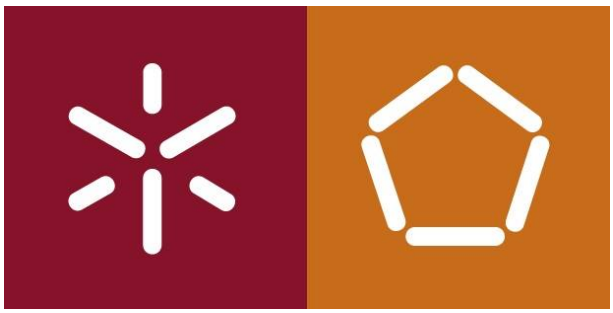

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Laboratórios de Informática III

Sistema de Gestão de Vendas

José Luís (a75481)
Margarida Faria (a71924)
Rui Azevedo (a80789)

12 de Abril de 2020

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

O presente relatório expõe todo o processo de concepção e desenvolvimento de uma aplicação de gestão de vendas no âmbito da cadeira do 2º ano, **Laboratórios de Informática III**.

O objectivo com o desenvolvimento da aplicação é aplicar estratégias e conceitos da área de **Engenharia de *Software*** e, para além disso, apresenta o desafio de processar grandes volumes de dados. A aplicação foi desenvolvida usando a linguagem de programação **C**.

Um dos principais desafios do trabalho é a escolha de estruturas de dados adequadas ao armazenamento e processamento dos dados fornecidos, sendo que é imperativo que o programa seja capaz de dar resposta a todas as perguntas solicitadas, no menor tempo possível.

Numa primeira fase, irão ser apresentados os requisitos do problema, bem como a concepção e desenho da resolução. De seguida, serão analisadas a complexidade e o tempo de resposta da aplicação. Por fim, irá ser dada uma análise crítica ao programa desenvolvido.

2 Análise e Especificação

Nesta secção irá ser apresentada, de uma forma breve, o objectivo do trabalho, os dados fornecidos ao programa e os requisitos para o seu desenvolvimento.

2.1 Descrição do Problema

O programa a desenvolver diz respeito a um **Sistema de Gestão de Vendas (SGV)** que contém clientes, produtos e vendas sendo que existe uma relação entre estas três entidades. É pretendido que se desenvolvam, obrigatoriamente, quatro módulos de dados. Os dois primeiros módulos, designados por catálogos, contém a lógica necessária ao armazenamento e acesso aos produtos e clientes, contendo apenas os códigos dos produtos e clientes. O terceiro módulo diz respeito há facturação de produtos e só deve referenciar apenas os códigos dos produtos e as respectivas facturações, não referenciando clientes. Por último, o quarto módulo desenvolvido diz respeito à gestão de filiais. O objectivo deste módulo é conter a relação entre clientes e respectivos produtos comprados.

2.2 Especificação de requisitos

2.2.1 Dados

O programa deverá ser capaz de ler três tipos de ficheiros que contêm uma estrutura bem definida. Os ficheiros **Clientes.txt** e **Produtos.txt** contêm, respetivamente, os códigos dos clientes e produtos que deverão ser carregados para o sistema. Os ficheiros **Vendas_1M.txt**, **Vendas_3M.txt** e **Vendas_5M.txt** contêm informações relativas às vendas que ocorreram no sistema e contêm, respetivamente, um milhão, três milhões e cinco milhões de vendas.

É de notar que todos os ficheiros podem conter informação inválida pelo que será necessário validá-la antes de carregar as estruturas de dados.

2.2.2 Modularidade

Uma das técnicas a adoptar no desenvolvimento do sistema é a programação modular. É pretendido que cada módulo seja independente e que cada um apenas contenha a funcionalidade necessária para trabalhar esse mesmo módulo. Cada um dos módulos contém uma *interface* que coloca à disposição as suas funcionalidades para permitir haver comunicação entre os mesmos. Esta técnica, muito importante para a **Engenharia de *Software*** faz com que o código esteja muito mais organizado uma vez que todas as componentes são independentes.

2.2.3 Encapsulamento

Outro aspecto importante no desenvolvimento deste tipo de sistemas é o encapsulamento dos dados. Os módulos de dados devem esconder o estado das suas estruturas de dados não permitindo acessos directos a essas mesmas estruturas. Para este fim, foram usados tipos opacos de dados, escondendo os detalhes da implementação das estruturas, onde o utilizador apenas necessita de conhecer o nome da estrutura de dados e os métodos necessários para a aceder, *i.e*, os designados *getters* e *setters*. Esta técnica tem elevada importância a nível de segurança de código pois é garantido que não há código externo ao módulo a alterar de uma maneira inesperada os dados das estruturas.

3 Concepção/Desenho da resolução

Nesta secção, irá ser apresentada a arquitectura do sistema de um módulo geral, bem como o desenho e análise de cada um dos módulos desenvolvidos.

3.1 Arquitectura do sistema

O sistema segue o *design pattern* ***Model-View-Controller (MVC)***. Este padrão permite dividir o código em três grandes elementos que comunicam entre si. O *model* contém a lógica de negócio contendo todas as estruturas de dados que armazenam os dados do sistema, bem como toda a lógica para manipular essas mesmas estruturas. A *view* diz respeito à apresentação visual dos dados, provenientes do *model*, para o utilizador. O *controller* é responsável por validar o *input* dado pelo utilizador, colectar os dados pretendidos e, posteriormente, passá-los à *view* para serem apresentados.

Na imagem abaixo, pode-se ver, de uma maneira geral, o fluxo do sistema.

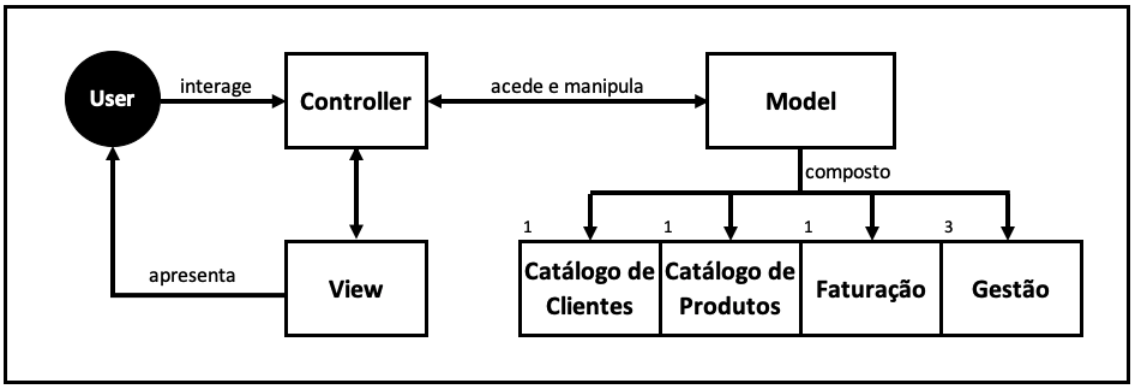

Figura 1: Arquitectura do sistema

3.2 Model

Esta secção irá apresentar as estruturas desenvolvidas para cada um dos módulos principais, *catalog.h*, *faturacao.h*, *gestao.h* e *interface.h*. É de salientar que, para além destes módulos de dados, foram criados também módulos auxiliares para que fosse possível dar uma resposta eficiente a todas as *queries*, mas que no entanto não serão vistos em detalhe uma vez que não fazem parte do foco principal do trabalho.

3.2.1 Catálogos

Tanto o catálogo de clientes como o catálogo de produtos, *clogClients.h* e *clogProducts.h*, respectivamente, foi desenvolvido com base num módulo genérico designado por *catalog.h*.

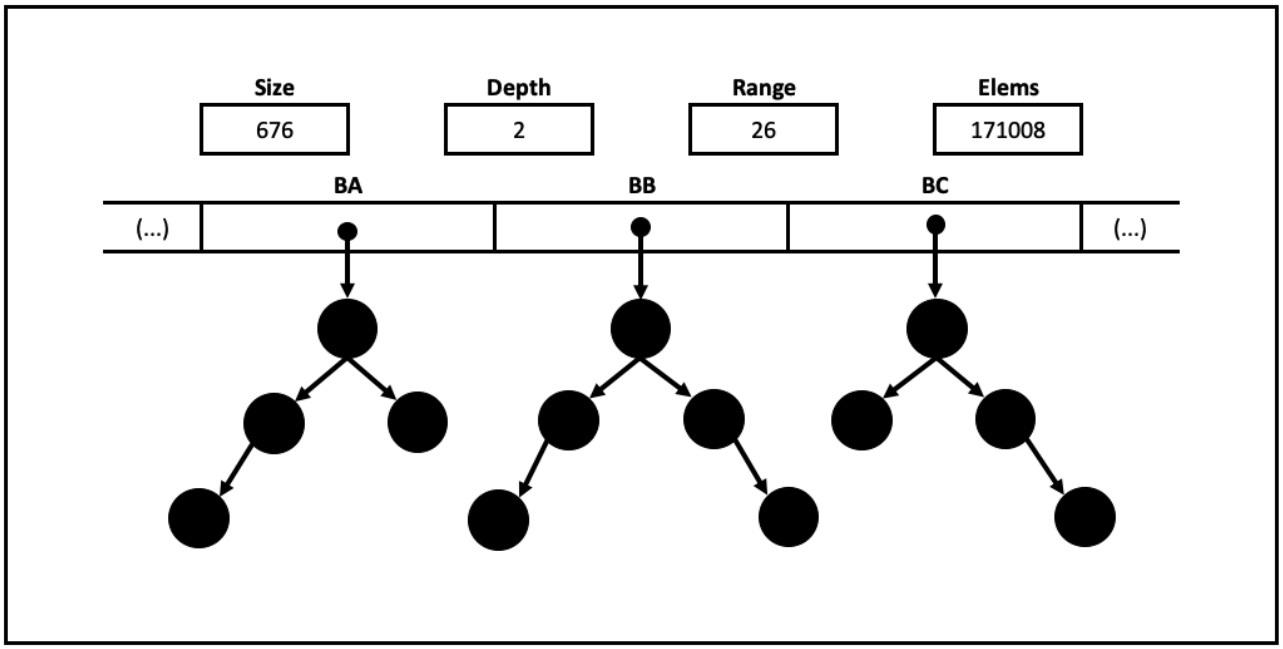

Figura 2: Estrutura do módulo de catálogos

Na figura acima, é apresentado o catálogo de produtos para exemplificar o comportamento do módulo *catalog.h*.

O catálogo é composto por um *array* de árvores balanceadas contêm os códigos dos produtos. Para além disso, é composto também pelo número de letras de um código (*depth*), a gama de valores de um código (*range*) que neste caso é 26 pois os códigos contêm todas as letras do alfabeto, pelo total de elementos (*elems)* e, por fim, o tamanho do *array* (*size*). As letras do código servem como elementos de procura no *array* havendo uma função de *hash* que

dado um código devolve o inteiro correspondente ao índice onde esse código, se existir, estará presente.

Algorithm 1 Função de *hash*

```
1: function HASH(code,depth,range)
2:     h ← 0
3:     for i ← 0 to depth − 1 do
4:         h+ = range ∗ (code[i] −′ A′)
5:     h+ = (code[i] −′ A′)
```

A decisão para esta estrutura, em alternativa ao índice ser apenas uma letra, foi reduzir o espaço de procura de um elemento no catálogo, fazendo então com que a procura seja mais eficiente.

A vantagem deste módulo genérico de catálogos é o facto de aceitar qualquer tipo de código composto por letras e dígitos. Por exemplo, se os códigos dos produtos fossem compostos apenas pelas três letras A, B e C e com quatro letras, iniciava-se o catálogo com $depth = 3$ e $range = 4$.

A estrutura de dados escolhida para guardar os produtos foi a árvore balanceada devido à sua complexidade de inserção e procura, uma vez que é, no pior dos casos, logarítmica. Para além disso, como se sabe à partida que o número de dígitos de um código é sempre quatro, pode-se afirmar que, no pior dos casos a complexidade de procura na árvore é de $T(N) = 1 + \log_2 9000 \simeq 14$. As árvores balanceadas são inicializadas com uma função de comparação de chaves para que as travessias nas árvores sejam feitas por ordem alfabética.

3.2.2 Facturação

O módulo de facturação, *faturacao.h*, relaciona os produtos às suas vendas mensais, sendo necessário ter a divisão dos diferentes valores, facturação e número de vendas, pelos diferentes modos de compra, normal ou em promoção. É de notar que este módulo deve referenciar também os produtos que não foram vendidos.

A estrutura de facturação é composta por dois *arrays* que, para cada mês do ano, contêm a informação global de facturação e de vendas para as *queries* que necessitem desta informação. Para além disto, é composta também por um catálogo de produtos em que a diferença para o catálogo anteriormente apresentado, os produtos contêm um valor correspondente. Este valor tem a informação sobre a facturação, quantidade comprada de um produto, número de linhas de venda desse produto e informação se um produto foi ou não comprado numa determinada filial (*flag*). Toda esta informação está organizada por filial e por mês.

Em alternativa a implementar uma matriz, decidiu-se defini-los como *arrays* de tamanho

$N = 12 * modos * filiais$. Existem, então, três *arrays* deste tipo, um para a facturação, outro para as vendas e outro para as linhas de venda.

A estrutura desenvolvida responde com eficiência às *queries* que necessitam de a usar. Caso seja preciso aceder à informação de um produto, o custo desta operação é, no pior caso, de $T(N) = 1 + \log_2 9000 \simeq 14$. O acesso à informação global de vendas e facturação, dado um mês, tem um custo constante de $T(N) = 1$.

Na imagem abaixo, pode-se observar a estrutura desenvolvida para representar as relações entre os produtos e as suas vendas. Para uma melhor ilustração, em vez de representar os *arrays* anteriormente mencionados, representou-se a matriz correspondente.

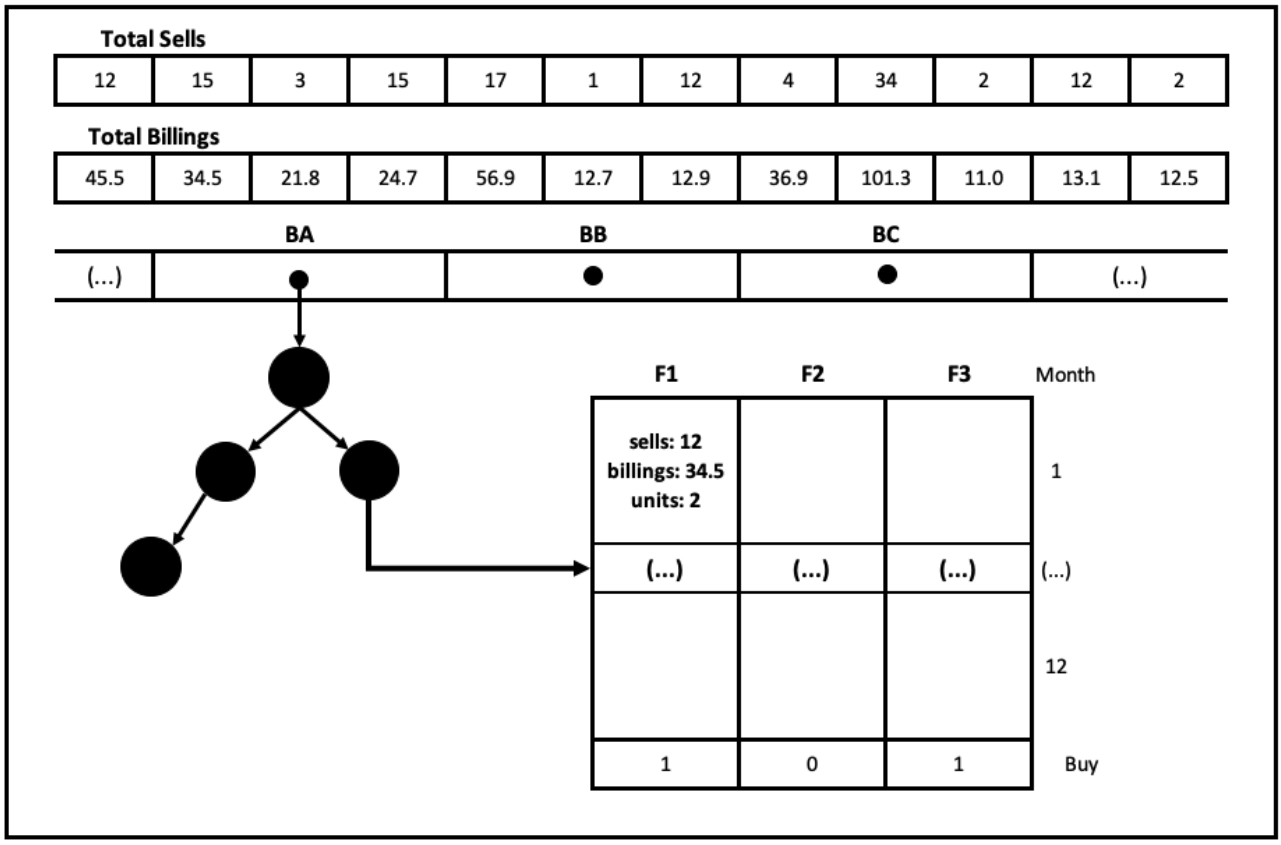

Figura 3: Estrutura do módulo de facturação

3.2.3 Gestão

O módulo de gestão de filial, *gestao.h*, é o módulo mais complexo a nível de estrutura de dados. É pretendido, neste módulo, fazer a correspondência entre clientes e produtos comprados pelos clientes.

Neste módulo, é usado, mais uma vez, a estrutura dos catálogos, mas desta vez para representar os clientes. Cada cliente contém uma estrutura composta por um *array* com o número total de produtos comprados em cada mês, e uma árvore balanceada. As chaves das árvores são o código do produto e os valores associados a cada chave são uma estrutura composta pelo número de vendas e total gasto nesse produto, dividido em modo normal e promoção, e ainda o número de compras desse produto organizado por mês.

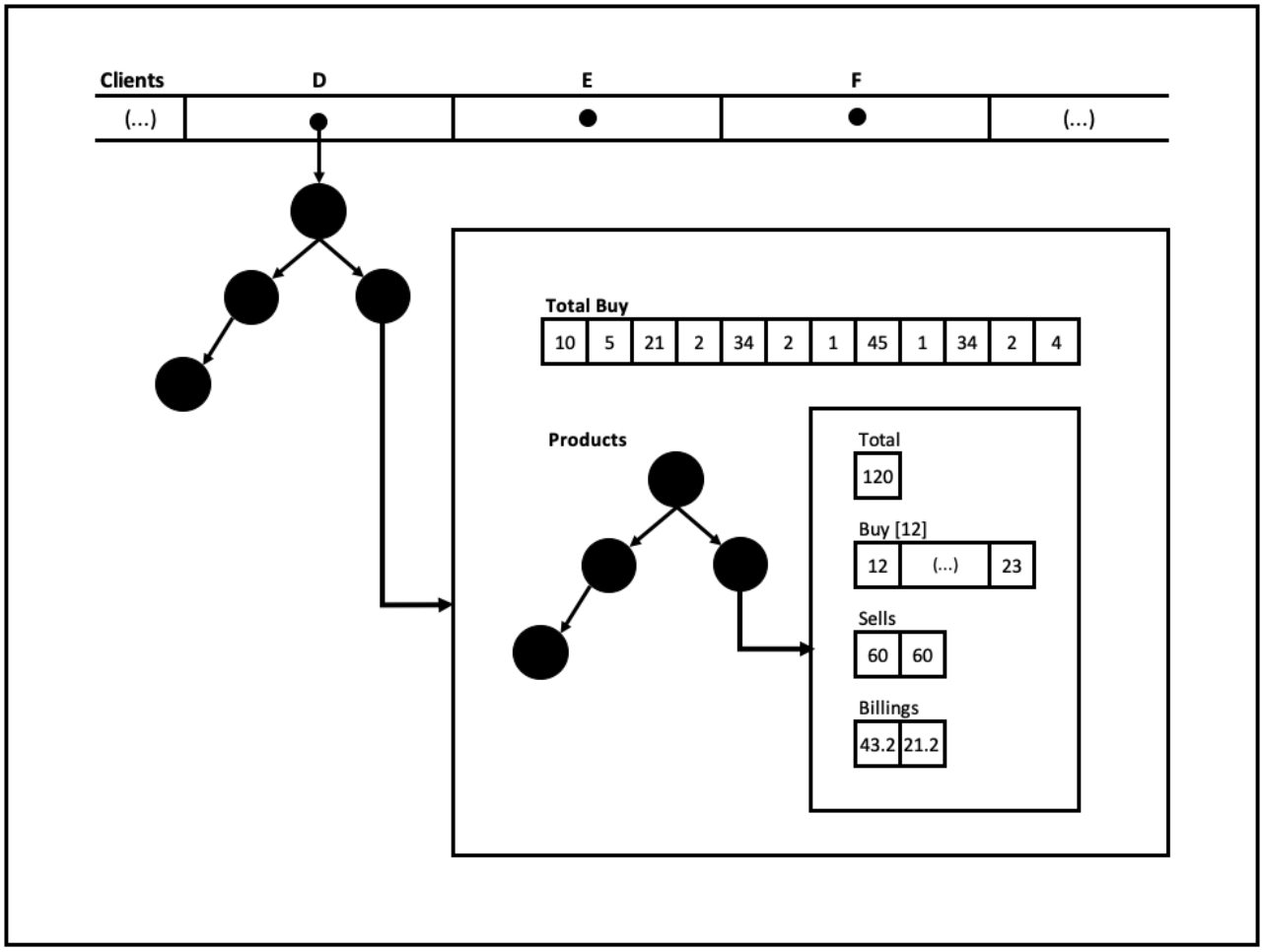

Figura 4: Estrutura do módulo de gestão de filial

Numa primeira implementação deste módulo, experimentou-se definir a árvore balanceada como um catálogo também mas o tempo de carregamento desta estrutura ficava exageradamente complexo, por isso, para um carregamento mais rápido e dado que o número de produtos nesta árvore tende a ser muito mais pequeno do que o número total de produtos no sistema, a implementação final contém uma simples árvore balanceada.

A complexidade para aceder à informação dos produtos de um determinado cliente é igual ao tempo de procura desse mesmo cliente, $T(N) = 1 + \log_2 9000 \simeq 14$. Uma vez obtida esta informação, os valores de compras totais podem ser acedidos em tempo constante. Quanto aos seus produtos, a procura de um produto é, neste caso de $T(N) = 1 + \log_2 m$, onde m é o número de produtos comprados por um cliente, que, nos piores dos casos, corresponde ao número total de produtos no sistema.

3.2.4 Collection

Embora este módulo não faça parte dos módulos de dados, é relevante pois é módulo que permite dar resposta à maior parte das *queries*.

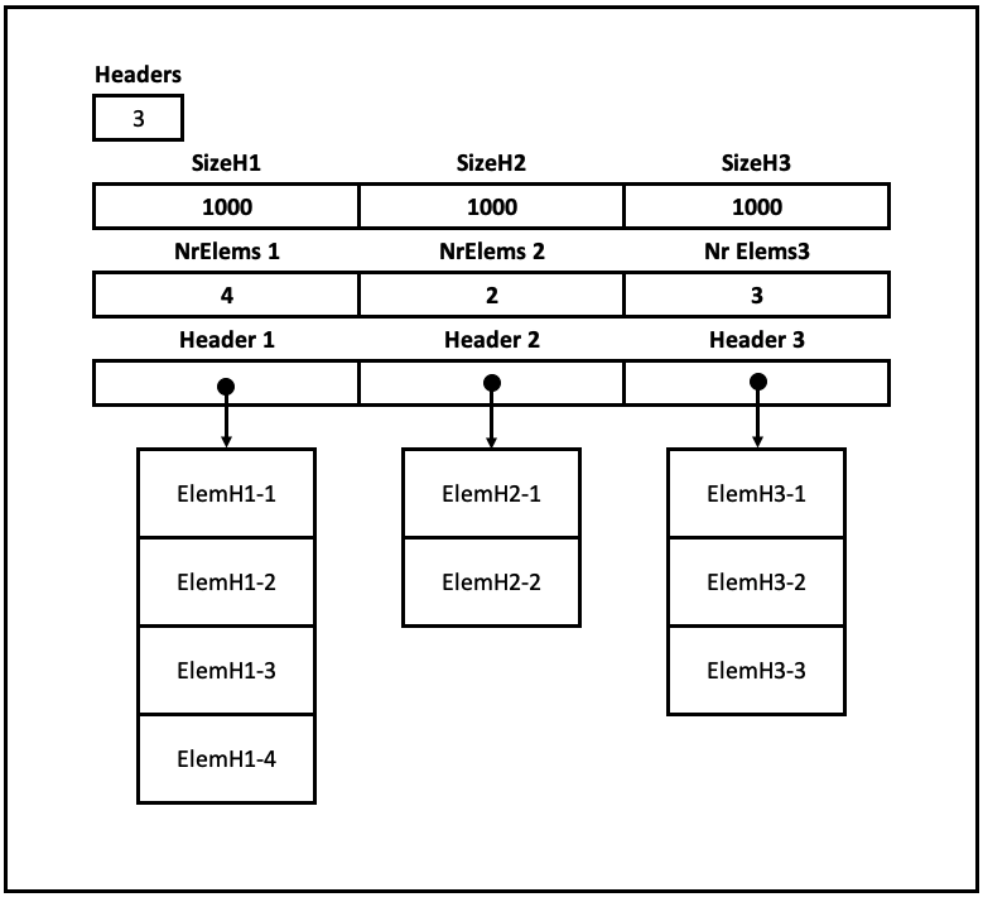

Figura 5: Estrutura do módulo *collection*

Uma vez que as respostas às questões colocadas tem todas um formato muito semelhante, este módulo visa cobrir essa semelhança e fazer com que a criação de respostas seja intuitiva

e já com um formato adequado para posterior impressão por parte da componente *view*.

Pode-se equiparar a estrutura da *collection* com a estrutura de uma tabela. Esta estrutura recebe como parâmetros de inicialização o número de cabeçalhos (*headers*) e o número inicial para alocação de memória. A estrutura interna deste módulo é composta por *array* designado de *content* com o tamanho igual ao número de cabeçalhos desejados. Cada posição desse *array* aponta para um *array* de *strings* onde está o contéudo de uma dada resposta. Para além disto, é também composto por mais dois *arrays*, cujo tamanho é também igual ao número de cabeçalhos e que têm a informação relativa ao número de elementos e tamanho alocado para cada um dos cabeçalhos.

O uso deste módulo é bastante intuitivo uma vez que para adicionar um elemento à estrutura invocar a função *addElem* e passar o cabeçalho ao qual se pretende adicionar o conteúdo, e o conteúdo em si.

Este módulo vai ser usado pelo módulo de paginação para obter um subconjunto da resposta, uma vez que a informação não deverá ser apresentada de um modo bruto.

3.2.5 Interface

Este módulo é o módulo principal da componente *model* pois é este que contém todas as estruturas de dados do sistema, anteriormente mencionadas, e é onde as respostas às questões estão definidas. Este módulo é então composto por:

- **Configuration**: estrutura de configuração inicial do sistema. Quando o programa arranca, esta estrutura é carregada com dados globais do sistema, nomeadamente, a profundidade do código de clientes e produtos, a gama de valores de um código, o número de filiais, o número de diferentes modos de pagamento e, por fim, o caminho, por omissão, para os ficheiros de *input* ao programa.
- **ClogProducts**: catálogo de produtos que contém todos os produtos válidos do sistema.
- **ClogClients**: catálogo de clientes que contém todos os clientes válidos do sistema.
- **Faturacao**: estrutura que relaciona os produtos às suas vendas.
- **Gestao[filiais]** : conjunto de estruturas de gestão de filial, uma para cada filial.
- **Stats**: estrutura que vai guardando dados estatísticos do programa, nomeadamente, o número de elementos válidos de cada tipo, o tempo de execução de cada *query* e o tempo de carregamento das estruturas. Para além disso, também mantém um ficheiro aberto onde vai escrevendo estes dados em tempo de execução.

3.3 View

A componente *view* desta arquitectura é implementada no módulo *view.h*. Este módulo apenas conhece o módulo de paginação para conseguir obter um subconjunto de uma determinada resposta. Esta componente apresenta algumas utilidades genéricas como por exemplo:

- mainMenu: imprime o menu principal.
- errorMessage: imprime uma mensagem de erro formatada
- clearScreen: limpa o ecrã para uma melhor visualização do conteúdo que está a ser apresentado.
- printQueryHeader: imprime o cabeçalho de uma *query*, *i.e*, o nome da *query* que está a ser executada.
- printRequest: imprime um pedido formatado de *input* ao utilizador.
- oneHeaderView: imprime uma tabela formatada com apenas um cabeçalho

Pode-se ver na imagem abaixo a resposta à *query* 1. O número definido de elementos por página foi de vinte. O utilizador pode navegar pelas páginas escolhendo o número da página pretendida. Caso prima a tecla 0, volta ao menu principal da aplicação.

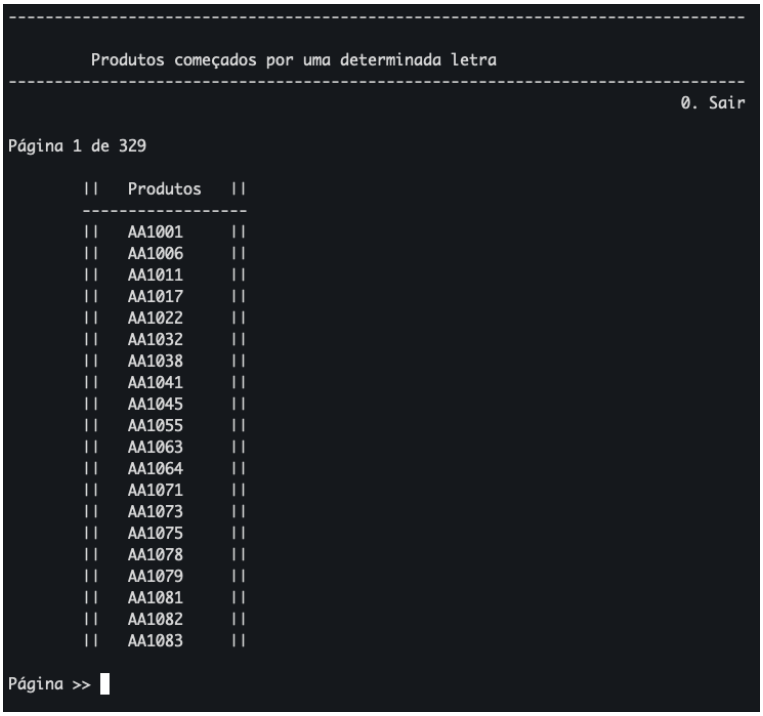

Figura 6: Vista da *query* 2

Para além destas funcionalidades, apresenta também outras que permitem apresentar a informação das diferentes *queries* com um formato específico.

3.4 Controller

Esta componente é implementada no módulo *controller.h* e é responsável por receber e validar o *input* do utilizador, comunicar com o *model* para obter a resposta à questão selecionada pelo mesmo e, passar a resposta obtida à *view* para ser apresentada.

Este módulo apenas contém uma função pública, designada por *controller*, usada apenas pela função *main* do programa, e contém treze funções privadas ao módulo, cada uma responsável por tratar uma resposta específica.

4 Testes e Desempenho

Nesta secção serão apresentados os tempos de carregamento das estruturas bem como o tempo que cada *query* demora a ser executada. É de notar que este tempo engloba, para além da parte algorítmica, o tempo de criar uma *collection* e adicionar os elementos de uma resposta a essa mesma colecção. Conta também com o tempo de escrever para ficheiro o resultado do tempo de execução de cada *query*.

Tabela 1: Tabela de desempenho das *queries*

Query	**Vendas_1M**	**Vendas_3M**	**Vendas_5M**
1	4.045569	13.766198	23.756104
2	0.002159	0.002158	0.002163
3	0.000015	0.000015	0.000014
4	0.031519	0.019110	0.019122
5	0.016590	0.016390	0.014512
6	0.033526	0.033697	0.034990
7	0.000017	0.000018	0.000016
8	0.000005	0.000005	0.000007
9	0.009816	0.014737	0.016108
10	0.000044	0.000088	0.000120
11	1.366366	3.433126	5.345670
12	0.000526	0.001046	0.001140

Pode-se observar que o que demora mais tempo é a leitura dos ficheiros e carregamento das estruturas como era de esperar. Isto deve-se ao facto de, tanto os produtos como os

clientes, terem que ser validados nas vendas antes da inserção na estrutura. Esta validação é o que está a fazer com que o carregamento demore tanto tempo, pois a maior parte do tempo de carregamento é gasto em validações. Como já foi referido anteriormente, por cada produto/cliente está-se a fazer, no pior dos casos, 14 acessos à memória, ou seja, no pior dos casos, imaginando que no ficheiro de vendas existem todos clientes e produtos, fazem-se no total $2 * 14 * 1000000$ acessos à memória num ficheiro com um milhão de vendas. O tempo de execução das diferentes *queries* nos diferentes ficheiros, não têm grande variação. Pode-se observar que muitas delas mantém praticamente o mesmo tempo de execução, com pequenas variações devido à quantidade de dados, mas não são variações significativas.

A *query* 11 é a *query* menos eficiente apresentada. Isto deve-se ao facto do seu fluxo interno. O processo da *query 11* é o seguinte:

- Fase 1: Criar uma árvore balanceada que irá colectar todos os produtos contidos no módulo gestão. A chave da árvore contém o código do produto e o valor correspondente contém uma estrutura, que irá sendo actualizada, com a informação relativa às vendas totais e filial a filial, assim como número total de clientes que o compraram e também filial a filial.

- Fase 2: Percorrer todos os clientes do módulo de gestão e, para cada cliente, percorrer todos os seus produtos para colectar a informação necessária. Isto tem um custo de $c = \sum_{n=1}^{N} np_i$, onde N representa o número de clientes e np_i representa o número de produtos comprados por um cliente. No fim da travessia, esta árvore contém toda a informação colectada mas não ordenada pelos produtos mais vendidos.

- Fase 3: Fazer uma travessia à árvore anterior, e, para cada elemento, inseri-lo numa *Heap*. O custo desta operação, dado que, no pior caso, o custo de inserção na *heap* é de $\log_2 N$, é de $TB * \log_2 N$, onde TB é o número total de produtos comprados. No fim desta operação a *heap* está organizada de maneira a devolver o *top* dos produtos mias vendidos.

- Fase 4: Por fim, são extraídos, *limit* elementos da *heap* e adicionados à estrutura *collection*.

Dado isto, é espectável que esta *query* seja a mais ineficiente devido a este processo todo que construção de uma resposta estruturada e ordenada.

5 Conclusão

De um modo geral, o grupo está satisfeito com o resultado final do trabalho. Todas as *queries* pedidas foram feitas e, os tempos de obtidos são satisfatórios, excepto o da *query 11* que é a *query* que tem o tempo de execução mais alto. As estruturas de dados usadas viram-se adequadas para a finalidade das respostas pedidas sendo prova disso o tempo de execução de cada uma delas.

No entanto, há aspectos que, neste momento, seriam feitos de maneira diferente. Um deles seria a implementação de um módulo de dados para árvores balanceadas. Embora que os módulos do *glib* tenham a complexidade desejada e, para além disso, já estarem feitos, não sendo preciso gastar tempo em criar um módulo novo, perdem no processo de *debug*. Uma vez que a *glib* usa tipos opacos de dados, não é possível, nem no *gdb* nem no *lldb* aceder ao estado interno da estrutura. Isto torna o processo de *debug* muito mais penoso, pois não se sabe o que se está a passar dentro do módulo, e, em muitos casos, os erros que o grupo teve, era bastante necessário aceder ao estado interno das árvores. Um dos exemplos, é um erro de libertação de memória que o programa tem ao destruir uma árvore. O erro obtido é um erro de libertação de memória de memória que não foi alocada, mas após uma profunda análise ao código, não foi possível perceber o problema e, se o módulo de árvores fosse criado pelo grupo, é provável que o erro conseguisse ser encontrado com facilidade.

Outro aspecto que podia ter sido melhorado, é relativo a *leaks* de memória. Uma técnica que o grupo deveria ter adoptado ao longo do processo de desenvolvimento do sistema era ir verificando se existem *leaks* de memória, através do programa *valgrind*, há medida que os diferentes módulos iam sendo implementados. Como isto não foi adoptado, uma vez que o sistema operativo usado para o desenvolvimento do projecto foi o *macOSX* e a ferramenta *valgrind* não tem o funcionamento esperado. Os *leaks* de memória foram posteriormente verificados, numa fase final do trabalho, numa máquina virtual com o sistema operativo *Ubuntu*. Foi possível ainda resolver alguns *leaks* de memória e, se o programa for iniciado e desligado de seguida, ou reiniciado com o conteúdo de outros ficheiros, verifica-se que não tem *leaks* de memória, após a análise do relatório do *valgrind*. Os *leaks* aparecem na execução das *queries*, nomeadamente, na destruição da estrutura de paginação. Após uma análise profunda ao código, o grupo não consegui identificar o estaria o erro.

Por fim, pode-se concluir que foi um trabalho bastante desafiante e, talvez, o mais complexo de toda a licenciatura. É um trabalho que obriga a pensar muito bem nas estruturas a escolher para o seu desenvolvimento pois têm um impacto crucial na *performance* do mesmo, e obriga também a adoptar métodos de programação que tornam o processo de desenvolvimento mais disciplinado e organizado. É indiscutível a importância que este trabalho tem na formação de um aluno de um curso de informática.

6 Anexos

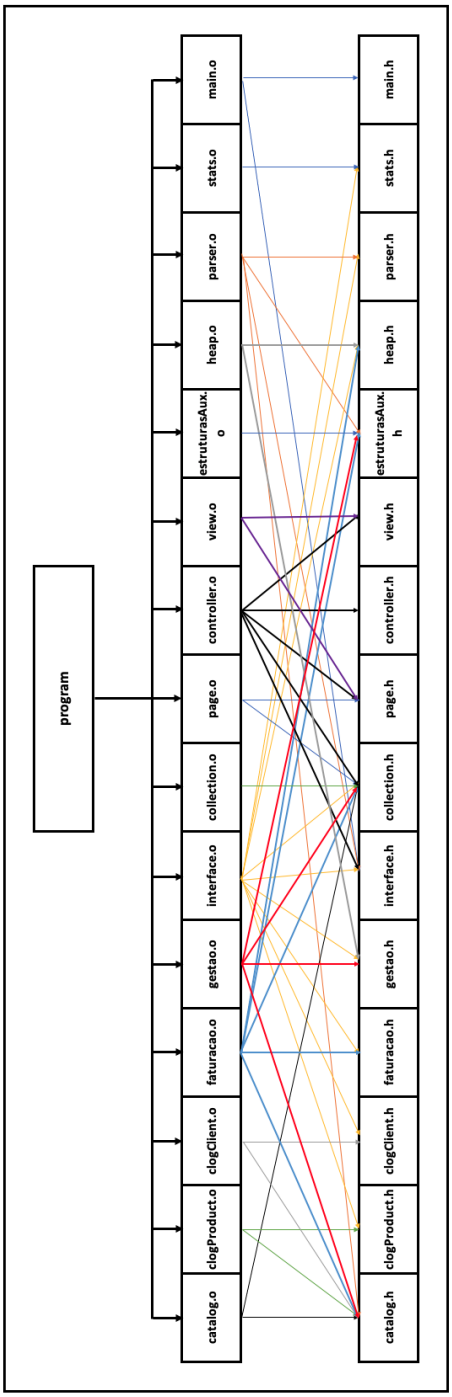

Figura 7: Grafo de dependências