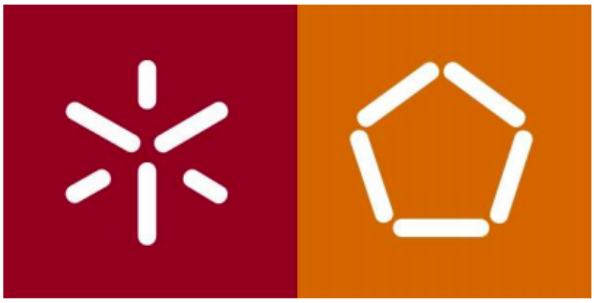


Universidade do Minho

Mestrado Integrado em Engenharia Informática

19/04/2020

Laboratórios de Informática III
Projeto em C

António Gonçalves (A85516)
Ricardo Costa (A85851)
Eduardo Conceição (A83870)

Contents

1 Introdução

Neste relatório iremos explicar as nossas escolhas e estratégias que utilizamos ao longo da realização do trabalho.

O objetivo deste trabalho é criar um Sistema de Gestão de Vendas (SGV), capaz de ler informação de ficheiros que contêm clientes, produtos e vendas. Após ler estes ficheiros, o utilizador poderá pedir várias *queries* para visualizar a informação agora presente no SGV.

Posto isto, um dos primeiros passos para realizar o projeto será criar estruturas capazes de guardar grandes quantidades de informação, de forma ordenada e eficaz, para que a sua consulta seja o mais rápida possível. Após estas serem criadas, será preciso ter a certeza que toda a informação presente nos ficheiros foi lida corretamente e que se encontra devidamente organizada, para que se possam realizar as *queries*.

2 Resolução

2.1 Módulos

Os vários ficheiros estão organizados de acordo com o paradigma MVC, seguindo a seguinte estrutura:

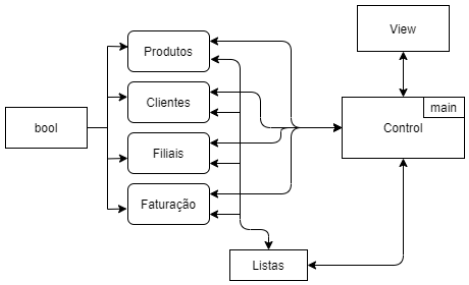


Figure 1: Diagrama da hierarquia dos ficheiros

Como tal,o *Model* corresponde aos vários ficheiros de dados que lidam diretamente com as estruturas de dados (Produtos, Clientes, Filiais e Faturação), juntamente com o módulo de booleanos e de estruturas auxiliares para as *queries*. A *View* do trabalho encontra-se toda no ficheiro "view.c", e o *Control*, juntamente com o SGV e a *main*, encontram-se em "control.c", onde está implementada a *interface* pedida no enunciado.

Associados a todos estes módulos estão os respetivos *header files*, onde estão declaradas as assinaturas dos métodos, e as declarações opacas das estruturas.

2.2 Leitura dos Ficheiros

A leitura dos vários ficheiros, em geral, é feita de forma semelhante. A leitura dos produtos e dos clientes é em tudo igual, validando cada linha do ficheiro *input* para verificar se o código que encontramos se conforma ao padrão dos produtos ou dos clientes, inserindo esse código de produto nas devidas estruturas, caso este seja válido, e colocando o seus valores booleanos associados a *false*(mais tarde explicaremos para que servem estes booleanos).

No entanto, a leitura das vendas é ligeiramente diferente, sendo que cada linha é separada num *array* de *strings*, cada *string* contendo um dos campos das vendas. Cada um dos campos é validado, sendo que os produtos e os clientes só serão válidos caso sigam o padrão e estejam nas estruturas respetivas. Caso não estejam, o código de produto ou cliente é considerado inválido e é descartado. Caso todos os campos estejam corretos e os códigos de produto e de cliente sejam válidos então os devidos campos da venda serão inseridos nas estruturas das filiais e da faturação total, e os booleanos dos produtos e dos clientes são atualizados para *true*. Esta atualização prejudica o tempo da *query* 1, mas diminui drasticamente o tempo da *query* 6. Uma forma alternativa de atualizar os booleanos seria aquando da procura, mas isto causaria o problema de um

produto poder ser válido, mas o cliente que o comprou não o ser, o que criaria uma incoerência nos dados.

Para além disto, em todos os casos, aquando da leitura dos ficheiros, a informação de cada (*path* para o ficheiro, número total de linhas lidas e de linhas válidas) é guardada numa estrutura auxiliar, FI, dentro do SGV, para a *query* 13.

2.3 Estruturas de Dados Principais

Uma vez que os ficheiros principais a ser lido são os dos Clientes e o dos Produtos, faz todo o sentido que cada um deste tenha a sua estrutura própria, às quais chamamos CatClientes e CatProdutos.

De seguida, é preciso guardar a informação de vendas. Para isso criámos duas estruturas, a FilHash e a HashTableFat. Mais à frente iremos explicar a utilidade de cada uma das estruturas criadas.

2.3.1 Catálogo de Clientes

Para guardar as informações dos clientes, criámos duas estruturas: CatClientes e AVLClientes. A primeira corresponde a uma implementação de uma *Hash Table* utilizando um *array*, com tamanho 26, que guarda 26 estruturas AVLClientes. Cada uma destas estruturas AVLClientes guarda todos os clientes começados por uma letra do alfabeto (ou seja, na primeira posição do *array* em CatClientes estão todos os clientes começados por 'A', na segunda todos começados por 'B', etc.), numa implementação de uma Árvore Binária de Procura AVL, em que, em cada nodo, encontramos um código de cliente e um booleano correspondente a se esse cliente comprou produtos ou não (um valor para ser usado na *query* 6).

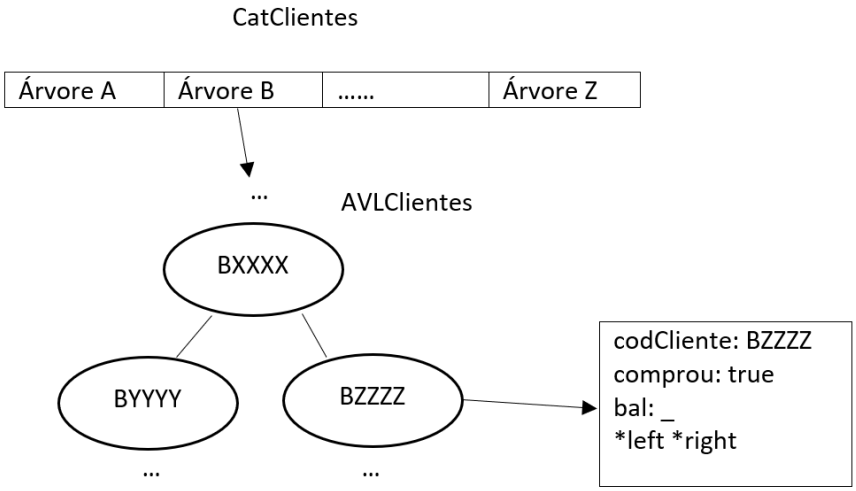
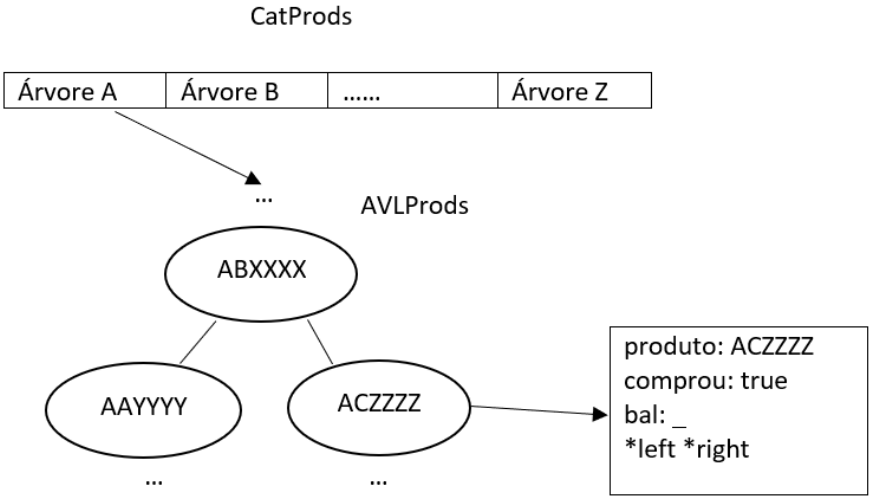


Figure 2: Estrutura usada em cliente.c

A estas estruturas estão associadas funções de inserção e de libertação de memória.

2.3.2 Catálogo de Produtos

As estruturas utilizadas para guardar os produtos são em tudo semelhantes às dos clientes. Temos mais uma vez uma *Hash Table* chamada CatProds, com um *array* de AVLProds. Como no caso dos clientes, esta última estrutura é uma Árvore Binária de Procura AVL, e em cada nodo encontramos um código de produto e um booleano que nos diz se o produto foi, ou não, comprado, um valor prático para várias *queries*.



2.3.3 Filiais

Como estrutura principal no que toca às filiais, temos a FilHash, que contêm várias AVLFils. Cada AVLFil representa a informação recebida dos Clientes iniciados por uma letra. Por exemplo, a primeira AVLFil representa a informação de todas as vendas efetuadas a Clientes cujo código começa pela letra 'A', na primeira filial. Para facilitar a travessia da árvore pretendida criamos uma função chamada hashFil, que a partir do código do Cliente nos dá a posição da árvore correspondente à primeira letra. A posição de cada uma destas árvores é calculada a partir do número da filial f e do resultado de hashFil(codCliente) p da seguinte forma:

```
Posição =  (hash->filiais)[(f-1)+p]
```

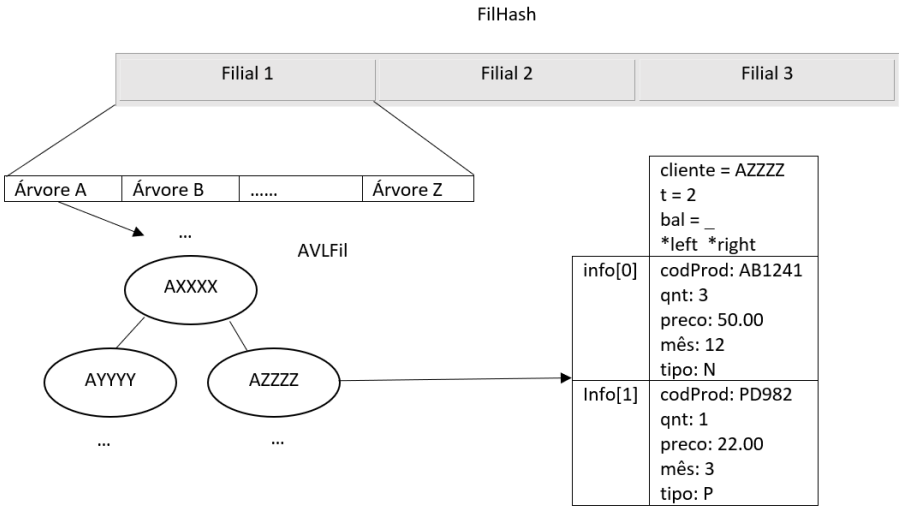


Figure 3: Estruturas usadas na filial.c

Estas árvores representam, como já referido, os vários clientes começados pela mesma letra e informações referentes às vendas dos mesmo. Assim, cada nodo da árvore irá representar um cliente, tendo nele, para além dos apontadores para os restantes nodos e o fator de balanceamento, um *array* info. Neste *array* iremos guardar a informação das vendas, como qual o produto que compraram, qual a quantidade, o preço, o mês em que foi efetuada a compra e o tipo de compra. Algumas destas informações foram adicionadas ao longo do projeto por tornarem as *queries* bastante mais simples e rápidas.

2.3.4 Faturação

A estrutura principal onde é armazenada a informação do módulo Faturação é a *HashTableFat*, que representa uma *hash table* de *AVLFat*, que por sua vez corresponde a uma estrutura do tipo AVL. Assim, cada posição da *hash table* é um apontador para uma estrutura do tipo *AVLFat* e a estrutura principal (*hash table*) é inicializada com 26 posições, uma para cada letra do alfabeto. Deste modo, as *AVLFat* estão organizadas pelos códigos dos produtos e cada uma contém apenas produtos iniciados pela mesma letra. Os produtos são inseridos segundo uma função de *hash* que nós permite descobrir de imediato a *AVLFat* na qual vão ser inseridos, tal como acontece no módulo Filiais. De seguida, apresenta-se o esquema completo da estrutura onde está armazenada a informação deste módulo:

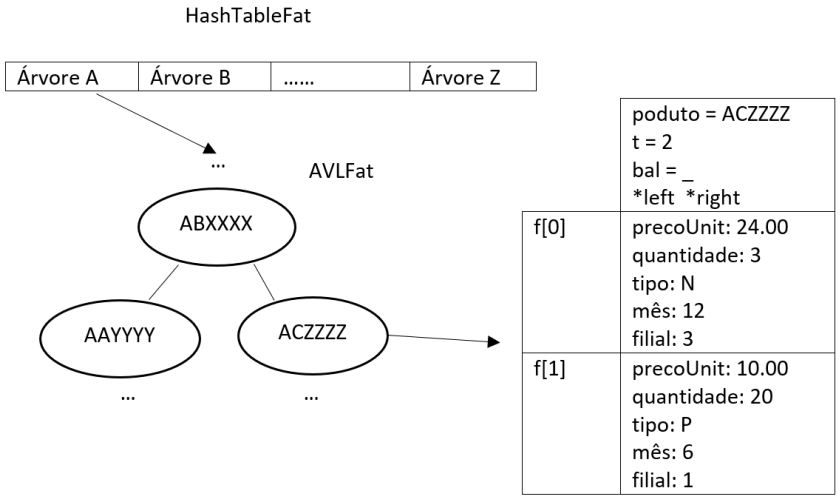


Figure 4: Estruturas usadas na fatutacao.c

Como podemos ver, cada nodo de cada *AVLFat* tem presente a seguinte informação: código de produto (por onde está organizada a árvore), balanceamento da *AVL*, apontadores para o ramo da esquerda e da direita e uma lista em que cada elemento é um apontador para uma estrutura *fatura* onde está armazenado o preço, a quantidade, o tipo, o mês e a filial de uma venda. Sempre que é encontrado no ficheiro de vendas uma linha com um código de produto igual, ao invés de adicionar um novo nodo na árvore, é adicionado antes um novo elemento a esta lista com estas informações e é incrementado o parâmetro *t*, que corresponde ao tamanho desta lista.

2.4 Queries

Nesta secção explicaremos em detalhe o funcionamento das várias *queries* pedidas neste projeto, bem como as estruturas que utilizamos para as resolver.

2.4.1 Query 1

A *query* 1 é referente ao preenchimento da estrutura SGV com os produtos, clientes e vendas. Para este efeito, damos a escolha ao utilizador de fornecer os seus próprios ficheiros, ou utilizar os que nos foram fornecidos no início do semestre por defeito. Depois disto, vai ser feita a leitura de cada ficheiro, como descrito na secção "Leitura dos Ficheiros"

2.4.2 Query 2

A *query* 2, dado uma letra, deverá devolver uma lista com todos os produtos começados por essa letra. Para esse efeito criámos uma estrutura Array, que contém um *array* de *strings* e o seu tamanho. Para preencher esta estrutura, apenas precisamos de percorrer a AVLProdutos no CatProds que corresponde à letra que é passada como argumento. Para impressão, criámos um navegador que permite visualizar um número dinâmico de valores de uma só vez. Este valor varia apenas com o tamanho da janela da linha de comandos.

2.4.3 Query 3

Nesta *query* é pretendido que, dado um código de produto e um mês, se apresente o número total de vendas e o total faturado com esse produto no dado mês, distinguindo ambos os totais no modo N e P e dependendo da escolha do utilizador, apresentar o resultado global ou distinguir os valores filial a filial.

Então, em primeiro lugar, obtemos as informações necessárias do utilizador, armazenando-as em variáveis diferentes. De seguida, como temos o código de produto conseguimos determinar qual a *AVLFat* da estrutura *HashTableFat* temos de percorrer, já que cada *AVLFat* apenas contém produtos iniciados por uma dada letra. Percorremos então a árvore à procura de um código de produto igual ao passado pelo utilizador. Ao encontrarmos esse produto, percorremos a lista presente nesse nodo à procura de um valor para o mês igual ao passado pelo utilizador. Só depois de passar por estas condições é que vemos o valor do parâmetro tipo para distinguir os totais em tipo N e P e se o utilizador tiver escolhido apresentar o resultado filial a filial ainda verificamos a filial para podermos distinguir também os totais das 3. Por fim, armazenamos estes valores em posições diferentes duma lista passada como argumento. Por exemplo, posição 0 para total de vendas do tipo N, posição 1 para total faturado do tipo N, posição 2 para total de vendas do tipo P e posição 3 para total faturado do tipo P. No caso de distinguir os resultados filial a filial, é necessário o triplo destas posições, mantendo, porém, esta ordem.

2.4.4 Query 4

Na *query* 4, dado um número de filial, devolve a lista dos produtos que não foram comprados nessa filial. Para isso, primeiro percorremos o CatProds, reunindo num Array todos os produtos que foram vendidos. Depois, percorremos a Faturacao, procurando cada produto. Se esse produto tiver alguma vez sido vendido na filial dada, descartamo-lo. Caso contrário, mantêmo-lo. No final, imprimímos a lista de produtos resultante, da mesma forma que a *query* 2.

2.4.5 Query 5

Nesta *query* é-nos pedido para criar uma lista com todos os Clientes que efetuaram compras em todas as filiais.

Assim, começámos por criar a lista de todos os clientes que fizeram compras, percorrendo o CatClientes. Com esta lista, iremos depois percorrer cada Cliente e verificar se este existe ou não em cada filial. Caso exista nas 3 filiais, o código do Cliente será adicionado a um *Array* que contém os restantes que compraram em todas as filiais, que depois serão imprimidos no ecrã, permitindo ao utilizador navegar pela lista.

2.4.6 Query 6

Para a *query* 6 são-nos pedidos os valores dos clientes que nunca compraram e dos produtos que nunca foram vendidos. Para este efeito, criamos uma estrutura denominada Pair, onde guardamos um par de inteiros com a informação descrita anteriormente. Para a obter, basta percorrer cada uma das árvores no CatClientes e CatProds, e, para cada nodo com o valor booleano ("comprado" no caso dos produtos, "comprou" no caso dos clientes) verificar se o mesmo é *false* e, se for, somamos um ao total. No final, guardamos os totais contados na estrutura Pair e devolvê-mo-la para poder ser apresentada.

2.4.7 Query 7

Nesta *query* é-nos pedido que, dando um código de cliente, seja criada uma tabela com a quantidade total de produtos comprados mês a mês, organizada por filial.

Começamos por inicializar um *array* de inteiros com 36 posições, sendo este usado para representar as quantidades compradas em cada mês por filial. Depois iremos à árvore a que pertence o cliente em cada filial ([p+26*i], sendo p o hash do codigo e i o número da filial, começando em 0) para depois percorrer as infos do mesmo, adicionando ao *array* com o resultado a quantidade comprada na posição certa((m+12*i)-1, sendo m o mês e i o número da filial).

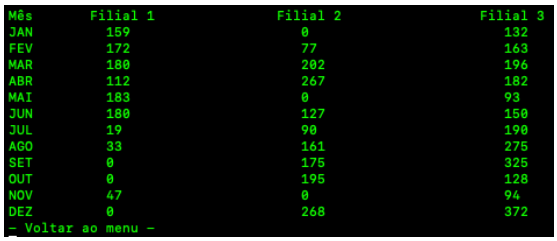


Figure 5: Exemplo de resultado da *query* 7.

2.4.8 Query 8

Nesta *query* é pretendido que a partir de um intervalo de meses dados pelo utilizador, determinemos o total de vendas registadas e o total faturado nesse intervalo.

Começámos então por perguntar ao utilizador o intervalo de meses que pretende analisar e de seguida validamos esse intervalo, verificando se os meses inseridos são válidos (1 a 12) e se o mês de fim é maior que o mês de início.

De seguida percorremos toda a estrutura *HashTableFat*, isto é, percorremos a lista de 26 *AVLFat* e os nodos todos de cada árvore. Em cada nodo percorremos a lista de estruturas *fatura* e verificámos se o mês dessa posição da lista está entre os meses dados pelo utilizador. Se estiver, incrementámos uma variável por 1, que é a variável onde vai estar a informação do total de vendas e incrementámos outra variável pela multiplicação dos parâmetros quantidade e preço presentes na estrutura *fatura*, para podermos obter o total faturado.

2.4.9 Query 9

Nesta *query* é-nos pedido que, dado um código de produto e uma filial, determinássemos os códigos de clientes que o compraram, distinguindo compras N e P.

Para isso, criámos uma nova estrutura, Cods, que tem dois *arrays* onde iremos guardar os codigos dos cliente que compraram os dois tipos diferentes, e dois inteiros que representam o tamanho de cada lista.

Depois de inicializar esta estrutura, iremos percorrer todas as AVLFils da filial dada, procurando clientes que tenham comprado o produto. Conforme o tipo de compra, este codigo de cliente será adicionado à devida componente da estrutura. No final de percorrer as AVLs, ficamos com a estrutura completa, e só é necessário retira-la.

2.4.10 Query 10

A *query* 10 trata-se de obter os produtos mais comprados por um cliente num dado mês, em termos de quantidade, e não de dinheiro gasto.

Para isto, percorremos a filial para encontrar o cliente que queremos. Uma vez que o encontremos, verificamos o seu *array* de Info, verificando em cada compra se esta foi feita no mês dado. Caso seja, inserimos a compra numa estrutura Array2, onde guardamos um *array* de *strings* e um de inteiros, ambos de tamanho igual, para guardar o código do produto e o número que foi comprado. Para este efeito, sempre

que fazemos uma inserção, verificamos se o produto já está no *array*. Se já estiver, somamos a quantidade comprada nova ao acumulado que já tínhamos, caso contrário, simplesmente adicionamos mais uma entrada no Array2. Uma vez que o cliente é percorrido, a lista é ordenada.

A ordenação é feita por ordem decrescente, com a ajuda de uma AVL criada apenas para este propósito. Esta estrutura é apenas uma Árvore binária de procura AVL com um *string* e um inteiro em cada nodo. A ordenação vem naturalmente se nós inserirmos todos os elementos na árvore (ao contrário da ordem natural, ou seja, os maiores váo para a esquerda e os menos para a direita) e depois fazer uma travessia *in order* para voltar a preencher o *Array2*, devolvendo-o e imprimindo a informação com o mesmo género de navegador utilizado noutras *queries*.

2.4.11 Query 11

Nesta *query* é pretendido criar uma lista dos N produtos mais vendidos, indicando o número total de clientes e de unidades vendidas para cada produto, distinguindo estes valores pelas 3 filiais.

Assim, começámos primeiro por perguntar ao utilizador quantos produtos é que pretende nesta lista.

Para resolver esta *query*, inicializamos uma estrutura denominada *Array2*, que contém uma lista de *char** para armazenar o código de cada produto e uma lista de inteiros para armazenar a quantidade de vezes que cada produto foi vendido. As posições das duas listas correspondem uma à outra (por exemplo, a quantidade de vezes que o código de produto na posição 0 da primeira lista foi vendido está na posição 0 da segunda lista). Esta estrutura contém também um inteiro para indicar o tamanho das duas listas (que é sempre o mesmo).

De seguida, percorremos toda a estrutura do módulo Faturação, *HashTableFat* e correspondentes árvores, e para cada nodo da estrutura *AVLFat*, percorremos a sua lista e vamos incrementando uma variável com o parâmetro *quantidade* de cada elemento da lista. Depois de percorrida a lista, chamámos uma função para adicionar o código de produto corresponde ao nodo e esta variável à estrutura *Array2* que tinhamos inicializado. Assim, no final, esta estrutura contém todos os códigos de produto numa lista e correspondente quantidade vendida noutra lista.

A seguir, convertemos esta estrutura para uma AVL (organizada pela quantidade) e de novo para a mesma estrutura, o que nos permite ordenar os produtos por ordem decrescente de quantidade vendida.

O próximo passo é dar *free* aos elementos das duas listas que não precisámos, isto é, a partir da variável N, cujo valor é dado pelo utilizador.

De seguida, inicializamos uma estrutura denominada *Array4*, que contém uma lista de *char**, 3 matrizes de inteiros (para cada filial) e um inteiro que indica o tamanho de todas as listas. Preenchemos a primeira lista com os códigos de produto da estrutura anterior (*Array2*) e por fim, percorremos toda a estrutura do módulo Filiais e para cada nodo de cada AVL, percorremos a lista em que cada elemento é uma estrutura *info* para cada um dos produtos existentes na estrutura *Array4*. Ou seja, em cada nodo, percorremos a sua lista várias vezes (para cada produto existente na estrutura *Array4*), porque várias produtos da estrutura *Array4* podem ter sido comprados pelo mesmo

cliente. Dependendo da filial, incrementámos as duas colunas de uma dada posição, com o número de clientes e a quantidade, respetivamente, da matriz correspondente (1, 2 ou 3) na estrutura *Array4*.

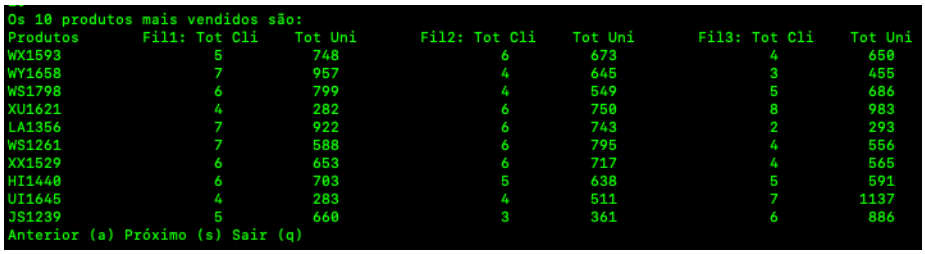


Figure 6: Exemplo de resultado da *query* 11.

2.4.12 Query 12

Para a *query* 12, o sistema deve, dado um código de cliente, devolver os N produtos em que o cliente gastou mais dinheiro ao longo do ano. Para auxiliar esta *query*, temos uma estrutura Array3, em tudo igual ao Array2, mas com um *array* de números reais e não de inteiros.

Assim, percorremos as Filiais em procura do cliente em questão. Uma vez que o encontramos, percorremos o seu *array* de info. Em cada iteração, adicionamos a compra ao Array3, da mesma forma que na *query* 10. Uma vez que tenhamos toda a informação do cliente nas três filiais recolhida, ordenamos o Array3 da mesma forma que o fazíamos na *query* 10. No entanto, ao invés de utilizarmos a estrutura AVL, utilizamos AVLF, bastante semelhante à anterior, mas com um número real em cada nodo em vez de um inteiro.

Uma vez que temos a lista ordenada, temos que eliminar os items para lá da N-ésima posição. Para isso, fazemos *free* de todas as *strings* excedentes, e fazemos um *resizing* dos apontadores dos *arrays* de *strings* e de *floats*, mantendo assim o uso desncessário de memória ao mínimo.

Uma vez que este processo está concluído, imprimimos os resultados na *bash* utilizando o mesmo navegador que nas outras *queries*.

No entanto, é de notar que esta *query*, à semelhança da 11, apresenta problemas quando executada repetidamente, mas, ao contrário da 11, apenas em sistemas Linux, e foi um problema que, no fim, não conseguimos resolver. Quando se executa a *query* repetidamente, ou a *query* 11 depois desta, o programa é parado com *Segmentation Fault*.

2.4.13 Query 13

Na *query* 13 devemos apresentar ao utilizador os *paths* para os ficheiros lidos, o número de linhas total e de linhas válidas para cada um. Para isso, basta aceder à informação guardada numa estrutura FI no SGV, que contém 3 *strings*, para cada um dos *paths* dos ficheiros, e 6 inteiros, 2 para as estatísticas de cada ficheiro.

Uma vez que se acede a estas informações, é só uma questão de imprimir os valores lidos.

3 Testes de Desempenho

	Ficheiro Vendas 1M	Ficheiro Vendas 3M	Ficheiro Vendas 5M
Q1	3.3184	10.94809	18.447773
Q4	0.039882	0.043499	0.042622
Q6	0.08632	0.08919	0.009209
Q7	0.000633	0.000576	0.000641
Q8	0.057558	0.145852	0.255811
Q9	0.016164	0.045505	0.072147
Q10	0.000095	0.000260	0.000342
Q11	0.944029	2.825218	4.496249
Q12	0.000405	0.001083	0.001903

Table 1: Tabela com os testes de *performance* (em segundos) de cada query.

Em geral, os valores são bastante bons. As *queries* 12 e 6, em particular, têm valores bons, uma vez que as implementações que escolhemos para as estruturas necessárias tornam as *queries* em pouco mais de uma ou duas travessias de árvores. Como essa travessia terá uma complexidade de $O(\log_2 N)$, isto seria de esperar. Os valores da *Query* 11 e 12 são referentes a 100 produtos, ou seja, valor de N igual a 100 (daí o valor da Query 11 com o ficheiro de 5M ser relativamente elevado, já que na nossa solução é necessário percorrer toda a estrutura, quer do módulo Faturação, quer do módulo Filiais por completo).

Outra forma de verificar o desempenho é através do uso de memória, nomeadamente, se existem *memory leaks*. Uma vez que, sempre que é alocada memória a uma estrutura, se ela deixa de ser necessária, essa memória é libertada, podemos assumir com confiança que, em geral, o uso de memória é adequado. O mesmo se aplica a apontadores mais simples como *char**, *int** ou *float**.

4 Documentação

A documentação deste projeto foi feita utilizando a ferramenta Doxygen. Pode ser encontrada no repositório do projeto, dentro do *folder* "html" na diretoria "docs", no ficheiro "index.html".

5 Conclusão

Em geral, acreditamos que conseguimos atingir o objetivo prinicipal do trabalho. Conseguimos implementar as funcionalidades pedidas, com a estrutura de módulos pedida. No entanto, acabamos por ter alguns problemas com as duas últimas *queries*, que, apesar de funcionarem, impedem o bom funcionamento do programa uma vez executadas, o que é, na nossa opinião, um aspeto bastante negativo. Contudo, e em jeito de resumo, acreditamos que, apesar dos problemas que ficaram, estes são menores no panorama geral do trabalho, pelo que consideramos que foi bem sucedido.