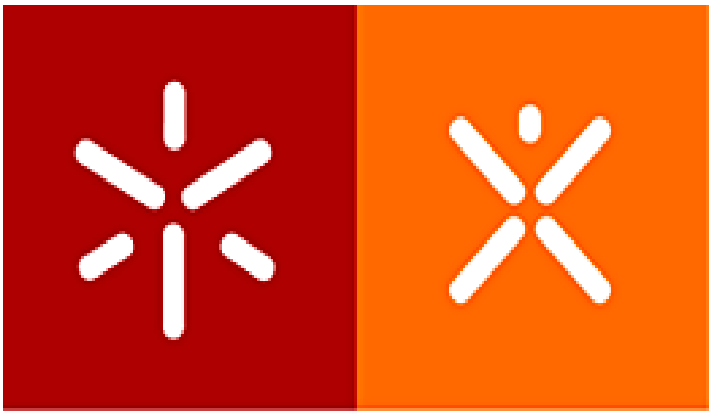


Universidade do Minho

Licenciatura em Engenharia Informática

Laboratórios de Informática III 2022/2023

Grupo 28

André Pereira (a 104275) Marco Gonçalves (a104614) Bruno Miguel (a94662)

Índice

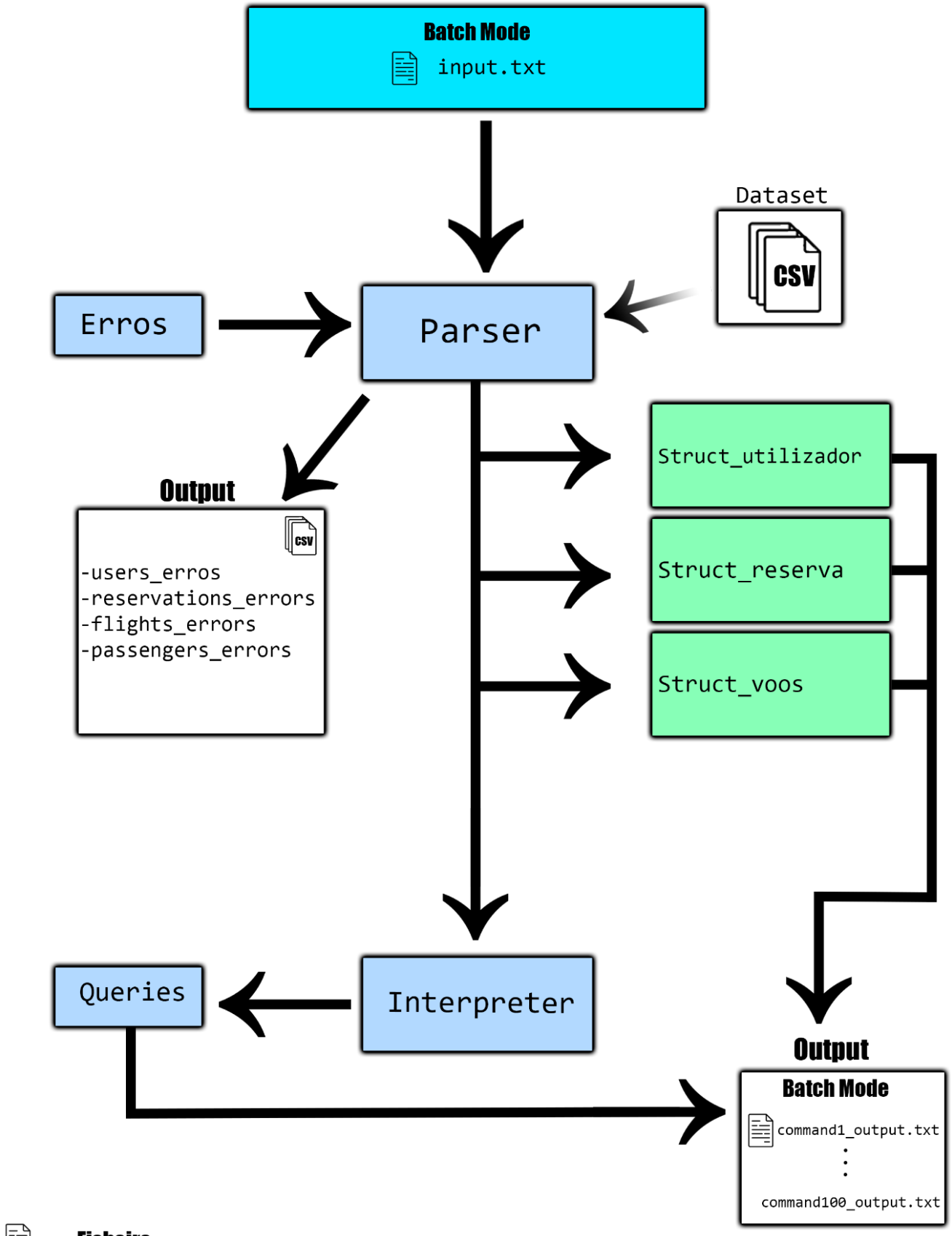
1- Introdução

1.1 Planeamento do Projeto

Nesta primeira fase do projeto fizemos um planeamento vago e simplificado , de forma a podermos focar-nos na implementação das funcionalidade de todos os módulos requisitados para um projeto que suporte as 6 queries que decidimos implementar. Como tal, também o resultado deste planeamento foi simples e ainda incompleto, demonstrado na representação gráfica abaixo **(1.2)**. As 6 queries já englobadas pelo nosso código são : 3,4,5,6,8 e 9.

Vale mencionar a falta de módulos que torna o projeto incompleto como uma separação entre módulos data e a falta de um módulo de output(neste momento repartido entre vários módulos) ,por exemplo, derivado do nosso foco na funcionalidade geral do programa para esta primeira fase.

1.2 Representação gráfica dos módulos



No Parser são identificados todos os erros através de funções no módulo “**erros**” e imprimidas nos ficheiros csv correspondentes; já com os dados filtrados estes passam por um novo processo de "parsing" antes de chamar o interpreter. O interpreter lê os inputs e chama ,através do módulo “**queries**”, uma função “**choose_querie**” que realiza e imprime os resultados com dependência nos módulos “struct” para manipular as estruturas de dados criadas.

2- Main

A nossa main é responsável por alocar o espaço inicial para todas as structs e matrizes dinâmicas que serão usadas ao longo do programa. Isto é algo que tencionamos modificar mais tarde, de forma a usufruir da modularidade.

Após a leitura dos argumentos e realizar todas as alocações de memória, é realizado um parsing genérico para todas as matrizes **(3)**. Depois disto, utilizamos um parsing específico**(4)** a cada struct, que nos garante boa performance na realização das queries.

De seguida é chamado o interpreter, que é responsável por separar os inputs e escolher a devida querie**(5)**.

Finalmente, libertamos todo o espaço aberto dinamicamente ao longo da execução do programa através de várias funções de “free”.

3-Parser e deteção de erros

É no parser que decidimos , para além de guardar todas as linhas de cada ficheiro csv do dataset na devida matriz, identificar e remover com antecedência todas as linhas incorretas ; para tal criamos um módulo “**erros**” que verifica toda e qualquer condição (função “**testa_linha**”) que invalide uma linha dependendo do conteúdo da mesma. Através de tentativa e erro conseguimos ,eventualmente, discernir todos os erros e facilitar o processo de parsing futuro **(4)**.

Como argumentos a funcao “**parser**” recebe o ficheiro a ler, a matriz onde deverá guardar as linhas, um “**select**” que serve para identificar o tipo de conteúdo (utilizadores,voos,...), o nome do ficheiro de erros onde imprimir os mesmos, uma

árvore onde insere todos os utilizadores inválidos, um vetor que acumula , através do id de voo, o número de passageiros em cada um deles,um vetor “**erros_voos**” que guarda todos os voos inválidos também através do id de voo, e finalmente um “**header**” que contém a linha inicial do ficheiro de erros a imprimir. (todas as impressões serão delegadas a um módulo output para a segunda fase)

No módulo “**parser**” incluímos ainda a função “**free_matriz**“ que será movida para um módulo de utilidades na segunda fase do projeto.

```
char** parser (FILE *parse,char ** total_dados,int select,char *ficheiro_erros,tree_erros* erros_utilizadores,int passageiros_voo [1000],int erros_voos[1000],char* header)
```

```
int testa_linha (char*linha,int ficheiro,tree_erros* erros_utilizadores,int erros_voos[1000])
{
    if(ficheiro == RESERVA)
    {
        if(testar_linha_reserva(linha,erros_utilizadores) == ERRO)
        {
            return ERRO;
        }
    }
    if(ficheiro == UTILIZADORES)
    {
        if(testar_linha_utilizador(linha) == ERRO)
        {
            return ERRO;
        }
    }
    if(ficheiro == VOO)
    {
        if(testar_linha_voo(linha) == ERRO)
        {
            return ERRO;
        }
    }
    if(ficheiro == PASSAGEIROS)
    {
        if(testar_linha_passageiros(linha,erros_utilizadores,erros_voos) == ERRO)
        {
            return ERRO;
        }
    }

    return CERTO;
}
```

4-Módulos

4.1 Struct_utilizador

-Contém todas as structs e tipos relativos aos utilizadores.

-Contém todas as funções relacionadas à gestão de utilizadores (Criação,inserção,etc...) inclusive para a gestão do tipo "**tree_erros**" onde são guardados os utilizadores invalidos (durante o parse generico **(3)**);

-Guarda os dados de cada um dos utilizadores (ainda em forma de string) num struct utilizador ("**cria_utilizador**","**guarda_utilizador_T**");

-É responsável pelo parser específico aos utilizadores, que guarda cada utilizador numa árvore binária ordenada alfabeticamente. ("**parser_utilizadores**","**inserir_utilizador**");

-Contém duas funções que realizam e imprimem a querie 9 ("**print_querie_9"** e "**print_querie_9F**" **(5)**);

-Por fim estão neste módulo funções específicas de "**free**" para libertar as suas árvores binárias ("**free_single_tree**","**free_erros**").

```
btree_utilizadores* parser_utilizadores(char** total_dados_utilizador,btree_utilizadores* arvore)
```

```
void print_querie_9 (btree_utilizadores* arvore,char* prefixo,FILE* ficheiro_output)
```

```
void print_querie_9F (btree_utilizadores* arvore,char* prefixo, FILE* ficheiro_output,int* contador)
```

4.2 Struct_reserva

-Contém todas as structs e tipos relativos às reservas.

-Contém todas as funções relacionadas à gestão de reservas (Criação,inserção,etc...);

-Guarda os dados de cada uma das reservas(ainda em forma de string) numa struct reserva ("**cria_reserva**","**guarda_reserva_T**");

-É responsável pelo parser específico às reservas, que guarda cada reserva numa Hash Table de árvores binárias ordenadas pelas datas de entrada(de forma crescente) ("**parser_reserva**","**inserir_reserva**");

-Para usufruir desta Hash Table temos uma função Hash que faz atoi do id de hotel. Desta forma conseguimos guardar BTrees com todas as informações sobre cada hotel de forma a garantir um acesso e procura rápida e eficaz nas queries pretendidas;

-Contém funções referentes à querie 3, 4 e 8 ("**Somar_ratings**" **(querie 3)**, "**print_querie_4**" , "**print_querie_4F**" , "**print_querie_8**" **(5)**);

-Por fim estão neste módulo funções específicas de “**free**” para libertar as suas árvores binárias (“**free_tree_reservas**” , ”**free_reserva, ”free_nodo”**).

```
reservas_arvore** parser_reserva (char** total_dados_reserva,reservas_arvore** hash_tree)
```

```
void print_nodo_q4F(reserva* nodo,FILE* ficheiro_output)
```

```
void print_querie_4 (reservas_arvore* arvore, FILE* ficheiro_output)
```

```
void print_nodo_q4(reserva* nodo,FILE* ficheiro_output)
```

```
void print_querie8(reservas_arvore* arvore,FILE* ficheiro_output,int data_inicio[3],int data_fim[3])
```

```
int hash_hotel (char* id_hotel)
{
    char* token;
    char* savetok;
    token = __strtok_r (id_hotel,"L",&savetok);
    token = __strtok_r (NULL,"\0",&savetok);
    int id = atoi (token);
    return id;
}
```

4.3 Struct_voos

-Contém todas as structs e tipos relativos aos voos.

-Contém todas as funções relacionadas à gestão de voos (Criação,inserção,etc...);

-Guarda os dados de cada um dos voos (ainda em forma de string) numa struct voo (“**cria_voo**”, “**guarda_voo_T**”);

-É responsável pelo parser específico aos voos, que guarda cada reserva numa Hash Table de “**aeroporto_arvore**(s)”(BTrees) ordenadas pelas datas de saída prevista (de forma crescente)(“**parser_voos**”,” **inserir_voo**”,);

-Para usufruir desta Hash Table temos uma função Hash que calcula uma hash de 0 a 999 dependendo da soma dos 3 caracteres que representam o aeroporto de origem. Desta forma conseguimos guardar BTrees com todas as informações sobre cada aeroporto de forma a garantir um acesso e procura rápida e eficaz nas queries pretendidas;

-Recebe como argumento a lista de passageiros por voo preenchida previamente no parser genérico **(3)**, podendo assim contabilizar o número de passageiros por cada ano (para usufruir na querie 6 dessa informação **(5)**; Para isso a struct "**AEROPORTO_ARVORE**" contém um valor para cada ano e um identificador do país de origem dos voos nesta árvore. Tivemos também em atenção a contabilização do número de passageiros nos aeroportos destino ("**contar_destination**"));

-Contém funções referentes à querie 5 e 6 ("**print_querie_5**","**print_querie_5F**","**print_querie_6**","**print_querie_6F**" **(5)**);

-Por último, estão neste módulo funções específicas de "**free**" para libertar a Hash de árvores ("**free_voos**","**free_aeroporto**""**free_voo**").

```c
struct AEROPORTO_ARVORE
{
    int vinte_um;
    int vinte_dois;
    int vinte_tres;
    char* pais;
    voo* Nodo;
    aeroporto_arvore* esquerda;
    aeroporto_arvore* direita;
};
```

```c
void print_querie5 (aeroporto_arvore* arvore_o,FILE* ficheiro_output,int data_inicio[6],int data_fim[6])
```

```c
void print_querie5F (aeroporto_arvore* arvore_o,FILE* ficheiro_output,int data_inicio[6],int data_fim[6],int* contador)
```

```c
void print_querie6_F(aeroporto_arvore** hash_tree,int ano, int N,FILE* ficheiro_output,int* headers)
```

```c
int hash_aeroporto(char* aeroporto)   //WARNING COLISÕES SE FIZEREM BLIND TESTS MAIS TARDE
{
    int i;
    int sum = 0;
    for(i = 0; aeroporto[i] != '\0';i++)
    {
        aeroporto[i] = toupper(aeroporto[i]);
    }

    sum = (aeroporto [0] *31 * 31) + (aeroporto[1] * 31) + aeroporto [2];
    sum = sum % HASH_AEROPORTO;
    return sum;
}
```

5- Interpreter e queries

5.1 Interpreter

No interpreter apenas é feita uma leitura das linhas no ficheiro de input seguida de uma separação dos seus dois campos: a querie a realizar em conjunto com o seu identificador ”F” ou falta dele e os detalhes relacionados a essa chamada que variam dependendo da querie. Após esse processo é chamada a função “**choose_queries**” que , com esses parâmetros e todos os dados necessários incluindo o nome do ficheiro onde será imprimido o resultado , identifica e calcula o resultado a querie pretendida.

5.2 Queries

Visto que apenas 6 das 10 queries estão funcionais neste momento, quando é chamada a função “**choose_queries**” para os valores 1,2,7 ou 10 é imprimido um ficheiro de output vazio.

5.2.1 Querie 3

Quando é chamada a querie 3 é identificado se existe um “F” ou não , se sim imprime-se imediatamente o “cabeçalho” pois este é sempre idêntico nesta querie e logo após é calculado o resultado através da função “**Somar_Ratings**”.

5.2.2 Querie 4

Quando é chamada a querie 4 é identificado se existe um “F” ou não , se sim é chamada uma variação da função que apenas diverge na altura de impressão mas necessita também de um contador para imprimir corretamente o “cabeçalho” das respostas.Como todos os hotéis foram guardados ordenadamente em árvore binárias esta querie é simplesmente uma impressão de árvore glorificada e garantidamente rápida e simples.O único aspeto que necessitou de trabalho extra foi o cálculo do preço total de cada reserva através de uma função “ **total_price**”;

5.2.3 Querie 5

Nesta querie começamos por calcular a hash do aeroporto correspondente aos detalhes na chamada da querie; depois guardamos as datas de início e fim em arrays para fácil comparação mais tarde.Mais uma vez é identificado se existe um “F” é chamada a função correspondente.Nas funções relativas a esta querie fazemos uma

procura pela árvore pelas datas que encaixam no “espectro” de datas pretendido e imprimimos os nodos necessários recursivamente garantindo a ordem correta e formato pretendido.

5.2.3 Querie 6

Na querie 6 guardamos o valor do ano e dos N aeroportos que teremos de imprimir. De seguida fazemos novamente a distinção do modificador “F” e chamamos a função correspondente. Já durante o cálculo do resultado percorremos a nossa hash de aeroportos guardando o valor de passageiros correspondente ao ano pretendido e num outro vetor de strings o identificador do País. Finalmente para imprimir o resultado usufruímos de uma função “**max_indi**” que retorna o índice do maior valor no vetor (alterado para 0 a cada ciclo) e imprimimos sequencialmente N vezes.

5.2.3 Querie 8

Começamos por recolher o id, descobrindo a hash de hotel, e por guardar as datas em vetores. À semelhança da querie 3 , caso seja identificado o modificador “F” imprimimos desde já o “cabeçalho”. Para calcular o custo total das noites de uma reserva usamos uma função que calcula a diferença de dias entre 2 datas (“**diferenca_dias**”) e simplesmente multiplicamos pelo “atoi” do custo por noite. O que tornou esta querie mais complexa foi a procura de todas as datas válidas na nossa árvore uma vez que exigiu a verificação de diversas condições possíveis que foram todas consideradas e implementadas na função “**somar_lucro**”.

5.2.3 Querie 9

Aqui chamamos imediatamente a função correspondente à falta de ou presença do modificador “F”. Nesta querie começamos por verificar se o nome na raiz da árvore contém ou não o prefixo com a função “**verificar_prefixo**” e ,dependendo do resultado desta função, imprimimos ou não e procuramos em um ou nos dois filhos da árvore por mais nomes que cumpram o requisito, garantindo que se encontram sempre todos os nomes que possam verificar a condição.

6- Conclusão

Estamos já bastante satisfeitos com certos aspetos do trabalho como a implementação de árvores binárias, a velocidade de execução e o espaço alocado;

ainda assim, tanto estes aspetos como alguns outros podemos e iremos melhorar para a segunda fase do projeto. Queremos simplificar certos aspetos do código e várias funções ,para além disso vamos usufruir mais eficientemente da modularidade e encapsulamento através de novos módulos. Entendemos também que certas funções contêm redundâncias e/ou inconsistências de sintaxe. Todas estas e quaisquer outras particularidades que achemos insatisfatórias serão trabalhadas.