Escola de Engenharia
Universidade do Minho

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

VENDUM

Sistema de Gestão de Vendas

Grupo 8

Renato Cruzinha
A75310

Ricardo Pereira
A73577

Braga, 31 de Maio de 2019

Conteúdo

1. Introdução

No âmbito da unidade curricular Laboratórios de Informática III do 2º Ano do Mestrado Integrado em Engenharia Informática foi-nos proposto o desenvolvimento de um projeto em linguagem *C* sobre uma gestão de vendas com três filiais que tem como objetivo ajudar à consolidação dos conteúdos teóricos e práticos e enriquecer os conhecimentos adquiridos nas u.c.'s de *Programação Imperativa* e *Algoritmos e Complexidade*.

Consideramos este projeto um grande desafio pela elevada complexidade que apresenta, uma vez que será uma aplicação com grandes volumes de dados, que é algo que a torna altamente complexa a nível algorítmico e estrutural. Nesse sentido, o desenvolvimento deste programa será realizado a partir dos princípios da modularidade (divisão do código fonte em unidades separadas coerentes), do encapsulamento (garantia de proteção e acessos controlados aos dados), da conceção de código reutilizável e da escolha otimizada das estruturas de dados.

2. Desenvolvimento

2.1 Descrição dos módulos e estruturas de dados

Como seria de esperar, este projeto foi desenvolvido através da construção de módulos, cada um deles com uma determinada função imprescindível. É extremamente importante fazê-lo, tendo em conta a deteção de erros, tornando-os mais evidentes e ainda a abstração do "mundo real", que nos ajuda a perceber e concretizar o nosso problema de uma forma mais objetiva.

Inicialmente, começamos por fazer uma análise dos dados que seriam importados para as estruturas e para isso, servimo-nos de estruturas que se apresentavam como *arrays* dinâmicos de *strings*, onde em cada um, guardávamos em cada posição, cada linha do respetivo ficheiro.

Rapidamente verificamos que as procuras por códigos de produtos e de clientes eram extremamente ineficientes. Verificamos que os tempos de validação dos registos de venda rondavam os dez minutos, algo que é extremamente ineficiente. A seguir, mostramos quais as decisões que tomamos para combater essa ineficiência.

2.1.1 Módulo árvores binárias balanceadas (*avl*)

Talvez um dos módulos mais importantes, o ***avl***, composto pelos ficheiros *avl.c* e *avl.h*, exporta, através deste último, a estrutura que otimiza de forma abruta o tempo necessário para resposta a todas as *queries*. A estrutura *AVL* é utilizada na maioria dos módulos que falaremos mais adiante, sendo uma "subestrutura"desses.

Uma *AVL*, é definida pelos conceitos "árvore binária de procura"e "balanceamento". Desde logo, sabemos que uma árvore binária de procura balanceada é uma árvore que armazena determinados dados ordenadamente, garantindo que, para cada nodo, cada um dos seus ramos descendentes não difere em mais do que uma unidade de altura. Os dados são armazenados em nodos, sendo que para cada um, existe uma chave, que será um tipo simples, ou algo que não seja uma estrutura de forma a que seja fácil a comparação para a correta inserção dos dados.

Este módulo exporta, entre outras, funções que operam sobre esta estrutura, nomeadamente: funções que executam outra função para cada nodo da árvore, funções que dada uma chave do nodo retornam o valor que lhe está associado, funções que inserem um novo par chave/valor, funções que removem uma chave e o respetivo valor, etc. A próxima figura mostra, de uma forma abstrata, a sua aparência.

A implementação desta estrutura foi feita recorrendo à *API* da *glib*, uma biblioteca de suporte ao desenvolvimento de *software*, a partir de onde temos acesso às estruturas de *AVL's* e respetivas funções.

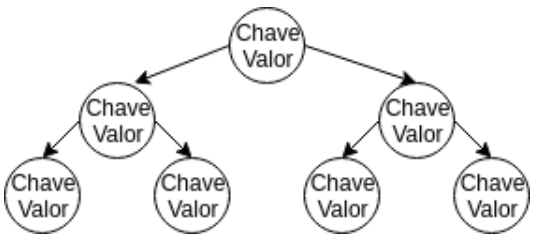

Figura 2.1: Abstração de uma *AVL*

2.1.2 Módulo produto (*produto*)

O produto é uma das principais entidades do nosso sistema de gestão de vendas. Para cada um existe um código com duas letras seguidas de quatro algarismos. Neste projeto, não é necessária qualquer estrutura para o caracterizar, não obstante, decidimos declará-la com um campo que é uma *string*. Caso os atributos desta entidade cresçam, teremos apenas que adicionar novos campos à estrutura.

2.1.3 Módulo cliente (*cliente*)

À semelhança do produto, o cliente apenas tem um código constituído por uma letra seguida de quatro algarismos. Foi também criada uma estrutura para caracterizar um cliente que, embora não fosse necessária, faz parte da abstração que caracteriza um cliente.

2.1.4 Módulo venda (*venda*)

Ao contrário do cliente e do produto, uma venda tem mais do que um atributo, isto é, cada linha disponibilizada pelo ficheiro *Vendas_1M.txt* engloba sete campos. Por esse motivo foi necessário criar uma estrutura onde pudessemos guardar cada campo associado a um registo de uma venda. Destes, existem dois campos que devem ser devidamente analisados, os do cliente e do produto, que devem existir no catálogo de clientes e de produtos, respetivamente. São estas operações que fazem com que a leitura dos ficheiros de dados, abordada mais à frente, seja muito demorada.

Neste módulo estão implementadas todas as funções que verificam cada campo de um registo, e ainda mais algumas, nomeadamente, a função que cria uma estrutura do tipo venda e inicializa os seus campos e a função que preenche cada campo de uma estrutura já criada.

2.1.5 Módulo catálogo de clientes (*cclientes*)

Falou-se, desde o ínicio do presente relatório, que a otimização dos tempos de execução deste sistema de gestão de vendas era impreterível. Essa otimização começa com o catálogo de clientes, estrutura onde são inseridos e organizados todos os clientes registados.

Um catálogo de clientes não é nada mais que um array de apontadores para *avl's* com vinte e seis posições. Este número deve-se ao facto de a cada árvore estar associada uma letra do alfabeto, isto é, criadas todas as *avl's* de cada posição, um cliente que seja válido terá como destino a árvore cuja letra associada é a única existente no seu código. Isto permite-nos fazer procuras (aproximadamente) vinte e seis vezes mais rápidas, assumindo que a atribuição de letras aos códigos é completamente aleatória. A estrutura contém ainda um inteiro que caracteriza o número de produtos comprados (não repetidos).

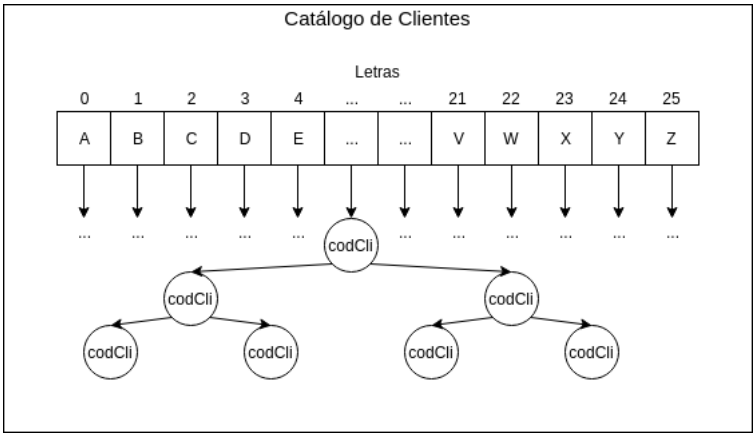

Figura 2.2: Catálogo de clientes

Como não poderia deixar de ser, este módulo alberga todas as funções que alteram ou criam estruturas

deste género. Para ficarmos a conhecer o apontador para uma *avl* associada a uma letra, temos disponível uma função que no-lo retorna quando esta é chamada.

2.1.6 Módulo catálogo de produtos (*cprodutos*)

Da mesma forma que dividimos os clientes por um *array* unidimensional, os produtos foram divididos por um *array bidimensional*, uma vez que o seu código é sempre constituído por duas letras. Desta vez, teremos um array de vinte e seis por vinte e seis posições em que cada uma aponta para uma *avl*. Cada linha está associada à primeira letra do código do produto e cada coluna, à segunda. As funções que identificamos no ponto anterior estão também implementadas neste módulo.

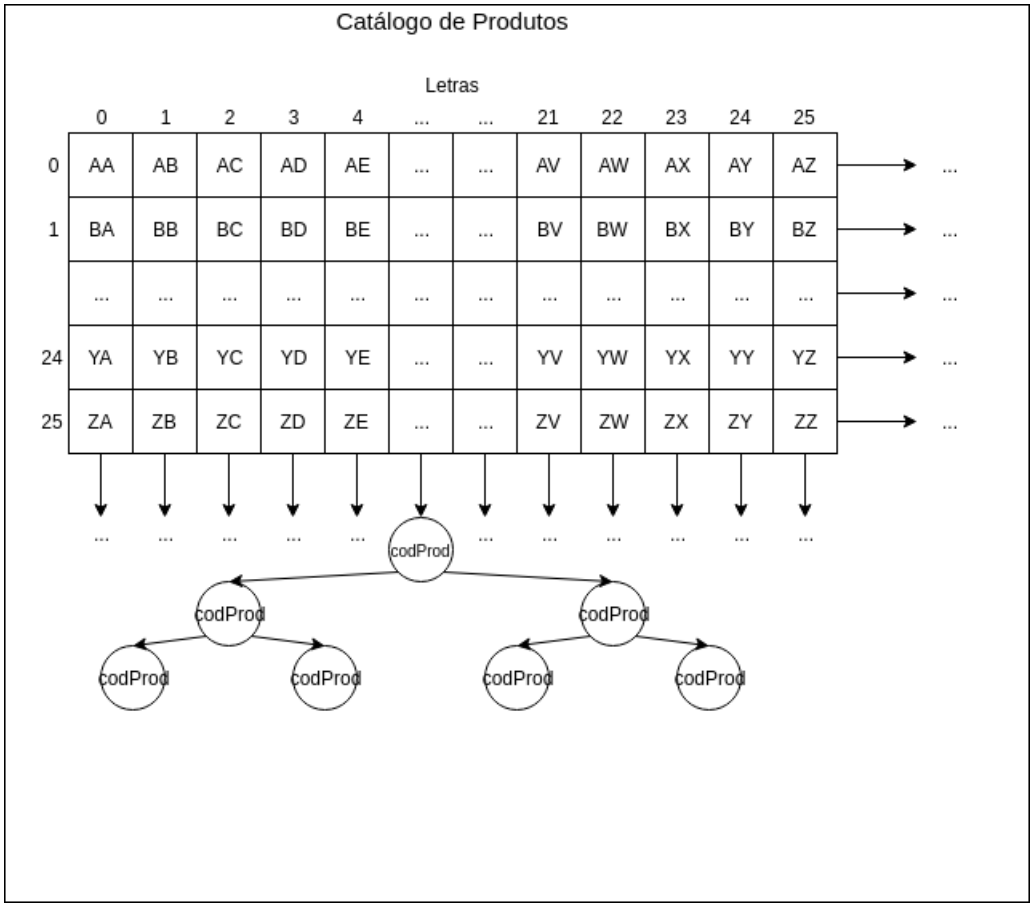

Figura 2.3: Catálogo de produtos.

2.1.7 Módulo catálogo de faturação (*cfaturacao*)

Foi-nos proposto que organizássemos num catálogo toda a faturação do sistema de gestão de vendas e após uma leitura cuidada e intensiva de todas as *queries* mencionadas, decidimos, sem fazer referência aos clientes, criar um módulo denominado *cfaturacao*. Este módulo exporta a estrutura de dados que vemos na seguinte imagem:

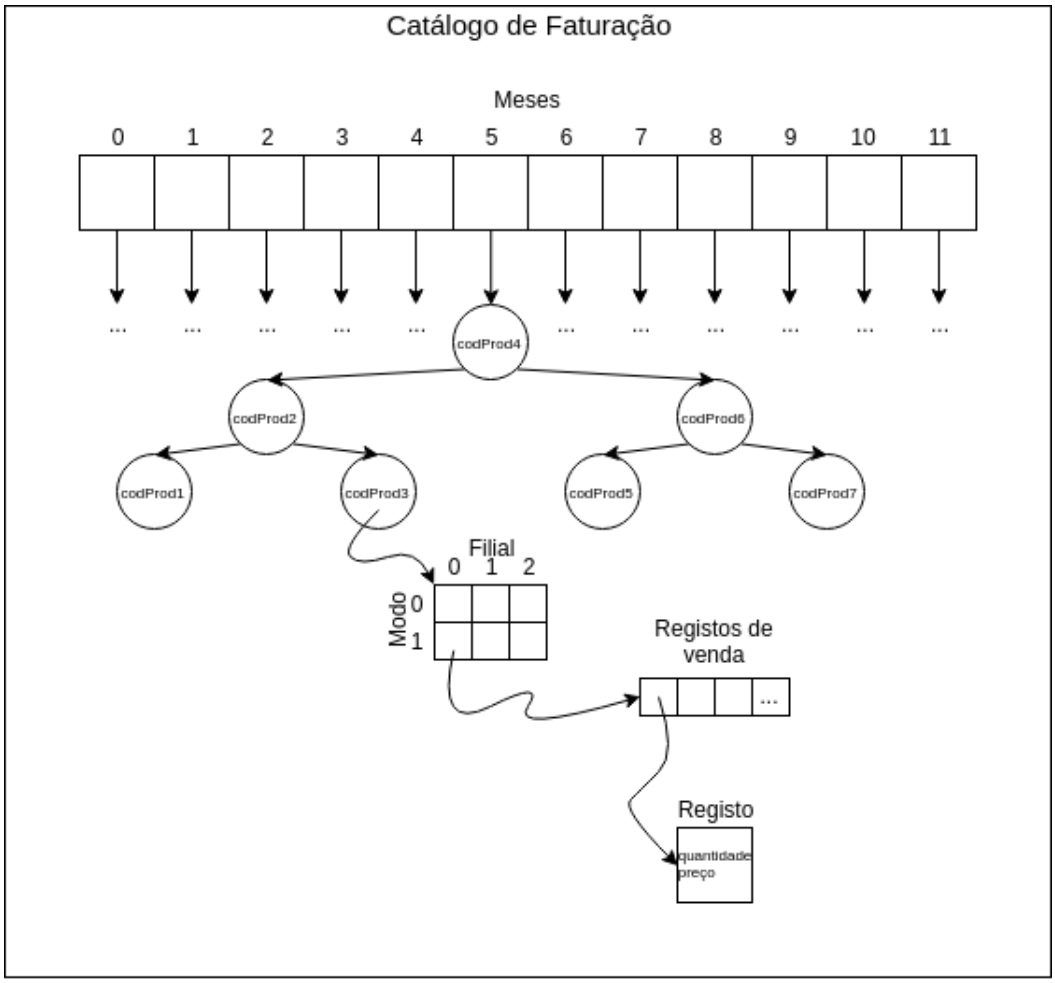

Figura 2.4: Catálogo de faturação.

Esta estrutura é, talvez, a mais complicada de compreender, visto que existem muitos detalhes que devem ser estudados. Para começar, achamos que a principal característica de um registo, atendendo àqueles doze requisitos, seria o mês, pelo que, construimos um *array* de apontadores para *avl's* com doze posições (número de meses), cujos nodos têm como chave um código de produto. Sendo assim, para cada nodo existe uma estrutura que apenas é inicializada quando de facto queremos lá colocar algo. Essa estrutura é um *array* bidimensional com as linhas associadas a uma filial e as colunas a um modo de compra. Até agora falamos de *arrays* cujo tamanho é sabido e, por esse motivo é que os organizamos dessa forma, no entanto, o número de registos de venda por produto num mês, filial e modo, varia bastante, e por esse motivo, não podemos definir uma estrutura com tamanho exato. Criamos então um *array* de apontadores dinâmico para estruturas do tipo *Registo*, cujo conteúdo terá sempre um preço e uma quantidade.

Como é evidente, qualquer função que atue sobre este módulo está lá implementada, como é o caso das funções que inicializam e inserem valores nas "subestruturas"ou então das funções que ajudam na resposta a determinadas *queries*.

2.1.8 Módulo catálogo de filiais (*cfiliais*)

Tal como aconteceu com o módulo *cfaturacao*, foi-nos pedido que criássemos um módulo *cfiliais* que relacionasse os clientes, os produtos e as quantidades, para além dos parâmetros mês, filial e modo, cuja existência é indispensável em qualquer um dos dois módulos agora mencionados.

No *cfiliais*, decidimos que, depois de um estudo exaustivo das interrogações impostas, deveríamos criar um array de apontadores para estruturas do tipo *AVL*. Cada nodo de cada uma dessas *avl's* tem

como chave, o código de um cliente e cada valor será um apontador para um *array* bidimensional (linhas correspondem aos modos de compra e as colunas aos meses) em que cada posição aponta para uma estrutura do tipo *avl*. Cada uma destas últimas, tem guardados os produtos comprados por cliente, filial, mês e modo, sendo o valor a quantidade que foi comprada.

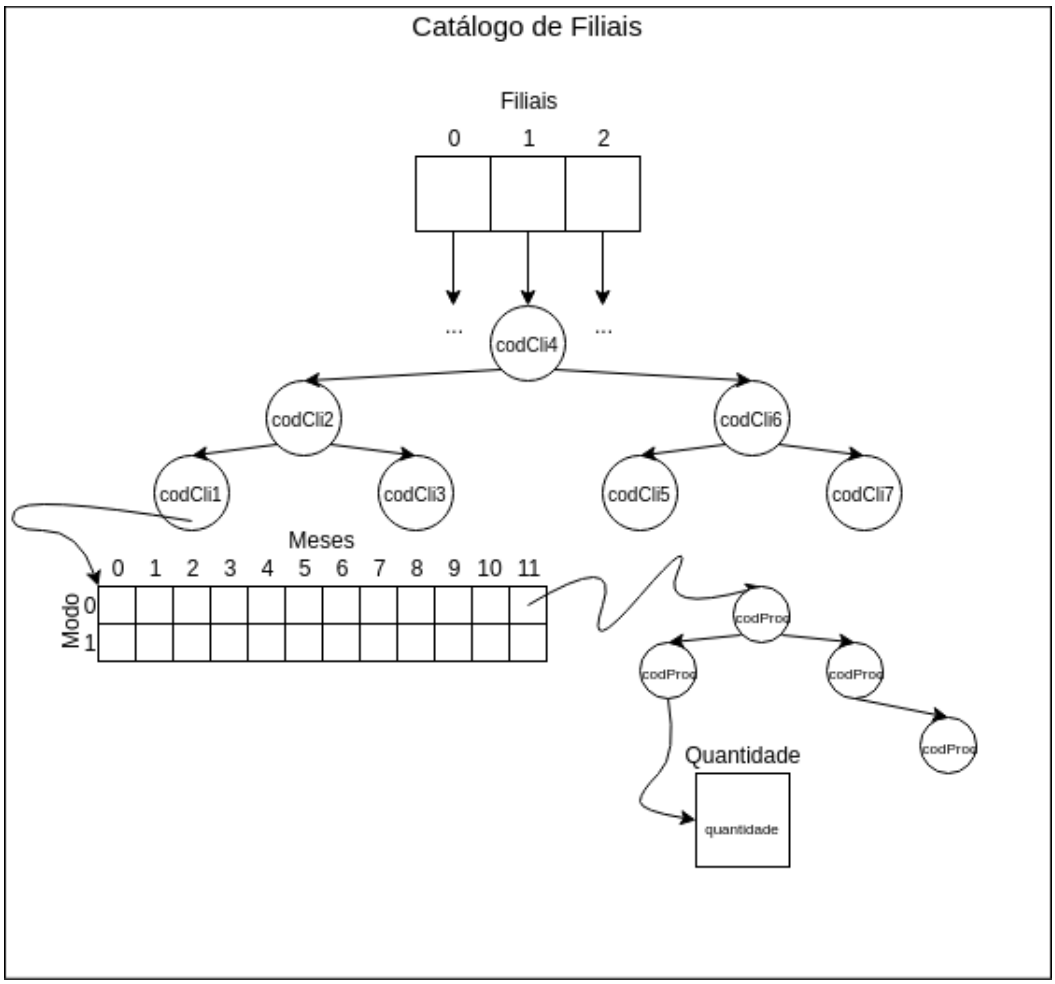

Figura 2.5: Gestor de filiais.

Tal como todos os outros descritos anteriormente, este módulo tem funções que retornam, por exemplo, a *avl* associada a uma filial, a *avl* associada a um cliente, etc. Implementa ainda as funções que criam a estrutura principal e as correspondentes "subestruturas". A técnica que adotamos para prencher esta estrutura passou por, associar um produto ao cliente no respetivo mês, modo e filial e sempre que aparecesse um registo nas mesmas condições de um que já existisse, apenas era feita a soma da quantidade do novo registo, à quantidade que já lá existia.

2.1.9 Módulo lista de *strings* (*lstrings*)

A função deste módulo é possibilitar ao utilizador a visualização dos resultados de uma forma prática. Sem este módulo, teríamos que imprimir os dados todos seguidos no terminal onde executamos a aplicação, ficando alguns indisponíveis, se a quantidade impressa fosse mesmo elevada.

Por esse motivo, importamos para um *array* dinâmico de *strings* todos os resultados que queremos imprimir. Esse *array* está englobado na estrutura *lstrings* juntamente com dois campos, que nos indicam o tamanho e a posição onde se irá "escrever"a próxima *string*.

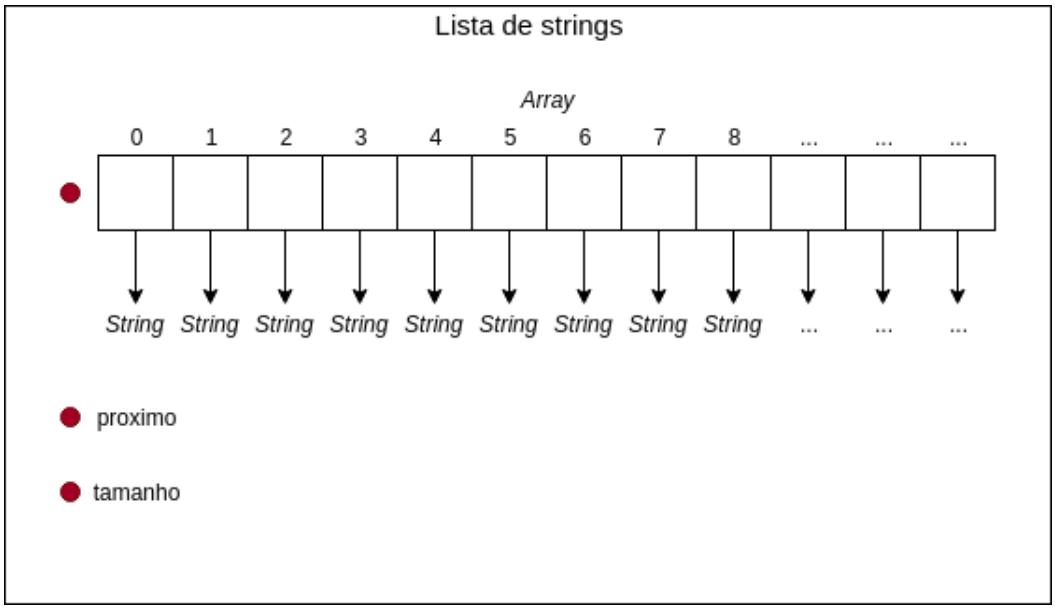

Figura 2.6: Listas de *strings*

Quando as *queries* assim o exigirem, o utilizador irá vizualizar a primeira página e conforme a instrução que submeter, poderá avançar ou retroceder nas várias "páginas"de resultados que existem.

2.1.10 Módulo leitura dos dados (*leituras*)

A leitura dos dados é crucial para o bom funcionamento do programa. É neste módulo que estão presentes as funções que leem os ficheiros *.txt* que contêm os dados e é nele que são chamadas as funções de inserção dos mesmos nas diversas estruturas.

2.1.11 Módulo interpretador (*main*)

A execução da aplicação é, no fundo, um conjunto de funções que são chamadas sempre que escolhemos a *querie* a ser executada. Ora, para podermos estabelecer esta comunicação com o sistema de gestão de vendas, é impreterível que tenhamos um módulo que trate deste ponto e que informe e peça todos os dados necessários ao utilizador para que a *query* escolhida apresente a informação correta.

2.1.12 Módulo globais (*globais*)

Em qualquer programa são necessários determinados parâmetros que, por se repetirem e por serem comuns a todos os outros módulos, não faz sentido que sejam constantemente declarados. No nosso caso, variáveis como o número de meses, o número de filiais e o número de modos de compra, são números exatos que não variam. Por esse motivo, decidimos criar um módulo que englobasse todas estas variáveis de tal forma que pudessem ser utilizadas por qualquer módulo.

2.1.13 Módulo *queries* (*queries*)

Finalmente, temos o módulo *queries* que apresenta para cada *query*, uma função que agrega todas as funções necessárias dos outros módulos para construir a resposta para o utilizador.

2.2 Arquitetura da aplicação

Como em qualquer programa em linguagem *C*, este inicia-se numa função *main* que gere todas as perguntas que são feitas pelo utilizador.

Por este motivo, o módulo interpretador, onde está a função *main*, inclui o módulo *queries* e todos aqueles que são necessários para conhecer as estruturas de dados que utilizamos, uma vez que é na *main* que estas são declaradas. O mesmo acontece com os restantes.

De uma forma geral, todos os módulos que querem fazer operações sobre estruturas ou então que necessitam de funções que não lhe pertencem, devem incluir outros módulos associados a essas mesmas operações/funções. Em baixo ilustramos um esquema de como estão a ser utilizados e incluidos uns pelos outros.

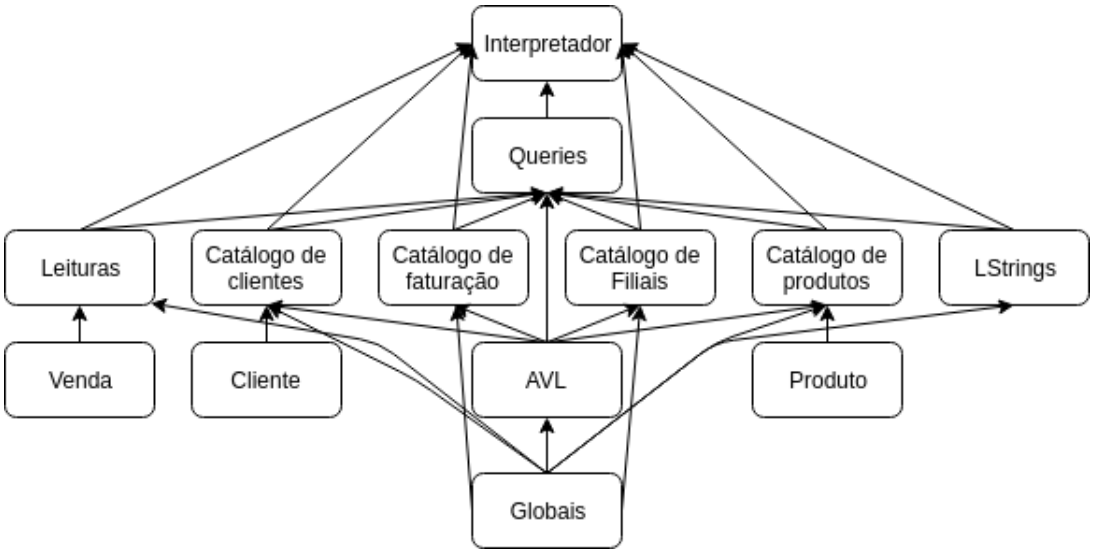

Figura 2.7: Arquitetura do *SGV*

2.3 Complexidade das estruturas

O objetivo, desde o início foi minimizar o tempo de obtenção das respostas referentes às diversas *queries*.

Como já foi explicado, uma forma instantânea de tornar o programa mais rápido foi dividir os clientes e os produtos por *avl's*, cada uma associada a uma letra. No caso dos produtos e dos clientes a complexidade no pior caso é, aproximada e respetivamente, $\log_2(N/676)$ (sendo *N* o número de produtos) e $\log_2(N/26)$ (sendo *N* o número de clientes).

Esclarecida a complexidade dos produtos e das vendas, explicamos agora a complexidade do catálogo das filiais e da faturação. Estas, não diferem em muito da complexidade das estruturas anteriores visto que também recorrem a *avl's*. Os casos em que existirá um maior trabalho por parte do programa, são o caso no qual terá de se percorrer a estrutura no seu todo e o caso onde se executam operações com o valor de um nodo da *avl*. Aqui, a complexidade será, geralmente, $\log_2(N)$ e está totalmente dependente do mês, da filial e do modo de compra em cada registo de venda.

Note-se que, tanto nas filiais como na faturação, decidimos não colocar no início da estrutura um *array* tridimensional, algo que pode parecer evidente, mas não é. Ao fazê-lo poderíamos estar a repetir nodos em demasia, nas diferentes *avl's* que cada posição do array teria, o que não é muito eficiente.

2.3.1 Resultados obtidos

Sabíamos desde o início, que era extremamente ineficiente trabalhar com *arrays* estáticos ou dinâmicos, isto porque, se quisermos fazer uma procura, de um determinado valor, teremos que o fazer de uma forma sequencial sem qualquer critério de procura.

Imagine-se um *array* com um milhão de posições, em que, sempre que quiséssemos fazer uma procura de um valor, no pior caso teríamos que consultar um milhão de posições. Imagine-se agora uma *avl* que, estando ordenada e balanceada, nos permitirá procurar esse mesmo valor consultando, no pior caso,

$\log_2(N)$ posições, a altura da árvore. Qual a melhor? Obviamente, a segunda opção é muito mais rápida e foi a mesma que nos permitiu passar de tempos de execução de dez minutos para dez segundos, para a leitura dos dados dos ficheiros de texto.

Para a primeira *query*, o tempo de execução ainda é considerável, não obstante, as restantes são executadas em tempo instantâneo, uma vez que o utilizador não chega sequer a esperar um segundo para obter cada resposta.

3. Conclusão

A unidade curricular de Laboratórios de Informática III revelou-se uma ótima oportunidade para aprimorarmos os nossos conhecimentos de linguagem *C*. Foi extremamente útil na medida em que ficamos com uma melhor preceção daquilo que é um projeto que lida com grandes volumes de dados.

A construção dos diferente módulos permitiu-nos começar a perceber como funcionam linguagens de alto nível, nomeadamente, a linguagem *Java*. Dividir o código escrito pelos diversos módulos não só é uma muito boa prática, como nos ajuda a abstrair a aplicação do mundo real para o mundo da programação.