Escola de Engenharia
Universidade do Minho

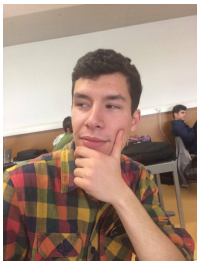
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Laboratórios de Informática III

Gestão de Vendas de uma cadeia de Distribuição com 3 filiais
GEREVENDAS

Grupo 84


Célia Figueiredo
a67637


Gil Gonçalves
a67738


Humberto Vaz
a73236


Ricardo Lopes
a72062

Braga, 28 de Abril de 2016

1. Introdução

No âmbito da unidade curricular de Laboratórios de Informática III do 2ºano do curso de MIEI, foi proposto o desenvolvimento de um projeto em linguagem C que tem por objetivo ajudar à consolidação dos conteúdos teóricos e práticos e enriquecer os conhecimentos adquiridos nas UCs de Programação Imperativa, de Algoritmos e Complexidade, e da disciplina de Arquitetura de Computadores.

O Projeto, denominado GereVendas, baseia-se num programa de gestão de hipermercados com 3 filiais que dependem de uma lista de clientes, uma lista de produtos e uma lista de vendas efetuadas. Cada uma destas listas estará num ficheiro .txt e para cada um dos ficheiros o programa percorre o ficheiro, executando operações que permitam guardar estes dados em memória. Para ajudar nesta tarefa repartir-se-á as tarefas em quatro módulos módulos. Estes módulos são: um catálogo de clientes; um catálogo de produtos; um módulo de faturação global; um módulo de gestão de filial.

De forma a preservar o encapsulamento de dados será disponibilizada uma API de forma a que o utilizador apenas possa aceder através destas funções públicas. Depois dos ficheiros serem carregados o utilizador será capz de executar uma lista de queries previamente fornecida pela equipa docente. Para responder às diferentes queries utilizam-se as funções definidas nas API dos diferentes módulos.

Este projeto considera-se um grande desafio, pelo facto de passarmos a realizar programação em grande escala, uma vez que se tratam de grandes volumes de dados e por isso uma maior complexidade. Nesse sentido, o desenvolvimento deste programa será realizado à luz dos princípios da modularidade (divisão do código fonte em unidades separadas coerentes) e do encapsulamento (garantia de proteção e acessos controlados aos dados).

Conteúdo

2. Descrição dos Módulos

A arquitetura da aplicação a desenvolver é definida por quatro módulos principais: Catálogo de clientes, Catálogo de produtos, Faturação Global e Vendas por Filial, cujas fontes de dados são três ficheiros de texto detalhados abaixo.

No ficheiro **Produtos.txt** cada linha representa o código de um produto vendável no hipermercado, sendo cada código formado por duas letras maiúsculas e 4 dígitos (que representam um inteiro entre 1000 e 1999), como no exemplo:

```
AB9012
XY1185
BC9190
```

O ficheiro de produtos contém cerca de 200.000 códigos de produto.

No ficheiro **Clientes.txt** cada linha representa o código de um cliente identificado no hipermercado, sendo cada código de cliente formado por uma letra maiúscula e 4 dígitos que representam um inteiro entre 1000 e 5000, segue um exemplo:

```
F2916
W1219
F2915
```

O ficheiro de clientes contém cerca de 20.000 códigos de cliente.

O ficheiro **Vendas_1M.txt**, no qual cada linha representa o registo de uma venda efectuada numa qualquer das 3 filiais da Cadeia de Distribuição. Cada linha (a que chamaremos compra ou venda, o que apenas depende do ponto de vista) será formada por um código de produto, um preço unitário decimal (entre 0.0 e 999.99), o número inteiro de unidades compradas (entre 1 e 200), a letra **N** ou **P** conforme tenha sido uma compra **Normal** ou uma compra em **Promoção**, o código do cliente, o mês da compra (1 ... 12) e a filial (de 1 a 3) onde a venda foi realizada, como se pode verificar nos exemplos seguintes:

```
KR1583 77.72 128 P L4891 2 1
QQ1041 536.53 194 P X4054 12 3
OP1244 481.43 67 P Q3869 9 1
JP1982 343.2 168 N T1805 10 2
IZ1636 923.72 193 P T2220 4 2
```

O ficheiro de vendas inicial, **Vendas_1M.txt** , conterá 1.000.000 (1 milhão) de registos de vendas realizadas nas 3 filiais da cadeia de distribuição. Existirão também os ficheiros **Vendas_3M.txt** e **Vendas_5M.txt** utilizados para as questões de performance da aplicação.

A aplicação possuiu uma arquitectura tal como apresentado na figura seguinte, em que se identificam as fontes de dados, a sua leitura e os módulos de dados a construir:

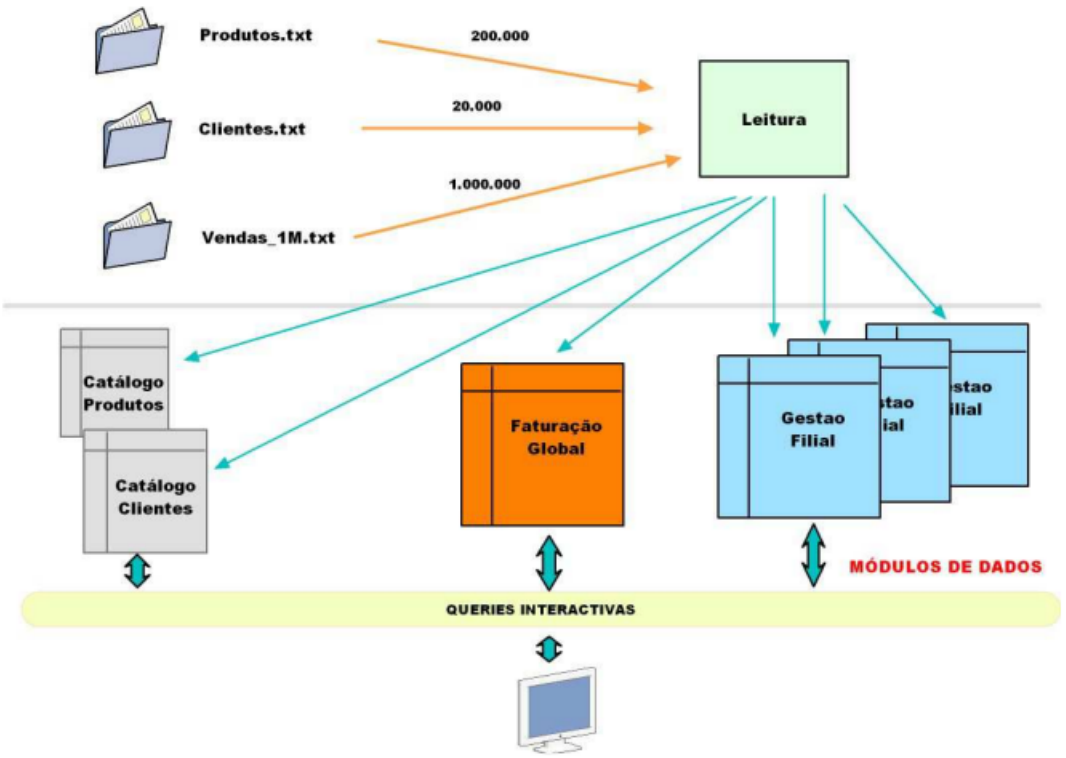


Figura 2.1: Arquitetura da aplicação

2.1 Catálogo de Clientes

É o módulo de dados onde são guardados os códigos de todos os clientes do ficheiro **Clientes.txt**, organizados por índice alfabético;

2.1.1 Clientes.h

Módulo de dados onde são guardados os códigos de todos os clientes do ficheiro **Clientes.txt**. O array de árvores este que é um array de 26 posições cujos índices se encontram organizados alfabeticamente. Cada índice contém um apontador para uma árvore correspondente à letra respetiva desse índice.

Tipos Opacos

```
typedef struct catalogo_clientes *CatClientes;
```

O typedef no ficheiro Clientes.h é a única informação que o utilizador têm relativamente à implementação de dados, não tendo acesso ao ficheiro .c dos clientes, não conseguindo conhecer a verdadeira implementação da estrutura AVL. Deste modo garantimos o encapsulamento de dados e a única forma de o utilizador interagir com o catálogo de clientes será através da API;

clientes.c

```
struct catalogo_clientes{
ARVORE indices[27];
};
```

Utilizamos um array de árvores para guardar os clientes, pois seria mais fácil a procura pelos mesmos. Cada indice do array corresponde a uma letra do alfabético, e quando procurarmos um determinado cliente já sabemos o indice onde ele se encontra, o que irá torna a procura muito mais rápida.

Usar AVLs executa as operações de inserção, busca e remoção em tempo O(log n), sendo rápida para aplicações que fazem uma quantidade excessiva de procuras, porém esta estrutura é um pouco mais lenta para inserção e remoção. Isso deve-se ao facto de as árvores AVL serem rigidamente balanceadas.

/*API*/

- **CatClientes inicializa_catalogo_clientes()** - A funcao inicializa_catalogo_clientes inicializa o modulo (array de 27 posicoes de avls), pondo estes 27 indices a null*;
- **void insertC(CatClientes c, char * valor)**- A funcao insertC insere um cliente na sua avl respectiva, isto é, a posicao do array a que corresponde a letra inicial;
- **void cat_remove_cliente(CatClientes cat, char *str)** - A funcao cat_remove_cliente remove da respectiva avl o codigo de cliente passado por argumento, fazendo free ao nodo onde este situava;
- **void free_catalogo_Clientes(CatClientes cat)** - A funcao free_catalogo_Cliente apaga as 27 avls, fazendo o respectivo free;
- **int existeCliente (char *cliente,CatClientes cat)** - A funcao existeCliente verifica se um codigo existe na respectiva avl;
- **int numeroClientes(CatClientes cat)** - a funcao numeroClientes conta quantos clientes existem nas avls, usando para isto a funcao generica do modulo avl, avl_count;
- **int numeroClientesLetra(CatClientes cat, char letra)** - a funcao numeroClientesLetra conta quantos clientes existem comecados por uma determinada letra, para isto só são contados os nodos da avl a que corresponde a posição dessa letra.

2.2 Catálogo de Produtos

Módulo de dados onde são guardados os códigos de todos os produtos do ficheiro **Produtos.txt**, organizados por índice alfabético, o que irá permitir, de forma eficaz, saber quais são os produtos cujos códigos começam por uma dada letra do alfabeto e saber quantos produtos são contabilizados.

2.2.1 Produtos.h

Tipos Opacos

```
typedef struct catalogo_produtos *CatProdutos;
```

Este typedef é a única informação que o utilizador tem relativamente à implementação de dados, não tendo acesso ao ficheiro .c dos produtos não consegue conhecer a verdadeira implementação da estrutura AVL. Assim, garantimos o encapsulamento de dados e a única forma de o utilizador interagir com o catálogo de produtos será através da API.

produtos.c

```
struct catalogo_produtos{
ARVORE indices[27];
};
```

Utilizamos um array de árvores para os produtos, pois seria mais fácil a procura pelos mesmos. Cada indice do array corresponde a uma letra do alfabético, e quando procurarmos um determinado produto já sabemos o indice onde ele se encontra, o que irá torna a procura muito mais rápida.

/*API*/

- **CatProdutos inicializa_catalogo_produtos()** - A funcão inicializa_catalogo_produtos aloca memória para o respetivo catalogo de todos os produtos e cria 27 AVLs para cada letra dos produtos através da funcão avl_create;
- **void insertP(CatProdutos c, char * valor)** - A funcão insertP insere na estrutura catalogo de produtos o id do novo produto;
- **void cat_remove_produto(CatProdutos cat, char *str)** - A funcão cat_remove_produto elimina e faz free ao nodo do produto em questão (utilizando a funcao avl_delete) e de seguida faz o respetivo free;
- **void free_catalogo_produtos(CatProdutos cat)** - A funcão free_catalogo_produtos destrói os dados do catalogo de produtos (cat) um a um (fazendo libavl_free) e no final liberta a memória do catálogo;
- **int existeProduto (char *produto,CatProdutos cat)** -A funcão existeProduto verifica se existe um produto (através do seu id) num catálogo (ambos passados por parâmetro) se existe devolve 1 se não 0;
- **int numeroProdutos(CatProdutos cat)** - A funcão numeroProdutos faz a soma de todos os nodos de cada avl, uma a uma. Ou seja através de um ciclo for e da funcão avl_count iremos obter a soma total de produtos das 27 avl's;
- **int numeroProdutosLetra(CatProdutos cat, char letra)** - A funcão numeroProdutos-Letra com o auxilio da funcão avl_count, dado um catalogo e letra ambos passados por parâmetro, calcula o nr de nodos da avl associada a essa letra;

- **ARRAY listaProdutosLetra(CatProdutos cat, char l)**- A funcão listaProdutosLetra, dado uma letra e um catalogo em parâmetro, através de um traverser associada à estrutura avl dessa letra, vai inserindo num array a, todos os ids de produtos começados por essa letra. Fazendo o respetivo free do traverser;

2.3 Faturação Global

Módulo de dados que contém as estruturas de dados responsáveis pela resposta a questões quantitativas que relacionam os produtos às suas vendas mensais, em modo Normal (N) ou em Promoção (P), para cada um dos casos guardando o número de vendas e o valor total de faturação de cada um destes tipos. Este módulo refecencia todos os produtos, mesmo os que nunca foram vendidos, não contém qualquer referência a clientes, mas é capaz de distinguir os valores obtidos em cada filial.

2.3.1 faturacao.h

Tipos Opacos

```
typedef struct faturacao *Faturacao;
typedef struct info *Info;
```

faturacao.c

```
struct faturacao{
int totalvendas[12];
float totalfaturado[12];
ARVORE produtos;
};
```

A estrutura facturação irá conter o total de vendas realizado numa filial, assim como o dinheiro facturado nessa filial; irá também conter a informação sobre os produtos que foram vendidos;

```
struct info{
char *code;
int vendasP[12][3];
int vendasN[12][3];
float faturadoN[12][3];
float faturadoP[12][3];
int quantidadeP[12][3];
int quantidadeN[12][3];
};
```

A estrutura info irá conter a informação do produtos comprados, utilizamos uma matriz 12 por 3 pois a procura é feita maioritariamente em meses, logo irá ser mais rápido a procurar. A estrutura terá informação do número de vendas e o seu valor distinguido por normal ou promoção.

/*API*/

- **Faturacao inicializa_faturacao()** - A funcão inicializa_faturacao aloca uma estrutura Faturacao e cria uma avl para os produtos, bem como inicializa os 12 nodos totalvendas e totalfaurado a zero;

- **void cont_regista_produto(Faturacao fat, char *prod)** - A funcão cont_regista_produto, dada a estrutura faturacão e um id de produto, insere-o através da funcão avl_insert;

- **void cont_insere_venda(Faturacao fat, char *produto, int q, float preco, char M,int mes, int filial)** - A funcão cont_insere_venda, dada a estrutura faturacão, atualiza as vendas, quantidades e faturado conforme o produto em questão ser Promocão (P) ou Normal (N);

- **void cont_remove_produto(Faturacao fat, char *produto)** - A funcão cont_remove_produto remove um produto da Faturacao passados ambos por parametro. Inicialmente calcula o id do produto através da funcão fat_procura_info para depois eliminar o produto em questão através da funcão avl_delete e logo a seguir faz free ao nodo eliminado através da funcão free_info;

- **void free_faturacao(Faturacao fat)**- A funcão free_faturacao elimina todos os nodos da faturacao através da funcão avl_destroy e faz free à estrutura faturacao passada por parâmetro;

- **float getTotalFatPFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getTotalFatPFilialX vai calcular o total faturado no modo Promocão (P) de um dado produto, num determinado mês e em determinada filial;

- **float getTotalFatNFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getTotalFatNFilialX vai calcular o total faturado no modo Normal (N) de um dado produto, num determinado mes e em determinada filial;

- **int getVendasNFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getVendasNFilialX calcula o nr de vendas em modo Normal (N) de um produto num determinado mês e filial. Inicialmente vai buscar o produto através do seu id, se o encontrar devolve o nr de vendas nas condicões anteriores;

- **int getVendasPFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getVendasPFilialX calcula o nr de vendas em modo Promocao (P) de um produto num determinado mês e filial. Inicialmente vai buscar o produto através do seu id, se o encontrar devolve o nr de vendas nas condicões anteriores;

- **int getQuantidadeNFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getQuantidadeNFilialX calcula a quantidade vendida em modo Normal (N) de um determinado produto num dado mês e filial;

- **int getQuantidadePFilialX (char* prod,int mes,Faturacao fat, int filial)** - A funcão getQuantidadePFilialX calcula a quantidade vendida em modo Promocão (P) de um determinado produto num dado mês e filial. Inicialmente vai buscar o nodo do produto através do id do produto e retorna finalmente a quantidade vendida;

- **ARRAY naoCompradosFilial(Faturacao fat, int filial)** - A funcão naoCompradosFilial percorre a estrutura da faturacao numa dada filial em todos os meses através do Traverser t, e verificia se este foi ou não comprado,se foi nada faz, se não insere-o num array a. No final faz free ao Traversser t e retorna o novo array;

- **ARRAY naoComprados(Faturacao fat)** - A funcão naoComprados percorre a estrutura da faturacao em todas as filiais e em todos os meses através do Traverser t, e verificia se este foi ou não comprado,se foi nada faz, se não insere-o num array a. No final faz free ao Traversser t e retorna o novo array;

- **int totalVendasMeses(Faturacao fat, int a, int b)** - A funcão totalVendasMeses calcula o total de vendas num determinado intervalo de meses;

- **float totalFatMeses(Faturacao fat, int a, int b)** - A funcão totalFatMeses calcula o total faturado num dado intervalo de meses;

- **ARRAY nMaisVendidos(Faturacao fat, int n)** - A funcão nMaisVendidos calcula os n produtos mais vendidos de todas as filiais através de um traversser t. Inicialza dois arrays a e b através da funcão inicializa_array . Copia os elementos do tipo Info para o array a, ordena-o através da funcão ordena e depois devolve o array b com os elementos ids dos produtos do array já ordenados;

- **int getQuantidadeFilial(Faturacao fat, char*prod, int filial)** - A funcão getQuantidadeFilial calcula a quantidade vendida de um determinado produto através do seu id de produto numa determinada filial em todos os meses.

2.4 Gestão da Filial

Módulo de dados que, a partir dos ficheiros lidos, contém as estruturas de dados adequadas à representação dos relacionamentos, fundamentais para a aplicação, entre produtos e clientes, ou seja, para cada produto, saber quais os clientes que o compraram, quantas unidades cada um comprou, o mês e a filial.

Para a estruturação optimizada dos dados deste módulo de dados tivemos em atenção que pretendemos ter o histórico de vendas organizado por filial para uma melhor análise, nunca esquecendo que existem 3 filiais nesta cadeia.

2.4.1 Filial.h

Tipos Opacos

```
typedef struct filial *Filial;
typedef struct icliente *Icliente;
typedef struct iprodutos *Iprodutos;
```

Filial.c

```
struct filial{
ARVORE infoCliente;
};
```

A estrutura filial tem a informação de todos os clientes que compraram numa dada filial;

```
struct icliente{
char *cliente;
int quantidade[12];
ARVORE infoprodutos[2];
};
```

A estrutura icliente irá ter o codigo do cliente válido, a quantidade dos produtos que comprou em cada mês e a informação dos produtos que comprou, ou de forma normal, ou de forma promocional;

```
struct iprodutos{
char *prod;
int quantidadeT;
float gastouT;
int quantidade[12];
float gastou[12];
};
```

A estrutura iprodutos terá um código de produtos, a quantidade total comprada, o total de dinheiro gasto, e irá ter também a quantidade desse produto comprado em vários meses, assim como o dinheiro que gastou por mês nesse produto;

/*API*/

- **Filial inicializa_filial()** - Inicializa a estrutura da filial;
- **void fil_regista_cliente(Filial fil, char *cliente)** - Insere um cliente na filial;
- **void fil_insere_prod(Filial fil, char *cliente, char *produto,int q, int mes, float preco, char p)** - A funcão fil_insere_prod insere um produto comprado por um dado cliente num dado mês, se o produto já tiver sido inserido a quantidade =quantidadeAntiga+quantidadeComprada, aumenta também o dinheiro gasto nesse produto. Senão tiver sido inserido a quantidade=quantidadeComprada e o preco=precoComprado;
- **int getQuantidadeMesCliente(Filial fil, char *cliente, int mes)** - A funcão getQuantidadeMesCliente retorna a quantidade dos produtos comprados por um dado cliente, num dado mês. Para tal, procura um cliente numa dada filial através da funcão fil_procura_cliente, retornando a quantidade comprada desse mês;
- **ARRAY naoCompraram(Filial fil)** - A função naoCompraram utiliza as funções inicializa_array e o avl_t_alloc, para alocar espaço ao array e para o traverser,que é uma estrutura que contém um apontar para o inicio, contem também um apontador para a árvore onde nos encontramos e uma stack com os restantes elementos,depois usamos a função avl_t_init, para inicializar o traverser com toda a informacao dos clientes que estão na estrutura Filial, depois com a ajuda do função avl_t_next percorremos o traverser e à medida que encontramos um cliente vemos se ele comprou ou não, se ele não tiver comprado nada, inserimos esse cliente num array dinâmico, no fim returnámos esse mesmo array;

- **ARRAY compraram(Filial fil)** - A função compraram utiliza avl_t_alloc para alocar um traverser e inicializa_array para inicializar um array dinâmico depois utiliza o avl_t_next para percorrer p traverser e sempre que encontra um cliente soma as quantidades de todos os meses e depois compara se a quantidade que ele comprou é maior que zero, se for ele inser no array dinamico no final retorna o array dinâmico

- **void clientesCompraram(Filial fil,ARRAY a)** - A funcão clientesCompraram remove do array todos os clientes que não compraram nenhum produto numa filial passada por parâmetro. Para isso cria um Traversser t percorredo a estrutura dos clientes, somando a quantidade comprada em cada mês, se esta for zero então remove o cliente do array através da funcão remove_posicão;

- **void free_filial(Filial fil)** - A função free_filial elimina todos os nodos da filial através da funcão avl_destroy e faz free à estrutura filial passada por parâmetro;

- **ARRAY topMaisGastou(ARRAY a)** - A funcão topMaisGastou irá calcular os 3 produtos que um cliente mais gastou retornando-os num array. Para isso recebe um array de Produtos comprados por um dado cliente, ordena esse array a (passado por parâmetro) e insere as chaves num array b, retornando-o;

- **ARRAY clientesCompraramProduto(Filial fil, char* produto)**- A função clientesCompraramProduto utiliza inicializa_iprodutos,inicializa_array e avl_t_alloc, para alucar espaço a um produto, array e a um traverser, depois com o avl_t_init pasa a informacao dos clientes de uma filial para o traverser e depois percorrer o traverser, quando encontra uma cliente que tenha comprado um produto inser o cliente no array dinamico. No fim retorna uma lista de clientes que compraram um produto numa dada filial;

- **int comprouProdutoN(Filial fil, char* cliente, char* produto)** - A função comprouProdutoN recebe com argumentos uma filial, cliente e um produto,ambos válidos,inicializa_icliente, inicializa_iprodutos para inicializar um cliente e um produtos, depois utiliza o avl_find para encontrar o cliente e o produto dentro dos produtos que o cliente comprou, se o cliente não tiver comprado esse produto em normal retorna zero, se o tiver comprado retorna 1;

- **int comprouProdutoP(Filial fil, char* cliente, char* produto)** - A função comprouProdutoP recebe com argumentos uma filial, cliente e um produto,ambos válidos,inicializa_icliente, inicializa_iprodutos para inicializar um cliente e um produtos, depois utiliza o avl_find para encontrar o cliente e o produto dentro dos produtos que o cliente comprou, se o cliente não tiver comprado esse produto em promoção retorna zero, se o tiver comprado em promoção retorna 1;

- **int getNumClientesFilial(Filial fil, char* produto)** - A função getNumClientesFilial utiliza inicializa_iprodutos,vl_t_alloc para alucar memorica a um produto e ao TRAVERSER, depois utiliza o avl_t_init para inserir a informação dos clientes no TRAVERSER, depois a medida que percorre o traverser verifica se o cliente comprou o produto em normal , ou em promoção, se ele comprou incrementa a variavel n. No fim retorna essa variável;

- **void getIProdMes(Filial fil, char* cliente, int mes, ARRAY a)** - A função getIProdMes utiliza as funções inicializa_icliente,avl_t_alloc para alocar memoria para um dado cliente e para o traverser, depois procura esse cliente na filial e quando o encontra retorna-o,

depois a função avl_t_init inicializa o traversar com os produtos comprados em modo normal, depois percorre o traverser quando encontra um produto faz uma copia da informação desse produto e vai buscar a possição onde ele se encontra, se ele ainda não tiver sido inserido, inser o produto, senão actualiza a quantidade do produto para esse mes. Depois irá fazer o mesmo para os produtos comprados em promoção;

- **ARRAY extraiPorQuantidade(ARRAY a,int mes)**- A função extraiPorQuantidade utilizada o inicializa_array para incializar o array b, depois utiliza a funcao ordena para ordenar os produtos por quantidade de um dado mes, depois cria uma copia do codigo do produto e insere ordenado no array b e retorna esse mesmo array;

- **void removeNaoCompraram(Filial fil, ARRAY a)** - A função removeNaoCompraram recebe como parametro um array dinâmico e uma filial, com informação lá dentro, depois com a função get_tamanho vai buscar o tamanho de um array, depois com o inicializa_icliente inicializa um cliente que irá ser retirado do array,de seguida com a função avl_find encontra esse cliente na filial e retorna a informação desse cliente. Por fim percorre soma a quantidade comprada dos meses todos, se essa quantidade for igual a zero remove do array dinâmico;

- **void removeCompraram(Filial fil, ARRAY a)**- A função removeCompraram, dado um array a com informacão dos clientes, recorre à funcao inicializa_icliente para inicializar o cliente da posição i, depois usa o avlfind para procurar esse cliente que foi inicializado e depois compara se ele comprou em normal ou em promocao, se ele tiver comprado remove do array.

3. Main

O programa é controlado pelo ficheiro leitura.c. Este que invoca as funções que estão inseridas no ficheiro querie.c este que carrega os ficheros de produtos, clientes e vendas, também é responsável pela interação com o utilizador.

- Menu - Uma função que imprime o menu ao utilizador, recebendo a opção.
- Funções de validação - As funções *existeCliente* e *existeProduto* que validam linha de vendas (querie 1).
- Carregar Vendas - Lê um ficheiro de vendas e carrega o módulo de vendas e faturação.
- Queries - Uma função para cada query
- Main - Carrega cada um dos módulos com um nome default de ficheiro. Após a colocação em memória, é chamada a função auxiliar querie, que servirá para selecionar a querie que o utilizador pretende executar. Escolhida a querie, com o auxílio de um switch temos acesso a funções que invocam a/as função/ões que faz exatamente o que a querie pede e mostra ao utilizador os resultados pretendidos, bem como para a alteração de ficheiros.

4. Interface do utilizador

Quando o utilizador executa o programa é-lhe pedido que escolha qual o documento de texto que pretende analisar, como podemos observar na figura seguinte:

Figura 4.1: Escolha do ficheiro de vendas a analisar

O ficheiro é carregado e de seguida aparece um menu com 12 opções, referentes às 12 queries do projeto, sendo que decidimos usar o [0] para sair do GereVendas. O objetivo é que o utilizador prima a tecla correspondente à opção do menu pretendida. E de seguida preencha os campos pedidos pela aplicação relativos à querie escolhida.

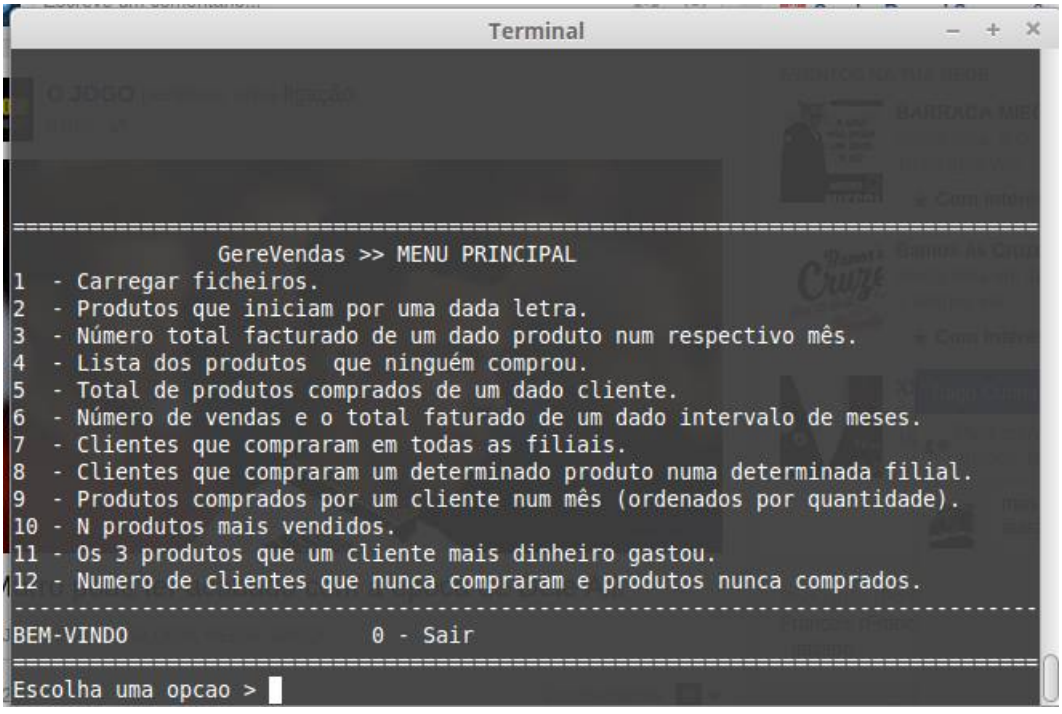


Figura 4.2: Menu principal da aplicação

5. Resultados e comentários sobre os testes de performance

Depois de desenvolver e codificar todo o projeto foi-nos proposto realizar alguns testes de performance que consistem em comparar os tempos de execução das queries 8, 9, 10, 11 e 12 usando os ficheiros Vendas_1M.txt (1000 000 vendas), Vendas_3M.txt (3 milhões de vendas) e Vendas_5M.txt (5 milhões de vendas). Uma vez que a quantidade de vendas vai aumentando de ficheiro para ficheiro é aceitável que os tempos de carregamento para os módulos aumente.

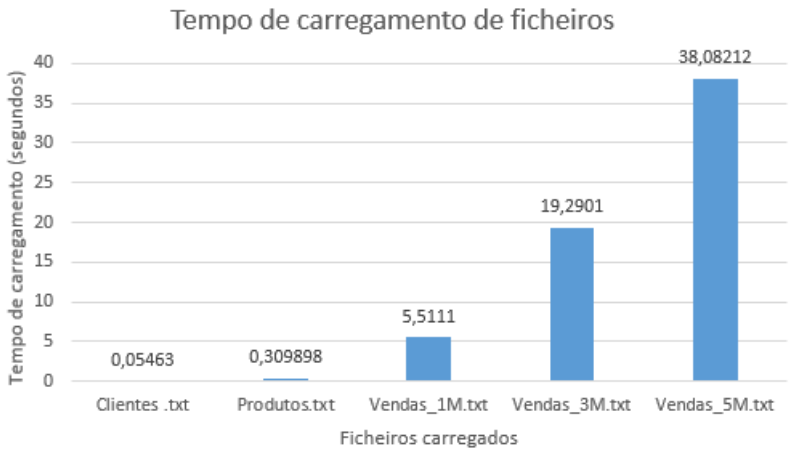


Figura 5.1: Gráfico do tempo de carregamento dos ficheiros de clientes, produtos e os 3 ficheiros de vendas

Verificou-se que os tempos de carregamento dos ficheiros Clientes.txt e Produtos.txt para os diferentes módulos mantiveram-se quase constantes.

Comparando os valores de execução das queries pretendidas, como podemos observar nos respetivos gráficos apresentados abaixo, verificamos que os tempos dos carregamentos dos módulos aumentam conforme o tamanho do ficheiro de vendas, este que era um resultado esperado.

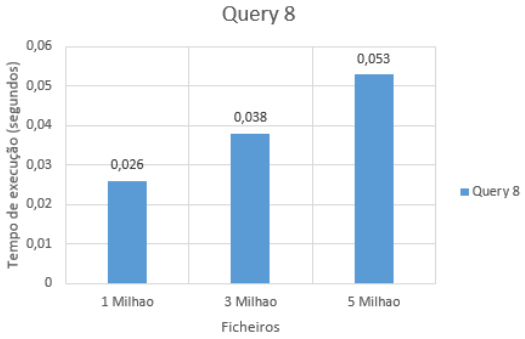


Figura 5.2: Tempos de execução da querie 8 para a filial 1 e o produto GI1298

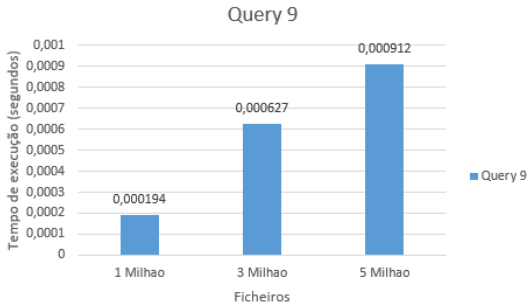


Figura 5.3: Tempos de execução da querie 9 para o cliente Z5000 para o mês 1

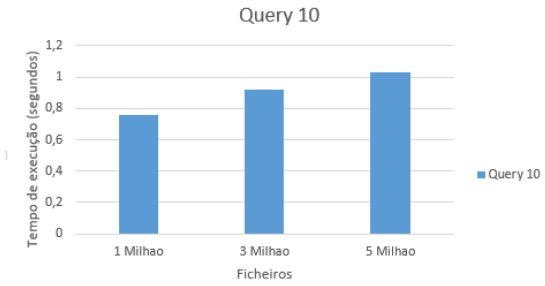


Figura 5.4: Tempos de execução da querie 10, para os 10 produtos mais vendidos

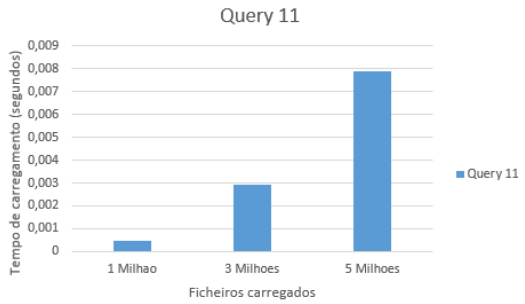


Figura 5.5: Tempos de execução da querie 11 para o cliente Z5000

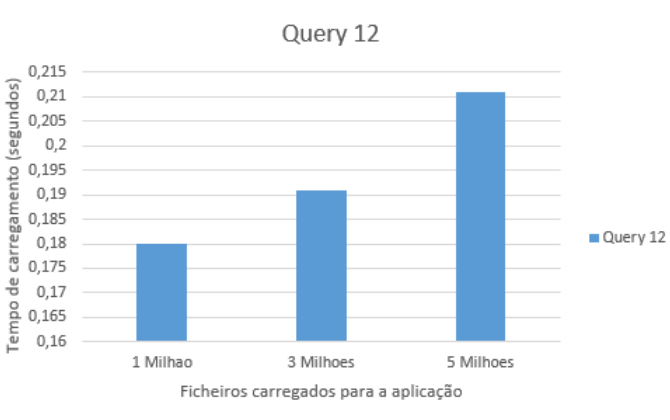


Figura 5.6: Gráfico do tempo de carregamento da querie 12 - nr de clientes que nunca compraram e produtos nunca comprados

6. Makefile e Grafo de dependências

A makefile permite correr todo o software escrevendo apenas *"make"* no terminal. Posto isto, apresenta-se a makefile utilizada cujas flags utilizadas como opção de compilação são –Wall –Wextra –ansi – pedantic –O2. Possui ainda a opção "make clean" que elimina todos os ".o" que foram criados quando se compilou o software.

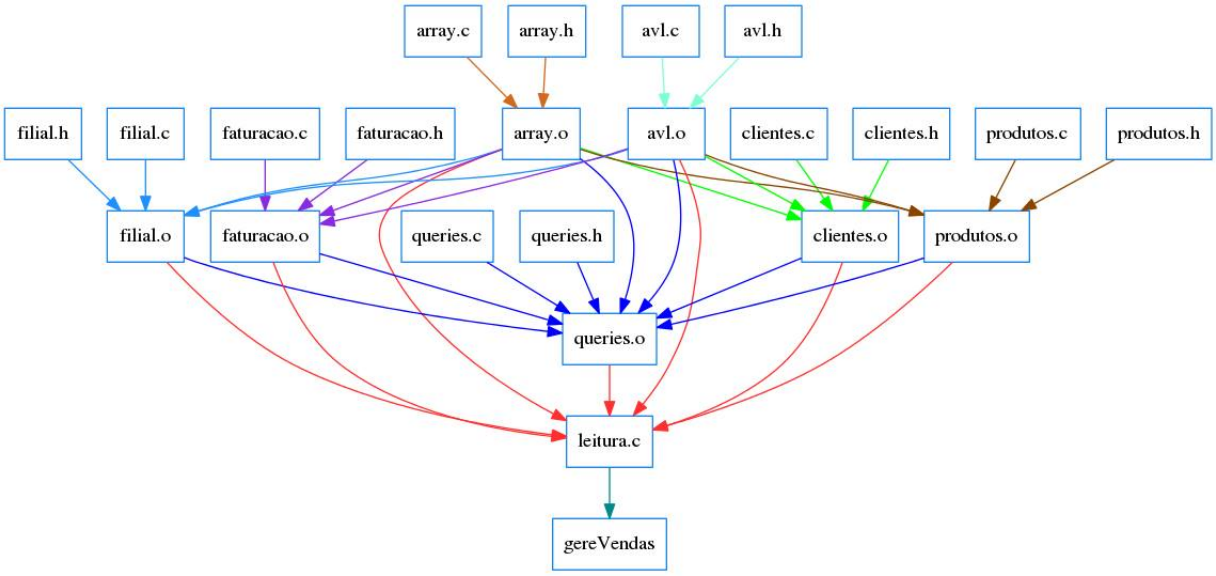


Figura 6.1: Grafo de depêndencias

```
objects = array.o avl.o clientes.o faturacao.o filial.o \
produtos.o queries.o

CFLAGS=-Wall -ansi -pedantic -O2

all:
make clean
make produtos
make array
make avl
make clientes
make faturacao
make filial
make queries
make leitura
```

```
leitura: src/leitura.c array.o avl.o clientes.o faturacao.o filial.o
 produtos.o queries.o
gcc src/leitura.c array.o avl.o clientes.o faturacao.o filial.o
 produtos.o queries.o $(CFLAGS) -o gereVendas -lm

queries: src/queries.c src/headers/queries.h
gcc src/queries.c -c $(CFLAGS)

clientes: src/clientes.c src/headers/clientes.h
gcc src/clientes.c -c $(CFLAGS)

produtos: src/produtos.c src/headers/produtos.h
gcc src/produtos.c -c $(CFLAGS)

array: src/array.c src/headers/array.h
gcc src/array.c -c $(CFLAGS)

faturacao: src/faturacao.c src/headers/faturacao.h
gcc src/faturacao.c -c $(CFLAGS)

filial: src/filial.c src/headers/filial.h
gcc src/filial.c -c $(CFLAGS)

avl: src/avl.c src/headers/avl.h
gcc src/avl.c -c $(CFLAGS)

.PHONY : clean
clean :
rm -f gereVendas
rm -f $(objects)
rm -f gesval
```

7. Conclusão

Uma vez que se tratou de um trabalho de uma dimensão já considerável comparando com o que estávamos habituados envolveu utilização de técnicas particulares e tivemos sempre como objetivo que este trabalho fosse concebido de modo a que seja facilmente modificável, e seja, apesar da complexidade, o mais optimizado possível a todos os níveis.

Inicialmente, tivemos dificuldades nas AVLs pois estávamos a fazer uma AVL para cada módulo. Depois de alguns problemas com o seu balanceamento, acabamos por apostar na utilização da biblioteca standard AVL da GNU, que nos facilitou não só o carregamento dos ficheiros em memória, mas também na realização de algumas queries, devido ao vasto conjunto de úteis funções que a biblioteca contém, evitando assim a repetição de código.

Tivemos dificuldades em conseguir resultados quando usámos o módulo das filiais e era necessário cruzar os dados com as 3 filiais existentes, pois o módulo filial era apenas para uma filial.

Em suma, podemos concluir que embora todas as queries estejam a funcionar corretamente há aspectos na interface com o utilizador que poderiam ser melhorados.