Laboratórios de Informática III

Frederico Cunha Afonso
a104001
LEI

Índice

Introdução

Este relatório objetiva apresentar uma análise detalhada sobre o projeto desenvolvido no âmbito da disciplina **Laboratórios de Informática III**. Este projeto deverá ser capaz de acessar a 4 ficheiros (.csv) relacionados a *utilizadores, voos, reservas (de um utilizador) e passageiros (de voos presentes no ficheiro .csv de voos e utilizadores no ficheiro .csv)*, fazer parsing das entradas válidas e guardar a informação essencial para responder a queries específicas.

Mais especificamente, esta aplicação tem de ser capaz de:

- Fazer parsing dos ficheiros de entrada e a sua validação;
- Ter um modo de operação batch;

  (onde o programa é executado pelo terminal com 3 argumentos: o seu executável, a diretoria dos ficheiros e o ficheiro de input das queries, e.g.: "***./programa-principal ../dataset/data ../dataset/input.txt***");

- Executar todas as 10 queries demonstradas no enunciado providenciado aos alunos;;
- Ter um modo de operação interativo, incluindo o menu de interação com o programa e um módulo
- de paginação para apresentação de resultados longos;
- Testes funcionais, que verificam a validação dos outputs criados, memória e tempo gasto durante a execução do programa;
- Respeitar os conceitos de modularidade e encapsulamento;
- Processar um dataset com uma ordem de grandeza superior.

Arquitetura

Antes de começar o desenvolvimento do programa, houve um pequeno período de tempo em que se discutiu as diversas estratégias que se podiam aplicar para ter o melhor rendimento possível em termos de tempo e memória gasta.

Estruturas de Dados

Houve vários aspetos a ter em conta na implementação de certas estruturas de dados no projeto, tendo-se separado as estruturas de dados em dois tipos:

- Estruturas de dados que visavam o armazenamento de informação diretamente relacionada com a informação de utilizadores, voos, reservas e passageiros;
- Estruturas de dados genéricas que tinham como objetivo armazenar informação genérica (**void ***) em grandes quantidades (auxiliares);

Flight

Quando se trata de voos, é usada a estrutura **Flight**, que contêm **strings** relativas ao seu identificador do voo, companhia aérea, modelo do avião, aeroporto de origem, aeroporto de destino, data e hora estimada e real de partida e data e hora estimada de chegada, tudo informação obtida do ficheiro "*flights.csv*", deixando a única informação que não se obtém diretamente desse ficheiro o número de passageiros válidos, sendo que esta é a única variável daqui que só é obtida através da leitura de um outro ficheiro.

Para determinar o número de passageiros válidos é necessário acessar ao ficheiro "*passengers.csv*" e assim contar todos os passageiros (que são utilizadores válidos) associados ao identificador do voo.

User

A estrutura **User** trata de guardar toda a informação relevante de um utilizador. Assim, esta contém o identificador, o nome, a data de nascimento, o género, o passaporte, o código do país de residência, a data de criação e estado da conta de um dado utilizador. Toda esta informação é obtida diretamente do ficheiro .csv relativo a utilizadores (*users.csv*).

É de notar que, de maneira a gastar o mínimo de memória possível, passei de usar uma **string** para guardar o género e estado de conta de um utilizador para usar um short, sendo que ambos só podem assumir dois estados (género feminino ou masculino e estado de conta ativo ou inativo);

Reservation

Como última estrutura *não auxiliar*, a **Reservation** contém o identificador da reserva, identificador do utilizador, identificador do hotel, nome do hotel, número de estrelas do hotel, percentagem do imposto da cidade (sobre o valor total), data de início e fim de estadia, preço por noite, inclusão de pequeno-almoço e classificação atribuída pelo utilizador de uma certa reserva. Tal como com a **User**, toda a informação contida desta estrutura é obtida ao acessar o ficheiro que lhe é relativo (*reservations.csv*)

Também como com a **User**, em vez de guardar toda a informação como **strings**, transformaram-se certos dados de **string** para outros tipos de forma a reduzir a quantidade de memória necessária para guardar os dados.

Como um **char *** requer 8 bytes de memória, houve uma reestruturação desta estrutura de maneira a usar o mínimo número de **char *** possíveis.

A classificação de uma reserva e o número de estrelas só pode variar entre 1 e 5 (ou, no caso da classificação, pode incluir 0 caso seja nula) , por isso passei a usar **char** (1 byte).

A percentagem do imposto da cidade e o preço por noite de uma reserva tem que ser um inteiro superior a 0, por isso (esperando que não seja maior que 10000) passei a usar um **short** (2 bytes).

O identificador de um hotel tem o formato “HTL1102”, logo, em vez de guardar os primeiros caracteres que estão presentes em todos os outros identificadores, apenas guardo o número seguido pelo “HTL” (e.g. de “HTL1102” será apenas guardado “1102”) num **short** (2 bytes)

A inclusão de pequeno-almoço só pode ter dois estados (incluir ou não incluir), passei a usar um **char** (1 byte).

O identificador da reserva tem o formato “BookXXXXXXXXXX” (alterando os X’s por números), por isso seria uma boa ideia guardar apenas o número (**int**) do id em vez da **string** toda (e.g. de “Book0000000002”, guardar apenas o 2), mas infelizmente isto só seria possível para um quarto de todas as possíveis reservas a serem feitas, pois como um **int** apenas consegue armazenar **2,147,483,647** números positivos e é possível haver **9,999,999,999** reservas (Book9999999999), isto não é possível. Logo, de maneira a não usar um **char ***, passei a guardar os últimos 9 dígitos do identificador como um **int** (4 bytes) e o primeiro dígito como um **char** (1 byte), usando assim apenas 5 bytes (e.g. para o identificador “Book1234567890”, será guardado como primeiro dígito ‘1’ e será guardado como **int** “234567890”).

Passando assim de usar **88 bytes** por **Reservation** a usar **46 bytes** (o que se torna numa grande diferença quando estivermos a tratar de datasets de maior escala).

Auxiliares

De maneira a responder às queries da forma mais eficiente possível, foram usadas estruturas de dados capazes de receber informação genérica com diferentes níveis de eficiência de inserção e obtenção de dados nos catálogos.

BTree

Btrees (árvores binárias de procura) são estruturas de dados incrivelmente eficientes na armazenação organizada de dados genéricos (**void ***) quando comparada a outras estruturas de dados. Tal como **HashTables** e **Stacks**, **BTrees** servem para guardar apontadores para estruturas de

dados criadas (**strings**, outras estruturas de dados, etc.), sendo assim eficientes em termos de memória.

Neste projeto em específico, foram implementadas nos catálogos de utilizadores (na armazenação de datas e horas de partida de vôos aos quais certos utilizadores tenha ido) e nos catálogos de voos (na armazenação de **Flights** organizados pela sua data e hora de partida em todos os nodos relacionados a aeroportos de algum dado voo).

Tempo de Inserção	Tempo de Procura	Organização
O(log n)	*O(log n)*	*Personalizada, feita por uma função dada à inserção*

Calendar

Diferente de **BTrees**, **Stacks** e **HashTables**, a estrutura **Calendar** serve para armazenação de várias métricas gerais da aplicação, tais como utilizadores, voos, passageiros, passageiros únicos (utilizadores que só estiveram num voo num dia/mês/ano) e reservas. Esta será mais tarde complementada por um catálogo que permite relacionar datas a esta estrutura.

Foi criada de maneira a responder à Query 10 da forma mais eficiente possível (em termos de tempo, pois é capaz de ocupar uma quantidade considerável de memória), sendo basicamente um array de 5 elementos e tendo funções que somam cada uma das métricas mencionadas anteriormente após serem validadas.

Todas as métricas de avaliação apresentadas foram feitas com o Catálogo de Datas em mente.

Tempo de Inserção	Tempo de Procura	Organização
O(1)	*O(1)*	*Feita pela data (de reserva, criação de conta ou partida)*

FHash

Esta estrutura é simplesmente uma **HashTable** com "closed addressing" em mente, sendo um array de **N** nodos em que cada um destes é uma lista ligada com de **void ***.

Esta é criada por uma função que recebe como argumento o número de nodos a serem criados (**N**). A função de inserção de dados (**função hash**) pode ser feita de 2 maneiras (dependendo do que o programa queira):

- Ao tornar uma dada **string** num **int** pela função de hash DJB2 (Catálogo de Datas, Catálogo de Reservas no tratar de hotéis e no Catálogo de Utilizadores);
- Ao tornar a sub-**string** 4 posições à frente da original num número inteiro (Catálogo de Voos e Catálogo de Reservas no tratar direto de reservas);

Apesar de ser uma estrutura ótima, para libertar a informação nela armazenada, é necessário especificar-lhe uma função personalizada que liberte a memória dos elementos armazenados. Uma otimização que poderia ser facilmente feita, seria ser capaz de dar à função de inserção e procura uma função de hash em vez de predefinir só dois tipos de *hash_functions*.

Tempo de Inserção	Tempo de Procura	Organização
O(1)	*O(1)*	*Feita pela string chave dada*

Stack

Uma das estruturas mais interessantes deste projeto, sendo ironicamente uma das mais simples.

Apesar de não ter qualquer tipo de organização (como **BTrees**, **FHashs** ou **Tries**), o tempo de inserção é de O(1), e contrário às outras estruturas, não aumenta consoante outros dados que já tinham sido inseridos na estrutura (o que é ótimo na inserção de uma grande quantidade de dados).

A inserção, criação e eliminação de uma **Stack** são extremamente diretas, o que difere esta estrutura das outras, é a maneira como obtém informação específica de dados que lhe foram inseridos. Para isto é necessário indicar o número de **strings** por entidade a serem guardadas, um inteiro que indique quantas entidades existem e uma função que, dada a tal entidade, copia para o array de **strings** resultantes todos os elementos a serem copiados compactados (como demonstrado na documentação da função *stack_to_char_array* no ficheiro "Stack.c").

Tempo de Inserção	Tempo de Procura	Organização
O(1)	*O(N * argumentos_desejados)*	—

Trie

Uma mistura entre uma Árvore Binária e um array, onde dado um nome e um id, vai percorrer o nome todo de letra a letra, e por cada **char** de 'A' a 'z' passa para um próximo "nível" da **Trie** (quando apresentada por um **char** com valores fora deste limite (e.g. 'Á', 'ö', etc.) continua para o próximo char do nome, após percorrer todo o nome, guarda ambos id e nome numa **Stack** juntos.

Esta estrutura, tal como a **Calendar**, foi criada com o objetivo de responder a uma query em específico, a Query 9. Assim, quando for preciso procurar por todos os nomes com um prefixo específico, só seria necessário percorrer a **Trie** com o prefixo, e com uma função personalizada, guardar todos os nomes e ids do "nível" específico da **Trie** relacionada ao prefixo e todos os "sub-níveis" seguintes, mais tarde organizando-os com a função *sort_strings*.

Tempo de Inserção	Tempo de Procura	Organização
O(k)	*O(k + N_elementos_dos_sub_níveis)*	*Feito pelo nome dado*

Catálogos

Catálogo de Passengers

Dado que o ficheiro *passengers.csv* apenas contém informação pertinente aos utilizadores e aos voos, não vi necessidade em criar uma estrutura específica para este (tal como com os utilizadores, voos e reservas), logo, tomei a decisão de apenas criar um array capaz de guardar o mesmo número de **ints** que há entradas no ficheiro *flights.csv*, para assim contar todos os passageiros válidos do *passengers.csv*.

Como a validação de voos depende da quantidade de passageiros que um voo tem, é necessário ter o *passengers.csv* em consideração na validação do *flights.csv*. Apesar de ser possível tornar o processo mais eficiente, infelizmente, não houve tempo suficiente para implementar uma versão melhor. Este processo pode ser visto da seguinte maneira:

1. *passengers.csv* (apenas para a contagem de lugares)
   1.1. Validação do tamanho da **string** do utilizador e voo
   1.2. Validação do utilizador
2. *flights.csv* (validação)
3. *passengers.csv* (validação)
   3.1. Validação do tamanho da **string** do utilizador e voo
   3.2. Validação do utilizador
   3.3. Validação do voo

Isto é incrivelmente ineficiente visto que estou a validar as entradas do *passengers,csv* duas vezes. Uma solução para isto seria, na contagem, associar cada entrada válida a um número e após a contagem, ao voltar a abrir o *passengers.csv*, só seria necessário validar o voo e verificar se esta é uma entrada válida pela linha guardada usando um contador presente na estrutura **Catalog**, assim substituindo os pontos 3.1 e 3.2 por uma simples comparação de **ints** em vez de perder tempo a verificar strings e procurar um user numa **HashTable** (esta implementação está presente no programa, mas infelizmente não se encontra funcional).

Calendar_Almanac

Semelhante à estrutura **Calendar**, este catálogo tem como objetivo responder à Query 10, sendo que estas têm funcionalidades diferentes que se complementam perfeitamente, visa que:

- A estrutura **Calendar** conta as reservas e voos feitos, utilizadores criados e passageiros normais e únicos num dado momento;
- O catálogo **Calendar_Almanac** guarda informação de acordo a uma certa data, guardando-a num ano, no mês do ano e no dia do mês.

Após a validação de um utilizador, reserva e voo, é usada a sua dada função da **Calendar** para contá-lo na sua, e tem-se em conta o *schedule_departure_date*, *account_creation* ou *begin_date* para utilizar o **Calendar** apropriadamente neste catálogo. É de notar que, para contar corretamente o número de passageiros únicos, foi necessário criar uma função específica (*is_unique_passenger*) que percorre a **BTree,** mencionada no **User_Almanac**, e compara a dada *schedule_departure_date* a todas as datas presentes na árvore e determina se o utilizador é (ou não) um passageiro único no dia, mês e/ou ano da *schedule_departure_date*.

Este catálogo usa **FHashs** para armazenar nodos especiais que contêm **Calendars** e outros **FHashs** dentro deles para "recursivamente" guardar a informação querida do "ano" ao "mês" ao "dia" da data, usando cada um destes como chave para procurar ou inserir o elemento.

Flight_Almanac

O **Flight_Almanac** é o catálogo relacionado aos voos presentes no ficheiro *flights.csv*, armazenando assim todas as entradas do ficheiro como **Flights** em diferentes estruturas, sendo estas:

- **HashTables** (Permitindo assim o acesso direto a informação de um voo ou nodo, dado o seu identificador como chave de procura);
- Nodos personalizados para:
    - Aeroportos (com o nome do aeroporto como identificador/chave do nodo), que contém uma árvore binária de **Flights** (ordenada pela data e hora de partida estimada dos voos de maneira a responder à Query 5), e uma lista de atrasos (correspondente à diferença entre a data e hora de partida estimada e real de cada voo);
    - Anos (com o ano como identificador/chave do nodo), que contém a quantidade de passageiros que estiveram num aeroporto (quer seja de origem ou de destino) e os tais aeroportos (usado mais tarde na execução da Query 6);
- **Stacks** (Permitindo assim um acesso rápido e geral a todos os aeroportos);
- Arrays (e.g. o *airport_names_delay* e o *airport_median_delays* servem para guardar o nome e a mediana de atrasos de um certo aeroporto após a validação de todos os voos presentes no ficheiro *flights.csv*, assim tratando da Query 7 eficientemente).

Reservation_Almanac

O **Reservation_Almanac** é o catálogo que trata de armazenar a informação de diferentes entradas do ficheiro *reservations.csv*, sendo que trata desta usando duas **HashTables**, uma que contém a estrutura **Reservations** e outra que contém nodos personalizados (**Hotéis**), sendo que estes apenas contêm a parte numérica do identificador de um hotel ("HTL**1104**") como identificador/chave do nodo, e uma **Stack** de todas as reservas com o mesmo identificador de hotel (respondendo à Query 3, 4 e 8 ao verificar todos os elementos da **Stack**).

User_Almanac

O **User_Almanac** é o catálogo que contém toda a informação relevante de todas as entradas válidas do ficheiro *users.csv*. Este último catálogo pode ser dividido em duas estruturas distintas:

- Uma **Trie** que contém os nomes e identificadores de todos os utilizadores válidos (ordenada pelo nome dos utilizadores, tratando assim da Query 9)
- Uma **HashTable** que contém nodos personalizados que representam cada utilizador, contendo estes a estrutura **User**, uma **Stack** com **Reservations** relacionadas às reservas feitas e outra **Stack** com **Flights** relacionados a voos a que o utilizador foi passageiro (respondendo assim à Query 2), e uma **BTree** (esta contém as datas e horas de partida dos voos presentes na **Stack**).

Parser

De forma a trabalhar com um **Parser** genérico, tratei de fazer com que os argumentos que a função recebesse fossem:

- A diretoria dos datasets (dada pelo utilizador no modo batch pelo terminal ou no modo interativo);
- O tipo (nome do ficheiro *.csv* a ser avaliado, ou seja, só podia ser um destes 4: *"users", "reservations", "flights", "passengers"*);
- O **Almanac** (catálogo geral);
- Uma função genérica que recebe o **Almanac** e uma **string** e retorna um **int** (1 se a função verificar que a **string** dada/entrada de um certo ficheiro é válida perante o ficheiro em que se encontra, e 0 caso contrário);

Assim a função apenas teria de juntar a **string** da diretoria dos datasets com o tipo dado, abrir o ficheiro presente, verificar que linhas do ficheiro são válidas com a função genérica e guardar a informação destas no **Almanac**.

Validação dos Utilizadores

Dado que cada entrada do *users.csv* tem de ser uma string com os seguintes elementos:

- id – identificador do utilizador;
- name – nome;
- email – email;
- phone_number – número de telemóvel;
- birth_date – data de nascimento;
- sex – sexo;
- passport – número do passaporte;
- country_code – código do país de residência;
- address – morada;
- account_creation – data de criação da conta;
- pay_method – método de pagamento;
- account_status – estado da conta.

Será necessário verificar que:

- O email de um utilizador tem que ter o formato: "<username>@<domain>.<TLD>". O <username> e o <domain> têm que ter pelo menos tamanho 1; O <TLD> tem que ter pelo menos tamanho 2. (Exemplo de erros: @email.com, john@.pt, john@email.a, john@email,pt, john.email.pt, …);
- O birth_date tem que vir antes de account_creation (o formato de ambos deverá ser sempre do tipo nnnn/nn/nn, onde n é um número entre 0 e 9, o mês deverá estar entre 1 e 12 e o dia entre 1 e 31, todos os casos são inclusivos);
- O country_code de um utilizador deverá ser formado por duas letras;
- O account_status de um utilizador deverá ter o valor "active" ou "inactive", sendo que diferentes combinações de maiúsculas e minúsculas também são válidas (e.g., "Active", "aCtive", e "INACTIVE" também são válidos);
- Os seguintes restantes campos têm que ter tamanho superior a zero: id, name, phone_number, sex, passport, address, pay_method;

Tudo isto pode ser verificado no ficheiro *users.csv* diretamente, logo é ideal começar o programa pelo parsing deste ficheiro.

Validação das Reservas

Dado que cada entrada do *reservations.csv* tem de ser uma string com os seguintes elementos:

- id – identificador da reserva;
- user_id – identificador do utilizador;
- hotel_id – identificador do hotel;
- hotel_name – nome do hotel;
- hotel_stars – número de estrelas do hotel;
- city_tax – percentagem do imposto da cidade (sobre o valor total);
- address – morada do hotel;

- begin_date – data de início;
- end_date – data de fim;
- price_per_night – preço por noite;
- includes_breakfast – se a reserva inclui pequeno-almoço;
- room_details – detalhes sobre o quarto;
- rating – classificação atribuída pelo utilizador;
- comment – comentário sobre a reserva.

Será necessário verificar que:

- O identificador do utilizador pertence a um utilizador válido (tem que estar presente no *users.csv* e tem de pertencer a um user válido);
- O número de estrelas de um hotel (hotel_stars) tem que ser um valor inteiro entre 1 e 5, inclusive. Exemplos de erros: 0, -3, 1.4, …;
- A percentagem de imposto da cidade de uma reserva (city_tax) tem que ser um valor inteiro maior ou igual a zero; Exemplos de erros: -3, 1.4, ...;
- O begin_date tem que vir antes do end_date (o formato de ambos deverá ser sempre do tipo nnnn/nn/nn, onde n é um número entre 0 e 9, o mês deverá estar entre 1 e 12 e o dia entre 1 e 31, todos os casos são inclusivos);
- O preço por noite de uma reserva (price_per_night) tem que ser um valor inteiro maior que 0. Exemplo de erros: 0, -3, 1.4, . . . ;
- As classificações de uma reserva (rating) têm que ter um valor inteiro entre 1 e 5, inclusive. Exemplos de erros: 0, -3, 1.4, ... Opcionalmente, podem estar vazias caso o utilizador não tenha classificado o hotel;
- Os seguintes restantes campos têm que ter tamanho superior a zero: id, user_id, hotel_id, hotel_name, address;

Antes de conseguir validar qualquer entrada neste ficheiro, é necessário verificar o ficheiro *users.csv*, impossibilitando a validação do *reservations.csv* antes da validação do *users.csv*.

Validação dos Voos

Dado que cada entrada do *flights.csv* tem de ser uma string com os seguintes elementos:

- id – identificador do voo;
- airline – companhia aérea;
- plane_model – modelo do avião;
- total_seats – número de lugares totais disponíveis;
- origin – aeroporto de origem;
- destination – aeroporto de destino;
- schedule_departure_date – data e hora estimada de partida;
- schedule_arrival_date – data e hora estimada de chegada;
- real_departure_date – data e hora real de partida;
- real_arrival_date – data e hora real de chegada;
- pilot – nome do piloto;
- copilot – nome do copiloto;
- notes – observações sobre o voo.

Será necessário verificar que:

- O número de lugares de um voo (total_seats) não poderá ser inferior ao número de passageiros nesse voo (número de utilizadores válidos presentes no *passengers.csv* associados ao id do voo);
- Os aeroportos de origem e destino de um voo tem que ser constituídos por 3 letras. Considera-se que dois aeroportos são iguais quando são formados pela mesma sequência de letras, mesmo que existam diferenças entre maiúsculas e minúsculas (e.g., OPO, opo, e opO são considerados o mesmo aeroporto);
- O schedule_departure_date tem que vir antes de schedule_arrival_date, e o real_departure_date tem que vir antes de real_arrival_date (o formato destes deverá ser sempre do tipo nnnn/nn/nn nn:nn:nn, onde n é um número entre 0 e 9, o mês deverá estar entre 1 e 12, o dia entre 1 e 31, a hora deverá estar entre 0 e 23, os minutos entre 0 e 59, e os segundos entre 0 e 59, todos os casos são inclusivos);
- Os seguintes restantes campos têm que ter tamanho superior a zero: id, airline, plane_model, pilot, copilot;

Antes de conseguir validar qualquer entrada neste ficheiro, é necessário verificar o ficheiro *passengers.csv*, impossibilitando a validação do *flights.csv* antes de efetuar uma contagem apropriada no *passengers.csv*.

Validação dos Passageiros

Dado que cada entrada do *passengers.csv* tem de ser uma string com os seguintes elementos:

- flight_id – identificador do voo
- user_id – identificador do utilizador

Será necessário verificar que:

- O identificador do utilizador pertence a um utilizador válido (tem que estar presente no *users.csv* e tem de pertencer a um user válido);
- O identificador do voo pertence a um voo válido (tem que estar presente no *flights.csv* e tem de pertencer a um voo válido);
- Os seguintes restantes campos têm que ter tamanho superior a zero: flight_id, user_id;

Antes de conseguir validar qualquer entrada neste ficheiro, é necessário verificar o ficheiro *users.csv* e *flights.csv*, impossibilitando a validação do *passengers.csv* antes da validação do *users.csv* e do *flights.csv*.

Assim, escolhi usar a ordem que aparentava ser a mais simples (e infelizmente, algo que inefeciente):

1. Validação do *users.csv;*
2. Validação do reservations*.csv;*
3. *Contagem no passengers.csv;*
4. Validação do *flights.csv;*
5. Validação do *passengers.csv;*

Interativo

O modo interativo foi feito usando todas as ferramentas disponibilizadas pela libc, não usando a biblioteca ncurses.h pois tinha como objetivo realizar um trabalho sem qualquer memory leak, independentemente do modo utilizado (isto veio a incentivar a criação de HashTables “personalizadas”).

Para analisar o modo interativo criado podemos separá-lo em 6 categorias:

- Executar uma Query;
- Visualizar o Menu de Queries já executadas
- Adicionar um input.txt preenchido de Queries já existentes;
- Adicionar um dataset;
- Apresentar o Output de uma Query executada;
- Paginação;

O modo interativo, antes de mais, não permite que o utilizador execute qualquer Query ou adicione algum input.txt até um dataset válido ter sido inserido, até lá, qualquer tentativa de executar alguma Query será ignorada. Após o utilizador adicionar um dataset válido, o programa executará como se estivesse no modo batch, mas não irá ainda executar qualquer Query, para isso será necessário o utilizador executar Queries individuais ou carregar um ficheiro input.txt com Queries válidas. Após uma das duas serem feitas, o programa irá adicionar a(s) Query(ies) a um ficheiro que tem em conta todas as Queries já executadas. Este ficheiro foi criado no início da execução do modo interativo e tem exatamente o mesmo formato que o ficheiro “*input.txt*” providenciado pelos professores.

Caso seja executada uma Query individualmente, o programa irá executá-la e guardar o resultado na diretoria “Resultados/”, apresentando o início desse ficheiro de texto no terminal, dando ao utilizador saltar para uma linha específica do ficheiro ou avançar/recuar um número constante de linhas dependendo do tamanho do terminal (paginação).

Finalmente, o utilizador pode verificar todas as Queries já executadas ao selecionar o Menu de Queries. Este pode ser visualizado como o resultado de uma Query executada, com uma adição: a capacidade de selecionar uma linha em específico do ficheiro para abrir a Query executada correspondente à linha (melhor explicado nas instruções).

Interpreter

Após o parsing dos ficheiros esperados, o programa deverá ser capaz de abrir o ficheiro *input.txt* (obtido no modo interativo pelo utilizador ou no modo batch como terceiro argumento ao executar o programa), ler cada linha presente e executá-la caso tenha o formato correto.

O executar de cada linha deve consistir em: criar um ficheiro com o formato *commandN_output.txt* (onde o N seria substituído pelo número da linha da Query executada) e posteriormente, inserir nesse ficheiro de texto o resultado esperado da Query, caso esta tenha um formato válido.

Query 1

Comando: `1 <ID>`

Objetivo: Listar o resumo de um utilizador, voo, ou reserva, consoante o identificador recebido por argumento. É garantido que não existem identificadores repetidos entre as diferentes entidades (Não deverão ser retornadas informações para utilizadores com account_status = "inactive").

Resposta: O programa irá usar uma das HashTables criadas para armazenar **Users**, **Flights** ou **Reservations** (dependendo do identificador recebido) e irá acessar à entidade desejada (caso exista), obtendo dela a informação esperada pela Query.

Query 2

Comando: `2 <ID> [flights|reservations]`

Objetivo: Listar os voos ou reservas de um utilizador, se o segundo argumento for flights ou reservations, respetivamente, ordenados por data (da mais recente para a mais antiga). Caso não seja fornecido um segundo argumento, apresentar voos e reservas, juntamente com o tipo (flight ou reservation). Para os voos, a data a ter em conta é o schedule_departure_date, para as reservas, a data a ter em conta é o begin_date. Em caso de empate, ordenar pelo identificador (de forma crescente). Utilizadores com account_status = "inactive" deverão ser ignorados.

Resposta: O programa irá usar a HashTable criada para armazenar nodos de utilizadores, sendo que estes contêm uma **Stack** que contém os voos (**Flights**) a que o utilizador tenha sido passageiro e uma **Stack** que contém as reservas (**Reservations**) a que o utilizador tenha feito e irá copiar apenas o id e o schedule_departure_date e/ou begin_date dos voos e/ou reservas para um array de strings desorganizado e mais tarde usará a função *sort_strings* (*O(N * log(N)*) para organizar o resultado como seria desejado.

Query 3

Comando: `3 <ID>`

Objetivo: Apresentar a classificação média de um hotel, a partir do seu identificador

Resposta: O programa irá usar uma das HashTables criadas para armazenar nodos que representam hotéis e irá copiar as classificações de todas as reservas (**Reservations**) feitas para o dado Hotel, e (após torná-las em **ints**) irá calcular a média desejada.

Query 4

Comando: `4 <ID>`

Objetivo: Listar as reservas de um hotel, ordenadas por data de início (da mais recente para a mais antiga). Caso duas reservas tenham a mesma data, deve ser usado o identificador da reserva como critério de desempate (de forma crescente).

Resposta: Tal como a Query anterior, o programa irá usar a HashTable que armazena nodos que representam hotéis e irá obter toda a informação esperada pela Query de todas as reservas (**Reservations**) feitas para o dado Hotel, e finalmente, irá organizá-las como a Query o indicou.

Query 5

Comando: `5 <Name> <Begin_date> <End_date>`

Objetivo: Listar os voos com origem num dado aeroporto, entre duas datas, ordenados por data de partida estimada (da mais recente para a mais antiga). Um voo está entre <begin_date> e <end_date> caso a sua respetiva data estimada de partida esteja entre <begin_date> e <end_date> (ambos inclusivos). Caso dois voos tenham a mesma data, o identificador do voo deverá ser usado como critério de desempate (de forma crescente).

Resposta: O programa irá usar a HashTable que armazena nodos que representam aeroportos, que contém uma árvore binária de **Flights**, e irá percorrer toda a árvore binária e imprimir diretamente no ficheiro de output qualquer **Flight** que se encontre entre a <begin_date> e <end_date>. Como a árvore encontra-se ordenada pela data e hora de partida estimada dos voos, ao imprimir os voos que se encontrem entre estas datas, a ordem será exatamente como a Query deseja sem ter de fazer qualquer tipo de ordenação.

Query 6

Comando: `6 <Year> <N>`

Objetivo: Listar o top N aeroportos com mais passageiros, para um dado ano. Deverão ser contabilizados os voos com a data estimada de partida nesse ano. Caso dois aeroportos tenham o mesmo valor, deverá ser usado o nome do aeroporto como critério de desempate (de forma crescente).

Resposta: O programa irá usar a HashTable que armazena nodos que representam anos no catálogo de voos, e irá imprimir diretamente os primeiros **N** elementos do array de **strings** com os nomes de aeroportos e os primeiros **N** elementos do array de **ints** que representam o número de passageiros presentes num dado aeroporto.

Query 7

Comando: `7 <N>`

Objetivo: Listar o top N aeroportos com a maior mediana de atrasos. Atrasos num aeroporto são calculados a partir da diferença entre a data estimada e a data real de partida, para voos com origem nesse aeroporto. O valor do atraso deverá ser apresentado em segundos. Caso dois aeroportos tenham a mesma mediana, o nome do aeroporto deverá ser usado como critério de desempate (de forma crescente).

Resposta: O programa irá usar o catálogo de voos, e, tal como a Query anterior, irá imprimir diretamente os primeiros **N** elementos do array de **strings** com os nomes de aeroportos e os primeiros **N** elementos do array de **ints** que representam a mediana de atrasos de um dado aeroporto.

Query 8

Comando: `8 <Id> <Begin_date> <End_date>`

Objetivo: Apresentar a receita total de um hotel entre duas datas (inclusive), a partir do seu identificador. As receitas de um hotel devem considerar apenas o preço por noite (price_per_night) de todas as reservas com noites entre as duas datas. E.g., caso um hotel tenha apenas uma reserva de 100€/noite de 2023/10/01 a 2023/10/10, e quisermos saber as receitas entre 2023/10/01 a 2023/10/02, deverá ser retornado 200€ (duas noites). Por outro lado, caso a reserva seja entre 2023/10/01 a 2023/10/02, deverá ser retornado 100€ (uma noite).

Resposta: O programa irá usar uma das HashTables criadas para armazenar nodos que representam hotéis e irá copiar o preço por noite, a data de início e fim de todas as reservas (**Reservations**) da **Stack**, mais tarde irá calcular a receita de cada reserva (caso se encontre dentro das datas dadas) e finalmente, irá somar todas as receitas e obterá assim a receita desejada.

Query 9

Comando: `9 <Prefix>`

Objetivo: Listar 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, deverá ser usado o seu identificador como critério de desempate (de forma crescente). Utilizadores inativos não deverão ser considerados pela pesquisa.

Resposta: O programa irá usar a **Trie** do catálogo de utilizadores, e irá percorrê-la de letra a letra do prefixo dado até este "acabar", aí o programa irá copiar todos os nomes presentes no nível atual e nos próximos "sub-níveis" da **Trie**.

Query 10

Comando: `10 [year [month]]`

Objetivo: Apresentar várias métricas gerais da aplicação. As métricas consideradas são: número de novos utilizadores registados (de acordo com o campo account_creation); número de voos (de acordo com o campo schedule_departure_date); número de passageiros; número de passageiros únicos; e número de reservas (de acordo com o campo begin_date). Caso a Query seja executada sem argumentos, apresentar os dados agregados por ano, para todos os anos que a aplicação tem registo. Caso a query seja executada com um argumento, year, apresentar os dados desse ano agregados por mês. Finalmente, caso a query seja executada com dois argumentos, year e month, apresentar os dados desse ano e mês agregados por dia. O output deverá ser ordenado de forma crescente consoante o ano/mês/dia.

Resposta: O programa irá usar o **Calendar_Almanac**, e, dependendo do número de argumentos presentes no comando, irá percorrer as HashTables presentes neste catálogo até alcançar ao nodo com o espaço de tempo desejado (para um comando com 1 argumento obterá a informação de todos os anos presentes no catálogo, com 2 argumentos obterá a informação de todos os meses presentes no catálogo de um dado ano e com 3 argumentos obterá a informação de todos os dias presentes no catálogo de um dado mês e ano).

6. Testes
7. Desempenho