|  |  |
| --- | --- |
| 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)

# 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.

## 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;

Note-se que ao colocarmos estas definições 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 seguinte 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