|  |  |
| --- | --- |
| EENG | **Escola de Engenharia**  Departamento de Engenharia Informática  Mestrado Integrado em Engenharia Informática  Laboratórios de Informática III |

Relatório de Laboratórios de Informática III

**GereVendas**

**(Trabalho Prático de C)**

Grupo 1:

João Martins (A68646)



João Pereira (A75273)



Manel Castro (A71646)



Índice

[1. Módulos de dados 3](#_Toc449370762)

[Bool 3](#_Toc449370763)

[Venda 3](#_Toc449370764)

[AVL 3](#_Toc449370765)

[Tipos enumerados 3](#_Toc449370766)

[*Typedefs* de funções 4](#_Toc449370767)

[Estruturas de dados e tipos exportados 5](#_Toc449370768)

[Funções da API 6](#_Toc449370769)

[Cliente 7](#_Toc449370770)

[Estruturas de dados 7](#_Toc449370771)

[Funções da API 8](#_Toc449370772)

[Produto 8](#_Toc449370773)

[Estruturas de dados 9](#_Toc449370774)

[Funções da API 9](#_Toc449370775)

[LStrings 10](#_Toc449370776)

[Estruturas de dados 10](#_Toc449370777)

[Número de Strings por página 11](#_Toc449370778)

[Funções da API 11](#_Toc449370779)

[CatalogoClientes 12](#_Toc449370780)

[Estruturas de dados 12](#_Toc449370781)

[Funções da API 14](#_Toc449370782)

[CatalogoProds 15](#_Toc449370783)

[Estruturas de dados 15](#_Toc449370784)

[Funções da API 16](#_Toc449370785)

[FaturacaoGlobal 17](#_Toc449370786)

[Macros 17](#_Toc449370787)

[Estruturas de dados 17](#_Toc449370788)

[Definição de tipos 19](#_Toc449370789)

[Funções da API 19](#_Toc449370790)

[Filial 22](#_Toc449370791)

[MemUtils 23](#_Toc449370792)

[Funções da API 23](#_Toc449370793)

[Leitura 23](#_Toc449370794)

[Macros 23](#_Toc449370795)

[Funções da API 24](#_Toc449370796)

[4. Módulo principal (main.c) 24](#_Toc449370797)

[Estruturas de dados 24](#_Toc449370798)

[Leitura dos ficheiros e invocação das queries 25](#_Toc449370799)

[Ficheiros lidos por omissão 25](#_Toc449370800)

# Módulos de dados

## Bool

O *standard* C89 (também conhecido como “ANSI C”) não define o tipo ***bool***, pelo que optamos por criar um pequeno ficheiro designado por **bool.h**, onde incluímos as seguintes definições:



Este tipo foi utilizado em todos os módulos em que existem variáveis, parâmetros ou valores de retorno que só podem tomar os valores lógicos TRUE ou FALSE.

## Venda

Para evitar que o utilizador tenha que utilizar valores numéricos ao se referir a cada um dos tipos de venda, optamos por criar um pequeno módulo de dados, designado por **venda.h**, onde colocamos as seguintes definições:



Ficando assim definidos o número de tipos de venda (N\_TIPOS\_VENDA) e o tipo enumerado **TipoVenda**, que passam a poder ser utilizados em qualquer módulo que faça referência aos dois tipos de vendas.

## AVL

A necessidade de ler e consultar grandes volumes de dados tornou indispensável a utilização de estruturas de dados nas quais a inserção e a procura sejam eficientes. Por esse motivo, optamos pela utilização de **árvores AVL genéricas** para armazenar a maior parte da informação lida a partir dos ficheiros de dados, garantindo assim que cada operação de inserção ou procura seja realizada em tempo .

### Tipos enumerados

Para assinalar o fator de balanço de cada nodo de uma AVL, definimos o tipo enumerado **FatorBalanco**. Note-se que o utilizador do módulo de AVLs não necessita de ter informação relativa ao fator de balanço dos nodos da AVL que está a utilizar, pelo que a definição deste tipo enumerado foi colocada em **AVL.c**, sendo expressa por:

**typedef enum** fatorBalanco{ESQ, EQ, DIR} **FatorBalanco**;

### *Typedefs* de funções

Dada a generalidade das AVLs utilizadas e a necessidade de comparação entre os valores de nodos em operações como a inserção, procura e travessia *inorder* de uma AVL, definimos a assinatura da função de comparação que o utilizador deve fornecer aquando da criação de uma árvore AVL. Esta definição tem que ser conhecida pelo utilizador, pelo que está disponível no ficheiro **avl.h**, estando expressa por:

**typedef** void(\***Comparador**)(void \*, void \*);

**Nota:** A função de comparação deve devolver um valor < 0 se o 1º argumento for menor que o 2º, um valor = 0 se os dois argumentos forem iguais e > 0 se o 1º argumento for maior que o 2º.

Além desta definição, temos também a definição dos seguintes tipos de funções:

* Função que atualiza o valor de um nodo de uma AVL:

**typedef** void (**\*Atualizador**) (void \*, void \*);

, onde o valor apontado pelo 1º argumento é atualizado com base no valor apontado pelo 2º argumento.

* Função usada para criar uma cópia do valor armazenado num dado nodo de uma AVL:

**typedef** void\* **(\*Duplicador)** (void \*);

* Função que liberta a memória alocada para o valor guardado no nodo de uma AVL:

**typedef** void **(\*LibertarNodo)** (void \*);

### Estruturas de dados e tipos exportados

No ficheiro **avl.c** encontra-se a definição de duas estruturas de dados:

**TCD\_AVL:**

**typedef struct** TCD\_AVL{

AVL\_NODO\* raiz;

Atualizador atualiza;

Comparador compara;

Duplicador duplica;

LibertarNodo liberta;

int tamanho;

} TCD\_AVL;

**raiz –** apontador para a raiz da AVL;

**atualizador** – apontador para uma função que permite atualizar o valor de um nodo da AVL. Se este apontador for NULL, a AVL permite repetições de valores;

**compara** – apontador para a função de comparação entre os valores de 2 nodos de uma AVL;

**duplicador** – apontador para uma função que devolve uma cópia do valor de um nodo.

**liberta** – apontador para uma função que liberta a memória alocada para o valor guardado num nodo. Se este for NULL, é usada a função free().

**Tamanho** – nº de nodos da AVL.

**AVL\_NODO:**

**typedef struct** nodoAVL{

void\* valor;

FatorBalanco fatorBalanco;

struct nodoAVL \*esquerda, \*direita;

} AVL\_NODO;

**valor** – referência do valor do nodo;

**fatorBalanco** – fator de balanço do nodo (ESQ, EQ ou DIR);

**esquerda –** apontador para a raiz da subárvore esquerda ou para NULL, se esta não existir;

**direita** – apontador para a raiz da subárvore direita ou para NULL, se esta não existir;

Ao colocarmos estas definições apenas no ficheiro **avl.c** estamos a preservar o encapsulamento, escondendo a implementação concreta das estruturas de dados utilizadas e evitando o acesso direto às mesmas. O utilizador, ao ter apenas acesso ao ficheiro **avl.h**,só conhece a declaração abstrata do tipo AVL:

**typedef struct** TCD\_AVL\* **AVL**;

Assim, qualquer tentativa por parte do utilizador de desreferenciar um apontador para a **struct TCD\_AVL** provoca um erro de compilação, pelo que a única forma de manipular uma AVL é utilizar as funções da sua API que apresentamos de seguida.

### Funções da API

* AVL **criaAVL** (

Atualizador atualiza,

Comparador compara,

Duplicador duplica,

LibertarNodo liberta

)

Cria uma AVL vazia, que vai utilizar a função de comparação 'compara'. A função de comparação é obrigatória. As restantes funções são opcionais. Se não for fornecida uma função de atualização, a AVL criada admite repetições. Se não for passada uma função de libertação de nodos, a AVL criada utiliza a função free() para libertar cada nodo.

**AVISO:** Se o utilizador não passar uma função de duplicação, não é criada uma cópia aquando da inserção de um valor na AVL e as funções **inorderAVL()** e **procuraAVL()** devolvem o conteúdo da própria AVL!

**Retorna:**

Em caso de sucesso é retornada a AVL criada. Quando não é fornecida uma função de comparação ou há uma falha de alocação, é devolvido NULL.

* AVL **insereAVL** (AVL arvore, void\* val)

Insere um valor numa AVL previamente criada. Caso tenha sido passado um duplicador aquando da criação da arvore, o valor inserido é uma cópia do original. Caso contrário, é o original.

**Retorna:**

Árvore com o valor inserido em caso de sucesso. NULL em casos de falha de alocação.

* AVL **apagaAVL** (AVL arvore)

Liberta a memória alocada para guardar a AVL passada como parâmetro.

**Retorna:**

É sempre retornado NULL.

* int **tamanhoAVL** (const AVL arvore)

**Retorna:**

Número de nodos da AVL passada como parâmetro.

* int **alturaAVL** (const AVL arvore)

**Retorna:**

Altura da AVL passada como parâmetro.

void\*\* **inorderAVL**(const AVL arvore)

**Retorna:**

Em caso de sucesso, é devolvido um array com os valores dos nodos (cópias se a função de duplicação tiver sido passada durante a criação da AVL) resultante da travessia inorder da AVL. Caso contrário, é devolvido NULL.

* void\* **procuraAVL** (const **AVL** arv, void\* val)

Procura o valor val (2º argumento) na AVL arv (1º argumento)

**Retorna:**

Se o valor procurado existir na AVL e tiver sido passada uma função de duplicação aquando da criação da AVL, é devolvida uma cópia do valor existente na AVL. Se o valor existir mas a AVL não tiver uma função de duplicação, é devolvido o próprio valor que está na AVL. Nos restantes casos é devolvido NULL.

* bool **existeAVL**(const AVL arv, void\* val)

Testa se o valor val (2º argumento) existe na AVL arv (1º argumento).

**Retorna:**

TRUE se o val existir na AVL arv. FALSE caso contrário.

**Nota:** o tipo AVL é o único tipo acessível ao exterior e as funções da API não fazem qualquer referência a **TCD\_AVL**.

## Cliente

O tipo **Cliente** foi definido para esconder do utilizador a representação interna do código de cada cliente, permitindo ao mesmo tempo que a API dos restantes módulos que lidam com clientes seja mais sugestiva e agradável.

### Estruturas de dados

No ficheiro **cliente.c** podemos encontrar a definição da estrutura de dados ilustrada a seguir:

**Cliente:**

**struct** cliente {

char\*codigoCliente;

};

**codigoCliente** – apontador para o código do cliente. Sendo declarado como um char\*, o código de cliente pode ter qualquer dimensão, uma vez que aponta para uma *String* alocada com a função malloc();

Embora a única informação armazenada na **struct cliente** seja um apontador para o código do cliente, o utilizador da API não tem que saber o que está guardado dentro desta **struct**, pelo que no ficheiro **cliente.h** temos apenas:

**typedef struct** cliente\* **Cliente;**

Deste modo, garantimos que o tipo **Cliente** é opaco, o que leva a que o utilizador só o possa manipular através das funções da API que passamos a descrever.

### Funções da API

* Cliente **criaCliente** (char\* codigoCliente)

Cria um cliente com o código passado como parâmetro.

**Retorna:**

O cliente criado em caso de sucesso. NULL em caso de falha de alocação.

* Cliente **apagaCliente** (Cliente c)

Liberta a memória alocada para guardar um cliente.

**Retorna:**

É sempre retornado NULL.

* char \* **obterCodigoCliente** (Cliente c)

**Retorna:**

Em caso de sucesso, é devolvida uma cópia do código do cliente passado como parâmetro. Se ocorrer uma falha de alocação, é devolvido NULL.

* char **inicioCodigoCliente** (Cliente c)

**Retorna:**

1ª letra do código do cliente passado como parâmetro.

* int **comparaCodigosCliente** (Cliente c1, Cliente c2)

Compara os códigos de dois clientes.

**Retorna:**

É devolvido um valor < 0 se o código de c1 for menor que o de c2, = 0 se os códigos de c1 e c2 forem iguais e > 0 se o código de c1 for maior que o de c2.

* Cliente **duplicaCliente** (Cliente c)

**Retorna:**

Em caso de sucesso é devolvida uma cópia do cliente passado como parâmetro. Se ocorrer uma falha de alocação é devolvido NULL.

## Produto

O tipo **Produto** é, em termos estruturais, semelhante ao tipo **Cliente** e tambémfoi definido pelas mesmas razões que o tipo **Cliente**, i.e.: esconder a representação interna do código de cada produto e tornar a API dos restantes módulos que lidam com produtos mais sugestiva.

### Estruturas de dados

No ficheiro **produto.c** encontra-se a definição da estrutura de dados a seguir ilustrada:

**Produto:**

**struct** produto {

char\*codigoProduto;

};

**codigoProduto** – apontador para o código do produto. Sendo declarado como um char\*, o código de produto pode ter qualquer dimensão, uma vez que aponta para uma *String* alocada com a função malloc();

Analogamente à **struct cliente**, a única informação armazenada na **struct produto** é um apontador para o código do produto, mas o utilizador, tendo apenas acesso ao ficheiro **produto.h** só conhece a seguinte definição do tipo abstrato de dados **Produto**:

**typedef struct** produto\* **Produto;**

Logo o tipo produto é também opaco.

### Funções da API

* Produto **criaProduto** (char\* codigoProduto)

Cria um produto com o código passado como parâmetro.

**Retorna:**

O produto criado em caso de sucesso. NULL em caso de falha de alocação.

* Produto **apagaProduto** (Produto p)

Liberta a memória alocada para guardar um produto.

**Retorna:**

É sempre retornado NULL.

* char\* **obterCodigoProduto** (Produto p)

**Retorna:**

Em caso de sucesso é devolvida uma cópia do código do produto passado como parâmetro. Se ocorrer uma falha de alocação, é devolvido NULL.

* char **inicioCodigoProduto** (Produto p)

**Retorna:**

1ª letra do código do produto passado como parâmetro.

* int **comparaCodigosProduto** (Produto p1, Produto p2)

Compara os códigos de dois produtos.

**Retorna:**

É devolvido um valor < 0 se o código de p1 for menor que o de p2, = 0 se os códigos de p1 e p2 forem iguais e > 0 se o código de p1 for maior que o de p2.

* Produto **duplicaProduto** (Produto p)

**Retorna:**

Em caso de sucesso é devolvida uma cópia do produto passado como parâmetro. Se ocorrer uma falha de alocação é devolvido NULL.

## LStrings

A grande quantidade de dados na forma de *Strings*, devolvidos por algumas das funções das APIs que serão apresentadas a seguir, tornou necessário o desenvolvimento de uma estrutura de dados navegável com a noção de índice e de página. Como a estrutura descrita é no fundo uma lista de *Strings* genérica, o grupo designou-a por **LStrings**.

Uma vez criada, uma **LStrings** permite que o utilizador decida se pretende ir para a página seguinte, para a página anterior, para uma página à sua escolha ou então para a primeira ou última página. Estando na página pretendida, o utilizador pode então pedir para obter uma cópia dos conteúdos dessa página e em seguida pode obter os conteúdos dessa página, linha a linha.

### Estruturas de dados

No ficheiro **LStrings.c**, encontram-se as definições dos tipos concretos de dados utilizados para implementar a lista de *Strings* e o tipo página:

|  |  |
| --- | --- |
| **LStrings:**  **struct** lStrings{  int total;  char\*\* strings;  int pag;  int índice;  };  **total** – número de *Strings* da lista de *Strings*;  **strings –** array de *Strings* com os conteúdos da lista;  **pag** – página atual;  **índice** – índice da primeira *String* da página atual; | **Pagina:**  **struct** pagina{  int total;  char\*\* strings;  int índice;  };  **total –** número total de *Strings* na página;  **strings** – array com as *Strings* da página;  **indice** – índice da *String* em que nos encontramos num dado momento; |

Mais uma vez, a implementação de cada uma das estruturas de dados apresentada foi escondida do utilizador declarando no ficheiro **.h** os tipos abstratos de dados como apontadores para os respetivos tipos concretos de dados, i.e.:

**typedef struct** lStrings\* **LStrings**;

**typedef struct** pagina\* **Pagina**;

Ficando assim assegurada a opacidade dos tipos **LStrings** e **Pagina**.

### Número de Strings por página

O número de *Strings* por página de cada **LStrings** criada é fixo e está definido no ficheiro **LStrings.h**, da seguinte forma:



### Funções da API

* LStrings **criaLStrings** (int total, char \*strings[])

Cria uma **LStrings** com de tamanho igual a total, cujas *Strings* são cópias de cada um dos elementos de strings.Se total for menor ou igual a 0, é criada uma lista vazia.

**Retorna:**

Para total >= 0 é devolvida, em caso de sucesso, uma **LStrings** com o número de elementos especificados, cujos conteúdos são cópias dos elementos do *array* strings. Para total < 0, em caso de sucesso, é devolvida uma **LStrings** vazia. Se ocorrer uma falha de alocação, é devolvido NULL.

.

* LStrings **apagaLStrings** (LStrings l)

Liberta a memória alocada para armazenar uma **LStrings**.

**Retorna:**

É sempre devolvido NULL.

* Pagina **obterPag** (LStrings l)

**Retorna:**

Cópia da página atual de uma **LStrings** (ou NULL em caso de falha de alocação).

* char \* **obterLinha** (Pagina pag)

**Retorna:**

Cópia da próxima linha de uma página (ou NULL em caso de falha de alocação).

* Pagina **apagaPag** (Pagina pag)

Liberta a memória alocada para uma página.

**Retorna:**

É sempre retornado NULL.

int **obterNumPag** (LStrings l)

**Retorna:**

Número da página atual da **LStrings** passada como parâmetro.

* int **obterTotal** (LStrings l)

**Retorna:**

Número total de *Strings* na **LStrings** passada como parâmetro.

* int **obterNumTotalPags** (LStrings l)

**Retorna:**

Número total de páginas da **LStrings**.passada como parâmetro.

* void **proxPag** (LStrings l)

Dada uma **LStrings**, passa para a página seguinte da mesma, caso a página atual não seja a última. Se a lista estiver na última página, esta função não tem qualquer efeito.

* void **pagAnt** (LStrings l)

Dada uma **LStrings**, passa para a página anterior da mesma, caso a página atual não seja a primeira. Se a lista estiver na primeira página, esta função não tem qualquer efeito.

* void **irParaPag** (int pag, LStrings l)

Dada uma **LStrings** e uma página pag, vai para a página escolhida se esta for válida. Se pag for inválida, esta função não tem qualquer efeito.

* void **primPag** (LStrings l)

Coloca uma **LStrings** na sua primeira página.

* void **ultimaPag** (**LStrings** l)

Coloca uma **LStrings** na sua última página.

## CatalogoClientes

O módulo **catalogoClientes** tem todas as funções e estruturas de dados utilizadas para guardar, por ordem alfabética, todos os clientes lidos a partir do ficheiro **clientes.txt**.

### Estruturas de dados

Tendo em consideração o facto de cada código de cliente começar por uma letra maiúscula e de existirem 26 iniciais possíveis, o grupo optou por estruturar o catálogo de clientes como um array de 26 AVLs que na posição de índice 0 tem uma AVL com todos os clientes cujo código começa por ‘A’, na posição de índice 1 tem uma AVL com todos os clientes cujo código começa por ‘B’ e assim sucessivamente. Estruturar o catálogo de clientes desta forma tem a vantagem de permitir inserções e procuras bastante mais eficientes, dado que em vez de termos uma AVL com todos os códigos de clientes temos 26 AVLs em que cada uma é relativamente pequena. Esta estruturação também facilita a obtenção de todos os códigos de cliente começados por uma dada letra.

Partindo da análise exposta no parágrafo anterior, o grupo declarou a **struct catClientes**-no ficheiro **catalogoClientes.c**, cuja ilustração passamos a apresentar:

**struct** catClientes {

AVL catalogo[MAX\_AVL];

};

**MAX\_AVL** – macro que corresponde ao valor 26;

**catalogo** – array de 26 AVLs. A AVL na posição de índice 0 tem os clientes cujo código começa por ‘A’, a posição de índice 1 tem os clientes com código começado por ‘B’ e assim sucessivamente;

|  |  |  |  |
| --- | --- | --- | --- |
| [0] | [1] | … | [25] |
| Clientes cujo código começa por ‘A’ | Clientes cujo código começa por ‘B’ | … | Clientes cujo código começa por ‘Z’ |

**Nota:** embora várias funções da API do catálogo de clientes recebam um elemento do tipo **Cliente**, a única informação que precisamos de guardar no catálogo é o código de cada cliente. Assim sendo, cada nodo das AVLs da **struct catClientes** tem apenas uma *String* correspondente ao código de um determinado cliente.

A implementação concreta do catálogo de clientes foi ocultada do utilizador através da declaração, no ficheiro .h, do tipo abstrato **CatClientes** como sendo um apontador para a **struct catClientes**. Assim, **catalogoClientes.h** temos a expressão:

**typedef struct** catClientes\* **CatClientes**;

### Funções da API

* CatClientes **criaCatClientes** ()

Aloca espaço e inicializa um novo catálogo de clientes.

**Retorna:**

Um catálogo de clientes vazio em caso de sucesso. NULL em caso de falha de alocação.

* CatClientes **insereCliente** (CatClientes, Cliente)

Recebe um catálogo de clientes e um cliente. Insere o cliente no catálogo passado para a função.

**Retorna:**

Em caso de sucesso é devolvido o catálogo de clientes após a inserção. Caso contrário, é devolvido NULL.

bool **existeCliente** (CatClientes, Cliente)

Testa se um cliente existe num catálogo de clientes.

**Retorna:**

TRUE se o cliente existir no catálogo passado como parâmetro. FALSE caso contrário.

* int **totalClientes** (CatClientes)

**Retorna:**

Número total de clientes no catálogo de clientes passado como parâmetro.

* int **totalClientesLetra** (CatClientes catC, char)

Recebe um catálogo de clientes e um carater.

**Retorna:**

Número total de clientes no catálogo passado como parâmetro cujo código começa pela letra especificada.

* CatClientes **apagaCatClientes** (CatClientes)

Liberta a memória alocada para um catálogo de clientes.

**Retorna:**

É sempre devolvido NULL.

* Cliente\* **todosClientes** (CatClientes catC)

**Retorna:**

Array de elementos do tipo **Cliente**, com todos os clientes do catálogo passado como parâmetro.

**Nota:** o encapsulamento dos dados é garantido pelo facto de não serem devolvidos os elementos do catálogo de clientes, mas sim novos elementos do tipo **Cliente**, criados a partir de cada um dos códigos de cliente existentes no catálogo catC.

* LStrings **clientesPorLetra** (CatClientes, char)

Recebe um catálogo de clientes e uma letra

**Retorna:**

Lista de *Strings* navegável (**LStrings**) com os códigos dos clientes começados pela letra especificada no 2º argumento.

## CatalogoProds

O módulo **catalogoProds** é semelhante ao módulo **catalogoClientes**, visto que contém as estruturas que permitem guardar, por ordem alfabética, todos os produtos lidos a partir do ficheiro **produtos.txt**.

### Estruturas de dados

Os códigos de produto, tal como os códigos de cliente, começam por uma das 26 letras maiúsculas do alfabeto. Além disso, o tipo de operações que se pretende realizar sobre um catálogo de produtos é idêntico ao tipo de operações efetuadas sobre um catálogo de clientes, logo a estruturação que escolhemos para o catálogo de produtos é idêntica à que já foi apresentada para o catálogo de clientes.

A única diferença relativamente ao catálogo de clientes é que neste caso estamos a guardar códigos de produtos e portanto temos uma estrutura designada por **struct catProds**-no ficheiro **catalogoProdutos.c**, que pode ser ilustrada por:

**struct** catProds{

AVL catalogo[MAX\_AVL];

};

**MAX\_AVL** – macro que corresponde ao valor 26;

**catalogo** – array de 26 AVLs. A AVL na posição de índice 0 tem os produtos cujo código começa por ‘A’, a posição de índice 1 tem os produtos com código começado por ‘B’ e assim sucessivamente;

|  |  |  |  |
| --- | --- | --- | --- |
| [0] | [1] | … | [25] |
| Produtos cujo código começa por ‘A’ | Produtos cujo código começa por ‘B’ | … | Produtos cujo código começa por ‘Z’ |

**Nota:** cada nodo das AVLs da **struct catProds** tem apenas uma *String* correspondente ao código de um determinado produto.

A definição do tipo abstrato de dados do catálogo de produtos é também semelhante às restantes definições de TADs apresentadas anteriormente, pelo que temos, no ficheiro **catalogoProdutos.h**, a linha:

**typedef struct** catProds\* **CatProds**;

### Funções da API

* CatProds **criaCatProds** ()

Aloca espaço e inicializa um novo catálogo de produtos.

**Retorna:**

Um catálogo de produtos vazio em caso de sucesso. NULL em caso de falha de alocação.

* CatProds **insereProduto** (CatProds, Produto)

Recebe um catálogo de produto e um produto. Insere o produto no catálogo passado para a função.

**Retorna:**

Em caso de sucesso é devolvido o catálogo de produtos após a inserção. Caso contrário, é devolvido NULL.

bool **existeProduto** (CatProds, Cliente)

Testa se um produto existe num catálogo de produtos.

**Retorna:**

TRUE se o produto existir no catálogo passado como parâmetro. FALSE caso contrário.

* int **totalProdutos** (CatProds)

**Retorna:**

Número total de produtos no catálogo de produtos passado como parâmetro.

* int **totalProdutosLetra** (CatProds catP, char)

Recebe um catálogo de produtos e um carater.

**Retorna:**

Número total de produtos no catálogo passado como parâmetro cujo código começa pela letra especificada.

* CatProds **apagaCatProds** (CatProds)

Liberta a memória alocada para um catálogo de produtos.

**Retorna:**

É sempre devolvido NULL.

* LStrings **prodsPorLetra** (CatProds, char)

Recebe um catálogo de produtos e uma letra

**Retorna:**

Lista de *Strings* navegável (**LStrings**) com os códigos dos produtos começados pela letra especificada no 2º argumento.

## FaturacaoGlobal

O módulo **FaturacaoGlobal** permite obter informações relativas às vendas mensais e/ou anuais dos vários produtos, como:

* O total de vendas e total faturado com um produto num dado mês, em modo Normal (**N**) e em Promoção (**P**) (***query* 3**);
* A lista ordenada dos códigos de produtos que ninguém comprou (***query* 4**);
* Total de vendas e total faturado num intervalo fechado de meses (***query* 6**);
* Quais foram os **N** produtos mais vendidos do ano (***query* 10**);

A faturação global possibilita também a distinção das vendas/faturação de cada filial e referencia todos os produtos, mesmo os não vendidos, de forma a garantir uma resposta eficiente à ***query* 4**. Note-se, contudo, que não é feita qualquer referência a clientes e qualquer informação relativa aos mesmos deve ser obtida no módulo **Filial**.

### Macros

No ficheiro **faturcaoGlobal.c** definimos as seguintes macros:



### Estruturas de dados

Ao analisarmos as *queries* a que a faturação global deve ser capaz de responder constatamos que duas delas (as *queries* 3 e 6) têm como chave de pesquisa o mês, enquanto que as outras duas (as *queries* 4 e 10) requerem informações sobre as vendas do ano. Tendo estes aspetos em consideração, optamos por ter duas estruturas principais distintas na faturação global:

* Uma estrutura em que são referenciados todos os produtos e que permite obter informação relativa às vendas anuais de cada um deles, para cada filial;
* Uma estrutura que tem informação sobre as vendas de cada mês e que permite distinguir compras **N** de compras **P**, bem como as vendas/faturação de cada filial;

Partindo da análise acima descrita, optamos pela seguinte estruturação dos dados:

**FatAnualProd** (nodo da AVL todosProdutos)

**struct** fatAnualProd{

**char\*** prod;

**int** totalUnids[N\_FILIAIS+1];

};

**prod** – *String*com o código de um produto;

**totalUnids** – *array* que na posição de índice i tem o número de unidades do produto prod vendidas durante o ano na filial i;

**FaturacaoGlobal:**

**struct** fatGlobal{

**AVL** todosProdutos;

**FatMes** fatMensal[N\_MESES+1];

};

**todosProdutos** – AVL com informação sobre as vendas anuais de todos os produtos;

**fatMensal** – *array* de estruturas com informação relativas à faturação de cada mês;

**Nota:** o *array* **fatMensal** tem comprimento N\_MESES+1 para que o possamos começar a indexar a partir da posição de índice 1.

**FatProdMes** (nodo da AVL fatProds)

**struct** fatProdMes{

**char\*** prod;

**int** unidsVend[N\_TIPOS\_VENDA][N\_FILIAIS+1];

**double** faturacao[N\_TIPOS\_VENDA][N\_FILIAIS+1];

};

**prod** – *String* com o código do produto vendido;

**unidsVend** – matriz que na linha 0 tem a quantidade vendida do produto prod em vendas normais, para cada filial, e na linha 1 tem a quantidade vendida em promoção, para cada filial;

**faturacao –** matriz análoga a **unidsVend**, que guarda informação sobre a faturação de prod;

**FatMes:**

**struct** fatMes{

**int** totalVendas;

**double** totalFaturado;

**AVL** fatProds;

};

**totalVendas** – total de vendas do mês;

**totalFaturado** – total faturado no mês;

**fatProds** – AVL com informação sobre a faturação de cada um dos produtos vendidos no mês;

### Definição de tipos

**Tipos auxiliares** (i.e.: definidos em **faturacaoGlobal.c**)

As *structs* **fatMes** e **fatAnualProd** são estruturas auxiliares. O utilizador não necessita de ter conhecimento das mesmas para obter as várias informações que possa desejar sobre a faturação dos produtos. Por esse motivo, a definição dos tipos **FatMes** e **FatAnualProd** foi colocada no ficheiro **faturacaoGlobal.c**, sendo expressa por:

**typedef struct** fatMes\* **FatMes**;

**typedef struct** fatAnualProd\* **FatAnualProd;**

**Tipos exportados** (i.e.: definidos em **faturacaoGlobal.h**)

O utilizador terá naturalmente que conhecer o tipo abstrato de dados **FaturacaoGlobal**, logo no ficheiro **faturacaoGlobal.h** temos:

**typedef struct** fatGlobal **FaturacaoGlobal**;

O grupo optou também por exportar o tipo abstrato **FatProdMes**, que não é mais do que a referência da **struct fatProdMes**, cuja implementação o utilizador desconhece. Assim, no ficheiro **faturacaoGlobal.h** temos também:

**typedef struct** fatProdMes **FatProdMes**;

A razão para exportamos este tipo está relacionada com a **query 3**, uma vez que nesta *query* o utilizador precisa de obter, para um dado produto e um dado mês, o total faturado em modos **N** e **P** (podendo optar por obter resultados globais ou por filial). Ora, a **struct** **fatProdMes** tem todas essas informações, portanto optamos por permitir que o utilizador obtenha, com a função **obterFatProdMes()**, a referência da **struct fatProdMes** para o produto e mês que pretende. Posto isto, o utilizador pode, com algumas das funções da API da faturação global, obter informação relativa aos campos que pretende. É importante notar que o tipo **FatProdMes** é opaco, logo o utilizador não tem conhecimento sobre a sua implementação, não consegue desreferenciá-lo e tem obrigatoriamente que usar as funções disponibilizadas na API de forma a conseguir as informações que deseja.

### Funções da API

* FaturacaoGlobal **criaFaturacaoGlobal** ()

Cria a faturação global.

**Retorna:**

A faturação global criada em caso de sucesso. NULL em caso de falha de alocação.

* FaturacaoGlobal **apagaFaturacaoGlobal** (FaturacaoGlobal fg)

Liberta a memória alocada para armazenar a faturação global.

**Retorna:**

É sempre retornado NULL.

1. FaturacaoGlobal **registaProduto** (FaturacaoGlobal fg, Produto p)

Regista um produto na faturação global.

**Retorna:**

A faturação global após a inserção, em caso de sucesso. NULL em caso de falha de alocação.

* FaturacaoGlobal **registaVenda** (

FaturacaoGlobal fg,

Produto p,

double precoUnit,

int quantidade,

TipoVenda tipo,

int filial,

int mês

)

Regista uma venda na faturação global.

**Retorna:**

A faturação global após a inserção, em caso de sucesso. NULL em caso de falha de alocação.

* int **totalVendasMes** (const FaturacaoGlobal fg, int mes)

**Retorna:**

Total de vendas num dado mês.

* double **totalFaturadoMes** (const **FaturacaoGlobal** fg, int mes)

**Retorna:**

Total faturado num dado mês.

* int **totalVendasIntervMeses** (const **FaturacaoGlobal** fg, int inicio, int fim)

Recebe uma faturação global e o início e o fim de um intervalo fechado meses.

**Retorna:**

Total de vendas registadas no intervalo de meses [inicio, fim].

**Nota:** se iniciofor maior que fim, considera-se que o intervalo é um conjunto vazio, pelo que o total de vendas retornado nesses casos é 0.

* double **totalFatIntervMeses** (const FaturacaoGlobal fg, int inicio, int fim)

Recebe uma faturação global e o início e o fim de um intervalo fechado meses.

**Retorna:**

Total faturado no intervalo fechado de meses [inicio, fim].

**Nota:** tal como na função **totalVendasIntervMeses()**, se iniciofor maior que fim, considera-se que o intervalo é um conjunto vazio, pelo que a faturação total retornada nesses casos é 0.

* FatProdMes **obterFatProdMes** (const FaturacaoGlobal fg, const Produto p, int mes)

Recebe uma faturação global, um produto e um mês.

**Retorna:**

Cópia da faturação do produto p, no mês especificado, em caso de sucesso. NULL

em caso de falha de alocação..

* FatProdMes **apagaFatProdMes** (FatProdMes fProdMes)

Liberta a memória alocada para guardar o resultado de **obterFatProdMes()**.

**Retorna:**

É sempre retornado NULL.

* int **totalUnidsProdMes** (const FatProdMes fProdMes, TipoVenda tipo)

Recebe a faturação de um produto num mês (obtida com **obterFatProdMes())** e um tipo de venda.

**Retorna:**

Total de unidades vendidas globalmente, que estão registadas em fProdMes, para o tipo de venda escolhido.

* int \* **unidsPorFilialProdMes** (const FatProdMes fProdMes, TipoVenda tipo)

Recebe a faturação de um produto num mês (obtida com **obterFatProdMes())** e um tipo de venda.

**Retorna:**

Array de que na posição de índice i tem o número de vendas registadas em fProdMes, para a filial i e para o tipo de venda escolhido.

**Nota:** de forma a preservar o encapsulamento, o array devolvido é uma cópia daquele que se encontra em fProdMes.

* double **faturacaoTotalProdMes** (const FatProdMes fProdMes, TipoVenda tipo)

Recebe a faturação de um produto num mês (obtida com **obterFatProdMes())** e um tipo de venda.

**Retorna:**

Faturação total a nível global, registada em fProdMes, para o tipo de venda escolhido.

* double \* **faturacaoPorFilialProdMes** (const FatProdMe**s** fProdMes, TipoVenda tipo)

Recebe a faturação de um produto num mês (obtida com **obterFatProdMes())** e um tipo de venda.

**Retorna:**

Array de que na posição de índice i tem a faturação total registada em fProdMes, para a filial i e para o tipo de venda escolhido.

**Nota:** de forma a preservar o encapsulamento, o *array* devolvido é uma cópia daquele que se encontra em fProdMes.

* int **quantosNaoComprados** (const FaturacaoGlobal fg)

Recebe uma faturação global.

**Retorna:**

Número de produtos que não foram comprados em nenhuma filial, durante o ano.

* LStrings **naoCompradosGlobal** (const FaturacaoGlobal fg)

Recebe uma faturação global.

**Retorna:**

Lista de *Strings* navegável com os produtos que não foram comprados em nenhuma f filial, durante o ano.

* LStrings \* **naoCompradosPorFilial** (const FaturacaoGlobal fg)

Recebe uma faturação global.

***Retorna:***

*Array de lista de strings em que a lista na posição i tem os códigos de todos os produtos que não foram comprados durante o ano na filial i.*

* char \*\* **NmaisVendidosFilial** (const FaturacaoGlobal fg, int N, int filial)

Recebe uma faturação global, uma filial e um número 'N'.

**Retorna:**

*Array* com os códigos dos 'N' produtos mais vendidos nessa filial, ordenados decrescentemente pelo total de vendas.

## Filial

O módulo **Filial** permite obter informações relativas às compras realizadas por cada cliente, discriminando os códigos dos produtos comprados por cada cliente, o mês e a filial onde foram comprados, os tipos de compras que foram realizados (P e/ou N), o total gasto na compra desses produtos e o total de unidades compradas desse produto por mês. Este módulo foi estruturado com o objetivo de ter uma resposta eficiente às seguintes *queries*:

* Nº total de produtos comprados pelo cliente, mês a mês e por filial (**Query 5**);
* Lista ordenada de códigos de cliente que realizaram compras em todas as filiais (**Query 7**)
* Códigos e nº total de clientes que compraram um produto, distinguindo as compras P das N (**Query 8**);
* Lista de códigos de produto que um cliente mais comprou (**Query 9**);
* Calcular os N produtos mais vendidos em todo o ano, indicando para cada filial, o total de clientes que o compraram e o total de unidades vendidas(**Query10**)
* Código dos 3 produtos em que um cliente mais gastou dinheiro durante o ano (**Query 11**)
* Número de clientes que não realizaram compras e número de produtos não vendidos (**Query 12**).

**Nota:** neste módulo só se faz referencia a clientes que fizeram compras e a produtos que foram efetivamente comprados, de maneira a tornar a estrutura mais leve e tornar as travessias/procuras na árvore mais rápidas.

### Estruturas de dados

Analisando as queries, rapidamente se percebe que a chave principal de pesquisa neste módulo é o código de clientes, uma vez que 5 das 6 queries que fazem uso deste módulo têm como uma das chaves o código de cliente. Atendendo a isto, estruturamos a Filial da seguinte forma:

### Definições dos tipos:

Como vimos anteriormente, existem 2 tipos(alias) para AVL's de forma a clarificar o tipo dos nodos:

**typedef AVL AVL\_ComprasPorCliente**;

**typedef AVL AVL\_ComprasDoProduto**;

Estes tipos estão definidos no ficheiro **filial.c** e não faz sentido que estejam no ficheiro **filial.h** porque o utilizador da API não precisa sequer de saber que estamos a implementar o módulo à custa de AVLs.

O único tipo exportado deste módulo é o tipo Filial, definido como:

**typedef struct filial\* Filial;**

Como o ficheiro filial.h não revela a constituição das estruturas, mantemos o tipo Filial completamente opaco aos utilizadores da API. Através das funções da API definidas abaixo, conseguimos obter toda a informação que precisamos para as queries, a partir de uma variável do tipo Filial.

### Funções da API

* Filial **criaFilial** ()

Aloca espaço e inicializa uma Filial.

**Retorna:**

A filial criada, em caso de sucesso. NULL em caso de falha de alocação.

* Filial **apagaFilial** (Filial filial)

Liberta o espaço alocado para guardar uma Filial.

**Retorna:**

É sempre devolvido NULL.

* Filial **registaCompra** (

Filial filial,

Cliente cliente,

Produto produto,

int mes,

TipoVenda tipoVenda,

int unidades,

double preco

)

Regista os dados de uma compra numa filial.

**Retorna:**

A filial resultante, em caso de sucesso. NULL em caso de falha de alocação.

* int\* **unidadesClientePorMes** (Filial filial, Cliente cliente)

**Retorna:**

Em caso de sucesso é devolvido um *array* com 13 entradas (12 válidas) em que a entrada i guarda o número de unidades compradas no mês i pelo cliente escolhido. Em caso de falha de alocação, é devolvido NULL.

* Cliente\* **clientesCompraramNaFilial** (Filial filial)

Recebe uma Filial.

**Retorna:**

*Array* com os clientes que compraram na Filial passada como parâmetro para a função.

* bool **clienteComprouNaFilial** (Filial filial, Cliente cliente)

**Retorna:**

TRUE se o cliente passado como 2º argumento tiver feito pelo menos uma compra na filial passada como 1º argumento. FALSE c.c.

* int **quantosClientesCompraram** (Filial filial)

**Retorna:**

Número total de clientes que compraram na filial passada como parâmetro.

* LStrings **produtosClienteMaisComprou** (Filial filial, Cliente c, int mes)

Recebe uma Filial, um cliente e um mês.

**Retorna:**

Em caso de sucesso, é devolvida uma lista de *Strings* com os códigos de produto que o cliente mais comprou (por quantidade) nessa filial. Em caso de falha de alocação é devolvido NULL.

* char \*\* **tresProdsEmQueMaisGastou** (Filial filial, Cliente c)

Recebe uma Filial e um cliente.

**Retorna:**

Em caso de sucesso é devolvido um *array* com os códigos dos 3 produtos em que o cliente passado como parâmetro gastou mais dinheiro durante o ano, ordenados decrescentemente pelo total gasto neles. Em caso de falha de alocação, é devolvido NULL.

* void **comprou** (Filial filial, Cliente cliente, Produto produto, bool \*comprouN, bool \*comprouP)

Dado um cliente, uma Filial e um produto, verifica se o cliente comprou em modo Normal e se comprou em modo de promoção. Os resultados são guardados em cada uma das variáveis passadas por referência.

* int **numeroClientesCompraramProduto** (Filial filial, char \*produto, int \*unidadesCompradas)

Recebe uma Filial e um código de Cliente.

**Retorna:**

Número de clientes que compraram o Produto. A quantidade comprada é devolvida na variável apontada por unidadesCompradas.

2. Módulo de alocação/desalocação de memória

## MemUtils

A necessidade frequente de alocar *arrays* de estruturas e de os libertar quando já não são necessários levou o grupo a criar o módulo **MemUtils**, de forma a simplificar o código dos restantes módulos e torná-lo mais legível e fácil de manter.

### Funções da API

* void \*\* **alocaMatriz** (int nlinhas, int ncolunas, size\_t nBytesElem)

Aloca memória para armazenar uma matriz genérica, de dimensões nlinhas X ncolunas em que o número de bytes de cada elemento é dado por nBytesElem (3º argumento).

**Retorna:**

Em caso de sucesso é devolvida a matriz alocada. Se ocorrer uma falha de alocação é devolvido NULL.

* void **apagaArray** (void\* array[], int total, void(\*apagaElem)(void \*))

Recebe um array de apontadores, o seu número total de elementos e a referência de uma função apagaElem(), a utilizar para libertar cada elemento do *array*. Liberta a memória alocada para guardar o array passado como 1º argumento.

3. Módulo auxiliar de leitura e processamento de input

## Leitura

No módulo principal (**main.c**) do programa **GereVendas** é bastante frequente termos que ler linhas e em seguida remover o(s) seu(s) caratere(s) de *newline*, sendo também bastante comum termos que ler inteiros ou simplesmente um único carater. Todo este código de leitura e processamento de *input* poderia ser colocado na **main.c**, mas nesse caso estaríamos a repetir código e a tornar a **main.c** mais extensa do que o necessário. Para evitar essa repetição de código e de forma a agrupar as funções mais gerais de leitura e processamento de *input*, criamos o módulo **leitura.c + leitura.h**, cujas macros e API apresentamos de seguida.

### Macros



A macro FLUSH\_STDIN() é utilizada para consumir os carateres que ficam no *buffer* do *stdin*, quando parte de uma linha introduzida pelo utilizador não é consumida. Deste modo, evita-se que as leituras do *standard input* recebam como *input* carateres que ficaram no *buffer* do *stdin* no final da leitura anterior.

### Funções da API

* char\* **leLinha** (char buffer[], int tamanho, FILE\* stream)

Lê a partir de uma stream uma linha com o tamanho especificado no 2º argumento e guarda-a no buffer passado como 1º argumento, sem o(s) caratere(s) de newline. São lidos no máximo 'tamanho' – 1 carateres, porque o útlimo carater é o '\0'. Se a stream for o *stdin* e *ficarem* caratéres no buffer do mesmo após a leitura da linha, é feito o *flush* do *stdin*.

**Retorna:**

Se for lido o EOF antes de qualquer outra carater, a função leLinha() devolve NULL, se não, devolve a linha lida.

* int **avancaEspacosInicio** (char str[])

Recebe uma *String*.

**Retorna:**

Índice do primeiro carater da *Strings* passada como parâmetro que não é um espaço. A definição de espaço está de acordo com a da função isspace() de <ctype.h>.

* int **leInt** ()

Lê um valor inteiro do *stdin* e faz o flush do mesmo, se necessário.

**Retorna:**

Inteiro lido. Se não for possível converter os carateres introduzidos para um valor interiro, é devolvido 0.

* double **leDouble** ()

Lê um *double* do *stdin* e faz o flush do mesmo, se necessário.

**Retorna:**

Double lido. Se não for possível converter os carateres introduzidos para um *double*, é devolvido 0.

* int **leChar** ()

Lê um carater do *stdin* e faz o flush do mesmo, se necessário.

**Retorna:**

É devolvido o carater lido, em caso de sucesso. Em caso de erro ou de *end of file*, é devolvido EOF.

# 4. Módulo principal (main.c)

Uma vez apresentados os módulos de dados, leitura e alocação de memória, passamos agora para a exposição dos aspetos mais importantes do módulo principal do programa **GereVendas**.

## Estruturas de dados

Todas as estruturas de dados utilizadas no ficheiro **main.c** são importadas dos vários módulos de dados expostos até agora. Assim sendo, antes da função **main()** temos:



**catClientes**  - catálogo de clientes que guarda os clientes do ficheiro **clientes.txt**;

**catProds** – catálogo de produtos que guarda os produtos do ficheiro **produtos.txt**;

**faturacaoGlobal** – referencia todos os produtos e guarda informação sobre a faturação mensal de cada produto e a faturação do ano todo;

**filiais** – *array* de comprimento 4, cuja posição de índice 0 guarda informação relativa a todas as compras de todas as filiais. As posições 1, 2 e 3 do *array* de filiais guardam informação relativa às filiais 1, 2 e 3, respetivamente.

**fichCarregados** – variável do tipo *bool* que indica se os ficheiros de dados estão carregados.

Ao declararmos as variáveis acima apresentadas como **static**, fora de qualquer função do ficheiro **main.**c, garantimos que estas são globais a **main.c** e ao mesmo tempo privadas aos restantes módulos, ficando assegurado o encapsulamento dos dados.

## Estruturação do programa principal

### Ecrã inicial

Quando o programa principal começa a ser executado é apresentado um *splash screen* e a indicação “prima enter para continuar”.

### Interpretador de comandos

Mal o utilizador prima ENTER no ecrã inicial, é invocada a função **interpretador()**, responsável por apresentar o menu com as opções do programa **GereVendas**, ler o *input* do utilizador e invocar a função **interpreta()** para o interpretar. O menu não é mais do que uma enumeração dos vários comandos que o utilizador tem à sua escolha. Para selecionar uma opção, o utilizador só precisa de ver qual é o seu número da mesma no menu e introduzi-lo.

Tal como já foi referido, após ser introduzida uma opção, esta é passada para função **interpreta()**. Esta função começa por avançar os espaços que o utilizador possa ter introduzido no início da linha, até encontrar um carater que não seja um espaço:

* Se o 1º carater que não é um espaço for o ‘\0’, o utilizador introduziu uma linha em branco e nesse caso regressamos à função **interpretador()** para que seja apresentado novamente o menu e lida uma nova opção;
* Se o 1º carater que não é um espaço for diferente de ‘\0’, é chamada a função atoi() para converter a linha, a partir desse carater, para um valor inteiro:
  + Se o resultado da conversão for menor ou igual a 0 ou maior que 12, a opção é inválida e é apresentada uma mensagem de erro;
  + Se a opção fizer parte do menu, mas corresponder a uma *query* e o utilizador não tiver os ficheiros carregados, é também apresentada uma mensagem de erro;
  + Caso contrário, a opção é válida e portanto é executada;

### Invocação da função correspondente a uma opção

Para evitar utilizar um bloco grande de “if (…) else if (…) else”, optamos pela utilização de uma *dispatch table*, i.e.: um array de apontadores para funções que na posição de índice i tem o endereço da função correspondente à opção i. No início do ficheiro **main.c** temos então:



### Query 1: Leitura dos ficheiros

Quando o utilizador seleciona a opção “Ler ficheiros”:

1. É feita a verificação do valor da *flag* **fichCarregados** e se esta tiver o valor lógico TRUE, é chamada a função auxiliar **apagaEstruturas()**, responsável por invocar as funções da APIs dos catálogos, faturação global e filial que libertam a memória alocada para as estruturas carregadas anteriormente. Antes de terminar, a função **apagaEstruturas()** coloca a *flag* **fichCarregados** com o valor lógico FALSE;
2. É invocada a função auxiliar **criaEstruturas()** que por sua vez invoca as funções **criaCatProds()**, **criaCatClientes(), criaFaturacaoGlobal()** e **criaFilial()** (para cada filial do *array* de filiais). Se alguma alocação falhar, a função **criaEstruturas()** devolve ERRO\_MEM (uma macro que corresponde ao valor -1 e permite que as várias funções auxiliares indicar falhas de alocação);
3. É chamada a função auxiliar **leFicheiros()**, responsável por dar a escolher ao utilizador se pretende, para cada um dos ficheiros (clientes, produtos e vendas), introduzir o caminho do mesmo ou premir ENTER para carregar o ficheiro por omissão. Para cada um dos ficheiros, existe também uma função auxiliar, responsável pela leitura e processamento do mesmo e pela invocação das funções de inserção nas estruturas de dados adequadas. No caso dos ficheiros de produtos e de clientes, assume-se que estes estão sintaticamente corretos, pelo que não é feita qualquer validação. Já no caso do ficheiro de vendas, os dados de cada linha poderão ser inválidos, pelo que cada linha é passada para a função auxiliar **insereSeValida()** que só regista os dados de uma venda na faturação global e nas filiais se estes forem válidos;

**Nota:** como a faturação global deve referenciar todos os produtos, ao lermos o ficheiro **produtos.txt**, inserimos uma cópia de cada código de produto no catálogo de produtos e outra na faturação global.

Os ficheiros por omissão estão definidos nas seguintes macros:



### Navegação

Antes de apresentarmos a query 2, convém primeiro referir como navegamos na quantidade considerável de *Strings* produzida por várias *queries* (nomeadamente a *query* 2).

Como já foi visto anteriormente, a API de **LStrings** já tem todas as funções necessárias para navegar numa lista de *Strings* e obter páginas da mesma. Assim sendo, no ficheiro **main.c** bastou definir uma função **navega()** que recebe uma **LStrings** (a que chamaremos lStr).

Se lStr for uma lista vazia, é apresentada a mensagem “Lista vazia” e não é feito qualquer tipo de navegação.

Se lStr tiver pelo menos uma *String*, é chamada a função auxiliar **imprimeInformacaoLStrings()** para imprimir no ecrã o número de entradas, o número total de páginas e o número de entradas por página de lStr. Em seguida entramos no ciclo que:

1. Limpa o ecrã;
2. Invoca a função **obterPag()**, para obter a página atual de lStr;
3. Passa a página obtida para a função auxiliar **apresentaPag()** que vai invocar ciclicamente **obterLinha()**, seguida de **puts()** para apresentar cada uma das linhas da página recebida;
4. Apresenta o menu da navegação:

 **Nota: <total>** indica o local onde é impresso o número total de páginas da **LStrings**.

1. Lê um inteiro com a opção do utilizador;
   1. Se a opção for inválida, apresenta uma mensagem de erro e regressa ao ponto número 1. ;
   2. Se a opção for válida e diferente de 7, executa-a e regressa ao ponto número 1;
   3. Se a opção for igual a 7 (Sair), sai da navegação;

A invocação das opções contidas no intervalo [1,6] é feita através de uma *dispatch table*, definida no início do ficheiro **main.c** como:



* **pagAnt()**, **proxPag()**, **primPag()** e **ultimaPag()** são funções da API de **LStrings**;
* **perguntaPag()** e **imprimeInformacaoLStrings()** são funções auxiliares definidas no ficheiro **main.c**, uma vez que realizam IO.
  + **perguntaPag()** pergunta ao utilizador para que página pretende ir e se essa página for válida, invoca a função **irParaPag()** da API de **LStrings**;
  + **imprimeInformacaoLStrings()**, apresenta novamente as informações que são impressas antes da 1ª página de uma **LStrings** não vazia ser apresentada;

### Query 2: Lista e nº total de produtos começados por uma letra

Como temos a função **prodsPorLetra()** na API do catálogo de produtos, ao implementarmos a função **query2()** no ficheiro **main.c** só precisamos de pedir ao utilizador para introduzir uma letra, validar a letra introduzida e, se esta for válida, invocar **prodsPorLetra()** para obter uma **LStrings** com os códigos dos produtos começados pela letra que o utilizador escolheu. A **LStrings** obtida é então passada para a função **navega()**, permitindo que o utilizador navegue no resultado.

### Query3: Quantidade vendida e faturação total de um produto num dado mês

A função **query3()** lê o código de um produto e um mês, testa a validade de cada um dos valores lidos e, se estes estiverem corretos, invoca a função **obterFatProdMes()** da API da faturação global, para obter a referência da estrutura com as várias informações sobre faturação do produto escolhido, no mês dado. A partir daqui basta perguntar ao utilizador se pretende resultados globais ou por filial e, consoante a resposta, invocar as funções da API da faturação global que permitem, a partir de um elemento do tipo abstrato de dados **FatProdMes**, obter os resultados que são pedidos.

### Query4: Lista de produtos não comprados globalmente ou por filial