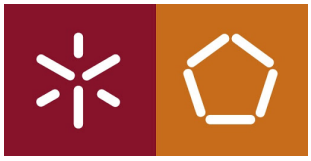
Universidade do Minho
Mestrado Integrado em Engenharia Informática



Projeto Laboratórios de Informática 3

GRUPO 65
Ana Teresa Gião Gomes - A89356
Maria Quintas Barros - A89325
Maria Beatriz Araújo Lacerda - A89535

19 de Abril 2020

Figure 1: A89536

Figure 2: A89525

Figure 3: A89535

1 Introdução

No âmbito desta unidade curricular, foi nos proposto o desenvolvimento de um Sistema de Gestão de Vendas (SGV) de uma cadeia de distribuição composta por 3 filiais, com o objetivo de que o utlizador consiga extrair o máximo de informação útil deste programa.

Nesta primeira fase do projeto, implementamos este sistema aplicando a linguagem C e, embora consideremos importante obtermos uma rápida execução deste programa resultando em tempos de execução muito reduzidos, focamo-nos também no encapsulamento e modularidade das nossas estruturas de dados.

2 Estruturação

2.1 SVG.h

```
struct sgv{
  char* path[3];
  Dados* clientes;
  Dados* produtos;
  VT* vendas;
  AVL* faturas;
  Filial* filial[3];
};
```

Figure 4: SGV

A estrutura que implemeta o nosso SGV é composta por: Um array com tamanho 3, para armazenar os caminhos para os dados dos clientes, produtos e vendas; Uma estrutura do tipo Dados, que contem todos os clientes válidos presentes no ficheiro, constituindo assim um catálogo de clientes; Uma estrutura do tipo Dados, que contem todos os produtos válidos presentes no respetivo ficheiro ,constituindo assim um catálogo de produtos; Uma estrutura do tipo VT, que contem as informações das vendas; Uma AVL que contem a informação das faturas; Um array de estruturas de tamanho 3 do tipo Filial, que armazena duas árvores binarias balenceadas que contem os produtos comprados e os clientes que compraram;

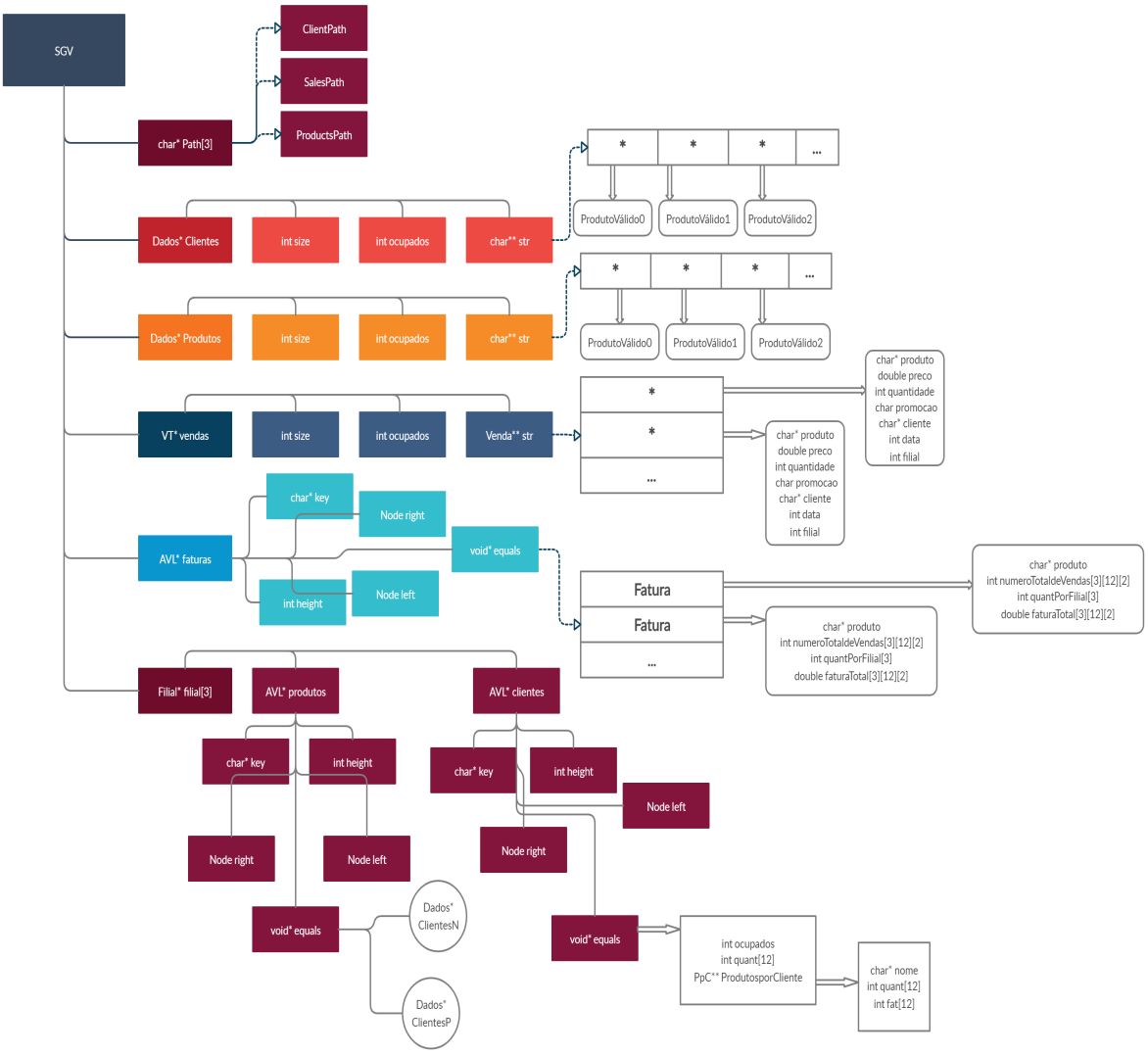


Figure 5: Ilustração da estrutura

2.2 Faturação

Cada fatura nossa é composta por um *char* produto* que nos indica o código deste. Contem ainda o *numeroTotalVendas[3][12][2]* que nos diz a totalidade de vendas por filial, mês e tipo de promoção. *quantPorFilial[3]* contem a quantidade de produto por filial e ainda o double *faturaTotal* que contem a faturação total por filial, por mes

```c
struct Node {
    char* key;
    struct Node *left;
    struct Node *right;
    int height;
    void* equals;
};
```

Figure 6: AVL

```c
struct fatura {
      char* produto;
      int numeroTotalVendas [3][12][2];
      int quantPorFilial [3];
      double faturaTotal [3][12][2];
};
```

Figure 7: Fatura

e por tipo de promoção. Esta estrutura foi criada com o propósito de ser utilizada nas queries. Na nossa estrutura AVL, temos um *char* key* que contem a chave da nossa árvore. Para além disso temos ainda a *struct Node *left* e a *struct Node *right* para ser possível navegar na nossa árvore. O inteiro height permite saber se a nossa árvore se encontra balanceada. Por último temos um *void* equals*. Neste void vamos introduzir outras estruturas como por exemplo uma estrutura *Fatura*, para poder associar a nossa *key* (Produto) com o *void**(Fatura).

2.3 Filial

Em filial estão contidas duas árvores binárias de procura, uma onde estão armazenados os produtos (comprados) e outra os clientes (que compraram) de cada filial.
Dentro da AVL dos produtos encontra-se a estrutura codigoProdutosP (ProdP). Esta estrutura, por sua vez, contem duas estruturas de Dados. Uma com o nome clienteN e outra com o nome clienteP. Esta estrutura foi criada com o propósito de ser possível distinguir entre os clientes com promoção e sem promoção quando nos pedido nas queries.
Dentro da AVL dos clientes temos a estrutura codigosClientes (ClieP). Esta estrutura contem o inteiro *ocupados* e ainda o array *quant[12]* que contem a quantidade de produtos comprados pelo cliente num determindado mês. Por último contem a estrutura produtosPorCliente *char* PpC*, que contem um *char* nome* para nos indica o código de cada cliente, um array *quant[12]* que contem a quantidade de compras feitas por mês por esse cliente daquele produto e ainda o array *fat[12]* que contem a faturação desse cliente por mês daquele produto.

2.4 Estruturas Elementares

As estruturas base do nosso trabalho e as primeiras a serem construidas foram a *Dados*, *vendas*, e *VT*.
A estrutura Dados é responsável por armazenar informação dos clientes e produtos válidos. Dentro desta estrutura temos dois inteiros: o *size* e o *ocupados*, que nos indicam o tamanho da estrutura e a ocupação desta, permitindo-nos fazer uma implementação dinâmica. Temos ainda outro inteiro, *lidas*, que foi criado com o propósito de fornecer o número de linhas válidas lidas (tal como nos foi pedido na querie 13). Por último te um array de apontadores? que irá armazenar os códigos dos clientes ou produtos válidos. Usamos esta estrura também para armazenar a informação dos clientes com promoção e sem promoção na estrutura codigoProdutosP(ProdP) já referida anteriormente.
A estrutura VT, tal como a estrutura Dados, contem um *size*, *ocupados* e um *lidas* para os mesmo propósitos. Contem ainda um array? do tipo Venda. Em cada posição deste array estará a estrutura venda que contem as informaçoes das vendas, nomeadamente o produto, preço, quantidade, tipo de promoção, o cliente que comprou o produto, o mês e a filial em que foi feita a venda. Esta estrutura permite-nos coletar a informação das vendas que irá depois ser distribuída por outras estruturas convenientemente, por exemplo para armazenar as informção na estrutura codigoProdutosP.

3 Modularização e Abstração de Dados

De maneira a fazer não só um código eficiente, mas também seguro, foi nos pedido para ter em conta a modularização. Para isto, separamos os nossos catálogos dos clientes, catálogo dos produtos, filial, faturação e faturações em diferentes módulos, reduzindo o número de funções visíveis para o mínimo. Por esta razão, definimos várias funções get, que devolvem uma cópia da informação que precisamos das nossas estruturas. Deste modo, temos acesso a informação que está em outros módulos preservando o encapsulamento. Para além disto, tornamos a nossa implementação de árvores binárias de procura balenceadas "reutilizável" ao trabalharmos com keys e valores do tipo void*, tornando, assim, mais fácil mudar a organização da árvore.

4 Queries

- **Querie 1:loadSGVFromFiles**

  Nesta querie, são carregados os ficheiros relativos aos produtos, clientes, vendas, faturação e filial. Esta querie é responsavél pela inicialização do programa e essencial para um bom funcionamento deste.

- **Querie 2: getProductsStartedByLetter**

  Criamos esta querie de modo a conseguir determinar a lista e o número total de produtos cujos codigos sao inicializados por uma dada letra introduzida pelo utilizador.

  Para isso, começamos por implementar um ciclo que percorre todos os codigos dos produtos válidos e, assim, verifica se o primeiro elemento do codigo corresponde à letra inserida pelo utilizador.

  Caso esta condição seja verdadeira, irão ser guardados num array os códigos dos produtos que respeitam as exigências.

- **Querie 3: getProductSalesAndProfit**

  Para responder a esta querie criamos uma estrutura (Q3) que armazena 4 valores: o número total de vendas tipo N, o número total de vendas tipo P, o total faturado tipo N e o total faturado tipo P.

  Existem duas cenários de resposta desta querie, dependendo da vontade do utilizador:

  1. O resultado apresentado é o global:

     Os 4 valores acima referidos são dados, cada um, como um total dos valores de cada filial, ou seja, nao fazendo divisão entre as filiais.

  2. O resultado apresentado é dado filial a filial para todas as 3 filiais:

     Para isso, é criado um ciclo que percorre todas as filiais válidas, de modo a obtermos os 4 valores acima referidos para cada uma das 3 filiais.

- **Querie 4:getProductsNeverBought**

  Criamos esta querie de modo a conseguir determinar a lista ordenada e o número total de produtos que nunca foram comprados.

  Existem duas cenários de resposta desta querie, dependendo da vontade do utilizador:

  1. O resultado apresentado é o global:
     Temos um ciclo que percorre todos os códigos de produtos válidos e, sem divisão dos valores por cada uma das filiais, verifica se cada produto foi comprado ou não, guardando apenas os que não foram nunca comprados.
  2. O resultado apresentado é dado filial a filial para todas as 3 filiais:
     Neste caso, também temos um ciclo que percorre todos os códigos de produtos válidos mas, desta vez, obtemos os valores correspondentes aos produtos que não foram comprados, para cada filial.

- **Querie 5: getClientsOfAllBranches**

  Esta querie tem como objetivo obter uma lista ordenada dos códigos do clientes que fizeram as suas compras em todas as 3 filiais.

  Temos um ciclo que percorre todos os códigos de clientes válidos e, para as 3 filiais, vamos procurar na AVL clientes os que compraram produtos em todas as filiais.

  Assim, forma-se uma lista ordenada com os códigos dos produtos que cumprem essas caracteristicas.

- **Querie 6: getClientsAndProductsNeverBought**

  Em primeiro lugar criamos uma estrutura (a Q6) que armazena dois inteiros (naoComprados e naoCompraram).

  No inteiro *naoCompraram* iremos armazenar o número de clientes que não compraram nada.

  Para isto vamos percorrer a lista de clientes válidos e, para as 3 filiais, vamos procurar estes clientes na AVL correspondente que contem todos os clientes que compraram produtos. Se o cliente não é encontrado então incrementamos a variável n.

Para o caso dos produtos, inicalizamos um array (*flag*) a 0 de tamanho 3 (correspondendo a cada filial). Em seguida, procuramos na AVL que contem todos os produtos comprados. Se não for encontrado o 0 modifica para 1. Em seguida, percorremos o array e se todos os elementos forem 0 incrementamos a variável *np* que foi depois guardada na estrutura criada anteriormente.

- **Querie 7: getClientsAndProductsNeverBought**

  Para implementar esta querie começamos por criar dois ciclos, um para controlar as filiais e outro para controlar os meses do ano.

  Dentro de ambos estes ciclos procuramos o cliente dado nos *inputs*:

  1. Se encontrar, invoca a função *getQuantProdMesCliente*. Esta função recebe o cliente, o mês e ainda a árvore dos clientes. Em seguida irá localizar o cliente em questão e atravês da estrutura ClieP retorna a quantidade daquele mês.
  2. Se não encontrar coloca 0 na tabela.

  Armazenamos os valores na estrutura Q7, que contem uma matriz (mes[filial][mes]).

- **Querie 8: getSalesAndProfit**

  Para responder a esta querie criamos uma estrutura que armazena dois inteiros (Q8).

  Para obter o número de vendas no intervalo pedido invocamos a função *quantidadeTAVL* (in faturacoes.c). Esta funçao vai somar na variável *valor*, o número total de vendas entre os meses escolhidos, para as três filiais(controlado por i) e ambas as promoçoes (controlado por k). Por sua vez, isto é feito através da função *getNumeroTotalVendas* (in faturacao.c).

  Posteriormente, para obter a faturação total no intervalo pedido invocamos funçao*faturacaoTAVL*, que funciona de uma maneira identica à funçao referida anteriormente.

- **Querie 9: getSalesAndProfit**

  Começamos por criar uma estrutura (Q9), que contem dois arrays de strings (strN para os clientes que compraram o *productID* na

filial *branch* sem promoção e strP para os que compraram o mesmo produto com promoção).

De seguida procuramos o produto pretendido na AVL dos produtos e na filial escolhida. Se este produto existir na AVL, é copiado toda a informação da str de auxN(correspondente a estrutura clientesN) para a string strN e o mesmo acontece na strP.

- **Querie 10: getSalesAndProfit**

  Para esta queries o procedimento foi o seguinte: Criamos 3 apontadores que irao percorrer as 3 AVL de clientes que compraram produtos(cada uma corresponde a uma filial) e encontramos o cliente em todas as árvores.

  Em seguida, através da função *insereSTR* inserimos os dados da estrutura PpC de um dado nodo.

  Por fim, ordenamos por ordem decrescente.

- **Querie 11: getTopSelledProducts**

  Esta queire cria uma lista dos N produtos mais vendidos, sendo N o limite introduzido pelo utilizador. Para além disso, indica o número de vendas e o número de clientes, filial a filial.

  Recorremos a um ciclo que liberta o espaço na memoria ocupado pelos códigos dos produtos que não pertencem aos N produtos.

  Implementamos também 3 ciclos, um que percorre os N produtos, outro as 3 filiais e por último os 12 meses, de modo a obter o número de vendas e de clientes.

- **Querie 12: getClientTopProfitProducts**

  Para esta queries o procedimento foi o seguinte: Criamos 3 apontadores que irao percorrer as 3 AVL de clientes que compraram produtos(cada uma corresponde a uma filial) e encontramos o cliente em todas as árvores.

  Em seguida, através da função *insereFat* inserimos os dados necessários (Faturação e o nome do produto) da estrutura PpC de um dado nodo.

  Por fim, ordenamos por ordem decrescente.

- **Querie 13: getTopSelledProducts**

  Através das estruturas utilizadas na querie 1, obtemos as diretorias dos ficheiros .txt e o número de linhas lidas e válidas por cada ficheiro.

5 Testes de Tempo

Ficheiro	**Tempo Global** *(s)*
1 Milhão	≈ 11.1
3 Milhões	≈ 36.5
5 Milhões	≈ 66.15

Querie	**1 Milhão** *(s)*	**3 Milhões** *(s)*	**5 Milhões** *(s)*
Querie 1	7.36	31.45	53.62
Querie 6	0.280	0.233	0.226
Querie 7	0.000038	0.000028	0.000034
Querie 8	0.211	0.23	0.511
Querie 9	0.000007	0.000014	0.000019
Querie 10	0.00009	0.00021	0.000394
Querie 11	0.446430	0.473	0.43517
Querie 12	0.000097	0.00042	0.000436
Querie 13	0.000005	0.000006	0.000004

Na nossa opinião consideramos que os tempos estão razoáveis, apesar de alguns lapsos. Parte das razões que justificam estes tempos devem-se ao facto de não conseguirmos libertar uma pequena percentagem da memória no modulo faturações. Contudo, houve claras melhorias visto que no inicio o nosso programa demorava 9 minutos a correr o ficheiro de 5M.

6 Grafo de Dependencias

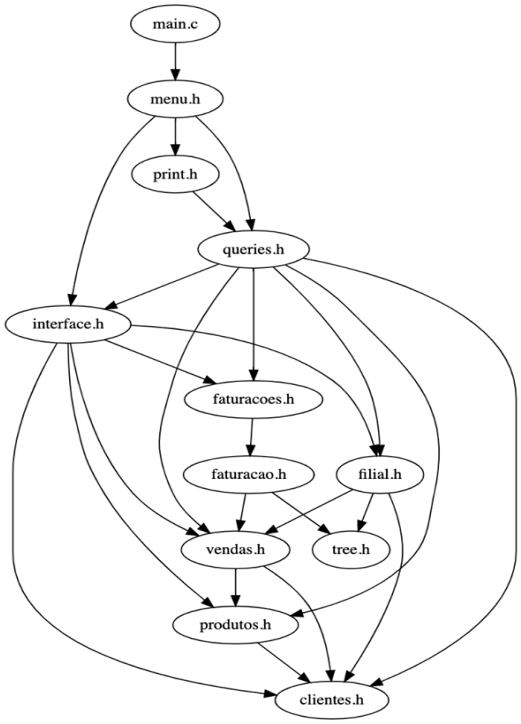


Figure 8: Grafo de Dependências

7 Conclusão

Em primeiro lugar, concluimos que a boa organização das estruturas de dados e a utilização destas nas funções que fomos implementando ao longo do nosso trabalho é um aspeto bastante importante para um bom funcionamento e uma melhor eficiencia do nosso programa.

Para além disso, consideramos importante manter acima de tudo o nosso código seguro de forma a evitar que, tanto o utilizador como outros, tenham acesso e consigam alterar as estruturas de dados.

Deste modo, para concluir, os nossos maiores desafios consistiram na organização das estruturas e na programação em grande escala devido ao facto de termos como nosso objetivo tornar este programa capaz de abranjir uma enorme quantidade de dados