

Universidade do Minho

Licenciatura em Engenharia Informática



Laboratórios de Informática III

Relatório 1ª Fase | Grupo 4

Diogo Ribeiro (A105995) Diogo Campos (A106920) Juliana Silva (105572)

Ano Letivo 2024/2025

ÍNDICE

	Pág.
1. INTRODUÇÃO.....	3
2. SISTEMA	4
2.1. Estrutura Aplicada.....	4
2.2. Funcionamento do projeto	4
2.3. Queries.....	5
2.3.1. Query 1.....	6
2.3.2. Query 2.....	6
2.3.3. Query 3.....	8
3. DISCUSSÃO	9
3.1 Análise do projeto	9
3.1.1 Estruturas de dados.....	9
3.1.2 Modularidade	9
3.1.3 Testes.....	9
4. CONCLUSÃO.....	11

1. INTRODUÇÃO

Este relatório apresenta informações referentes à 1ª Fase do Projeto da Unidade Curricular Laboratórios de Informática III da Universidade do Minho. Neste projeto, tivemos de utilizar estruturas de dados altamente eficientes para armazenar e manipular os ficheiros .csv fornecidos pela equipa docente. Estes ficheiros contêm dados de um programa de músicas e o nosso objetivo foi armazená-los e desenvolver métodos de pesquisa entre os mesmos. Além dos ficheiros, a equipa docente, instruiu-nos a implementar três queries, cada uma com um objetivo específico. Para tal, utilizamos estruturas de dados como *Hash Tables*, *Arrays*, Listas ligadas e Árvores binárias de pesquisa, escolhidas pela sua adequação às diferentes necessidades do projeto.

Dada a complexidade deste projeto, um objetivo importante desta 1ª fase foi a implementação rigorosa de modularidade e encapsulamento, facilitando futuras manutenções e alterações no código. Outro ponto crucial foi a implementação de *parsing* e a validação dos dados de entrada, essenciais para garantir a integridade e a precisão dos dados processados.

2. SISTEMA

2.1. Estrutura Aplicada

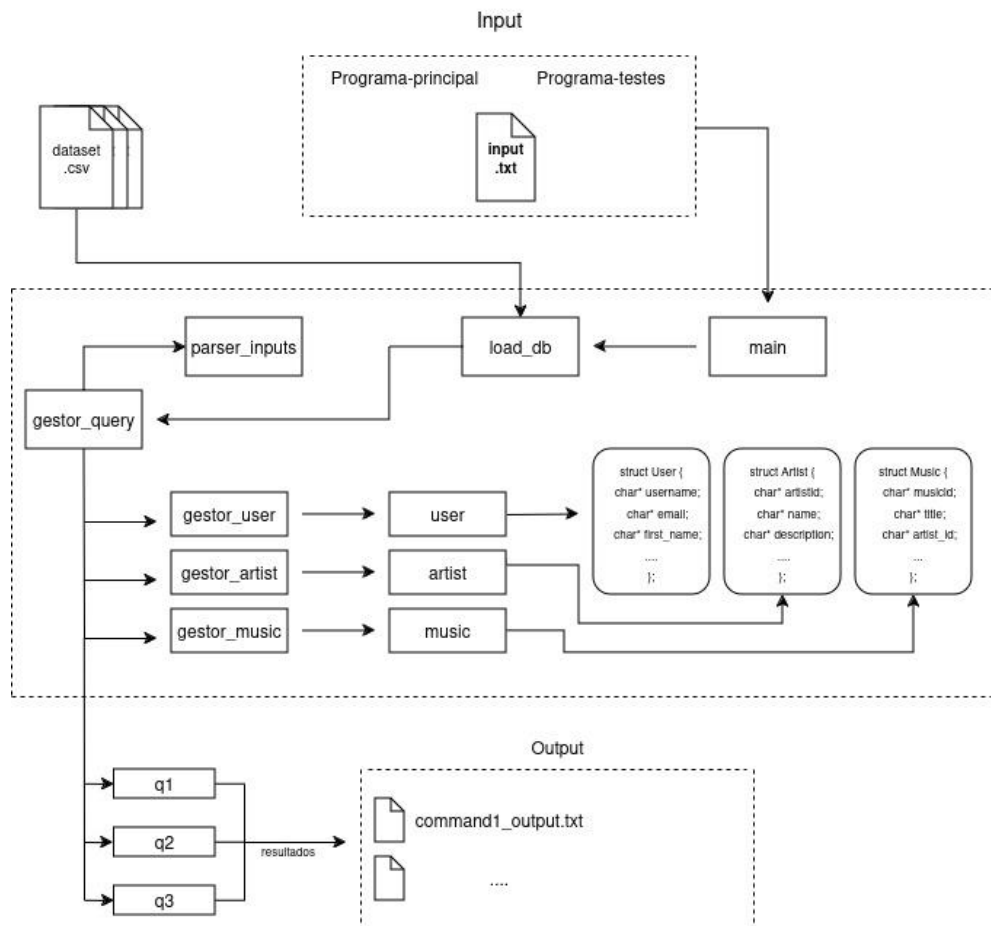


Figura 1 - Diagrama do programa em C

2.2. Funcionamento do projeto

A execução deste projeto começa na função `main` que, para além de ler argumentos de entrada, é também responsável por abrir os dados a partir de arquivos como o `dataset.csv` e o `input.txt`. Estes arquivos fornecem informações que o programa irá usar para preencher as suas estruturas de dados. A função `main` chama então a função `load_db` que, por sua vez, invoca as três funções principais para carregar dados específicos: `load_users`, `load_musics` e `load_artists`. Cada uma destas funções é dedicada a carregar os seus respetivos dados (utilizadores, músicas e artistas), inicializando os gestores de dados correspondentes para cada tipo de informação.

Estes gestores de dados (*gestorUser*, *gestorMusic* e *gestorArtist*) têm funções especializadas para manipular os dados armazenados. Todos estes gestores utilizam uma estrutura *GHashTable* para facilitar o acesso rápido aos dados.

O *gestorUser* inclui funções como *insereUser* e *procuraUser*, que permitem inserir novos utilizadores e procurar informações específicas através do *username*. A estrutura *User* inclui informações tais como, *username*, *email*, primeiro nome, último nome, data de nascimento, país, subscrição e gostos em músicas.

De uma forma muito semelhante, temos o *gestorArtist* usa funções como a *insereArtist* e a procura *Artist* que insere e procura um artista específico pelo *artist_id*. A estrutura *Artist* contém informações como o *id*, nome, descrição, *recipe_per_stream*, *id_constituent*, país e tipo.

Temos ainda o *gestorMusic* que possui as funções *insereMusic* e *procuraMusic* para inserir e localizar músicas através do *id* da música. *Music* é estruturada por informações como *id*, título, *artist_id*, duração, género, ano e *lyrics*.

Uma vez que estes dados são carregados e os gestores inicializados, o programa chama então a função *gestor_query* que processa o *input* seleccionando a respetiva query. A q1 recebe uma linha de comando referente a utilizadores. Já a q2 e a q3, lidam tanto com o gestor de artistas como o de músicas. Cada uma destas queries utiliza o *parser_inputs* que faz o tratamento do *input* consoante a query requerida.

Após a execução das queries, os *outputs* são gerados como arquivos de texto com o nome *commandX_output.txt* que registam os resultados das funções realizadas.

2.3. Queries

Neste projeto tivemos de realizar 3 queries cada uma com um objetivo diferente.

Inicialmente, os ficheiros com os *datasets* são abertos e a função *fgets* (com recurso a um ciclo *while*) é utilizada para percorrer todos os ficheiros linha a linha. Durante todo o ciclo, os campos de cada *user*, *artist* e *music* são inicializados a *null*, a função *strtok* separa esses mesmos campos e passa-os à função *userField*, *artistField* e *musicField*, respetivamente. Estas são responsáveis por inserir a informação em cada campo correspondente.

2.3.1. Query 1

Lista o resumo de um utilizador, consoante o identificador recebido por argumento.

A inserção dos *users* na *Hash Table* só é feita de acordo com 4 condições: o e-mail tem um formato válido, a subscrição é apenas “normal” ou “premium”, a data tem um formato válido e é anterior a 9 setembro de 2024 e todas as músicas no campo *liked_musics* existem. As funções responsáveis por estas verificações são, respetivamente: *emailValida*, *subValida*, *dataValida* e *listaValida*. Se todas estas funções retornarem um valor não negativo, o *user* é inserido na *Hash Table*, caso contrário a função erro cria e escreve no ficheiro a informação do user que contém campos inválidos.

Para responder ao que foi pedido pela Query 1, isto é, imprimir a informação relativamente a um determinado user num ficheiro, foram desenvolvidas as seguintes funções:

- *getUserTable* – *getter* para devolver a *Hash Table*
- *criaGestorUser* – aloca memória para o gestor e cria a *Hash Table*
- *freeGestorUser* – destrói e liberta a memória do gestor
- *insereUser* – insere um *user* na *Hash Table*. A função *g_hash_table_insert*¹ recebe a *Hash Table* como argumento, um *username* e insere-o em memória
- *procuraUser* – recebe como argumentos o gestor e o *username_id* e devolve, utilizando a função da biblioteca “Glib” *g_hash_table_insert*, o *user* correspondente (caso exista)

2.3.2. Query 2

Consiste em obter o top N de artistas com maior discografia dado ou não um país.

A inserção dos *artists* na *Hash Table* só é feita de acordo com 1 condição: se o *type* dos *artists* for individual, então o *id_constituent* não poderá ter elementos. A função responsável por esta validação é a *validaElementos*. Se ela for verdade, o *artist* é inserido na *Hash Table*, caso contrário a função erro cria e escreve no ficheiro a informação do *artist* que contém campos inválidos.

A Query 2 tem como objetivo imprimir a informação relativamente a um conjunto N de *artists* e para isso, foram feitas as seguintes funções:

- *getArtistHashTable* – *getter* para devolver a *Hash Table*
- *criaGestorArtist* – aloca memória para o gestor e cria a *Hash Table*
- *freeGestorArtist* – destrói e liberta a memória do gestor

1 Função Glib que insere uma nova chave e um valor numa GHashTable

- `insereArtist` – insere um *artist* na *Hash Table*. A função `g_hash_table_insert` recebe como argumento, um *artist_id* e insere-o na memória.
- `procuraArtist` – recebe como argumentos o gestor e o *artist_id* e devolve, utilizando a função da biblioteca “Glib” `g_hash_table_lookup`², o *artist_id* (caso exista).
- `acumulaDuracaoArtistas` – recebe como argumentos o gestor de artistas e o gestor das músicas e percorre todas as músicas, extraíndo cada uma e convertendo-a em segundos através de uma função auxiliar `durationToSeconds`. Identifica os artistas associados a cada música e soma todas as suas durações.
- `durationToSeconds` – recebe como argumento a duração e faz a validação do formato da duração. Se o formato for válido, ela transforma a duração em segundos para facilitar a soma.
- `filtraArtistasPorPais` – é do tipo `GList`³ e recebe a *Hash Table* de artistas e um país e com a ajuda da `g_hash_table_iter_init`⁴ e da `g_hash_table_iter_next`⁵, percorre a *Hash Table* permitindo assim consultar cada entrada da tabela com o país especificado. Depois, utiliza a `g_list_append`⁶ para facilitar a adição dos elementos que queremos no final da lista.
- `ordenaArtistasPorDuracao` – recebe a `GList` feita na função `filtraArtistasPorPais` e ordena-a por ordem decrescente utilizando a `g_list_sort`⁷, `GCompareFunc`⁸ e a função auxiliar `comparaArtistaDuration`.

² Função Glib que procura uma chave numa *GHashTable*

³ Estrutura de lista duplamente ligada fornecida pela Glib para armazenar sequências de elementos de qualquer tipo

⁴ Função Glib que inicializa o ponteiro para um *Hash Table* que permite percorrer todos os pares chave-valor armazenados na tabela

⁵ Função Glib que avança o ponteiro para o próximo valor-chave na *Hash Table* associada a ele

⁶ Função Glib usada para adicionar um elemento ao final de uma lista do tipo `GList`

⁷ Função Glib que ordena os elementos de uma `GList` por uma ordem específica

⁸ Especifica o tipo de uma função de comparação entre dois valores. A função deve retornar um inteiro negativo se o primeiro valor vier antes do segundo, 0 se eles forem iguais, e um inteiro positivo se o primeiro valor vier depois do segundo

- `resetArtistDurations` – recebe como argumento o gestor de artistas e redefine a duração dos artistas para zero para evitar que os valores das execuções anteriores alterem os resultados atuais.
- `imprimeTopNArtistas` – recebe como argumentos a `GList` de artistas, um valor `N` e a linha atual e devolve com a ajuda da função auxiliar `printArtist`, o top `N` de artistas pedido na query 2

2.3.3. Query 3

Consiste em verificar quais são os géneros de música mais populares numa determinada faixa etária.

Para resolver esta *query* foi criada uma árvore Binária de procura do *Glib* usando a `HashTable` de utilizadores já criada acima e que em cada `Node` guarda o valor da idade de 0 a 100. Ela utiliza a `HashTable` das músicas e a função `getGenre()` e guarda em cada node a contagem de cada um dos géneros.

Depois, passa para a *query* 3 onde se recebe os parâmetros de início e fim (intervalo de idades requeridas) do `gestor_parser` e manda estes para o `count_genres_in_range` junto com a árvore onde procede a soma dos géneros ao longo do intervalo. Estes são guardados na estrutura `GenreCount genres` que faz o *sorting* para ordenar de forma decrescente e mandando para o `write_genre_counts_to_file` para *printar* os géneros. Usa também a função `all_zeros` para verificar se todos valores da `GenreCount` são 0 procedendo à criação de um ficheiro vazio se assim o for.

- `write_genre_counts_to_file` - escreve os resultados da query 3 no ficheiro
- `count_genres_in_range` - faz o somatório dos géneros de músicas ao longo do intervalo de anos pedido
- `binarytree` - recebe a `Hash Table` de utilizadores e de músicas criando uma árvore binária de procura da *Glib* usando com valor de procura com as idades do intervalo de 0 a 100 e o número total de *likes* de cada um dos géneros para aquela faixa etária num *array* seguindo uma ordem para cada posição⁹
- `insertMusic`- calcula o índice consoante a ordem das músicas e incrementa essa posição do *array*

⁹ *Metal* ->0; *Pop* ->1; *Rock* -> 2; *Blues* ->3; *Hip Hop* ->4; *Classical* ->5; *Country* ->6; *Electronic* ->7; *Jazz* ->8; *Reggae* ->9

3. DISCUSSÃO

3.1 Análise do projeto

3.1.1 Estruturas de dados

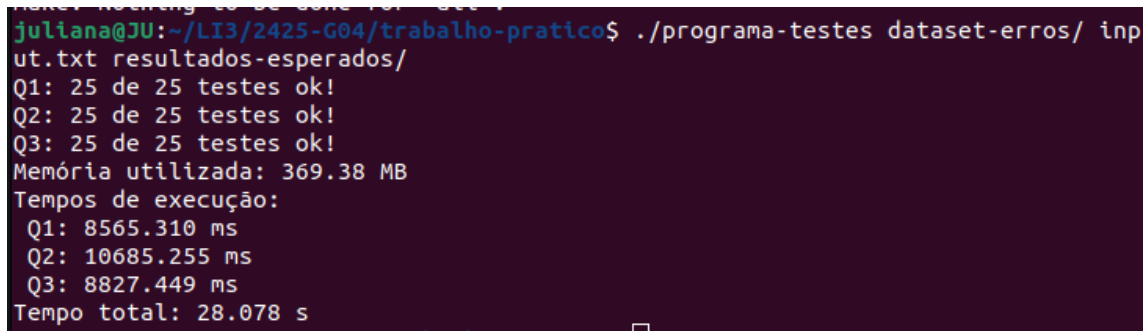
Neste projeto decidimos utilizar diferentes estruturas de dados na resolução das queries. Usamos *Hash Tables* para todas as queries uma vez que é uma estrutura de dados simples e eficiente. Na query 2, recorremos ainda ao uso de listas ligadas para armazenar os artistas com os países que desejávamos. Na última query decidimos utilizar árvores binárias de procura uma vez que, tínhamos de verificar se a idade do utilizador.

3.1.2 Modularidade

Quanto à modularidade, tentamos fazer o projeto com o máximo de módulos possíveis para que possíveis modificações de funções no futuro fossem facilitadas tanto para nós como para possíveis programadores que precisam de entender um código que não foi feito pelos mesmos.

3.1.3 Testes

Aqui temos os resultados de cada aluno que fez este projeto:



```
juliana@JU:~/LI3/2425-G04/trabalho-pratico$ ./programa-testes dataset-erros/ inp
ut.txt resultados-esperados/
Q1: 25 de 25 testes ok!
Q2: 25 de 25 testes ok!
Q3: 25 de 25 testes ok!
Memória utilizada: 369.38 MB
Tempos de execução:
  Q1: 8565.310 ms
  Q2: 10685.255 ms
  Q3: 8827.449 ms
Tempo total: 28.078 s
```

Figura 2- Computador da Juliana - CPU Intel Core i5 quad-core 6th geração

```

diogo@DI-PC:~/Downloads/2425-G04-main(3)/2425-G04-main/trabalho-pratico$ ./program
a-testes dataset-erros/ input.txt resultados-esperados/
Q1: 25 de 25 testes ok!
Q2: 25 de 25 testes ok!
Q3: 25 de 25 testes ok!
Memória utilizada: 369.25 MB
Tempos de execução:
Q1: 4326.295 ms
Q2: 5423.808 ms
Q3: 4375.261 ms
Tempo total: 14.125 s
diogo@DI-PC:~/Downloads/2425-G04-main(3)/2425-G04-main/trabalho-pratico$

```

Figura 3- Computador do Diogo R. - CPU Intel Core i5 octa-core 12th geração

```

Memória utilizada: 369.25 MB
Tempos de execução:
Q1: 5655.510 ms
Q2: 6983.194 ms
Q3: 5535.931 ms
Tempo total: 18.175 s

```

Figura 4 - Computador do Diogo C. - CPU AMD Ryzen 7 5800H octa-core

Como podemos verificar, a diferença de tempo da execução das queries é notória de computador para computador devido às especificações de cada um. Além disso, e falando mais concretamente do projeto, verificamos que a query 2 é a que demora mais tempo a ser executada e isso leva-nos a crer que talvez as suas funções não sejam tão eficientes quanto poderiam ser. As restantes queries, têm um tempo de execução parecido, mas temos de ter em conta que o tamanho do ficheiro de músicas e do ficheiro de utilizadores têm um tamanho bastante diferente. Assim concluímos que também poderíamos talvez deixar a query 1 um pouco mais rápida e eficiente.

Todos os testes foram executados sem problemas e obtiveram o resultado que era suposto então acreditamos que estejam corretas.

Temos ainda a informação da memória utilizada na execução deste programa que achamos que está satisfatória e corresponde às expectativas do que nos era proposto nesta fase.

4. CONCLUSÃO

Apesar dos imensos desafios enfrentados ao longo desta 1ª Fase, o projeto foi executado da melhor forma possível, com todas as tarefas devidamente concluídas.

Ao longo deste projeto, adquirimos conhecimentos valiosos sobre modularidade, encapsulamento, gestão de memória e diversas outras competências relacionadas à linguagem programação C. Essa experiência foi fundamental para consolidar a nossa compreensão e habilidades técnicas nesta linguagem.

Reconhecemos que ainda existem oportunidades para pequenas melhorias em termos de eficiência, e pretendemos abordá-las na próxima fase. De um modo geral, estamos muito satisfeitos com o resultado e acreditamos que ele cumpre com o solicitado, não só em relação às Queries, mas também em termos de modularidade, encapsulamento e eficiência.

Os maiores desafios que enfrentamos foram: o *parsing*, a criação dos programas de teste e a otimização da Query 3. O *parsing* foi especialmente desafiador, pois todos os campos precisavam de ser formatados corretamente para serem passados para as funções adequadas. Pequenos detalhes, como a não remoção do *char* ‘\n’, por exemplo, poderiam ser suficientes para causar bugs ou gerar resultados inesperados, e esses pequenos detalhes muitas vezes são difíceis de identificar.