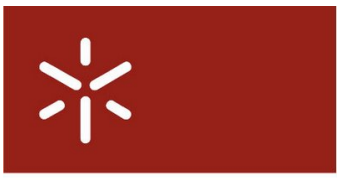


Universidade do Minho

Relatório - Laboratórios de Informática III

MIEI - 2º ano - 2º semestre
Universidade do Minho

Gestão das Vendas de uma Cadeia de Distribuição com 3 Filiais

Grupo 33

Dinis Peixoto
A75353

Ricardo Pereira
A74185

Marcelo Lima
A75210

16 de Novembro de 2017

Conteúdo

1 Introdução **2**

2 Descrição dos Módulos **2**
2.1 Catálogo de Clientes . . . 3
2.1.1 Estrutura . . . 3
2.1.2 Typedef's . . . 4
2.1.3 Funções . . . 4
2.2 Catálogo de Produtos . . . 5
2.2.1 Estrutura . . . 5
2.2.2 Typedef's . . . 5
2.2.3 Funções . . . 5
2.3 Facturação Global . . . 7
2.3.1 Estrutura . . . 7
2.3.2 Typedef's . . . 8
2.3.3 Funções . . . 8
2.4 Vendas por filial . . . 10
2.4.1 Estrutura . . . 10
2.4.2 Typedef's . . . 11
2.4.3 Funções . . . 11

3 Programa Principal **13**

4 Apresentação da Makefile **14**
4.1 Estrutra . . . 14
4.2 Comandos disponíveis . . . 14
4.3 Grafo de dependências . . . 15

5 Interface para o utilizador **15**

6 Testes de Performance **18**

7 Conclusão **21**

1. *Introdução*

Este projeto foi nos solicitado pelos docentes da unidade curricular *Laboratórios de Informática III* e tem como principal objetivo a realização de um programa que faça a gestão de vendas de uma Cadeia de Distribuição com vários filiais. Outros objectivos estão também associados a este, como a consolidação de conhecimentos adquiridos em unidades curriculares anteriores, dentro das quais se incluem *Programação Imperativa*, *Algoritmos e Complexidade* e *Arquitetura de Computadores*. O principal desafio do projecto seria a programação em larga escala, uma vez que iam passar pelo nosso programa milhões de dados, aumentando assim a complexidade do trabalho. Para que a realização deste projecto fosse possível, foram-nos introduzidos novos príncipios de programação, com especial relevo para a *Modularidade e encapsulamento de dados*.

2. *Descrição dos Módulos*

A arquitetura da aplicação é desenvolvida conforme quatros módulos principais: Catálogo de Clientes, Catálogo de Produtos, Facturação global e Vendas por filial. Adjacente a estes módulos existe também um módulo responsável pela leitura dos ficheiros como input para aplicação: *Clientes.txt*, *Produtos.txt* e *Vendas_1M*, *Vendas_3M* ou *Vendas_5M*, conforme desejado. Todos estes módulos estão diretamente relacionados com uma interface onde o utilizador final pode desfrutar de todas as funcionalidades que a aplicação implementa.

Ficheiros

Clientes.txt:

Ficheiro com o código de 20.000 clientes. O código de cada cliente é representado por uma letra maiúscula seguida de quatros digitos que representam um número entre 1000 e 5000.

Produtos.txt:

Ficheiro com o código de 200.000 produtos. O código de cada produto é representado por duas letras maiúsculas seguidas de quatros digitos que representam um número entre 1000 e 1999.

Vendas_1M.txt:

Ficheiro com o código de 1.000.000 de vendas. O código de cada venda inclui: um código de produto, um preço (entre 0 e 999.99), uma quantidade (entre 1 e 200), uma letra indentificando o tipo de compra (Normal ou Promoção), um código de cliente, o mês da compra e por fim o filial onde ocorreu a venda.

Vendas_3M.txt:

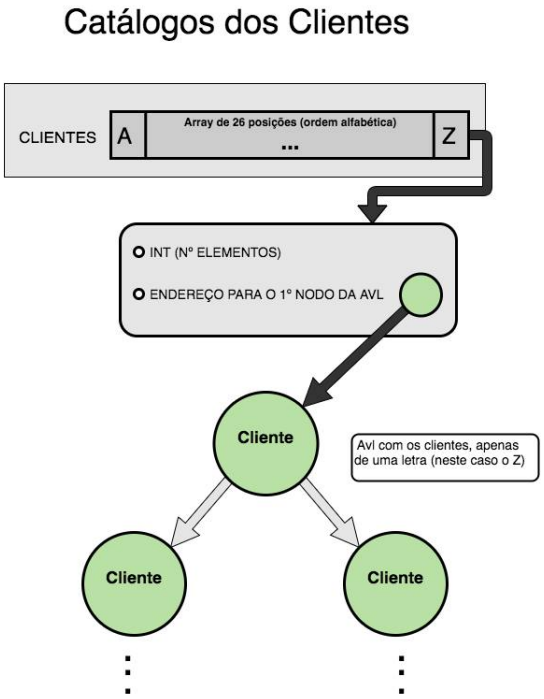
Ficheiro com o código de 3.000.000 de vendas. Estrutura idêntica ao ficheiro anterior.

Vendas_5M.txt:

Ficheiro com o código de 5.000.000 de vendas. Estrutura idêntica aos ficheiros anteriores.

2.1 Catálogo de Clientes

2.1.1 Estrutura



O Catálogo de Clientes é o módulo onde são armazenados todos os clientes que se encontram no ficheiro *Clientes.txt*. Como se trata de uma larga quantia de clientes, necessitámos de os guardar numa estrutura adequada:

MY_AVL CatClients[SIZE_ABC]

Correspondendo a um array de 26 estruturas MY_AVL (nas quais se encontra uma AVL com todos os clientes cujo código começa por uma determinada letra e um inteiro com o número de elementos presentes nessa mesma AVL).

2.1.2 Typedef's

.h

typedef struct client *CLIENT;
typedef struct catc *CATALOG_CLIENTS;

2.1.3 Funções

- **CATALOG_CLIENTS initClients();**

  Inicia o Catálogo dos Clientes, alocando espaço para o mesmo.

- **CLIENT initClie();**

  Inicia a estrutura do Cliente, alocando espaço para a mesmo.

- **CATALOG_CLIENTS insertClient(CATALOG_CLIENTS,CLIENT);**

  Insere um Cliente no respetivo Catálogo. Para garantir o encapsulamento a função recebe como argumento uma estrutura CLIENT em vez de um char*.

- **BOOL testClient(CLIENT);**

  Testa se um Cliente tem uma estrutura correta.

- **BOOL existClient(CATALOG_CLIENTS, CLIENT);**

  Verifica se um determinado Cliente existe num Catálogo de Clientes.

- **void freeClient(CLIENT);**

  Remove um Cliente, libertando o espaço ocupado por este.

- **void removeCatClients(CATALOG_CLIENTS);**

  Limpa o Catálogo de Clientes.

- **int totalClientsLetter(CATALOG_CLIENTS,char);**

  Calcula quantos Clientes, começados por uma determinada letra, existem no Catálogo de Clientes.

- **int totalClients(CATALOG_CLIENTS);**

  Calcula quantos Clientes existem num Catálogo de Clientes.

- **char* getClient(CLIENT);**

  Retorna um código de um Cliente. Para garantir o encapsulamento deste cliente, a string (código) devolvida é uma cópia do código original do cliente.

- **CLIENT setClient(CLIENT,char*);**

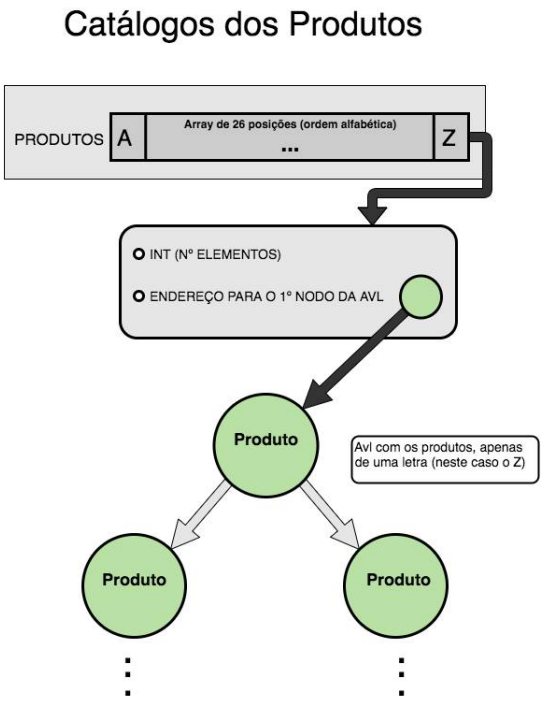
  Altera a String de um Cliente. Para garantir o encapsulamento do cliente, a string do cliente é uma cópia da string dada como argumento pelo utilizador.

- **MY_AVL getC(CATALOG_CLIENTS,int);**

  Retorna a MY_AVL presente num determinado índice do Catálogo de Clientes. Para garantir o encapsulamento a estrutura retornada é uma cópia da estrutura original.

2.2 Catálogo de Produtos

2.2.1 Estrutura



O Catálogo de Produtos é o módulo onde são armazenados todos os produtos que se encontram no ficheiro *Produtos.txt*. Tal como no caso anterior, como se trata de uma larga quantia de produtos, necessitámos de os guardar numa estrutura adequada:

MY_AVL CatProducts[SIZE_ABC]

Correspondendo a um array de 26 estruturas MY_AVL (nas quais se encontra uma AVL com todos os produtos cujo código começa por uma determinada letra e um inteiro com o número de elementos presentes nessa mesma AVL).

2.2.2 Typedef's

.h

```
typedef struct product *PRODUCT;
typedef struct catp *CATALOG_PRODUCTS;
```

2.2.3 Funções

- **CATALOG_PRODUCTS initProducts();**

  Inicia o Catálogo dos Produtos, alocando espaço para o mesmo.

- **PRODUCT initProd();**

  Inicia a estrutura do Produto, alocando espaço para a mesmo.

- **CATALOG_PRODUCTS insertProduct(CATALOG_PRODUCTS, PRODUCT);**

  Insere um Produto no respetivo Catálogo. Para garantir o encapsulamento a função recebe como argumento uma estrutura PRODUCT em vez de um char*.

- **BOOL testProduct (PRODUCT);**

  Testa se um Produto tem uma estrutura correta.

- **BOOL existProduct(CATALOG_PRODUCT,PRODUCT);**

  Verifica se um determinado Produto existe num Catálogo de Produtos.

- **void freeProduct(PRODUCT);**

  Remove um Produto, libertando o espaço ocupado por este.

- **void removeCatProds(CATALOG_PRODUCTS);**

  Limpa o Catálogo de Produtos.

- **int totalProductsLetter(CATALOG_PRODUCTS,char);**

  Calcula quantos Produtos, começados por uma determinada letra, existem no Catálogo de Produtos.

- **int totalProducts(CATALOG_PRODUCTS);**

  Calcula quantos Produtos existem num Catálogo de Produtos.

- **char* getProduct(PRODUCT);**

  Retorna o código de um Produto. Para garantir o encapsulamento deste produto, a string (código) devolvida é uma cópia do código original.

- **PRODUCT setProduct(PRODUCT,char*);**

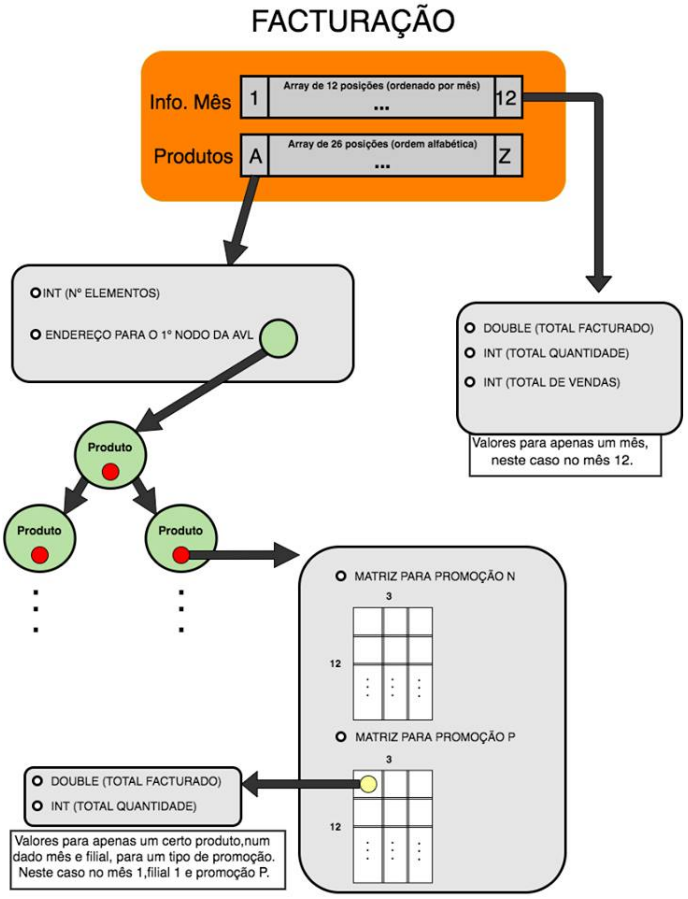
  Altera a String de um Produto. Para garantir o encapsulamento do produto, a string do produto é uma cópia da string dada como argumento pelo utilizador.

- **MY_AVL getP(CATALOG_PRODUCTS,int);**

  Retorna a MY_AVL presente num determinado índice do Catálogo de Produtos. Para garantir o encapsulamento a estrutura retornada é uma cópia da estrutura original.

2.3 Facturação Global

2.3.1 Estrutura



O módulo da Facturação apresenta uma estrutura mais complexa comparada aos catalógos apresentados anteriormente.
A sua estrutura inicial, possui dois array's. Um array, indexado por mês, tendo em cada posição uma estrutura que com valores referentes, a quantiadade total, total facturado e número de vendas totais, de um dado mês.
O outro array, possui a mesma estrutura que o Catalógo de Produtos. É um array de 26 posições, indexado pelas letras iniciais dos produtos, para uma melhor repartição dos mesmos.
No entanto, em cada nodo da AVL, que representa um produto, possui mais uma estrutura comparado ao Catálogo dos Produtos. Essa estrutura possui a toda a informação do Produto. Estruturado com duas matrizes, de dimensões 12 x 3 (Nº meses x Nº filiais), sendo uma matriz para promoção (P) e a outra matriz para sem promoção (N). Cada posição da matriz, possui uma outra estrutura com o total facturado e a quantidade total. Estes valores são referentes a um dado produto, mês, filial e promoção.

2.3.2 Typedef's

.h

typedef struct fact *FACTURACAO;
typedef struct dados *DADOS;

.c

typedef struct totalMes *TOTAL_MES;
typedef struct priceQuantity *PRICE_QUANTITY;
typedef struct info *INFO;

2.3.3 Funções

- **FACTURACAO initFact();**

  Inicia a estrutura da Facturação, alocando espaço para a mesma.

- **DADOS initDADOS();**

  Inicia a estrutura dos dados, alocando espaço para a mesma.

- **FACTURACAO copyProducts(FACTURACAO,CATALOG_PRODUCTS);**

  Copia os produtos do Catálogo de Produtos para a Facturação. Ao copiar os produtos, são feitas cópias dos produtos e só depois copiados, isto para garantir o encapsulamento.

- **FACTURACAO insereFact(FACTURACAO,SALES);**

  Insere uma Venda na estrutura Facturação. Ao fazer a inserção é feita uma cópia da estrutura para garantir o encapsulamento da estrutura de retorno.

- **void freeFact(FACTURACAO);**

  Remove a estrutura da Facturação, libertando o espaço ocupado por esta.

- **void freeInfo(void*);**

  Remove a estrutura Info, libertando o espaço ocupado por esta.

- **double* getDadosP(DADOS);**

  Retorna um apontador para um array com a quantidade total facturada nas várias filiais.

- **int* getDadosQ(DADOS);**

  Retorna um apontador para um array com a quantidade total de produtos comprados nas várias filiais.

- **double getDadosTP(DADOS);**

  Retorna o total facturado num Mês.

- **int getDadosTV(DADOS);**

  Retorna a quantidade total de produtos comprados num Mês.

- **double getTotalMP(DADOS);**

  Retorna a quantidade total facturada.

- **int getDadosTQ(DADOS);**

  Retorna a quantidade total .

- **int getTotalVendas(DADOS);**

  Retorna a quantidade total de produtos vendidos.

- **DADOS totalM(FACTURACAO,DADOS,int,int);**

  Retorna a quantidade total de produtos vendidos, na estrutura DADOS.

- **LISTA_STRINGS listaProducts(FACTURACAO,LISTA_STRINGS,int);**

  Constrói uma lista dos produtos, nunca comprados, de forma Total ou por Filial. Para garantir o encapsulamento da estrutura, de cada vez que é adicionado um elemento à estrutura LISTA_STRINGS é criada uma cópia do mesmo.

- **DADOS updatePriceQuantity(FACTURACAO,DADOS,int,int,char*);**

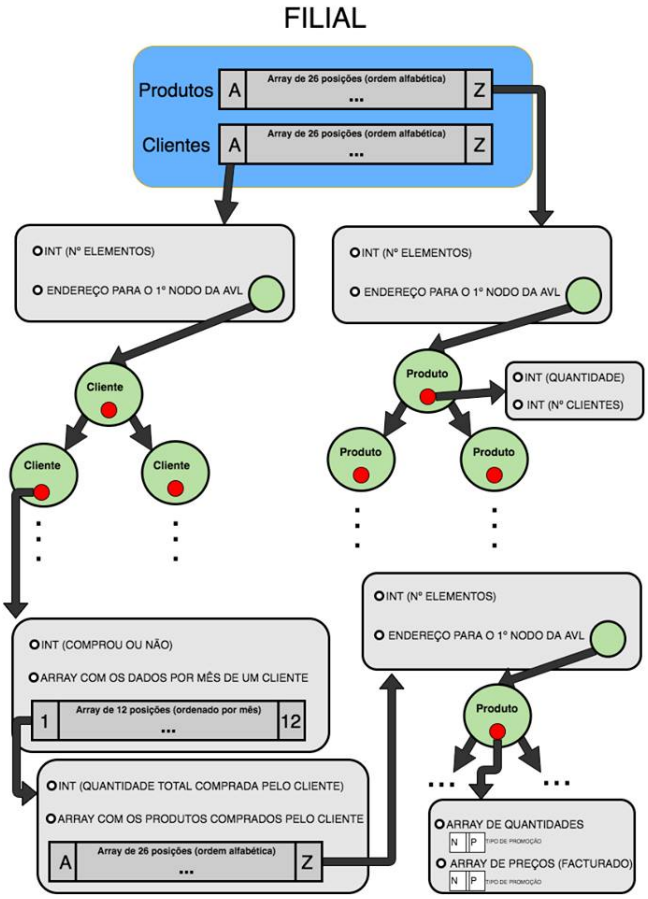
  Recolhe os dados sobre um produto, num dado mês e promoção.

- **int querie12Products(FACTURACAO);**

  Calcula o número de produtos nunca comprados.

2.4 Vendas por filial

2.4.1 Estrutura



O módulo da Filial apresenta uma estrutura apenas para uma das várias Filiais. Esta apresenta a estrutura mais complexa comparado aos vários módulos apresentados anteriormente.
A estrutura inicial apresenta-se com dois array's. Cada array tem 26 posições, e indexam para a letra inicial, no entanto um array é para os Clientes e outro é para os Produtos.
No array dos Produtos, temos em cada posição uma AVL com os produtos com a mesma letra inicial. Para além disso, em cada nodo dessa AVL, que representa um Produto, existe uma estrutura que possui os valores referentes a quantidade total comprada do produto e ao número total de vendas, ou seja, o número total de clientes diferentes que compraram aquele produto.
No array dos Clientes, também temos os clientes separados pela letra inicial, em cada posição do array. Com isto, em cada posição podemos encontrar uma AVL com os clientes inicados pela mesma letra. Em cada nodo dessa AVL, que representa um Cliente, temos uma estrutura que possui, um inteiro indicando se um cliente comprou ou não e um array indexado por mês. Em cada posição do array temos outro inteiro que representa a quantidade total comprada pelo cliente

num dado mês, e um array de 26 posições, que indexa a letra inicial dos produtos comprados pelo Cliente.
Além disso, em cada posição do array, os produtos com a mesma letra inicial estão organizados por uma AVL. Em cada nodo da AVL, que respresenta um produto, temos uma estrutura com dois array's de 2 posições. Um array possui as quantidades e outro possui os preços, correspondentes ao total facturado. As posições representam se os valores provém de uma promoção (posição 1) ou sem promoção (posição 0). Desta forma temos uma grande repartição, o que torna os acessos aos dados muito mais eficientes.

2.4.2 Typedef's

.h

typedef struct filial *FILIAL;
typedef struct dadosFilial *DADOS_FILIAL;

.c

typedef struct infoMes *INFO_MES;
typedef struct infoClient *INFO_CLIENT;
typedef struct infoProduct *INFO_PRODUCT;
typedef struct productInfo *PRODUCT_INFO;

2.4.3 Funções

- **FILIAL initFilial();**

  Inicia a estrutura da Filial, alocando espaço para a mesma.

- **DADOS_FILIAL initDadosFilial();**

  Inicia a estrutura DADOS_FILIAL, alocando espaço para a mesma.

- **FILIAL insertFilial(FILIAL,SALES);**

  Insere uma Venda na estrutura Filial. Ao fazer a inserção é feita uma cópia da estrutura para garantir o encapsulamento da estrutura de retorno.

- **FILIAL copyP(FILIAL,CATALOG_PRODUCTS);**

  Copia os Produtos do Catálogo para a Filial. Ao copiar os produtos, são feitas cópias dos produtos e só depois copiados, isto para garantir o encapsulamento.

- **FILIAL copyCPO(FILIAL,CATALOG_CLIENTS);**

  Copia os Clientes do Catálogo para a Filial. Ao copiar os clientes, são feitas cópias dos clientes e só depois copiados, isto para garantir o encapsulamento.

- **void freeFilial(FILIAL);**

  Remove a estrutura Filial, libertando o espaço ocupado por esta.

- **DADOS_FILIAL updateQuant_DadosFilial(DADOS_FILIAL,int,int);**

  Actualiza os dados relativos a uma dada Filial num dado mês.

- **int getDadosFilialQuantity(DADOS_FILIAL,int);**

  Retorna a quantidade de produtos comprados numa dada Filial, num dado Mês.

- **LISTA_STRINGS dontBuyClient(FILIAL,LISTA_STRINGS);**

  Constrói uma lista de Strings, com os clientes que nunca compraram nada.

- **Heap highCostProd(FILIAL,Heap char*);**

  Constrói uma Heap, ordenada por os produtos com maior facturação.

- **Heap querie10Fil(FILIAL,Heap);**

  Constrói uma Heap, ordenada pelos produtos mais vendidos em quantidade. Ao construir uma Heap, antes de adicionar os elementos à mesma é feita uma cópia destes para garantir assim o encapsulamento da Heap.

- **Heap moreBuy(FILIAL,Heap,char*,int);**

  Constrói uma Heap, ordenada pelos produtos com maior gastos. Ao construir uma Heap, antes de adicionar os elementos à mesma é feita uma cópia destes para garantir assim o encapsulamento da Heap.

- **LISTA_STRINGS productNeP(FILIAL,char,LISTA_STRINGS, LISTA_STRINGS);**

  Constrói duas Listas de Strings, tendo uma os produtos que um cliente comprou sem promoção e outra com promoção. Para garantir o encapsulamento da estrutura, de cada vez que é adicionado um elemento à estrutura LISTA_STRINGS é criada uma cópia do mesmo.

- **LISTA_STRINGS checkClientsValeN (FILIAL,LISTA_STRINGS);**

  Constrói uma Lista de Strings, com os Clientes que compraram numa filial. Para garantir o encapsulamento da estrutura, de cada vez que é adicionado um elemento à estrutura LISTA_STRINGS é criada uma cópia do mesmo.

- **DADOS_FILIAL valoresFilial(FILIAL,DADOS_FILIAL,char*);**

  Recolhe os dados de uma filial, sobre um dado cliente.

3. *Programa Principal*

```c
int main(){

    int running =1,i;

    CATALOG_CLIENTS CatClients = initClients();
    CATALOG_PRODUCTS CatProducts = initProducts();
    FILIAL Filiais[3];
    FACTURACAO Fact = initFact();
    for(i=0;i<3;i++)
        Filiais[i] = initFilial();

    while(running){
        running = interpretador(CatClients,CatProducts,Filiais,Fact);
        if(running == -1){
            freeMemory(CatClients,CatProducts,Filiais,Fact);
            CatClients = initClients();
            CatProducts = initProducts();
            Fact = initFact();
            for(i=0;i<3;i++) Filiais[i] = initFilial();
            running = 1;
        }
    }
    freeMemory(CatClients,CatProducts,Filiais,Fact);
    return 0;
}
```

O nosso programa principal começa por inicializar todas as estruturas necessárias para armazenar a informação contida nos ficheiros de texto que o programa se prepara para ler.
Após isso, dentro de um ciclo *while* é chamada a função *interpretador*, função esta responsável por apresentar uma interface gráfica ao utilizador, permitindo que o mesmo possa desfrutar de todas as funcionalidades do nosso programa, dentro das quais estão incluídas as queries propostas.
O retorno desta função é associado à variável *running*, sendo que esta é muito importante para a continuação do ciclo while. A variável *running* pode tomar 3 valores distintos:

- 1. Se o comando utilizado pelo utilizador correu bem, sendo que o ciclo pode continuar, podendo assim o utilizador continuar a desfrutar da interface.

- -1. Se o comando de leitura foi usado após as estruturas já estarem preenchidas e o utilizador desejar limpar estas estruturas para poder reler novamente os ficheiros de texto.

- 0. Se o utilizador desejar sair da interface, fechando assim o programa principal também, tendo em conta que a condição para o ciclo while deixa de ser satisfeita.

No último caso, antes do programa terminar definitivamente é libertada toda a

memória utilizada até então pelas estruturas de armazenamento de dados.

4. *Apresentação da Makefile*

```
CC = gcc

CFLAGS = -ansi -Wunreachable-code -O2 -Wuninitialized -Wunused-parameter -Wall -Wextr

OBJECTS = obj/main.o obj/avl.o obj/CatClients.o obj/CatProducts.o obj/facturacao.o \
          obj/filial.o obj/heap.o obj/interpretador.o obj/listaStrings.o \
          obj/queries.o obj/readFiles.o obj/Sales.o

DOC = doc/Doxyfile

compile: $(OBJECTS)
    $(CC) $(CFLAGS) -o gereVendas $(OBJECTS)

obj/%.o: src/%.c
    @mkdir -p obj
    $(CC) $(CFLAGS) -o $@ -c $<



run: $(OBJECTS)
    $(CC) $(CFLAGS) -o gereVendas $(OBJECTS)
    ./gereVendas

clean:
    rm -f gereVendas
    rm -f debug
    rm -f $(OBJECTS)

debug: $(OBJECTS)
    $(CC) $(OBJECTS) -g -o debug
    gdb debug

.PHONY: doc
doc:$(OBJECTS)
        doxygen $(DOC)
```

4.1 Estrutra

A Makefile permite-nos compilar todo o nosso programa para que seja possível executá-lo.
Como podemos verificar pela figura, estámos a guardar na variável *CC* o compilador que estámos a utilizar (*gcc*), seguida da variável *CFLAGS* onde adicionámos todas as flags utilizadas por opção ao compilar. Seguido destas temos *OBJECTS* onde guardamos todos os ficheiros objectos que vamos utilizar, e por fim em *DOC* o Doxyfile necessário ao gerar a Documentação.
Está também presente no Makefile as dependências de cada ficheiro objecto gerado.

4.2 Comandos disponíveis

- **make ou make compile**

  Comando default do Makefile para compilar o nosso programa.

- **make run**

  Comando para compilar e de seguida executar o programa. No caso deste já estar compilado e não houver qualquer alteração ao compilar novamente, apenas executa o programa.

- **make clean**

  Comando para remover o executável, o executável com informação para o debugger e os objectos, criados no acto de compilação.

- **make debug**

  Comando para compilar o programa gerando um executável com informação para o debugger, o qual de seguida é executado no *gdb*.

- **make doc**

  Comando para gerar a documentação feita no código do programa.

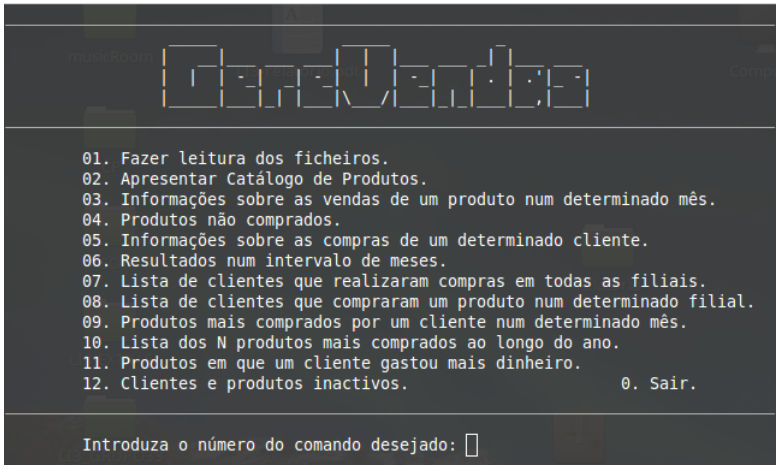
4.3 Grafo de dependências

```
obj/main.o: src/headers/CatClients.h src/headers/CatProducts.h src/headers/facturacao.h \
            src/headers/interpretador.h src/headers/filial.h
obj/avl.o: src/headers/avl.h
obj/CatClients.o: src/headers/CatClients.h src/headers/avl.h
obj/CatProducts.o: src/headers/CatProducts.h src/headers/avl.h
obj/facturacao.o: src/headers/facturacao.h src/headers/Sales.h src/headers/avl.h
obj/filial.o: src/headers/filial.h src/headers/avl.h src/headers/CatClients.h \
              src/headers/CatProducts.h src/headers/Sales.h
obj/heap.o: src/headers/heap.h
obj/interpretador.o: src/headers/interpretador.h src/headers/CatClients.h src/headers/CatProducts.h \
                     src/headers/filial.h src/headers/facturacao.h src/headers/readFiles.h \
                     src/headers/queries.h src/headers/listaStrings.h
obj/listaStrings.o: src/headers/listaStrings.h
obj/queries.o: src/headers/queries.h src/headers/facturacao.h src/headers/filial.h src/headers/heap.h \
               src/headers/listaStrings.h
obj/readFiles.o: src/headers/readFiles.h src/headers/CatClients.h src/headers/CatProducts.h \
                 src/headers/facturacao.h src/headers/filial.h
obj/Sales.o: src/headers/Sales.h src/headers/CatClients.h src/headers/CatProducts.h
```

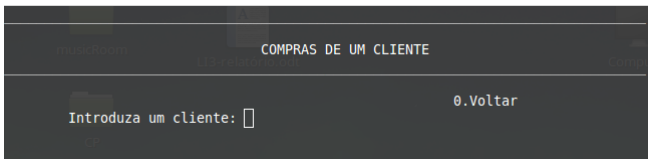
O grafo de dependências encontra-se em anexo na última página deste documento.

5. *Interface para o utilizador*

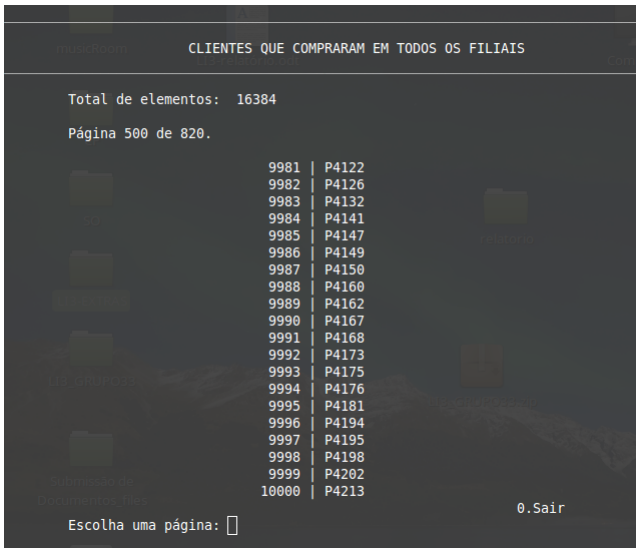
Quando o utilizador executa o nosso programa, a primeira coisa que lhe é apresentada é um Menu, onde estão presentes todas as funcionalidades do nosso programa. Este não pode, no entanto, aceder a qualquer uma destas sem antes utilizar o comando *1* responsável pela leitura dos ficheiros, conseguindo com isto armazenar dados nas estruturas para posteriormente utilizar as diferentes funcionalidades do nosso programa.



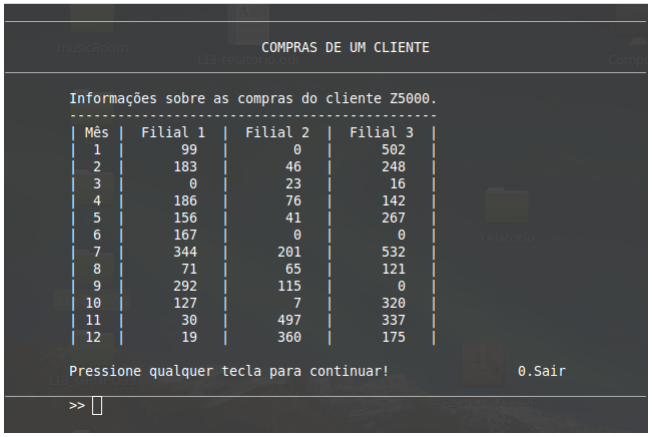
A nossa interface recebe grande parte das vezes números como comandos, é exemplo disso o comando *0* que funciona sempre como um comando de Voltar/Sair. Existem no entanto exeções a esta regra em algumas das queries presentes na interface, quando pedem um Cliente/Produto em específico, por exemplo.

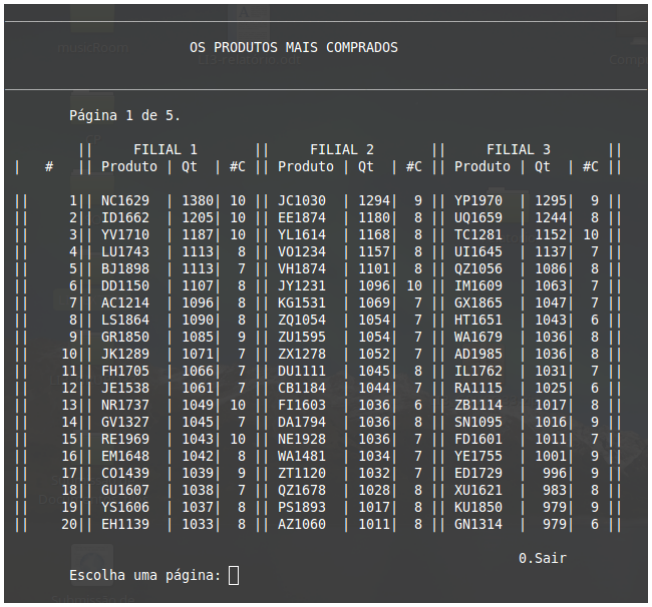


Tendo em conta o facto de que os comandos presentes na interface são, salvo raras excepções, números, decidimos que seria confortável para o utilizador ter uma apresentação por páginas, com a opção do utilizador escolher especificamente uma página do catálogo que lhe é apresentado, em vez da utilização de outro tipo de comandos que fizessem avançar/recuar apenas uma página.



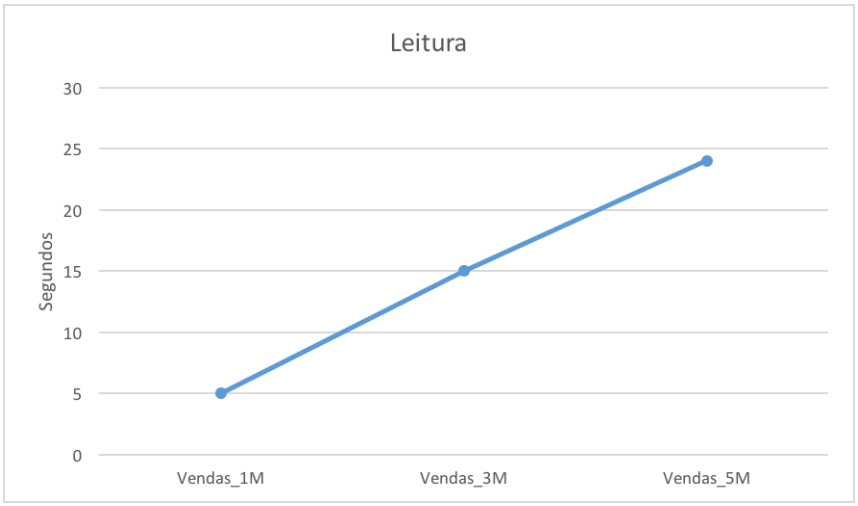
Houve ainda outras queries em que a informação era demasiada, e para conforto do utilizador esta informação teve de ser bem organizada com a utilização de tabelas.

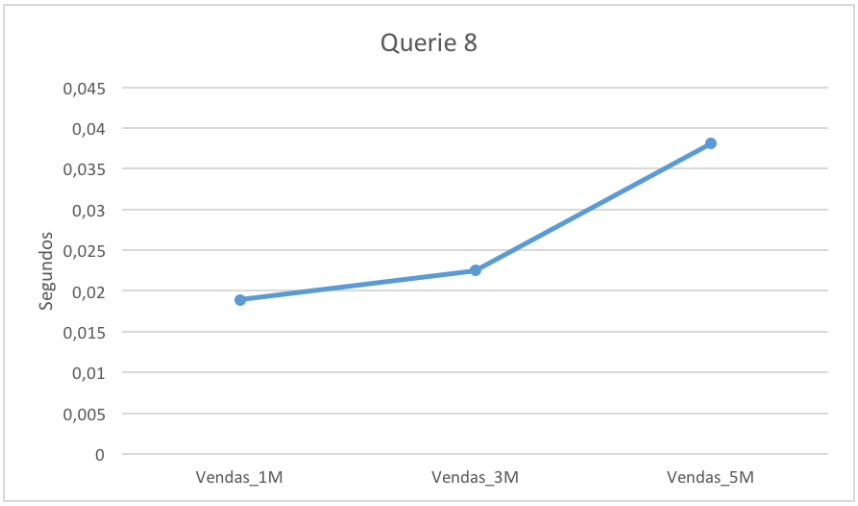


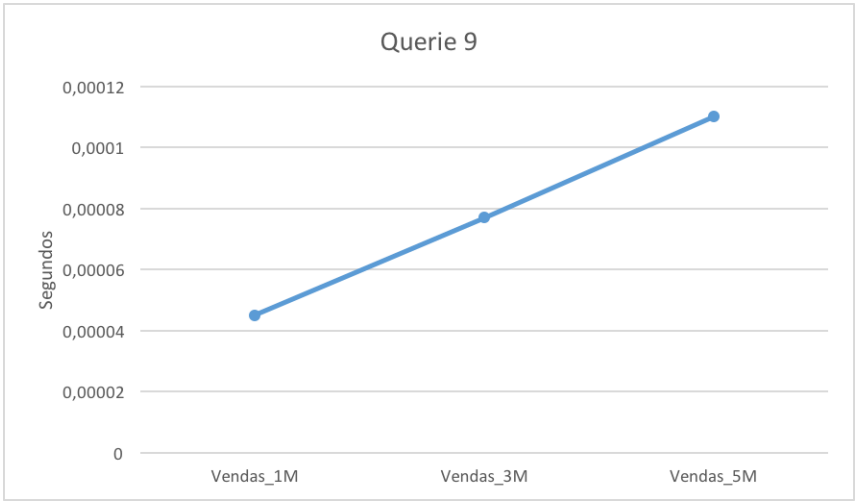


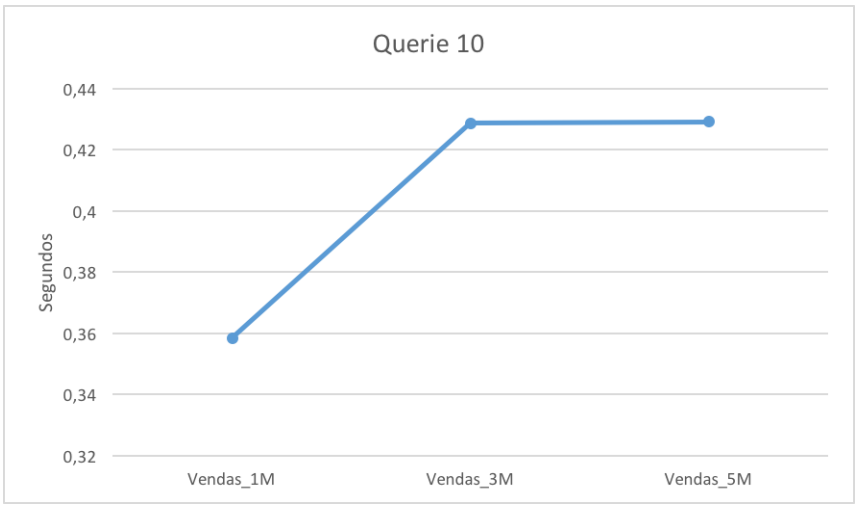
Toda esta interface para o utilizador, apesar de simples, foi feita por nós de maneira a que o utilizador podesse tirar o maior proveito da mesma, com comandos simples e fáceis de memorizar. Foi também preocupação nossa fazer com que a interface fosse fácil para o utilizador se acostumar, com pouco tempo de utilização.

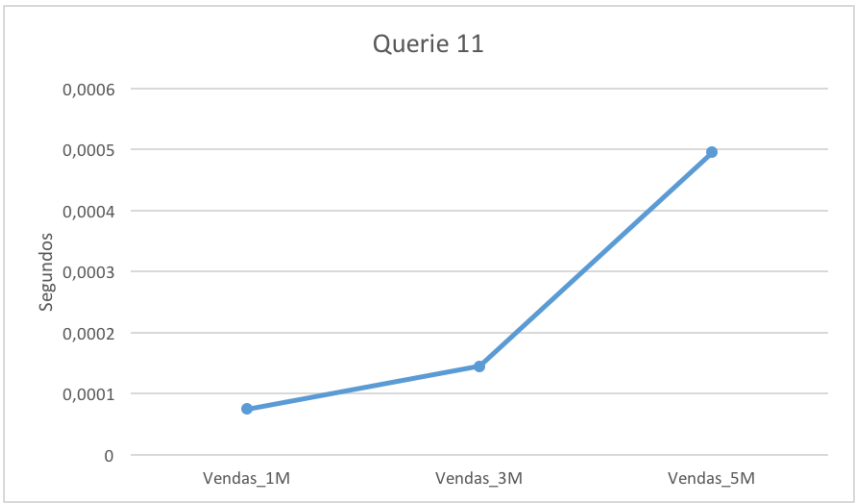
6. *Testes de Performance*

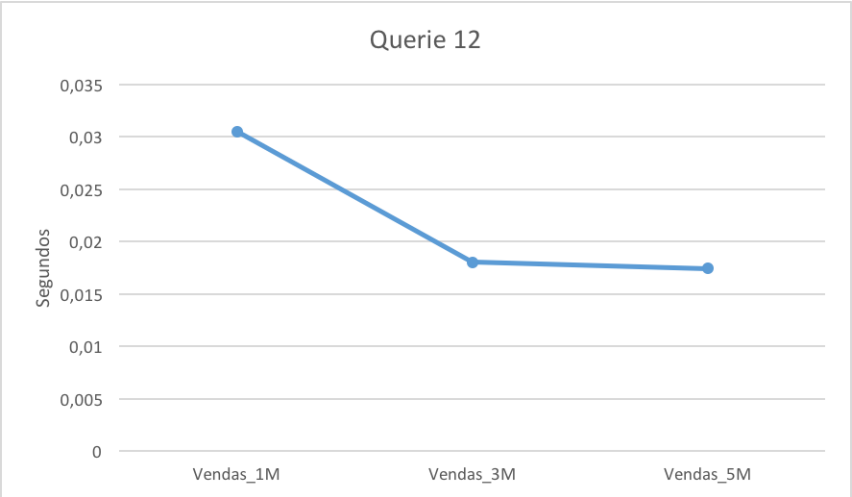





Querie 9
0,00012
0,0001
0,00008
0,00006
0,00004
0,00002
0
Segundos
Vendas_1M
Vendas_3M
Vendas_5M


Querie 10
0,44
0,42
0,4
0,38
0,36
0,34
0,32
Segundos
Vendas_1M
Vendas_3M
Vendas_5M

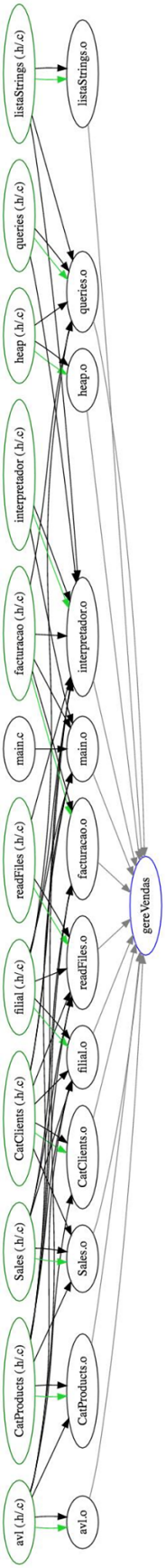

Querie 11
0,0006
0,0005
0,0004
0,0003
0,0002
0,0001
0
Segundos
Vendas_1M
Vendas_3M
Vendas_5M



Os gráficos apresentados, são resultados de testes de performance com os ficheiros Vendas_1M.txt, Vendas_3M.txt e Vendas_5M. Os testes realizados são dirigidos à leitura dos ficheiros, e também à execução das queries 8,9,10,11 e 12, para os diferentes ficheiros.
Como podemos verificar, grande parte dos gráficos apresenta um acréscimo com o aumento do número de vendas, o que é esperado, no entanto a querie 10 e 12 não obtiveram essa variação.
Na querie 10, apartir de 3 milhões de vendas, o tempo de execução praticamente não varia. Este facto deve-se ao facto de ao fim de inserir 3 milhões de vendas, a estrutura de Produtos da filial, fica com os nodos todos preenchidos, o que torna o tempo de acesso constante apartir deste valor de vendas, para a querie 10.
Na querie 12 a razão da diminuição do tempo de execução com o aumento do número de vendas, deve-se ao facto do número de produtos não comprados e clientes que nunca compraram diminuir, logo o tempo para calcular os seus valores é menor.

7. Conclusão

Este foi o projecto que até ao momento gerou mais dificuldades a todos os elementos do grupo. Ao longo da realização do mesmo fomos confrontados com situações novas que puxaram pelo nosso espírito de autonomia, só com este e muito trabalho conseguimos ultrapassar estas dificuldades e poder apresentar um trabalho do qual nos orgulhamos.
Nos módulos dos catálogos (clientes e produtos) foi necessária a implementação de AVLs, algo que tinhamos abordado em Algoritmos e Complexidade, e que não foi de todo uma das nossas grandes dificuldades.
O mesmo não aconteceu nos dois restantes módulos, em que necessitamos de construir estruturas que tornassem a leitura dos dados o mais eficiente possível. Estas tiveram de ser bem pensadas, e mesmo depois de aplicadas foram alteradas diversas vezes até ao resultado final em que tentámos ao máximo manter os baixos tempos de leitura dos dados conciliados com os tempos praticamente instantâneos de todas as queries propostas.
O facto de haver a necessidade de utilizar tipos opacos foi também uma das nossas grandes dificuldades, algo novo para nós e que nos obrigou a estruturar o trabalho de maneira completamente distinta do que estavamos habituados até então, como por exemplo a clonagem de uma string/estrutura antes de a retornar, caso contrário o utilizador poderia ter acesso à mesma pelo apontador.
Todas estas dificuldades foram, no entanto, ultrapassadas tanto com a aplicação de matéria lecionada em cadeiras anteriores como com o trabalho em equipa do grupo.



Grafo de dependências