Projeto Laboratórios de Informática III

Fase II

Projeto desenvolvido por:
Rafael Lopes Seara (A104094), Sara Catarina Loureiro da Silva (A104608) e Zita Magalhães Duarte (A104268)

Grupo 80

Licenciatura em Engenharia Informática

Universidade do Minho
Escola de Engenharia

Departamento de Informática
Universidade do Minho

Índice

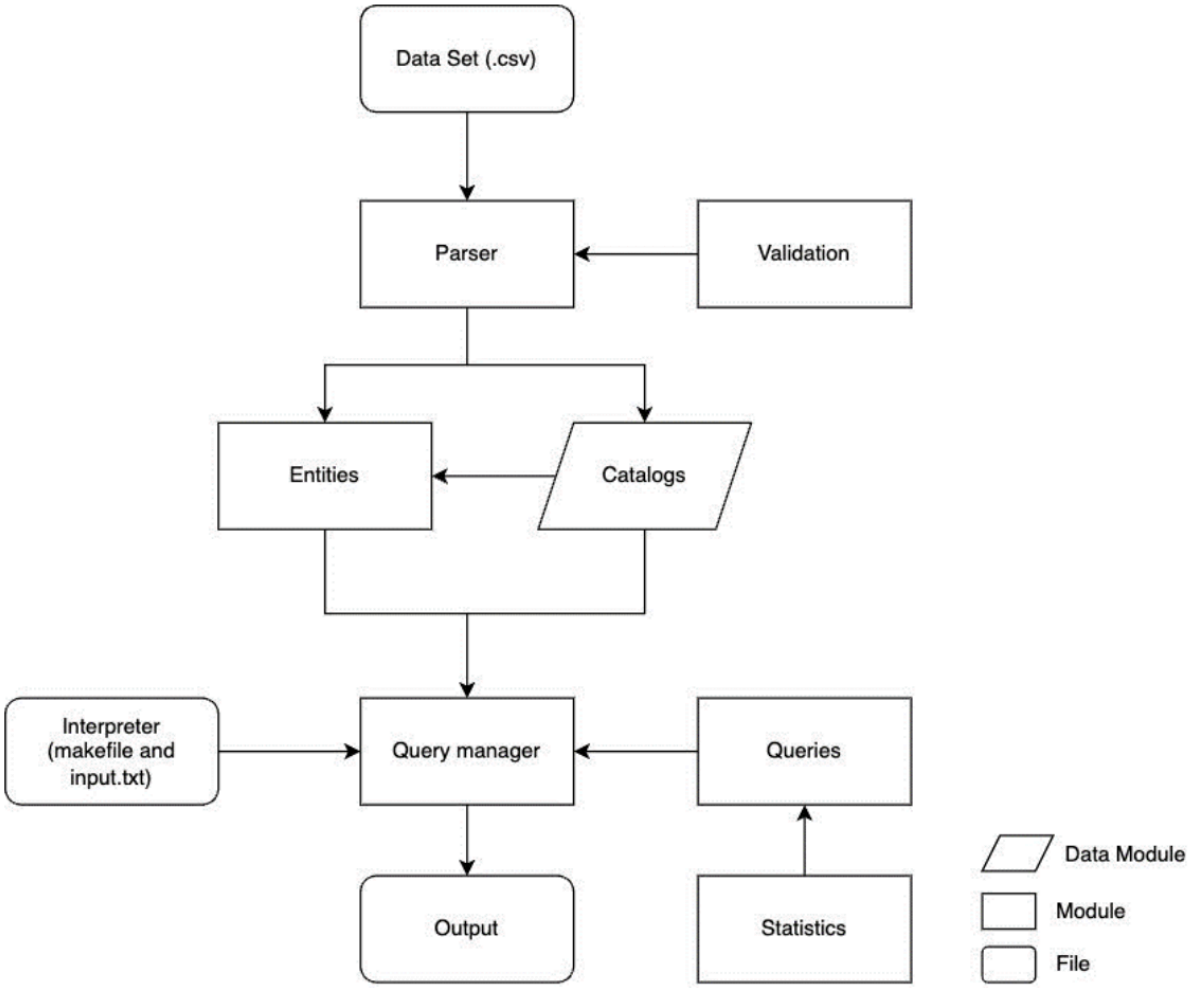
Capítulo 1
Introdução

Foi-nos proposto, na unidade curricular de Laboratórios de Informática III, do ano 2023/2024, o desenvolvimento deste projeto. Nele temos de implementar uma base de dados em memória que armazene dados fornecidos pelos docentes. Com este trabalho pretende-se desenvolver capacidades no que toca a conhecimentos essenciais da linguagem C e de Engenharia de Software, nomeadamente modularidade e encapsulamento, estruturas dinâmicas de dados, validação funcional, e medição de desempenho (computacional, consumo de memória, etc), como também ganho de experiência no uso de ferramentas essenciais ao desenvolvimento de projetos em C, nomeadamente compilação, linkagem, definição de objetivos de projeto com base nas suas dependências, depuração de erros, avaliação de desempenho e consumo de recursos, e gestão de repositórios colaborativos
O projeto está dividido em duas fases, e esta segunda fase consiste em implementar o modo interativo, o que recai mais sobre a segunda parte dos conhecimentos referidos acima. Neste modo, o programa é executado sem argumentos. Nele, o grupo disponibilizará um menu interativo que contém toda a informação (instruções) necessária para a execução de cada comando (query). Aqui, cada comando é interpretado e executado individualmente, com o respetivo resultado apresentado no terminal. Para isso, o programa deverá, inicialmente, perguntar ao utilizador qual o caminho do dataset a processar. Este modo fornece, ainda, uma funcionalidade de paginação para permitir navegar mais facilmente quando o output é longo.

Capítulo 2

Desenvolvimento

Para uma maior facilidade na execução e organização do código, o nosso grupo começou por organizar e planear. Assim, conseguimos uma visão mais clara daquilo que temos e queremos fazer, para que consigamos evitar ter de refazer grandes partes de código devido a implementações ineficientes ou menos práticas. Começamos por imaginar uma arquitetura de como seria a ligação entre os diferentes módulos e a forma como iriam comunicar entre si. Tendo em consideração o mencionado, chegamos à seguinte arquitetura:



Em primeiro lugar, fazemos o *parser* dos ficheiros *.csv*, tal como a verificação dos *users*, dos flights, das reservations, e dos passengers. Nesta fase também verificamos se as *reservations* são válidas através do catálogo dos *users* e se os *passengers* são válidos através do catálogo users e do catálogo flights. Neste processo também adicionamos, aos catálogos previamente iniciados, as *entities* necessárias.

Outra implementação criada foram os *setters* e os getters para cada uma das *entities*, tal como funções adicionais para uso dos catálogos. Temos também um módulo de estatísticas usado nas queries para ajudar a modularidade do código.

Ao fim destes passos, reparamos que era necessário reconstruir o parser para que a memória utilizada baixasse. Para isso, em vez de utilizarmos o *sscanf*, utilizamos o *fgets* e passamos às nossas funções auxiliares uma cópia de cada linha. Isto fez com que a memória utilizada pelo parser baixasse de mais de 5 GB para 2,5 GB. Foi necessário também modificar as entidades e a validação para implementar essas mudanças.

Devido ao encapsulamento, tivemos que adaptar também algumas queries para garantir que elas não o violassem. Abordaremos este exemplo mais adiante.

QUERIES

Query 1:
A query 1 consiste em listar o resumo de um utilizador, voo, ou reserva, consoante o identificador recebido por argumento. Assim, e tendo em atenção ao que nos é pedido, fizemos uma função que calcula o preço da estadia, uma que calcula o número de noites da estadia, uma que calcula o atraso do voo, uma que conta o número de passageiros no voo e a que executa a *query*. Quando o argumento recebido é um *user_id*, começamos por verificar se o *user_id* recebido está na tabela. Se o *user_id* está na tabela, vamos buscar o nome do user, bem como o seu sexo, a data de nascimento, o código do país e o número do passaporte. Além disso, criamos uma função que calcula a idade do *user*. De seguida, declaramos toda esta informação. Por outro lado, quando o argumento recebido é um *reservation_id*, começamos por verificar se o *reservation_id* recebido está na tabela. Se o *reservation_id* está inserido na tabela, vamos buscar o nome do hotel, o id do hotel, a data de começo da reserva, bem como o seu fim. Para além disso, vamos ver se a reserva inclui pequeno-almoço e vamos buscar também o preço de cada noite e o city tax. Com esta informação criamos duas funções que calculam quer o preço da estadia e o número de noites. Depois de retirarmos toda esta informação, declaramo-la. Em última instância, quando o argumento recebido é um *flight_id*, vamos verificar se o *flight_id* é diferente de zero e se este *flight_id* está presente na tabela. Se o *flight_id* está inserido na tabela, vamos buscar a airline, bem como o modelo do voo, a sua origem, a data esperada de partida, a data esperada de regresso, a data real de partida, a data esperada de chegada. Além disso, vamos buscar o piloto e o copiloto. Com os dados que retiramos, calculamos o número de passageiros desse *flight* bem como o atraso associado ao voo.

Query 3:
A query 3 consiste em apresentar a classificação média de um hotel, a partir do seu identificador. A função *exec_query_3* recebe como parâmetros um identificador de um hotel, *hotel_id*, e o catálogo dos hotéis, *hotels* catalog. Para realizar esta query, vamos buscar informação ao catálogo dos hotéis, através do *hotel_id* que nos é fornecido e, desta forma, vamos buscar ao catálogo a informação que necessitamos para efetuar a conta que nos permite calcular o rating médio de um hotel, ou seja, vamos buscar o rating do hotel e o número de *ratings* desse hotel. De seguida verificamos se o número de ratings é superior a zero e, se tal acontecer, fazemos a divisão entre o *rating* e o *number_of_ratings* e, dessa forma, calcular o *rating* médio do hotel.

Query 4:
Listar as reservas de um hotel, ordenadas por data de início (da mais recente para a mais antiga). Para isso, fizemos uma função que compara as datas iniciais de duas reservas e ordena as de forma crescente. Essa função recebe dois ponteiros const void* que apontam para as reservas a serem comparadas, converte esses ponteiros para ponteiros do tipo *RESERVATION* para que possam ser utilizados e obtém as datas de início das reservas (por meio da função *getBeginDate*). As reservas são ordenadas primeiro pela data de início em ordem decrescente e, em caso de empate, pelo identificador da reserva em ordem crescente. Posteriormente, implementamos a função que executa a *query*. Esta recebe o *ID* de um hotel, um *array* de estruturas reservation, que representa o catálogo de reservas e o tamanho desse catálogo, filtra as reservas que pertencem ao hotel específico com base no *ID* fornecido, usa um loop para percorrer o catálogo de reservas e copiar as reservas correspondentes ao hotel filtrado para um novo array chamado *filtered_reservations* e ordena esse novo *array* de reservas filtradas utilizando a função *qsort*, passando como argumento a função de comparação *compare_reservations*. Por fim, imprime os detalhes das reservas filtradas e ordenadas no console, usando informações como *ID* da reserva, datas de início e fim, *ID* do usuário, avaliação, preço total, nome e detalhes do hotel, entre outros. A função *g_array_free* é usada para libertar a memória alocada para a lista de reservas. A memória alocada para strings (*begin_date_query4, end_date_query4, user_id_query4*) é libertada dentro do loop.

Query 5:
A query 5 lista os voos com origem num dado aeroporto, entre duas datas, ordenados por data de partida estimada, sendo imprimidos os valores da mais antiga para a mais recente. Sendo que, um voo está entre a *begin date* e a *end date* caso a sua data estimada esteja entre esses dois valores. Contudo, no caso dos dois voos terem a mesma data, temos como critério de desempate o identificador do voo, sendo tal feito de forma crescente. Para fazer esta query, começamos por criar uma função que recebe como argumentos o catálogo dos *Flights*, uma vez que este contém toda a informação que necessitamos, bem como nos é fornecido os parâmetros necessários, sendo estes um array que contém o aeroporto, a *begin*

date e a *end date*. Iniciamos a função por fazer a divisão do array que recebemos, separando essa informação em três strings, sendo elas o *aeroport*, *begin_date* e *end_date* e inicializamos um array vazio chamado *resultados* onde guardamos a informação que necessitamos. De seguida, criamos uma função no catálogo dos *Flights*, chamada de *information_query5*, onde são fornecidos os parâmetros(sendo estes os mesmos que os fornecidos na query 5), o catálogo dos *flights* e um *array* onde guardaremos os resultados (array criado na função da query 5) e realizamos na mesma a divisão em 3 strings, as mesmas que na query 5. Nesta função *information_query5*, iteramos ao longo do catálogo dos flights, guardado o nome dos aeroportos que encontramos (aeroporto de origem) bem como a sua begin date do voo.Para além disso, verificamos se o *found_aeroport* é igual ao *aeroport* fornecido e se a *found_begin_date* está entre a *begin_date* e a *end_date*; se tal acontecer guardamos esses valores no array resultados. A função *information_query5* é chamada na *query5*, onde ordenamos o array dos resultados (através da função *g_array_sort*) e percorremos esse array retirando a informação que necessitamos e de seguida imprimimos as informações.

Query 6:

A *query 6* pretende listar o top N aeroportos com mais passageiros, para um dado ano. Aqui, deverão ser contabilizados os voos com a data estimada de partida no respetivo ano. Caso dois aeroportos tenham o mesmo valor, deverá ser usado o nome do aeroporto como critério de desempate (de forma crescente).

A função *execute_query6* `recebe` um catálogo de voos (*FLIGHTS_CATALOG*), uma string dos parâmetros da consulta e o caminho para o arquivo de saída. Desta forma, a função analisa os parâmetros da consulta para obter o caminho do arquivo de saída (ano e quantidade), que foi anteriormente aberto para escrita. Cria, posteriormente, um catálogo de aeroportos (*GHashTable*) e dois arrays para armazenar a origem e o destino dos voos. Seguidamente chama a função *information_query6* para obter informações relevantes sobre os voos e preenche um array com os aeroportos e ordena-os usando a função comparativo. Por fim, escreve no arquivo de saída os aeroportos com mais passageiros de acordo com a quantidade especificada e liberta a memória alocada, fechando o arquivo de saída.

A função ‘comparativo’ retorna um inteiro que representa a comparação entre os dois aeroportos. Um valor negativo indica que o primeiro aeroporto (a) é menor, um valor positivo indica que o segundo aeroporto (b) é menor, ou seja, tem menor número de passageiros, e o zero indica que são iguais. Se o número de passageiros for igual, compara os nomes dos aeroportos usando strcmp.

A função ‘ano’ recebe uma *string* que representa uma data no formato "ano/mês/dia hora:minuto:segundo" e retorna o ano extraído da *string* de data. De forma a evitar a perda de encapsulamento, alteramos algumas partes desta *query*, nomeadamente ao introduzir uma parte dela nos catálogos *(information_query6)*, uma vez que itera sobre a *hashtable*.

Query 8:

A *query 8* consiste em apresentar a receita total de um hotel entre duas datas (inclusive), a partir do seu identificador. Para isso, criamos duas funções que nos auxiliaram na execução da *query*. A primeira converte a data para um número que representa a quantidade de dias desde uma referência fixa que, neste caso, é uma simplificação para fins de cálculo. Ela recebe uma data no formato "ano/mês/dia" e extrai o ano, mês e dia da string da data. A segunda recebe uma RESERVATION, uma data inicial e uma data final como parâmetros, obtém as datas de início e fim da reserva e o preço por noite. Seguidamente, verifica se as datas da reserva estão dentro do intervalo fornecido e converte-as para um formato numérico para facilitar a comparação, calcula o número de noites entre as datas de início e fim dentro do intervalo especificado e multiplica o número de noites pelo preço por noite para obter a receita daquela reserva dentro do intervalo. Para terminar, retorna o valor da receita. 7 A função que executa a query 8 recebe um *hash table* (*GHashTable*) de reservas, uma string de parâmetros e um inteiro que representado a linha. Esta função separa os parâmetros fornecidos (ID do hotel, data inicial e data final), calcula um caminho para criar um arquivo de saída para armazenar o resultado da query, inicializa um contador (receita) para calcular a receita total do hotel e itera sobre o *hash table* de reservas, procurando reservas associadas ao ID do hotel. Para cada reserva encontrada, utiliza a função busca_receita para calcular a receita considerando as datas fornecidas e soma o valor retornado pela função busca_receita à receita e, por fim, escreve o total da receita no arquivo de saída. Alteramos algumas partes desta query, introduzindo uma parte dela nos catálogos *(information_query8),* uma vez que itera sobre a hashtable, para evitar a perda de encapsulamento, tivemos que alterar algumas partes desta *query*, introduzindo uma parte dela nos catálogos, uma vez que itera sobre a *hashtable*.

Query9:

Listar todos os utilizadores cujo nome começa com o prefixo passado pelo argumento, ordenados por nome (de forma crescente). Fizemos uma função de comparação que é utilizada para ordenar as informações de usuário. Ela recebe dois ponteiros *gconstpointer* que apontam para estruturas e converte-os para USER_INFO(Estrutura usada para guardar os usernames e ids). De seguida compara os usernames dos utilizadores utilizando a *g_utf8_collate*. Se os usernames forem iguais, compara os IDs dos usuários e retorna um valor inteiro que representa a relação de ordenação entre os dois usuários para os ordenar de forma crescente. De seguida, implementamos uma função que verifica se o utilizador possui um prefixo específico. Recebe um utilizador (*USER*) e uma *string* como argumentos, obtém o nome do usuário através da função *getName* e compara os primeiros caracteres do nome do utilizador com a *string* fornecida. No fim, retorna 1 se o prefixo estiver presente no nome do utilizador, caso contrário, retorna 0. Por fim, implementamos a função que executa a *query*, que recebe um catálogo de usuários (*USERS_CATALOG*), uma *string* de argumentos (*args*) e o nome do arquivo de saída (*output),* abre um arquivo de saída especificado em modo de escrita. Seguidamente, inicializa um iterador para percorrer o catálogo de usuários

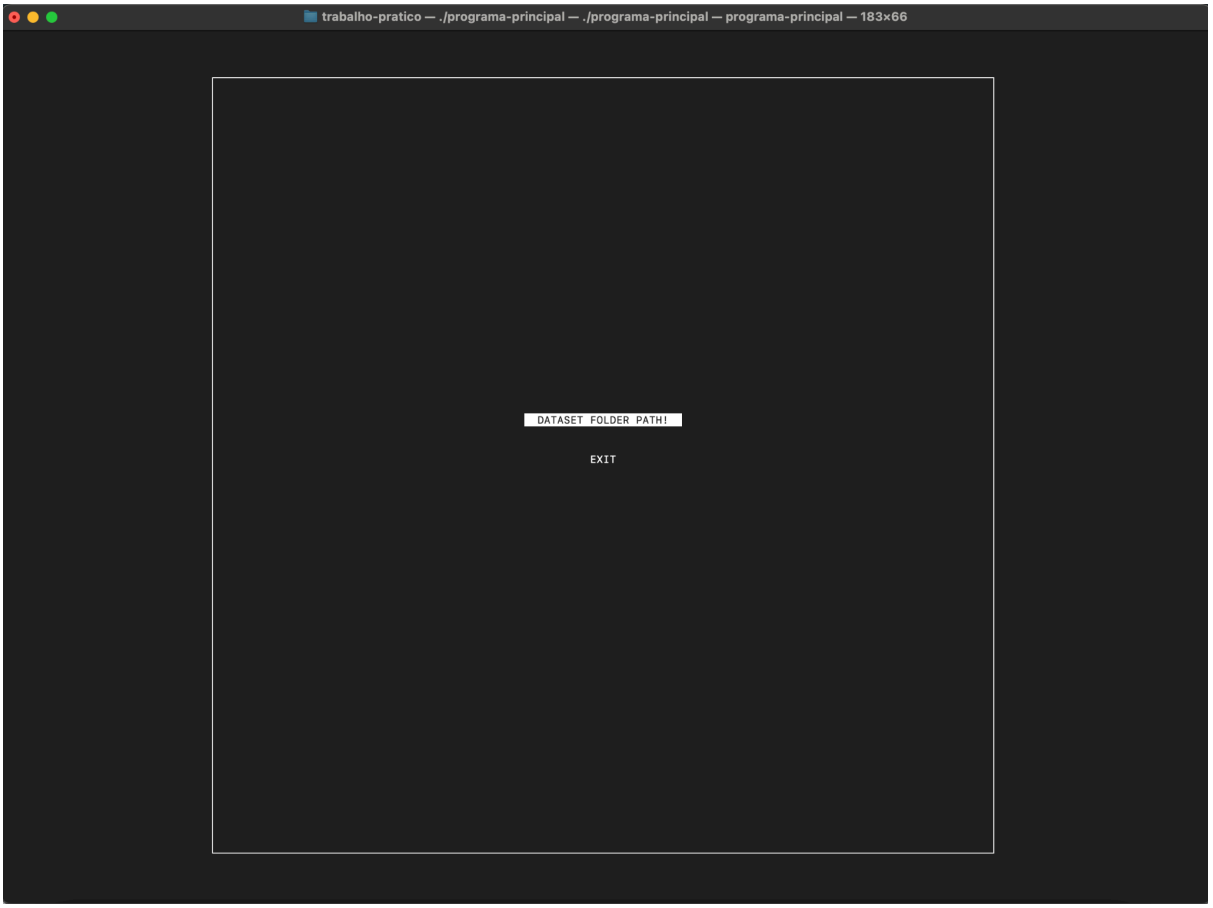
e cria um array (*userArray*) para armazenar informações sobre os usuários que atendem aos critérios estabelecidos. Esta função percorre o catálogo de usuários: Para cada usuário ativo que tenha um nome com um prefixo correspondente aos argumentos fornecidos, cria uma estrutura *USER_INFO*, contendo o nome de usuário e o *ID* do usuário, e a adiciona ao *userArray*. Ordena o *userArray* utilizando a função *compare_user_info* e escreve as informações dos usuários ordenados no arquivo de saída, no formato *user_id;username.* Para terminar, fecha o arquivo de saída e liberta a memória alocada para o *userArray*.
Para evitar a perda de encapsulamento, tivemos que alterar algumas partes desta *query,* introduzindo uma parte dela nos catálogos, uma vez que itera sobre a *hashtable*.

Capítulo 3
Modo interativo

Inicialização:

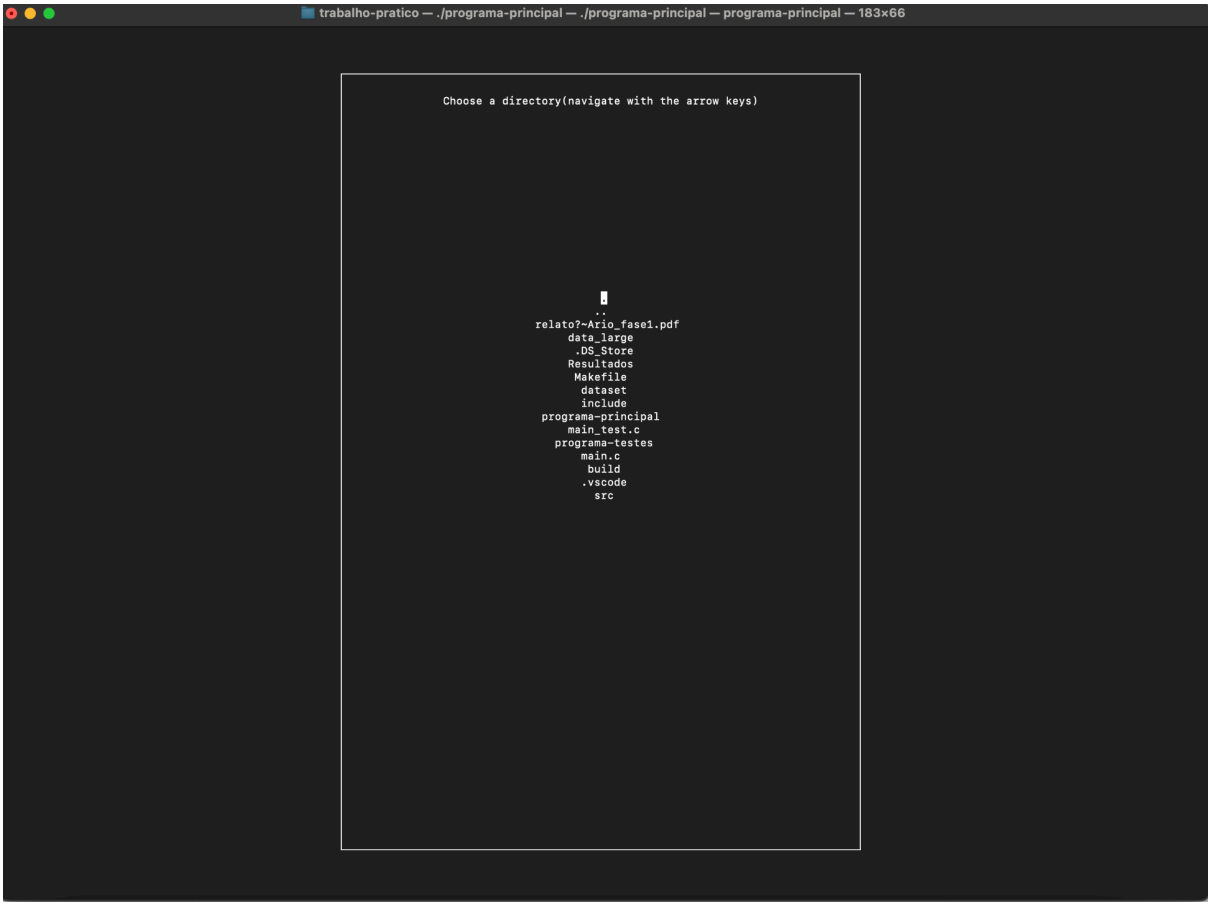
O programa inicializa a interface do menu usando a função initscr() da biblioteca ncurses. Depois, configura várias opções como entrada do teclado, sem ecoar caracteres, e esconde o cursor usando funções como keypad(), noecho(), e curs_set(0).

Menu Principal:
O menu principal apresenta duas opções: "CAMINHO DA PASTA DO CONJUNTO DE DADOS!" e "SAIR".
O utilizador navega entre estas opções usando as teclas de seta e seleciona uma opção pressionando Enter.

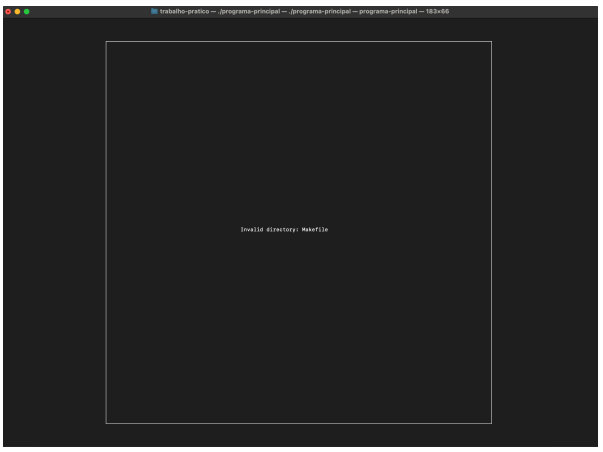


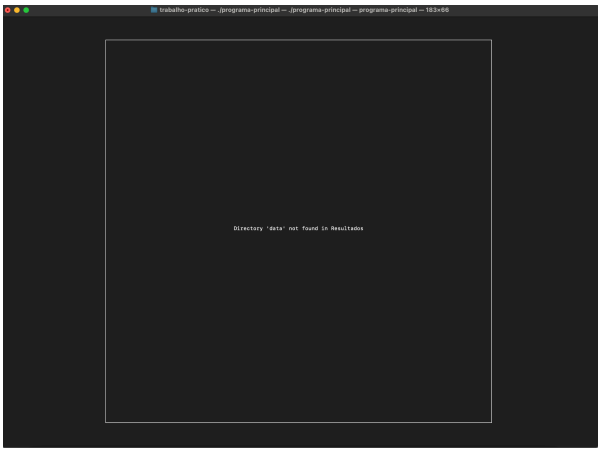
Se o utilizador selecionar "CAMINHO DA PASTA DO CONJUNTO DE DADOS!", o programa pede ao utilizador para selecionar um diretório.
A função *choose_path_menu()* é chamada para fornecer uma interface de navegação de diretório.

O utilizador pode navegar pelos diretórios usando as teclas de seta (esquerda/direita) e selecionar um diretório pressionando Enter.



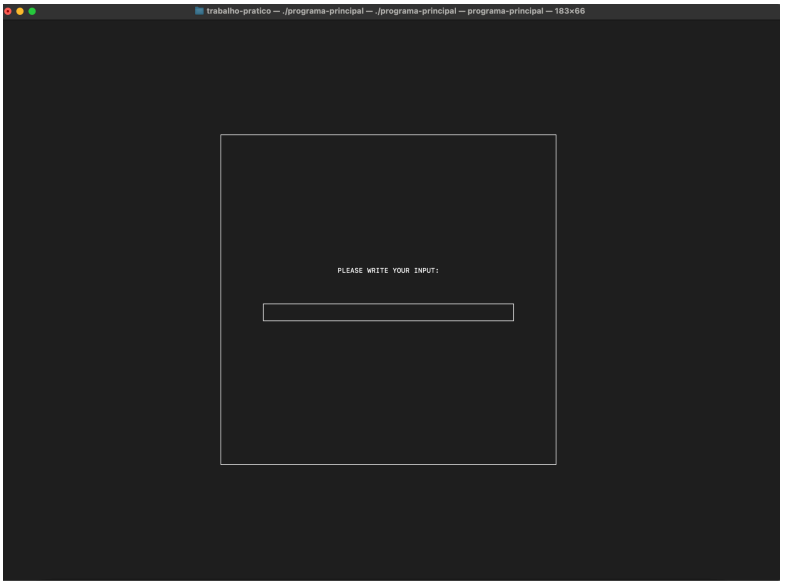
Se o caminho for invalido ou não conter o caminho data, o seguinte aparece:





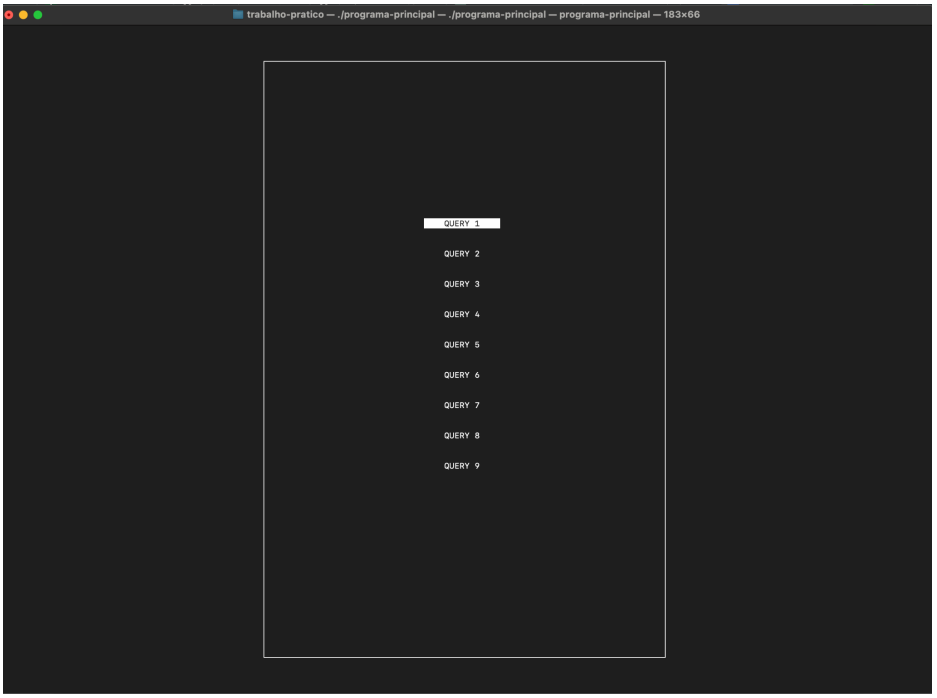
Escolha da Consulta:

Após selecionar um caminho para a pasta do conjunto de dados, o programa pede ao utilizador para selecionar uma query a realizar.
A função *choose_query_menu()* apresenta uma lista de opções de consulta (neste caso, numeradas de 1 a 9).
O utilizador navega pelas opções de consulta usando as teclas de seta (para cima/para baixo) e seleciona uma consulta pressionando Enter.



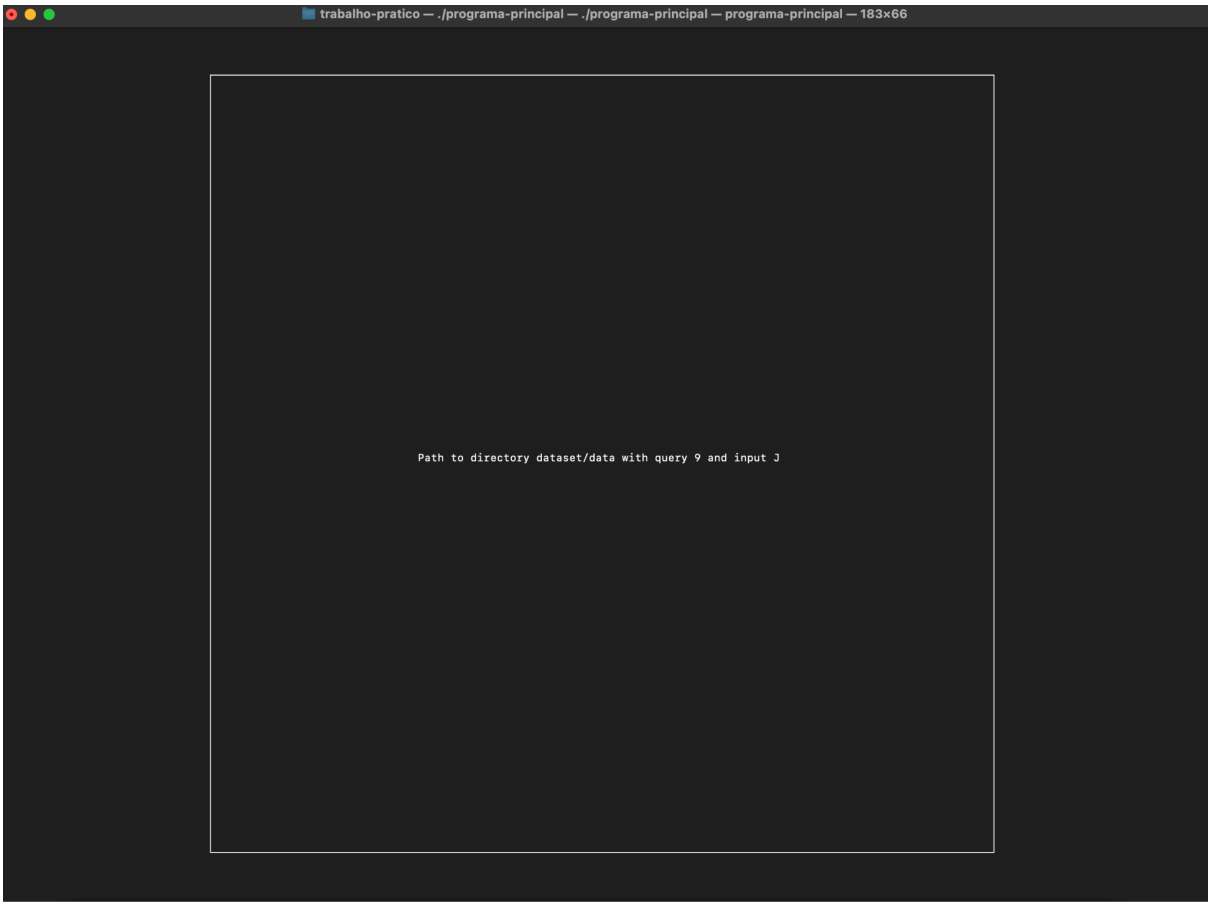
Introdução de Parâmetros da Consulta:

Depois da query selecionada, o programa solicita ao utilizador que introduza parâmetros adicionais.
A função *choose_input_menu()* permite ao utilizador introduzir parâmetros, se necessário.



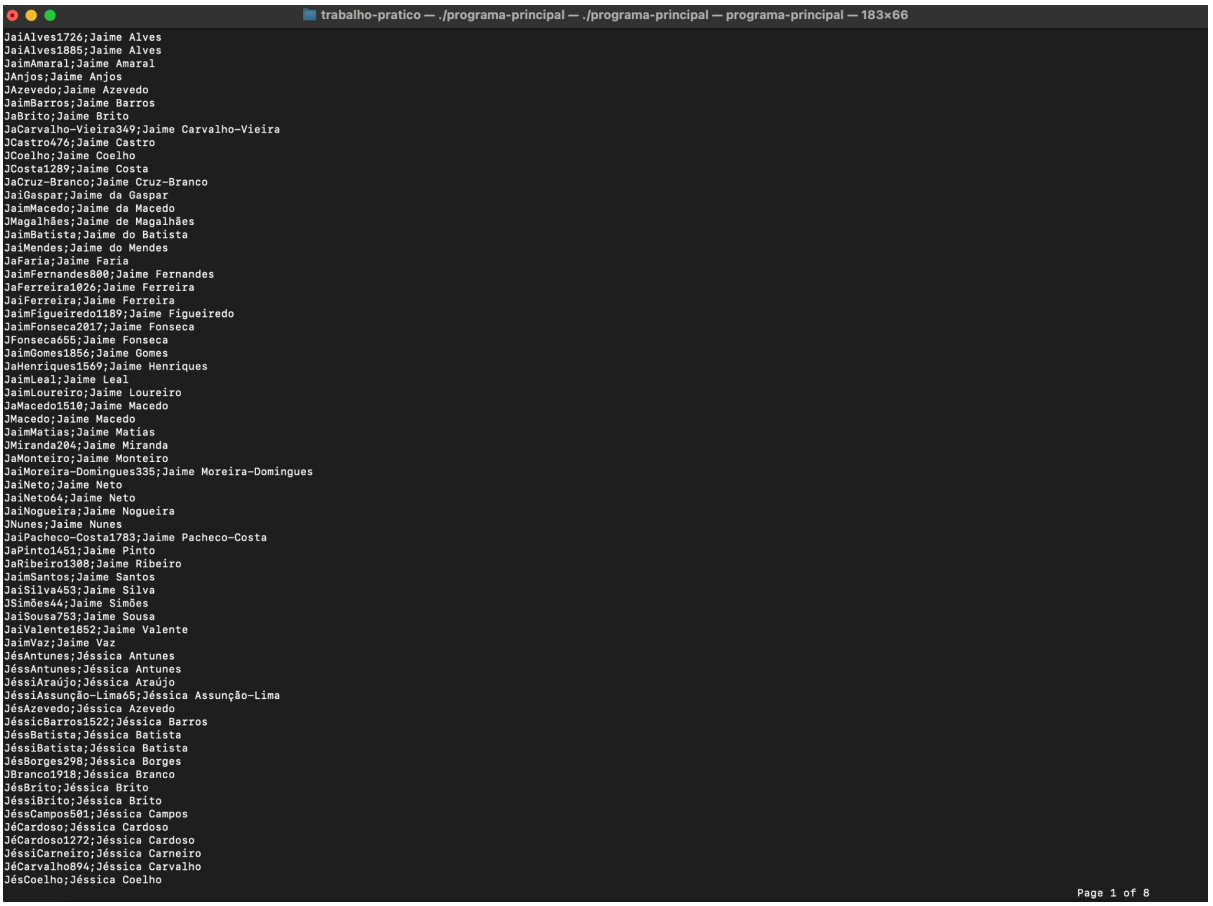
Execução da Consulta:

Uma vez selecionado o caminho da pasta do conjunto de dados e os parâmetros da consulta, o programa executa a consulta escolhida.
A função *batch_for_interactive()* processa a consulta selecionada usando o conjunto de dados fornecido e os parâmetros de entrada.



Exibição dos Resultados:

Após executar a consulta, o programa exibe os resultados num formato de menu rolável.
A função *result_menu()* exibe os resultados da consulta executada, permitindo ao utilizador navegar por várias páginas, se necessário.
O utilizador pode navegar pelos resultados usando as teclas de seta (para cima/para baixo) e sair da visualização dos resultados pressionando Enter.



Sair do Programa:

Em qualquer momento, o utilizador pode optar por sair do programa selecionando a opção "SAIR" no menu principal.

Resumindo, este menu interativo facilita aos utilizadores a seleção de um diretório contendo dados, a escolha de queries específicas para processar esses dados, a entrada de parâmetros, e a visualização dos resultados. O utilizador pode navegar pelos diretórios, selecionar queries numeradas, introduzir parâmetros de consulta e, em seguida, executar as consultas para obter resultados, tudo através de uma interface de menu intuitiva. Após a execução, os resultados são exibidos em um menu rolável, permitindo uma fácil visualização.

Capítulo 4
Dificuldades Sentidas

Durante o desenvolvimento deste projeto, diversas dificuldades foram enfrentadas, afetando diferentes áreas do processo de desenvolvimento. Uma das principais dificuldades surgiu em relação à utilização de Macs devido à gestão de memória, que muitas vezes apresentava problemas de desempenho e alocação inadequada de recursos. Isso resultou em dificuldades para lidar eficientemente com vazamentos de memória e garantir uma execução estável do código. Além disso, a incompatibilidade entre diferentes computadores da equipe complicou ainda mais a colaboração, tornando a padronização do ambiente de desenvolvimento um desafio contínuo.

Além disso, problemas de comunicação entre os membros da equipe surgiram ao longo do desenvolvimento. A falta de comunicação eficaz levou a mal-entendidos, atrasos na entrega de tarefas e inconsistências na implementação das funcionalidades. Essas dificuldades ressaltam a importância de uma comunicação clara e aberta dentro da equipe para garantir um desenvolvimento fluido e eficiente do projeto.

Capítulo 5
Desempenho e Testes

Este capítulo detalha o processo de teste realizado no projeto, focando nos conjuntos de dados normal e grande. Os testes desempenham um papel crucial no desenvolvimento de software, garantindo a qualidade e confiabilidade do sistema. Neste contexto, realizamos testes abrangentes nos conjuntos de dados para validar o desempenho, a precisão e a escalabilidade.

Dataset normal:

	Tempo de execução	Memória gasta
Teste 1 (ASUS vivobook 15)	0.508965 seg	24620 KB
Teste 2 (Lenovo thinkpad)	0.514409 seg	24716 KB
Teste 3 (HP Envy)	0.522655 seg	24628 KB

Dataset grande:

	Tempo de execução	Memória gasta
Teste 1	138.554705 seg	4988524 KB
Teste 2	140.333211 seg	4988520 KB
Teste 3	131.377954 seg	4978740 KB

Capítulo 6
Conclusão

Resumindo, este projeto foi, para nós, um grande desafio, principalmente esta segunda fase. Isto deve-se ao facto de terem sido introduzidos muitos conceitos novos e de implementação complexa. Sentimos que melhoramos o nosso trabalho em relação à fase anterior, apesar de os resultados não serem tão positivos quanto foram na fase anterior.

Como grupo, temos a noção de que poderíamos ter trabalhado melhor como um, mesmo tendo superado algumas dificuldades que enfrentamos na primeira fase em relação a este aspeto. Pretendemos, em projetos futuros, continuar a melhorar e a superar os aspetos menos bons que enfrentamos.

Apesar disso, aplicamos corretamente alguns dos fundamentos base deste projeto, como a modularidade e encapsulamento, que pensamos serem conceitos fundamentais que qualquer engenheiro de software deveria ser capaz de aplicar.