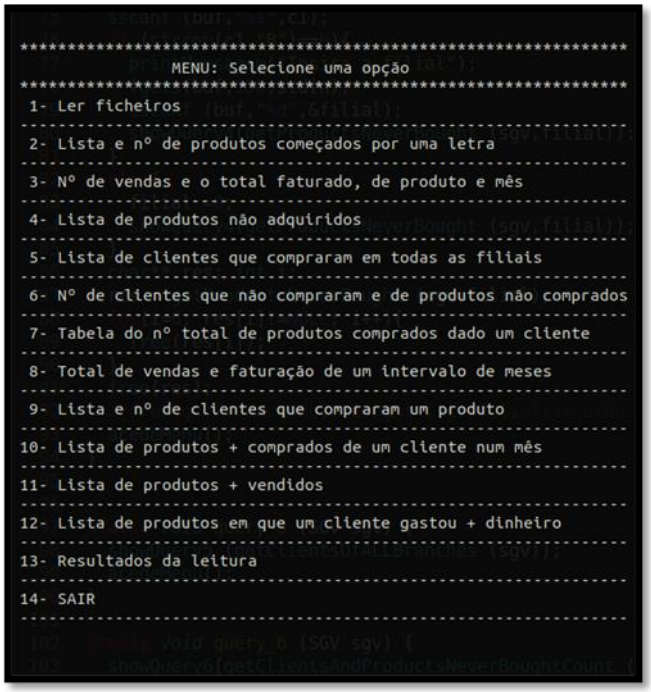
Relatório do Projeto de C
Sistemas de Gestão de vendas – SGV



(A89464) Raquel Costa

(A89518) Ema Dias

(A89537) Leonardo Marreiros

1 ÍNDICE

2 INTRODUÇÃO

No âmbito da disciplina LI3, desenvolvemos o projeto *Sistemas de Gestão de Vendas – SGV*, que nos desafiou a uma programação em grande escala, tendo em conta o grande volume de dados que nos foram apresentados nos ficheiros.

Para o correto tratamento destes dados, tivemos que nos debruçar nos novos princípios de programação indicados e imprescindíveis a este tipo de programação, já que esta tem uma complexidade estrutural bastante elevada. Estes princípios focaram principalmente na modularidade e no encapsulamento de dados, assim como nos levaram a uma grande preocupação na escolha das estruturas de dados a utilizar e da sua eficiência na resposta a cada uma das queries apresentadas.

Para a realização do projeto foram apresentados 3 ficheiros, ficheiros estes que constam numa função de leitura (query 1), e cujos dados são posteriormente guardados em módulos com as devidas estruturas definidas. Portanto, desenvolvemos um módulo para armazenar os códigos dos clientes (clientes.c), outro para os códigos dos produtos (produtos.c), assim como um módulo para a faturação (faturacao.c) e outro para a gestão da filial (filial.c).

Iremos abordar de seguida, em pormenor, aquilo que estruturamos em cada um destes módulos.

3 MODEL-VIEW-CONTROLLER

Sendo que um dos principais objetivos deste trabalho é aplicar de forma acertada os conceitos de Modularidade e Encapsulamento, isto foi uma constante preocupação do grupo. Portanto, ao longo do trabalho tentamos sempre dividir o código de modo a facilitar a deteção de eventuais erros, e tivemos sempre em conta a reutilização do mesmo de forma coerente e organizada. Além disso, tentamos sempre garantir a proteção e o acesso controlado aos dados, privatizando aqueles que são unicamente utilizados num módulo.

3.1 ARQUITETURA DO PROJETO

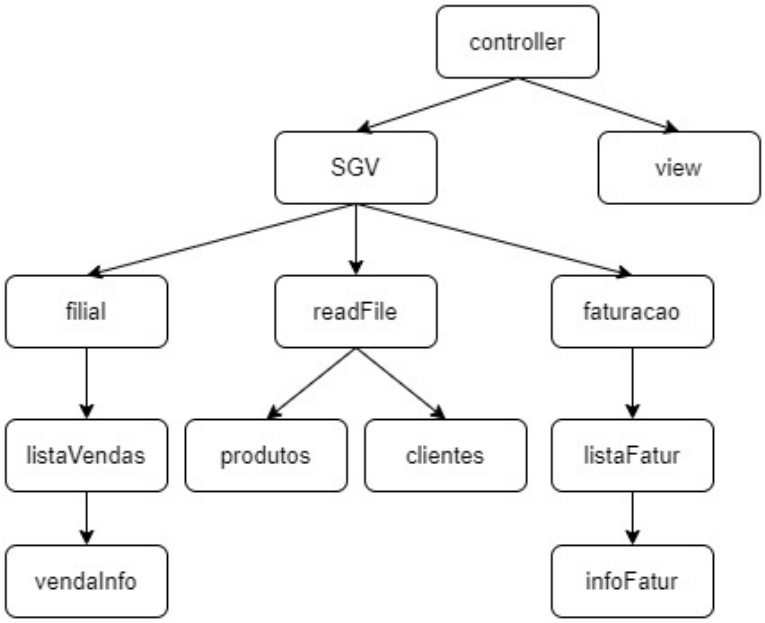


Figura 1-Estrutura do Projeto

- **Model:** SGV.c; readFile.c; faturacao.c ; Filial.c; ListaVendas.c ; ListaFatur.c; Produtos.c; Clientes.c; VendaInfo.c; InfoFatur.c;
- **View:** view.c (inclui o navegador)
- **Controller:** controller.c

Uma típica interação com o nosso programa caracteriza-se por: Primeiramente o utilizador faz um pedido de acordo com aquilo que lhe é apresentado pelo módulo "view.c", ou seja o menu que contém toda as queries. O "controller.c" responsabiliza-se por "recolher" o pedido do utilizador e atualizar os dados ou consulta-los nos módulos pertencentes ao Model. Após isto, o Model envia os dados atualizados ou aqueles que o utilizador pretende consultar de novo para o Controller, já este atualiza o View com os dados obtidos de modo a serem finalmente apresentados ao utilizador.

4 MÓDULOS E API

4.1 READFILE

O objetivo deste módulo é fazer uma leitura dos ficheiros escolhidos pelo utilizador, sejam eles os "default" (Produtos.txt, Clientes.txt, Vendas_1M.txt) ou outros. Este módulo desempenha um papel fulcral no projeto, uma vez que é aqui onde as estruturas responsáveis pela faturação global, pela gestão de vendas em cada filial e pelos catálogos de Produtos e Clientes são preenchidas. Posteriormente, é com o auxilio destas que conseguimos responder às queries.

É importante referir que fazemos "include" de outro módulo (VendasValidas.c) que faz a procura binária nos catálogos de Produtos e Clientes, de modo a validar os códigos presentes no ficheiro das vendas. Além disso, também verifica os outros campos de cada venda.

4.2 CATÁLOGOS

Nestes módulos (produtos.c e clientes.c) pretendíamos uma estrutura capaz de armazenar as informações de todos os produtos e clientes. Para o efeito decidimos utilizar um array de procura binária alocado, e organizado por ordem alfabética. Ora, para isto necessitamos de criar funções de inserção, organização e procura.

4.2.1 Limitações / Pontos Positivos

Poderíamos ter utilizado outras estruturas ao invés do array, no entanto a sua facilidade de utilização e acesso e o facto de ser uma forma conveniente de armazenar dados do mesmo tipo e tamanho acabou por justificar a nossa escolha. Apesar disso, reconhecemos que ao definir os catálogos com uma dimensão fixa (Ex: lista[200000]) não foi a melhor decisão em termos de gestão de memória.

4.3 FILIAL

Neste módulo encontra-se a estrutura de dados responsável pelas vendas de uma Filial. A decisão do grupo foi optar por uma estrutura AVL, uma vez que é um conceito com o qual nós estamos mais familiarizados e, em termos de procura e gestão de dados corresponde às nossas necessidades, para tal recorremos ao auxílio da livraria GLib . Em primeiro lugar, criamos o módulo "VendasInfo.c" que contém a "struct Info_vendas" responsável pelo armazenamento de cada "campo"/informação de uma só linha de venda. Após isso, decidimos criar noutro módulo (ListaVendas.c) uma lista ligada das várias "Info_vendas", ou seja, todas as compras que um cliente fez numa determinada filial ficam ligadas. Deste modo, é mais fácil termos acesso às compras de cada cliente e às respetivas informações.

Resumindo, a key dos nodes da GTree é o código de cada cliente e o value é uma lista ligada de uma struct que contém as informações sobre cada compra do cliente na filial em questão.

Neste módulo definimos funções do tipo "get", para conseguirmos aceder a certos campos da "struct info_venda" fora deste módulo, além disso existem outras funções que percorrem a árvore toda de modo a devolverem informações necessárias à resolução das queries.

4.3.1 Limitações / Pontos Positivos

Algo que o nosso grupo teve em conta ao construir as estruturas de dados referentes a este módulo foi a sua eficiência a dar resposta às queries. Há queries que pedem a lista de clientes que compraram um certo produto, e como as keys da nossa árvore são os códigos de cada cliente temos de percorrer a árvore toda para obter essa lista o que poderá trazer custos relativamente ao tempo de execução.

Uma das alternativas que o nosso grupo pensou para evitar isto foi criar duas árvores para a Filial .Uma tal como a que nós temos e outra que tem como key dos nodes o código de cada produto. Deste modo, para obter a lista de clientes que compraram um determinado produto bastaria percorrer a "árvore dos produtos" até encontrar o produto em causa, o que teoricamente iria custar menos em tempo mas ocupa mais memória, o que poderá ser uma desvantagem. Sendo assim, podemos dizer que o que está em causa é um "trade-off" entre tempo de execução e consumo de memória. Nós acabamos por optar pelo uso de uma só árvore que tem como key o código do cliente,

uma vez que na prática a sua travessia não demonstra um impacto considerável no tempo de execução. Acabamos por valorizar a "poupança" no consumo de memória, que consideramos que seja mais importante para o nosso programa.

4.4 FATURAÇÃO

Sobre a estrutura utilizada para gerir a Faturação Global de cada produto decidimos usar algo parecido com o que fizemos no módulo da Filial.

Portanto, usamos uma estrutura AVL da biblioteca do GLib com o auxílio das suas funções. Tal como pedido no enunciado este módulo não contém qualquer referência a clientes, mas é capaz de distinguir os valores obtidos em cada filial e consegue responder a questões relacionadas com as vendas mensais de um produto, tanto em modo Normal como em Promoção, guardando para cada um o número de vendas e a faturação total . Tendo em conta isto, construímos uma "struct Info_fatur" no módulo "InfoFatur.c" que guarda todas estas informações. De seguida no módulo "ListaFatur.c" definimos uma lista ligada tal como fizemos no módulo Filial. A diferença é que aqui quando encontramos uma venda de um produto que foi feita na mesma filial e no mesmo mês de uma outra venda que já guardamos, apenas adicionamos a faturação dessa venda à "Info_fatur" da venda já existente e incrementamos o seu número de vendas total e dependendo se esta venda foi feita em modo N ou P incrementamos o respetivo nr_vendas desse modo.

Basicamente, cada produto tem a sua lista ligada com as informações da faturação, quantidade vendida e número de vendas (tanto em modo N como P) por cada mês e filial.

Decidimos usar como key dos nodes da nossa AVL o código de cada produto existente no catálogo de produtos, e como value as "ListasFatur's", mas inicialmente estas encontram-se com todos os campos a 0. À medida que fomos lendo o ficheiro das vendas fomos construindo as Info_fatur's e adicionando-as às ListaFatur's de cada produto. Sendo assim, os produtos cujas listas têm tudo a 0 , são produtos que não foram comprados.

4.4.1 Limitações / Pontos Positivos

Escolhemos a implementação desta ideia uma vez que para responder às queries pedidas achamos que é mais fácil ir buscar certos campos à struct, como por exemplo, se queremos saber a faturação do modo P num certo mês e filial basta ir buscar à lista do produto em causa, sem ter de percorrer a lista até ao fim. Mas também reconhecemos que se precisássemos de saber em concreto o preço de cada compra feita, com este módulo não iríamos conseguir saber isso, iríamos ter de recorrer ao módulo da Filial, o que provavelmente poderia demorar mais tempo. E tal como esta questão, podem surgir outras que, pelo facto da nossa "struct" não ter um caráter tão específico em relação às compras, não consiga responder.

Referimos portanto mais uma vez que nós ao construir estas estruturas tivemos em conta que estávamos a tratar de uma "faturação global" para cada produto, não especificando então as informações de cada compra pois temos acesso a isso através da Filial, e além disso, tivemos sempre em mente a eficiência na resposta a cada query.

Também queremos acrescentar que, provavelmente também poderíamos ter armazenado menos campos na struct "Info_fatur", ou seja, nós, por exemplo, temos 3 campos para a faturação (um para a faturação em modo N, outro para o modo P e, por fim, um para a faturação global) e, visto isto, bastaria apenas armazenar dois campos, uma vez que o global é a junção destes dois. Desta maneira, não estaríamos a consumir memória inutilmente com uma informação que poderíamos obter através do uso dos outros campos da "struct".

4.5 CONTROLLER

O nosso controller contém a main do programa, é neste módulo que podemos aceder à opção escolhida pelo utilizador, com o objetivo de restribuir pelas funções declaradas e definidas no SGV.h e .c . Deste modo, temos um *do while* que permite através de um switch, com o comando selecionado e obtido , invocar as funções (*query_*) que estão definidas para obter os restantes inputs necessários para as queries (utilizamos fgets e sscanf), e por sua vez chamar as funções do módulo SGV para obter a respostas.

Além disso, invocamos , no final de cada query, as funções definidas na view (*showquery*) ,para apresentar os resultados , e de seguida permitimos a opção de voltar ao menu.

4.6 SGV

O módulo SGV é aquele que tem acesso a todas as estruturas deste trabalho, portanto apresenta como struct uma fusão de todas as outras. Deste modo, contém a struct Clientes, a Produtos, 3 Filial, Faturação e Ficheiro, este último para que seja possível armazenar a informação que se obtém aquando da leitura.

Para a correta utilização desta estrutura SGV, definimos uma função que a inicializa (*init_sgv*) e uma função que liberta a memória da mesma (*destroySGV*).

Além disso, é este módulo que contém as queries definidas. Estas utilizam, como auxiliares, funções dos outros módulos e posteriormente, as queries são chamadas no controller, de forma a respeitar a estrutura MVC.

Em suma, é neste módulo que é possível agregar todas as estruturas, de maneira a conciliar todos os módulos e, portanto, ser capaz de responder às perguntas que o utilizador pretende.

4.7 VIEW (NAVEGADOR)

Este módulo corresponde à apresentação dos dados, de acordo com a arquitetura MVC. É o módulo responsável por qualquer output, texto, tabelas, etc... ou seja, onde os dados que o utilizador pede são exibidos. Este módulo é independente e não tem conhecimento de qualquer informação do modelo ou controlador, apenas se dedica a criar estruturas genéricas de apresentação as quais funcionariam para qualquer outro trabalho, dado o input correto.

Assim sendo, desenvolvemos as funções "showQuery" para apresentar arrays de strings, inteiros, e tabelas. Para o caso específico das listas de strings criamos a função "listDivider" que apresenta estas listas de forma organizada, dividida e navegável. Alguns dos pontos positivos desta função são as funcionalidades de escolher página e procura por elemento. Além destas

funções, o view inclui também os menus e outras interfaces de apresentação, assim como funções que validam o input do utilizador.

Recapitulando, o módulo View reflete o estado do modelo, e muda com ele.

4.7.1 Limitações / Pontos Positivos

Podíamos ter feito uma função mais global para as funções que têm o mesmo output, em particular as funções que utilizam a função "listDivider" (queries 2,4,5 e 12).

Idealmente o navegador poderia receber como input o número de elementos por página para, desta forma, decidir acerca deste assunto tendo em conta a dimensão da lista, ou o resultado pretendido, ou então, poderia ser o utilizador a decidir sobre a dimensão de cada página.

É de valorizar o facto de incluirmos casos particulares para clientes e/ou produtos inexistentes assim como casos de listas vazias. Além disso, as funções de validação de inputs aliados a uma cuidada apresentação estética, tornam o trabalho organizado e fácil de navegar.

5 TESTES DE DESEMPENHO

5.1 GLOBAL

Ao longo do projeto, fomos utilizando várias ferramentas para testar a eficiência do nosso código e a gestão de memória. Foi o caso do Valgrind, que usamos para verificar a existência de memory leaks e da biblioteca time.h para calcular os tempos de execução com a qual efetuamos vários testes de desempenho das várias queries com diferente número de vendas.

5.2 POR QUERY

	VENDAS_1M	VENDAS_3M	VENDAS_5M	INPUT
QUERY 1	7.735307	31.632494	79.444721	-
QUERY 2	0.009401	0.004729	0.00906	(A)
QUERY 3	0.0001173	0.000084	0.000137	(AF1184, 1)
QUERY 4 (A)	0.080802	0.013473	0.038617	-
QUERY 4 (B)	1.460728	1.607361	0.201097	2
QUERY 5	0.027568	0.023585	0.043214	-
QUERY 6	0.092389	0.072136	0.045571	-
QUERY 7	0.000160	0.000138	0.000403	(E4717)
QUERY 8	0.250400	0.284301	1.309241	(1, 4)
QUERY 9	0.062053	0.036584	0.637705	(AF1184, 2)
QUERY 10	0.000131	0.000116	0.000295	(E4717, 4)
QUERY 11	2.482037	1.859707	14.021737	(10)
QUERY 12	0.000223	0.000157	0.000438	(E4717,60)
QUERY 13	0.000061	0.000039	0.000063	-

Tabela 1- Tempo médio em segundos do resultado da execução de cada query para cada ficheiro de vendas

5.3 Reflexão sobre os Resultados

O tempo de execução de algumas queries poderia ter sido melhorado.

6 Conclusão

6.1 Reflexão Crítica

- Binary Trees vs Hash Tables

Apesar de inicialmente termos optado por escolher as AVL para realizar o tratamento de dados, já que consideramos que seria a melhor estrutura para as necessidades do programa, sentimos no final, já com todas as queries realizadas, que o tempo de execução poderia ter sido mais otimizado. Portanto, decidimos estudar as outras opções de estruturas que poderiam ter sido utilizadas.

Com isto, chegamos à conclusão que se tivéssemos optado pelas Hash Tables teríamos tipo uma maior facilidade em acessos, principalmente quando se trata de programação em grande escala. As funções de hash conseguem mapear com uma melhor técnica de pesquisa, sendo que, maioritariamente, o pior caso é O(1) e além disso, tem o melhor tratamento de colisões.

Assim, percebemos que as Binary Trees, apesar de terem uma complexidade uniforme, têm de tempo referente a lookups e travessias de O(log n) e a comparações de $O(\log_2 n)$, ao contrário das Hash Tables que o tempo de acesso depende da função de hash escolhida, mas geralmente é de complexidade O(1), exceto quando é preciso aumentar o espaço que é O(n).

- Reutilização de Código

Ao longo do desenvolvimento do trabalho, tivemos o cuidado de criar código reutilizável, de maneira a tornar o projeto mais simples e planeado, e com menos casos particulares. A título de exemplo, temos a struct ClientesProd que foi utilizada em várias queries, pois conseguimos construí-la de forma a agregar as componentes necessárias para cada uma delas, e apenas inicializar aquelas que seriam utilizadas. Assim como, a função *auxForEach*.

No entanto, temos consciência, que poderíamos, por vezes, ter tido mais versatilidade em algumas funções, e seria algo que tentaríamos mudar num projeto futuro.

- Consumo de Memória

Relativamente à memória, sabemos perfeitamente que a nossa gestão da mesma poderia ser melhor. Apesar ter sido a nossa prioridade inicial do projeto, e de incessantemente termos recorrido a alocação dinâmica e consequentemente, à libertação da mesma, sabemos que existe desperdício de memória.

Deliberamos antecipadamente quais seriam as estruturas que íamos utilizar, pois acreditávamos que seria a melhor opção. Neste momento, com todos os conhecimentos que adquirimos sobre este tema, optaríamos por uma gestão mais eficaz dos dados.

6.2 Reflexão Final

Mediante o exposto neste relatório, o nosso programa é capaz de responder aos requisitos, e efetuar todos os comandos à disposição do utilizador. Assim sendo, tentaríamos melhorar o referido acima para uma melhor gestão de memória e um menor tempo de acesso.