ao recebido numa matriz (FlightID), e fazemos uma travessia pela mesma comparando todos os ID’s de voo com os ID’s presentes nessa mesma matriz, em caso de igualdade copiamos a informação relevante para a struct Info user, que vai servir para dar print aos outputs depois de ordenada, por ordem decrescente das datas através da função *“comparar_infos_decrescente”*. Em caso de duas datas serem iguais , o critério de desempate é o respetivo ID.

Em caso do tipo de output esperado ser “reservations” , apenas percorremos o array de reservas , comparando o ID das mesmas com o ID recebido no input . Mais uma vez, quando essa condição se provar verdadeira , copiamos a informação para a struct nova e repetimos o processo.

Não sendo especificado o tipo de output esperado , realizamos os dois processos .

2.2 Query 4

Esta query tem como objetivo listar as reservas de um hotel, ordenadas da data mais recente para a mais antiga (utilizando como parâmetro a data de início da reserva). Caso a data de início de duas reservas coincida, o critério de desempate passa pelo identificador da reserva (de forma crescente).

Para a implementação da mesma, decidimos percorrer o Array de Estruturas *Reservation,* e caso o ID do mesmo fosse igual ao fornecido à query, esta reserva seria "copiada" para outro array (também do tipo *Reservation*). Para a comparação de identificadores, utilizamos o "*strcmp*". Já com todas as devidas reservas copiadas, ordenamos, através do *Quicksort*, o array, com recurso à função de ordenação "*comparaReservas*" que tem como objetivo ordenar duas reservas, através da data, tendo também em conta os casos onde as datas iniciais coincidem.

Por fim, para imprimir os resultados fazemos uma travessia pelo array ordenado, de forma a listar as reservas.

2.3 Query 5

Esta query tem como objetivo listar os voos com origem num dado aeroporto, com restrição de datas, ordenados da data mais recente para a mais antiga (utilizando como parâmetro a data de partida estimada). Tal como na query 4, caso a data coincida, o critério de desempate passa pelo identificador do voo.

Para implementar esta query, a estratégia utilizada é bastante semelhante à da query 4. Começamos por percorrer o array de Voos, e fazemos a verificação através da disjunção de duas condições: A origem do voo, a ser analisado na interação, deve coincidir com a origem fornecida à query e o mesmo deve estar entre as datas fornecidas. Caso o voo cumpra ambos os requisitos será copiado para outro array.

Para ordenar, recorremos ao *Quicksort*, tal como nas outras queries, utilizando a função "*comparaVoos*" como critério de ordenação.

Por fim, para imprimir os resultados fazemos uma travessia pelo array ordenado, de forma a listar os voos.

2.4 Query 6

Esta query requer que apresentemos o aeroporto com o maior número de passageiros num determinado ano, tendo em conta os voos que de lá partem , e que os voos que lá aterram respetivamente. Em primeiro lugar,
usamos a função "**ajustaSigla2"* como forma de converter uma string
para maiúsculas . A função "*comparaPassageiros*" é capaz de ordenar um array do tipo Airport com base no número de passageiros, e em caso de empate, utilizando a comparação do nome do próprio aeroporto. Copiamos a informação relevante para uma struct Airports, a qual ordenamos e utilizamos para dar print aos outputs esperados .

2.5 Query 7

Esta query, tal como a anterior, requer informações indiretas sobre os voos, pois é pedido para apresentar o top N de aeroportos com a maior mediana de atrasos. A função *"inserirAeroporto"* procura um aeroporto com um ID específico dentro de um array de aeroportos. Com a função "*comparaMediana*" comparamos as medianas de dois aeroportos e devolvemos em ordem decrescente. Se as medianas forem iguais, use o nome do aeroporto como critério de desempate.

2.6 Query 8

Esta query pede-nos o total gasto numa determinada reserva que esteja dentro de um certo limite de tempo, tendo em conta a data que nos é dada no input (begin_date) e o respetivo ID do hotel. Através da função "*get_hotel_price_per_night"*, utilizamos a get_reservation_price_per_night e o número de noites para chegar ao gasto total dessa mesma reserva (através da fórmula dada no enunciado, que está implementada no catálogo statistics.c).
Após todos os cálculos , realizamos o print desse mesmo gasto total (double) no ficheiro dos outputs , ou no terminal dependendo do modo .

2.7 Query 9

Esta query lista todos os utilizadores cujo nome começa com o prefixo passado por argumento, ordenados por nome, de forma crescente. Caso dois utilizadores tenham o mesmo nome, o critério de desempate passa pelo seu identificador (ordenado de forma crescente). Esta função não tem em conta os usuários que têm o “Account Status” definido como “inativo”.

Para a implementação da mesma, utilizamos uma estratégia idêntica às outras queries.

Realizamos uma disjunção de duas condições: O nome do usuário deve conter o prefixo e o mesmo deve estar ativo.

Caso as duas condições sejam cumpridas, “copiamos” o usuário para outro array. O processo repete-se para o array inteiro.

Após a análise do array, ordenamos o array de usuários filtrados, recorrendo à função “*comparaUsuários*”.

Por fim, para imprimir os resultados fazemos uma travessia pelo array ordenado, de forma a listar os usuários.

2.8 Query 10

Esta query está incompleta, logo não foi incluída no nosso programa devido a alguns problemas encontrados na sua realização .Ela apresenta várias métricas sobre o programa, como números de novos utilizadores registados, número de voos, número de passageiros, número de passageiros únicos e número de reservas. Dependendo do input, os outputs esperados podem conter informaçoẽs sobre dias de um determinado mês, sobre os meses de um determinado ano , ou sobre todos os anos relevantes neste mesmo programa (por exemplo , caso o ano esteja especificado no input e o mês não , damos print às informações dos meses correspondentes a esse ano).

Nota

Em todas as query em que se usa uma função comparativa, quer seja para ordenar alguma informação, quer seja para comparar uma mediana de atrasos de aeroportos ou até para comparar o número de passageiros de um determinado aeroporto podem ser depois usadas como argumento para a função "*qsort*" para ordenar um determinado array. Tanto as query implementadas na fase anterior como as implementadas nesta fase, tem dois argumentos comuns a todas elas, int counter e int mode que nos ajudam a identificar o número de consultas realizadas ao imprimir o output, o que é útil para quando estamos a visualizar o mesmo e a fazer a distinção entre o modo batch e o modo interativo, respectivamente.

3. Dificuldades sentidas nesta fase

Uma das maiores dificuldades que sentimos foi como libertar a memória utilizada , isto é, tentar diminuir ao máximo as memory leaks. Não era viável simplesmente usar a função “*free*” indiscriminadamente, pois poderia resultar na perda de informações cruciais para o desempenho de outras queries. Essa preocupação foi crucial em situações em que tínhamos múltiplos processos em execução e, assim, precisávamos manter uma alocação de memória eficiente.

Ao lidar com entradas inválidas o problema era semelhante, já que a alocação excessiva de memória poderia levar a atingir rapidamente o limite disponível. Portanto, precisávamos equilibrar a alocação de memória com a garantia de não causar perdas desnecessárias.

No final das contas , apesar de se ter provado desafiante , conseguimos remover todos os memory leaks do nosso programa.

4. Modo interativo

Neste modo, o programa é iniciado sem argumentos, e a detecção de input é feita exclusivamente pelo teclado. No início da execução, o utilizador final é solicitado a inserir o caminho do dataset desejado. Utilizando s*canf("%s", path)*, o programa aguarda a entrada do usuário necessária para inicializar este modo.

Usando a função "*fopen*", as informações relativas aos usuários, voos, passageiros e reservas são processadas. Em seguida, verifica-se se todos os arquivos foram abertos corretamente. Em caso positivo, informa-se ao usuário e solicita-se uma verificação do caminho fornecido. Se todos os arquivos foram abertos corretamente, o parser dos arquivos dos usuários, voos, passageiros e reservas é executado. Posteriormente, todos esses arquivos são fechados, e informações adicionais são fornecidas ao usuário.

A seguir, instruções básicas são apresentadas sobre como inserir os inputs, utilizando o formato *<query> <arg1> <arg2> <arg3>*. Um exemplo é fornecido: *1 anTonioFaARIA10*. Além disso, são fornecidas orientações sobre como sair do modo interativo.

Finalmente, o programa executa o tratamento do input e libera a memória alocada para as estruturas de dados.

Em resumo, criamos um ambiente interativo, onde o utilizador final fornece o caminho contendo arquivos CSV, os arquivos são abertos, os dados são parseados e o programa entra em um loop para receber consultas até que o usuário decida sair.

5. Testes funcionais e de desempenho

O módulo "src/tests.c" é responsável pela implementação do programa de testes da aplicação. A função *"compare"* tem a finalidade de comparar o resultado do output recebido com o esperado. Os caracteres são comparados um a um, e a função retorna verdadeiro se os caracteres são iguais e falso se houver diferença. A função *"check"* é encarregada de avaliar o resultado da função "*compare"* para cada teste, identificando o teste em execução.

Para cada query implementada, existe uma função "*qx_test"* que recebe pointers para as structs de dados primárias. Essa função verifica o segundo caractere da string, embora o propósito específico não tenha sido detalhado.

O programa de testes inicia sua execução como um modo interativo, começando com o comando *system("clear")* para limpar o terminal. Posteriormente, solicita o caminho dos arquivos CSV. São criados dois diretórios, um para os erros de saída e outro para os resultados dos testes executados.

Durante a execução, o parser dos dados é acionado, e os arquivos de leitura são fechados quando o processo é concluído. É alocada memória para os arquivos de input, e no final, toda a memória previamente alocada é liberada.

Em todas as queries implementadas na fase anterior como as implementadas nesta fase existem 3 linhas de código comuns. Com *clock_t start, end* conseguimos medir o tempo de execução do programa, armazenado em *double cpu_time_used.* Por fim, registamos o tempo atual do relógio marcando o início da contagem do tempo através de *start = clock()*.

Esse pedaço de código é usado para medir o tempo de CPU utilizado por uma seção específica do código entre o start e o end. A diferença entre esses dois tempos, quando dividida pelo número de ciclos de relógio por segundo, fornecerá o tempo de CPU real utilizado pelo programa. O tempo de CPU é útil para avaliar o desempenho de um determinado trecho de código.