

# LibRCG: biblioteca de funções C

## Versão 2.1.2

Rui Carlos A. Gonçalves  
rcgoncalves.pt@gmail.com

7 de Fevereiro de 2009

## Conteúdo

<a href="#">1 LibRCG</a>	1
<a href="#">2 Índice de Estruturas de Dados</a>	1
<a href="#">3 Índice de Ficheiros</a>	2
<a href="#">4 Documentação das Estruturas de Dados</a>	3
<a href="#">5 Documentação dos Ficheiros</a>	9

## 1 LibRCG

A [LibRCG](#) é um conjunto de bibliotecas de funções em C, desenvolvidas por [Rui Carlos A. Gonçalves](#) <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>.

Entre outras coisas, estas bibliotecas contêm:

- Funções de leitura de dados do teclado, de ordenação, etc;
- Funções que manipulam *strings*;
- Implementação de funções de programação linear;
- Definição de estruturas de dados derivadas de "listas", respectivas funções que as manipulam;
- Implementação de funções finitas (árvores binárias de procura equilibradas e tabelas de *Hash*);
- Implementação de um iterador.

### 1.1 Licença

Este software é licenciado sob a [CC-GNU LGPL](#).

### 1.2 Downloads

- [Código Fonte](#)
- [Documentação HTML](#)
- [Documentação PDF](#)

## 2 Índice de Estruturas de Dados

### 2.1 Estruturas de Dados

Lista das estruturas de dados com uma breve descrição:

<a href="#">SArray (Definição da estrutura do array)</a>	3
--	---

<a href="#">SHashMap</a> (Estrutura de uma tabela de hash )	3
<a href="#">SHashNode</a> (Estrutura do nodo de uma tabela de hash )	4
<a href="#">SIterator</a> (Estrutura do iterador )	4
<a href="#">SList</a> (Estrutura da lista )	5
<a href="#">SListNode</a> (Estrutura do nodo da lista )	5
<a href="#">SPQueue</a> (Estrutura da pqueue )	6
<a href="#">SPQueueNode</a> (Estrutura do nodo da pqueue )	6
<a href="#">SQueue</a> (Estrutura da queue )	6
<a href="#">SQueueNode</a> (Estrutura do nodo da queue )	7
<a href="#">SStack</a> (Estrutura de uma stack )	7
<a href="#">SStackNode</a> (Estrutura do nodo de uma stack )	8
<a href="#">STreeMap</a> (Estrutura de uma árvore )	8
<a href="#">STreeNode</a> (Estrutura do nodo da árvore )	8

## 3 Índice de Ficheiros

### 3.1 Lista de Ficheiros

Lista de todos os ficheiros documentados com uma breve descrição:

<a href="#">array.c</a> (Implementação de um array dinâmico )	9
<a href="#">array.h</a> (Implementação de um array dinâmico )	14
<a href="#">hashmap.c</a> (Implementação de uma tabela de hash )	19
<a href="#">hashmap.h</a> (Implementação de uma tabela de hash )	24
<a href="#">iterator.c</a> (Implementação de um iterador )	30
<a href="#">iterator.h</a> (Implementação de um iterador )	34
<a href="#">list.c</a> (Implementação de uma lista (duplamente) ligada )	39
<a href="#">list.h</a> (Implementação de uma lista (duplamente) ligada )	45
<a href="#">mainpage.h</a>	??
<a href="#">pqueue.c</a> (Implementação de uma pqueue como lista ligada )	52
<a href="#">pqueue.h</a> (Implementação de uma queue com prioridades )	56
<a href="#">queue.c</a> (Implementação de uma queue como lista ligada )	60

<a href="#">queue.h</a> (Implementação de uma queue como lista ligada )	64
<a href="#">rlp.c</a> (Implementação de funções de programação linear )	68
<a href="#">rlp.h</a> (Implementação de funções de programação linear )	71
<a href="#">rstring.c</a> (Implementação de funções que manipulam strings )	73
<a href="#">rstring.h</a> (Implementação de funções que manipulam strings )	76
<a href="#">stack.c</a> (Implementação de uma stack como lista ligada )	79
<a href="#">stack.h</a> (Implementação de uma stack como lista ligada )	83
<a href="#">treemap.c</a> (Implementação de uma árvore binária de pesquisa equilibrada )	86
<a href="#">treemap.h</a> (Implementação de uma árvore binária de pesquisa equilibrada )	98
<a href="#">util.c</a> (Implementação de funções auxiliares básicas )	105
<a href="#">util.h</a> (Implementação de funções auxiliares básicas )	108

## 4 Documentação das Estruturas de Dados

### 4.1 Referência à Estrutura SArray

#### 4.1.1 Descrição Detalhada

Definição da estrutura do array.

Definido na linha 22 do ficheiro array.h.

#### Campos de Dados

- int [capacity](#)  
*Capacidade máxima de elementos do array.*
- int [size](#)  
*Número de elementos do array.*
- void \*\* [array](#)  
*Array de apontadores.*

### 4.2 Referência à Estrutura SHashMap

#### 4.2.1 Descrição Detalhada

Estrutura de uma tabela de hash.

Definido na linha 77 do ficheiro hashmap.h.

### Campos de Dados

- `int(* hash)(void *)`  
*Função de hash.*
- `int(* equals)(void *, void *)`  
*Função que compara duas chaves.*
- `int size`  
*Número de elementos contidos na tabela.*
- `int length`  
*Tamanho do array que constitui a tabela.*
- `float factor`  
*Factor de "reestruturação" da tabela.*
- `HashNode * elems`  
*Tabela de apontadores.*

## 4.3 Referência à Estrutura SHashNode

### 4.3.1 Descrição Detalhada

Estrutura do nodo de uma tabela de hash.

Definido na linha 59 do ficheiro hashmap.h.

### Campos de Dados

- `void * key`  
*Apontador para a chave do nodo.*
- `void * value`  
*Apontador para o valor associado ao nodo.*
- `struct sHashNode * next`  
*Apontador para o próximo nodo.*

## 4.4 Referência à Estrutura SIterator

### 4.4.1 Descrição Detalhada

Estrutura do iterador.

Definido na linha 18 do ficheiro iterator.h.

### Campos de Dados

- int `capacity`  
*Dimensão do iterador.*
- int `size`  
*Número de elementos.*
- int `pos`  
*Posição actual.*
- void \*\* `values`  
*Apontadores para os elementos do iterador.*

## 4.5 Referência à Estrutura SList

### 4.5.1 Descrição Detalhada

Estrutura da lista.

Definido na linha 37 do ficheiro list.h.

### Campos de Dados

- int `size`  
*Número de elementos da lista.*
- `ListNode first`  
*Apontador para o início da lista.*
- `ListNode last`  
*Apontador para o fim da lista.*

## 4.6 Referência à Estrutura SListNode

### 4.6.1 Descrição Detalhada

Estrutura do nodo da lista.

Definido na linha 19 do ficheiro list.h.

### Campos de Dados

- void \* `inf`  
*Apontador para a informação do nodo.*
- struct `sListNode` \* `prev`  
*Apontador para o nodo anterior.*

- struct sListNode \* [next](#)  
*Apontador para o nodo seguinte.*

## 4.7 Referência à Estrutura SPQueue

### 4.7.1 Descrição Detalhada

Estrutura da pqueue.

Definido na linha 51 do ficheiro pqueue.h.

#### Campos de Dados

- int(\* [comp](#))(void \*, void \*)  
*Função que compara dois elementos.*
- int [size](#)  
*Número de elementos da pqueue.*
- [PQueueNode head](#)  
*Apontador para o início da pqueue.*

## 4.8 Referência à Estrutura SPQueueNode

### 4.8.1 Descrição Detalhada

Estrutura do nodo da pqueue.

Definido na linha 35 do ficheiro pqueue.h.

#### Campos de Dados

- void \* [inf](#)  
*Apontador para a informação do nodo.*
- struct sPQueueNode \* [next](#)  
*Apontador para o nodo seguinte.*

## 4.9 Referência à Estrutura SQueue

### 4.9.1 Descrição Detalhada

Estrutura da queue.

Definido na linha 35 do ficheiro queue.h.

### Campos de Dados

- `int size`  
*Número de elementos da queue.*
- `QueueNode head`  
*Apontador para o início da queue.*
- `QueueNode last`  
*Apontador para o fim da queue.*

## 4.10 Referência à Estrutura SQueueNode

### 4.10.1 Descrição Detalhada

Estrutura do nodo da queue.

Definido na linha 19 do ficheiro queue.h.

### Campos de Dados

- `void * inf`  
*Apontador para a informação do nodo.*
- `struct sQueueNode * next`  
*Apontador para o nodo seguinte.*

## 4.11 Referência à Estrutura SStack

### 4.11.1 Descrição Detalhada

Estrutura de uma stack.

Definido na linha 35 do ficheiro stack.h.

### Campos de Dados

- `int size`  
*Número de elementos da stack.*
- `StackNode top`  
*Apontador para o topo da stack.*

## 4.12 Referência à Estrutura SStackNode

### 4.12.1 Descrição Detalhada

Estrutura do nodo de uma stack.

Definido na linha 19 do ficheiro stack.h.

#### Campos de Dados

- void \* [inf](#)  
*Apontador para a informação do nodo.*
- struct sStackNode \* [next](#)  
*Apontador para o nodo seguinte.*

## 4.13 Referência à Estrutura STreeMap

### 4.13.1 Descrição Detalhada

Estrutura de uma árvore.

Definido na linha 67 do ficheiro treemap.h.

#### Campos de Dados

- int(\* [keyComp](#))(void \*, void \*)  
*Função que compara duas chaves.*
- int [size](#)  
*Número de elementos da árvore.*
- [TreeNode](#) [root](#)  
*Apontador para a raiz da árvore.*

## 4.14 Referência à Estrutura STreeNode

### 4.14.1 Descrição Detalhada

Estrutura do nodo da árvore.

Definido na linha 43 do ficheiro treemap.h.

#### Campos de Dados

- void \* [key](#)  
*Apontador para a chave do nodo.*

- void \* [value](#)  
*Apontador para o valor associado à chave.*
- [BFactor bf](#)  
*Factor de balanceamento.*
- struct sTreeNode \* [super](#)  
*Apontador para o nodo superior.*
- struct sTreeNode \* [left](#)  
*Apontador para a subárvore esquerda.*
- struct sTreeNode \* [right](#)  
*Apontador para a subárvore direita.*

## 5 Documentação dos Ficheiros

### 5.1 Referência ao Ficheiro [array.c](#)

#### 5.1.1 Descrição Detalhada

Implementação de um array dinâmico.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro [array.c](#).

**Funções**

- [Array newArray](#) (int size)  
*Cria um array.*
- void [arrayDelete](#) ([Array](#) array)  
*Elimina um array.*
- int [arrayInsert](#) ([Array](#) array, int index, void \*inf, int replace)  
*Insere um elemento numa determinada posição de um array.*
- int [arrayRemove](#) ([Array](#) array, int index, void \*\*inf)

*Remove um elemento numa determinada posição de um array.*

- `int arrayAt (Array array, int index, void **inf)`  
*Verifica qual o elemento numa determinada posição de um array.*
- `int arrayResize (Array array, int size)`  
*Altera a capacidade do array.*
- `int arraySize (Array array)`  
*Determina o número de elementos de um array.*
- `int arrayCapacity (Array array)`  
*Determina a capacidade de um array.*
- `int arrayMap (Array array, void(*fun)(void *))`  
*Aplica uma função aos elementos de um array.*
- `Iterator arrayIterator (Array array)`  
*Cria um iterador a partir de um array.*

## 5.1.2 Documentação das Funções

### 5.1.2.1 `int arrayAt (Array array, int index, void ** inf)`

Verifica qual o elemento numa determinada posição de um array.

Se não existir nenhum elemento na posição indicado é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*array* array.

*index* posição do elemento que procuramos.

*inf* endereço onde será colocado o resultado.

#### Retorna:

0 se existir elemento na posição pretendida;  
1 se não existir elemento na posição indicada.

Definido na linha 95 do ficheiro array.c.

### 5.1.2.2 `int arrayCapacity (Array array)`

Determina a capacidade de um array.

Devolve o valor do campo *capacity* do array.

**Parâmetros:**

*array* array.

**Retorna:**

capacidade do array.

Definido na linha 149 do ficheiro `array.c`.

**5.1.2.3 void arrayDelete (Array array)**

Elimina um array.

**Atenção:**

apenas liberta a memória referente à estrutura do array; não liberta o espaço ocupado pelos elementos nele contidos.

**Parâmetros:**

*array* array.

Definido na linha 36 do ficheiro `array.c`.

**5.1.2.4 int arrayInsert (Array array, int index, void \* inf, int replace)**

Insere um elemento numa determinada posição de um array.

A posição, especificado pelo argumento *index*, tem que ser maior do que 0. Se o tamanho actual do array não permitir a inserção do novo elemento, o tamanho é aumentado e passa a ser igual ao valor de *index+1* (se não for possível aumentar o tamanho do array o elemento não é inserido). Caso a posição já esteja ocupada, a variável *replace* determina se o valor antigo é ou não substituído (caso seja 0 não há substituição, caso tenha outro valor o novo elemento é inserido).

**Atenção:**

se o elemento a inserir tiver o valor NULL, este não é inserido, não sendo, como tal, incrementado o número de elementos do array.

**Parâmetros:**

*array* array.

*index* posição em que será inserido.

*inf* endereço do elemento que queremos inserir.

*replace* variável que determina se elementos já existente são ou não substituídos.

**Retorna:**

- 0 se o elemento for inserido;
- 1 se a posição já estava ocupada;
- 2 se o valor de *index* não for válido;
- 3 se não for possível aumentar o tamanho do array.

Definido na linha 44 do ficheiro `array.c`.

### 5.1.2.5 Iterator arrayIterator (Array *array*)

Cria um iterador a partir de um array.

Se ocorrer algum erro a função devolve NULL.

#### Ver Também:

[Iterator](#)

#### Parâmetros:

*array* array.

#### Retorna:

iterador criado ou NULL.

Definido na linha 176 do ficheiro array.c.

### 5.1.2.6 int arrayMap (Array *array*, void(\*) (void \*) *fun*)

Aplica uma função aos elementos de um array.

A função *fun* tem que ser do tipo: void fun(void\*).

#### Parâmetros:

*array* array.

*fun* função a ser aplicada.

#### Retorna:

0 se o array não estiver vazio;

1 se o array estiver vazio.

Definido na linha 156 do ficheiro array.c.

### 5.1.2.7 int arrayRemove (Array *array*, int *index*, void \*\* *inf*)

Remove um elemento numa determinada posição de um array.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se o elemento pretendido não existir será colocado o valor NULL em *inf*. Considera-se que não existe elemento numa posição do array, se o seu valor for NULL.

#### Atenção:

esta função não liberta a memória ocupada pela elemento removido.

#### Parâmetros:

*array* array.

*index* posição do elemento que queremos remover.

*inf* endereço onde é colocado o valor removido (ou NULL).

**Retorna:**

- 0 se o elemento for removido;
- 1 se não existir elemento na posição indicada.

Definido na linha 72 do ficheiro `array.c`.

**5.1.2.8 `int arrayResize (Array array, int size)`**

Altera a capacidade do array.

Apenas permite aumentar a capacidade do array.

**Parâmetros:**

- array* array.
- size* nova capacidade do array.

**Retorna:**

- 0 se a dimensão for alterada;
- 1 se ocorrer algum erro durante o redimensionamento do array;
- 2 se o valor da nova dimensão for inferior à dimensão actual.

Definido na linha 116 do ficheiro `array.c`.

**5.1.2.9 `int arraySize (Array array)`**

Determina o número de elementos de um array.

Devolve o valor do campo *size* do array.

**Parâmetros:**

- array* array.

**Retorna:**

- número de elementos do array.

Definido na linha 142 do ficheiro `array.c`.

**5.1.2.10 `Array newArray (int size)`**

Cria um array.

Um array trata-se de um vector de elementos associados a uma posição, posição essa a que podemos aceder em tempo linear. Na criação do array é indicado a dimensão inicial, dimensão essa que pode posteriormente ser alterada através da função `arrayResize`.

Se não for possível criar o array devolve `NULL`.

**Parâmetros:**

- size* dimensão pretendida.

**Retorna:**

- array inicializado ou `NULL`.

Definido na linha 12 do ficheiro `array.c`.

## 5.2 Referência ao Ficheiro array.h

### 5.2.1 Descrição Detalhada

Implementação de um array dinâmico.

Esta biblioteca disponibiliza funções que permitem alocar dinamicamente o array com o comprimento que desejarmos, tendo o comprimento do array disponível.

O array tem também a capacidade de se redimensionar sempre que necessário, mantendo toda a informação já existente.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

09/2009

Definido no ficheiro [array.h](#).

**Estruturas de Dados**

- struct [SArray](#)  
*Definição da estrutura do array.*

**Definições de Tipos**

- typedef [SArray](#) \* [Array](#)  
*Definição do array.*

**Funções**

- [Array newArray](#) (int size)  
*Cria um array.*
- void [arrayDelete](#) ([Array](#) array)  
*Elimina um array.*
- int [arrayInsert](#) ([Array](#) array, int index, void \*inf, int replace)  
*Insere um elemento numa determinada posição de um array.*
- int [arrayRemove](#) ([Array](#) array, int index, void \*\*inf)  
*Remove um elemento numa determinada posição de um array.*
- int [arrayAt](#) ([Array](#) array, int index, void \*\*inf)

Verifica qual o elemento numa determinada posição de um array.

- `int arrayResize (Array array, int size)`  
*Altera a capacidade do array.*
- `int arraySize (Array array)`  
*Determina o número de elementos de um array.*
- `int arrayCapacity (Array array)`  
*Determina a capacidade de um array.*
- `int arrayMap (Array array, void(*fun)(void *))`  
*Aplica uma função aos elementos de um array.*
- `Iterator arrayIterator (Array array)`  
*Cria um iterador a partir de um array.*

## 5.2.2 Documentação das Funções

### 5.2.2.1 `int arrayAt (Array array, int index, void ** inf)`

Verifica qual o elemento numa determinada posição de um array.

Se não existir nenhum elemento na posição indicado é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*array* array.

*index* posição do elemento que procuramos.

*inf* endereço onde será colocado o resultado.

#### Retorna:

0 se existir elemento na posição pretendida;  
1 se não existir elemento na posição indicada.

Definido na linha 95 do ficheiro array.c.

### 5.2.2.2 `int arrayCapacity (Array array)`

Determina a capacidade de um array.

Devolve o valor do campo *capacity* do array.

#### Parâmetros:

*array* array.

**Retorna:**

capacidade do array.

Definido na linha 149 do ficheiro array.c.

**5.2.2.3 void arrayDelete (Array array)**

Elimina um array.

**Atenção:**

apenas liberta a memória referente à estrutura do array; não liberta o espaço ocupado pelos elementos nele contidos.

**Parâmetros:**

*array* array.

Definido na linha 36 do ficheiro array.c.

**5.2.2.4 int arrayInsert (Array array, int index, void \* inf, int replace)**

Inserir um elemento numa determinada posição de um array.

A posição, especificado pelo argumento *index*, tem que ser maior do que 0. Se o tamanho actual do array não permitir a inserção do novo elemento, o tamanho é aumentado e passa a ser igual ao valor de *index+1* (se não for possível aumentar o tamanho do array o elemento não é inserido). Caso a posição já esteja ocupada, a variável *replace* determina se o valor antigo é ou não substituído (caso seja 0 não há substituição, caso tenha outro valor o novo elemento é inserido).

**Atenção:**

se o elemento a inserir tiver o valor NULL, este não é inserido, não sendo, como tal, incrementado o número de elementos do array.

**Parâmetros:**

*array* array.

*index* posição em que será inserido.

*inf* endereço do elemento que queremos inserir.

*replace* variável que determina se elementos já existentes são ou não substituídos.

**Retorna:**

- 0 se o elemento for inserido;
- 1 se a posição já estava ocupada;
- 2 se o valor de *index* não for válido;
- 3 se não for possível aumentar o tamanho do array.

Definido na linha 44 do ficheiro array.c.

### 5.2.2.5 Iterator arrayIterator (Array array)

Cria um iterador a partir de um array.

Se ocorrer algum erro a função devolve NULL.

#### Ver Também:

[Iterator](#)

#### Parâmetros:

*array* array.

#### Retorna:

iterador criado ou NULL.

Definido na linha 176 do ficheiro array.c.

### 5.2.2.6 int arrayMap (Array array, void(\*)(void \*) fun)

Aplica uma função aos elementos de um array.

A função *fun* tem que ser do tipo: void fun(void\*).

#### Parâmetros:

*array* array.

*fun* função a ser aplicada.

#### Retorna:

0 se o array não estiver vazio;

1 se o array estiver vazio.

Definido na linha 156 do ficheiro array.c.

### 5.2.2.7 int arrayRemove (Array array, int index, void \*\* inf)

Remove um elemento numa determinada posição de um array.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se o elemento pretendido não existir será colocado o valor NULL em *inf*. Considera-se que não existe elemento numa posição do array, se o seu valor for NULL.

#### Atenção:

esta função não liberta a memória ocupada pela elemento removido.

#### Parâmetros:

*array* array.

*index* posição do elemento que queremos remover.

*inf* endereço onde é colocado o valor removido (ou NULL).

**Retorna:**

- 0 se o elemento for removido;
- 1 se não existir elemento na posição indicada.

Definido na linha 72 do ficheiro `array.c`.

**5.2.2.8 `int arrayResize (Array array, int size)`**

Altera a capacidade do array.

Apenas permite aumentar a capacidade do array.

**Parâmetros:**

- array* array.
- size* nova capacidade do array.

**Retorna:**

- 0 se a dimensão for alterada;
- 1 se ocorrer algum erro durante o redimensionamento do array;
- 2 se o valor da nova dimensão for inferior à dimensão actual.

Definido na linha 116 do ficheiro `array.c`.

**5.2.2.9 `int arraySize (Array array)`**

Determina o número de elementos de um array.

Devolve o valor do campo *size* do array.

**Parâmetros:**

- array* array.

**Retorna:**

- número de elementos do array.

Definido na linha 142 do ficheiro `array.c`.

**5.2.2.10 `Array newArray (int size)`**

Cria um array.

Um array trata-se de um vector de elementos associados a uma posição, posição essa a que podemos aceder em tempo linear. Na criação do array é indicado a dimensão inicial, dimensão essa que pode posteriormente ser alterada através da função `arrayResize`.

Se não for possível criar o array devolve `NULL`.

**Parâmetros:**

- size* dimensão pretendida.

**Retorna:**

- array inicializado ou `NULL`.

Definido na linha 12 do ficheiro `array.c`.

## 5.3 Referência ao Ficheiro hashmap.c

### 5.3.1 Descrição Detalhada

Implementação de uma tabela de hash.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro [hashmap.c](#).

#### Funções

- static int [reHash](#) ([HashMap](#) hmap)  
*Redimensiona uma tabela de hash.*
- [HashMap](#) [newHash](#) (int size, float factor, int(\*hash)(void \*), int(\*equals)(void \*, void \*))  
*Cria uma tabela de hash.*
- int [hashSetHash](#) ([HashMap](#) hmap, int(\*hash)(void \*))  
*Altera a função de hash associada a uma tabela.*
- int [hashSetEquals](#) ([HashMap](#) hmap, int(\*equals)(void \*, void \*))  
*Altera a função que compara duas chaves de uma tabela.*
- int [hashSetFactor](#) ([HashMap](#) hmap, int factor)  
*Altera o factor de "reestruturação" da tabela.*
- void [hashDelete](#) ([HashMap](#) hmap)  
*Elimina uma tabela de hash.*
- int [hashInsert](#) ([HashMap](#) hmap, void \*key, void \*value, int replace)  
*Insere um par chave/valor numa tabela de hash.*
- int [hashRemove](#) ([HashMap](#) hmap, void \*key, void \*\*value, void(\*del)(void \*))  
*Remove um elemento de uma tabela de hash.*
- int [hashGet](#) ([HashMap](#) hmap, void \*key, void \*\*value)  
*Procura um elemento numa tabela de hash.*
- int [hashSize](#) ([HashMap](#) hmap)  
*Determina o número de elementos de uma tabela de hash.*

- **Iterator** `hashKeys` (`HashMap hmap`)

Cria um iterador a partir das chaves de uma tabela de hash.

- **Iterator** `hashValues` (`HashMap hmap`)

Cria um iterador a partir dos valores associados às chaves de uma tabela de hash.

### 5.3.2 Documentação das Funções

#### 5.3.2.1 `void hashDelete (HashMap hmap)`

Elimina uma tabela de hash.

##### Atenção:

apenas liberta a memória referente à estrutura da tabela; não liberta o espaço ocupado pelos elementos nela contidos.

##### Parâmetros:

*hmap* tabela de hash.

Definido na linha 122 do ficheiro hashmap.c.

#### 5.3.2.2 `int hashGet (HashMap hmap, void *key, void **value)`

Procura um elemento numa tabela de hash.

Devolve o valor associado a uma chave se ela existir. Se a chave não existir é colocado o valor NULL em *value*.

##### Atenção:

esta função coloca em *value* o endereço do valor pretendido; depois de executar esta função é aconselhável fazer uma cópia do mesmo e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

##### Parâmetros:

*hmap* tabela de hash.

*key* chave que procuramos.

*value* endereço onde é colocado o resultado.

##### Retorna:

0 se o elemento existir;

1 se o elemento não existir.

Definido na linha 210 do ficheiro hashmap.c.

#### 5.3.2.3 `int hashInsert (HashMap hmap, void *key, void *value, int replace)`

Insere um par chave/valor numa tabela de hash.

Caso a chave já exista, a variável *replace* determina se o valor antigo é ou não substituído (caso seja 0 não há substituição, caso tenha outro valor o novo elemento é inserido).

**Parâmetros:**

*hmap* tabela de hash.

*key* chave.

*value* valor associado à chave.

*replace* variável que determina se elementos já existente são ou não substituídos.

**Retorna:**

0 se o elemento for inserido;

1 se a tabela já possuía a chave indicada;

2 se não for possível alocar memória para o novo elemento.

Definido na linha 141 do ficheiro `hashmap.c`.

**5.3.2.4 Iterator `hashKeys` (`HashMap hmap`)**

Cria um iterador a partir das chaves de uma tabela de hash.

Coloca as referências para as chaves num iterador. Se ocorrer algum erro devolve `NULL`.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

iterador criado ou `NULL`.

Definido na linha 239 do ficheiro `hashmap.c`.

**5.3.2.5 `int hashRemove` (`HashMap hmap`, `void *key`, `void **value`, `void(*) (void *) del`)**

Remove um elemento de uma tabela de hash.

Permite devolver o valor removido, caso o valor de *value* seja diferente de `NULL`. Se a chave não existir ou o elemento não for removido é colocado o valor `NULL` em *value*.

**Atenção:**

esta função não liberta o espaço ocupado pelo valor associado à chave; já o espaço ocupado pela chave removida, se *del* for diferente de `NULL`, será libertado.

**Parâmetros:**

*hmap* tabela de hash.

*key* chave que queremos remover.

*value* endereço onde é colocado o elemento removido (ou `NULL`).

*del* função que elimina uma chave (ou `NULL`).

**Retorna:**

0 se o elemento for removido;

1 se a chave não existir.

Definido na linha 180 do ficheiro `hashmap.c`.

**5.3.2.6 int hashSetEquals (HashMap *hmap*, int(\*) (void \*, void \*) *equals*)**

Altera a função que compara duas chaves de uma tabela.

O valor de *equals* não pode ser NULL.

**Parâmetros:**

*hmap* tabela de hash.

*equals* nova função.

**Retorna:**

1 se *equals* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 98 do ficheiro hashmap.c.

**5.3.2.7 int hashSetFactor (HashMap *hmap*, int *factor*)**

Altera o factor de "reestruturação" da tabela.

O novo valor tem que ser maior do 0.1.

**Parâmetros:**

*hmap* tabela de hash.

*factor* novo valor.

**Retorna:**

1 se *factor* for menor do que 0.1;  
0 caso contrário.

Definido na linha 110 do ficheiro hashmap.c.

**5.3.2.8 int hashSetHash (HashMap *hmap*, int(\*) (void \*) *hash*)**

Altera a função de hash associada a uma tabela.

O valor de *hash* não pode ser NULL.

**Parâmetros:**

*hmap* tabela de hash.

*hash* nova função.

**Retorna:**

1 se *hash* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 86 do ficheiro hashmap.c.

### 5.3.2.9 `int hashSize (HashMap hmap)`

Determina o número de elementos de uma tabela de hash.

Devolve o valor do campo *size* da tabela.

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

número de elementos da tabela.

Definido na linha 232 do ficheiro `hashmap.c`.

### 5.3.2.10 `Iterator hashValues (HashMap hmap)`

Cria um iterador a partir dos valores associados às chaves de uma tabela de hash.

Coloca as referências para os "valores" num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

iterador criado ou NULL.

Definido na linha 262 do ficheiro `hashmap.c`.

### 5.3.2.11 `HashMap newHash (int size, float factor, int(*)(void *) hash, int(*)(void *, void *) equals)`

Cria uma tabela de hash.

Se não for possível criar a tabela devolve NULL. Têm que ser especificadas a função de hash e a função que compara chaves, caso contrário a tabela não será criada. Estas funções podem ser alteradas a qualquer momento, utilizando as funções `hashSetHash` e `hashSetEquals`.

É ainda necessário indicar um valor (*factor*) que determina quando a dimensão da tabela deve ser aumentada; isso acontecerá quando:

$size > factor * length$

Esta variável terá que ser superior a 0.1.

**Parâmetros:**

*size* dimensão inicial da tabela.

*factor* factor de "reestruturação" da tabela.

*hash* função de hash.

*equals* função que compara duas chaves.

**Retorna:**

tabela inicializada ou `NULL`.

Definido na linha 58 do ficheiro `hashmap.c`.

**5.3.2.12 static int reHash (HashMap *hmap*) [static]**

Redimensiona uma tabela de hash.

Duplica a dimensão do array que suporta a tabela e recoloca os elementos contidos na tabela antiga na nova tabela.

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

1 se a alocação de espaço para a nova tabela falhar;  
0 caso contrário.

Definido na linha 23 do ficheiro `hashmap.c`.

## 5.4 Referência ao Ficheiro `hashmap.h`

### 5.4.1 Descrição Detalhada

Implementação de uma tabela de hash.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma tabela de hash.

Sempre que existam colisões os elementos são inseridos numa lista ligada. Quando o número de elementos atingir uma determinada percentagem do número de posições da tabela, a sua dimensão é aumentada.

Na criação de uma tabela de hash é necessário especificar algumas funções que manipulam os tipos de dados utilizados. De seguida descrevem-se essas funções e apresentam-se exemplos (para o tipo `char*`):

- `int hash(void *)`

Função de hash (usada pelas funções `hashInsert`, `hashRemove` e `hashGet`). Associa a uma chave um número único.

```
int hash(void* key)
{
    int i,x;
    char* aux=key;
    for(i=0,x=0;i<32&&aux[i]!='\0';x+=aux[i],i++);

    return x;
}
```

- `int keyEquals(void* key1,void* key2)`

Função que compara duas chaves (usada pelas funções `hashInsert`, `hashRemove` e `hashGet`); deve devolver 0 se `key1` igual a `key2` e um valor diferente de 0 caso contrário; pode ser alterada através da função `hashSetEquals`.

```
int keyEquals(void* key1, void* key2)
{
    if(key1&&key2) return strcmp((char*)key1, (char*)key2);
    else return 1;
}
```

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro [hashmap.h](#).

**Estruturas de Dados**

- struct [SHashNode](#)  
*Estrutura do nodo de uma tabela de hash.*
- struct [SHashMap](#)  
*Estrutura de uma tabela de hash.*

**Definições de Tipos**

- typedef [SHashNode](#) \* [HashNode](#)  
*Definição do apontador para os nodos da tabela de hash.*
- typedef [SHashMap](#) \* [HashMap](#)  
*Definição da tabela de hash.*

**Funções**

- [HashMap newHash](#) (int size, float factor, int(\*hash)(void \*), int(\*equals)(void \*, void \*))  
*Cria uma tabela de hash.*
- int [hashSetHash](#) ([HashMap](#) hmap, int(\*hash)(void \*))  
*Altera a função de hash associada a uma tabela.*
- int [hashSetEquals](#) ([HashMap](#) hmap, int(\*equals)(void \*, void \*))  
*Altera a função que compara duas chaves de uma tabela.*
- int [hashSetFactor](#) ([HashMap](#) hmap, int factor)  
*Altera o factor de "reestruturação" da tabela.*

- void `hashDelete` (`HashMap` `hmap`)  
*Elimina uma tabela de hash.*
- int `hashInsert` (`HashMap` `hmap`, void \*`key`, void \*`value`, int `replace`)  
*Insere um par chave/valor numa tabela de hash.*
- int `hashRemove` (`HashMap` `hmap`, void \*`key`, void \*\*`value`, void(\*`del`)(void \*))  
*Remove um elemento de uma tabela de hash.*
- int `hashGet` (`HashMap` `hmap`, void \*`key`, void \*\*`value`)  
*Procura um elemento numa tabela de hash.*
- int `hashSize` (`HashMap` `hmap`)  
*Determina o número de elementos de uma tabela de hash.*
- Iterator `hashKeys` (`HashMap` `hmap`)  
*Cria um iterador a partir das chaves de uma tabela de hash.*
- Iterator `hashValues` (`HashMap` `hmap`)  
*Cria um iterador a partir dos valores associados às chaves de uma tabela de hash.*

## 5.4.2 Documentação das Funções

### 5.4.2.1 void hashDelete (HashMap *hmap*)

Elimina uma tabela de hash.

#### Atenção:

apenas liberta a memória referente à estrutura da tabela; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*hmap* tabela de hash.

Definido na linha 122 do ficheiro hashmap.c.

### 5.4.2.2 int hashGet (HashMap *hmap*, void \* *key*, void \*\* *value*)

Procura um elemento numa tabela de hash.

Devolve o valor associado a uma chave se ela existir. Se a chave não existir é colocado o valor NULL em *value*.

#### Atenção:

esta função coloca em *value* o endereço do valor pretendido; depois de executar esta função é aconselhável fazer uma cópia do mesmo e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*hmap* tabela de hash.

*key* chave que procuramos.

*value* endereço onde é colocado o resultado.

**Retorna:**

0 se o elemento existir;

1 se o elemento não existir.

Definido na linha 210 do ficheiro hashmap.c.

**5.4.2.3 int hashInsert (HashMap *hmap*, void \* *key*, void \* *value*, int *replace*)**

Inserir um par chave/valor numa tabela de hash.

Caso a chave já exista, a variável *replace* determina se o valor antigo é ou não substituído (caso seja 0 não há substituição, caso tenha outro valor o novo elemento é inserido).

**Parâmetros:**

*hmap* tabela de hash.

*key* chave.

*value* valor associado à chave.

*replace* variável que determina se elementos já existentes são ou não substituídos.

**Retorna:**

0 se o elemento for inserido;

1 se a tabela já possuía a chave indicada;

2 se não for possível alocar memória para o novo elemento.

Definido na linha 141 do ficheiro hashmap.c.

**5.4.2.4 Iterator hashKeys (HashMap *hmap*)**

Cria um iterador a partir das chaves de uma tabela de hash.

Coloca as referências para as chaves num iterador. Se ocorrer algum erro devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

iterador criado ou NULL.

Definido na linha 239 do ficheiro hashmap.c.

**5.4.2.5 `int hashRemove (HashMap hmap, void * key, void ** value, void(*) (void *) del)`**

Remove um elemento de uma tabela de hash.

Permite devolver o valor removido, caso o valor de *value* seja diferente de NULL. Se a chave não existir ou o elemento não for removido é colocado o valor NULL em *value*.

**Atenção:**

esta função não liberta o espaço ocupado pelo valor associado à chave; já o espaço ocupado pela chave removida, se *del* for diferente de NULL, será libertado.

**Parâmetros:**

*hmap* tabela de hash.

*key* chave que queremos remover.

*value* endereço onde é colocado o elemento removido (ou NULL).

*del* função que elimina uma chave (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a chave não existir.

Definido na linha 180 do ficheiro `hashmap.c`.

**5.4.2.6 `int hashSetEquals (HashMap hmap, int(*) (void *, void *) equals)`**

Altera a função que compara duas chaves de uma tabela.

O valor de *equals* não pode ser NULL.

**Parâmetros:**

*hmap* tabela de hash.

*equals* nova função.

**Retorna:**

1 se *equals* for NULL (não é efectuada qualquer alteração);

0 caso contrário.

Definido na linha 98 do ficheiro `hashmap.c`.

**5.4.2.7 `int hashSetFactor (HashMap hmap, int factor)`**

Altera o factor de "reestruturação" da tabela.

O novo valor tem que ser maior do 0.1.

**Parâmetros:**

*hmap* tabela de hash.

*factor* novo valor.

**Retorna:**

1 se *factor* for menor do que 0.1;  
0 caso contrário.

Definido na linha 110 do ficheiro `hashmap.c`.

**5.4.2.8 `int hashSetHash (HashMap hmap, int(*) (void *) hash)`**

Altera a função de hash associada a uma tabela.

O valor de *hash* não pode ser NULL.

**Parâmetros:**

*hmap* tabela de hash.

*hash* nova função.

**Retorna:**

1 se *hash* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 86 do ficheiro `hashmap.c`.

**5.4.2.9 `int hashSize (HashMap hmap)`**

Determina o número de elementos de uma tabela de hash.

Devolve o valor do campo *size* da tabela.

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

número de elementos da tabela.

Definido na linha 232 do ficheiro `hashmap.c`.

**5.4.2.10 `Iterator hashValues (HashMap hmap)`**

Cria um iterador a partir dos valores associados às chaves de uma tabela de hash.

Coloca as referências para os "valores" num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*hmap* tabela de hash.

**Retorna:**

iterador criado ou NULL.

Definido na linha 262 do ficheiro `hashmap.c`.

#### 5.4.2.11 `HashMap newHash (int size, float factor, int(*) (void *) hash, int(*) (void *, void *) equals)`

Cria uma tabela de hash.

Se não for possível criar a tabela devolve NULL. Têm que ser especificadas a função de hash e a função que compara chaves, caso contrário a tabela não será criada. Estas funções podem ser alteradas a qualquer momento, utilizando as funções `hashSetHash` e `hashSetEquals`.

É ainda necessário indicar um valor (*factor*) que determina quando a dimensão da tabela deve ser aumentada; isso acontecerá quando:

`size > factor * length`

Esta variável terá que ser superior a 0.1.

##### Parâmetros:

*size* dimensão inicial da tabela.

*factor* factor de "reestruturação" da tabela.

*hash* função de hash.

*equals* função que compara duas chaves.

##### Retorna:

tabela inicializada ou NULL.

Definido na linha 58 do ficheiro `hashmap.c`.

## 5.5 Referência ao Ficheiro `iterator.c`

### 5.5.1 Descrição Detalhada

Implementação de um iterador.

##### Autor:

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

##### Versão:

2.0.2

##### Data:

02/2009

Definido no ficheiro `iterator.c`.

##### Funções

- `Iterator newIt (int size)`  
*Cria um iterador.*
- `void itDelete (Iterator it)`

*Elimina um iterador.*

- int `itAdd` (Iterator it, void \*val)

*Adiciona um elemento a um iterador.*

- int `itNext` (Iterator it, void \*\*val)

*Devolve o próximo elemento de um iterador.*

- int `itHasNext` (Iterator it)

*Verifica se existe "próximo" elemento num iterador.*

- int `itPrev` (Iterator it, void \*\*val)

*Devolve o anterior elemento de um iterador.*

- int `itHasPrev` (Iterator it)

*Verifica se existe elemento "anterior" num iterador.*

- int `itAt` (Iterator it, int n, void \*\*val)

*Verifica qual o elemento numa determinada posição do array de valores de um iterador.*

- int `itSetPos` (Iterator it, int n)

*Altera a posição actual do iterador.*

- int `itGetPos` (Iterator it)

*Determina a posição actual do iterador.*

## 5.5.2 Documentação das Funções

### 5.5.2.1 int itAdd (Iterator it, void \* val)

Adiciona um elemento a um iterador.

Verifica se a capacidade do iterador já foi atingida, em caso afirmativo o elemento não é adicionado e é devolvido o valor 1.

#### Parâmetros:

*it* iterador.

*val* valor a inserir.

#### Retorna:

0 se o valor for adicionado;

1 se o iterador não tiver espaço livre.

Definido na linha 44 do ficheiro iterator.c.

### 5.5.2.2 int itAt (Iterator it, int index, void \*\* val)

Verifica qual o elemento numa determinada posição do array de valores de um iterador.

Se o valor de *index* não for válido é colocado o valor NULL em *val*.

**Parâmetros:**

- it* iterador.
- index* posição pretendida.
- val* local onde será colocado o resultado.

**Retorna:**

- 0 se o elemento existir;
- 1 se o valor de *index* não for válido.

Definido na linha 107 do ficheiro `iterator.c`.

**5.5.2.3 void itDelete (Iterator it)**

Elimina um iterador.

Esta função limita-se a libertar o espaço ocupado pelo array de apontadores e pela estrutura do iterador.

**Parâmetros:**

- it* iterador.

Definido na linha 36 do ficheiro `iterator.c`.

**5.5.2.4 int itGetPos (Iterator it)**

Determina a posição actual do iterador.

Devolve o valor do campo *pos* do iterador.

**Parâmetros:**

- it* iterador.

**Retorna:**

- posição do iterador.

Definido na linha 138 do ficheiro `iterator.c`.

**5.5.2.5 int itHasNext (Iterator it)**

Verifica se existe "próximo" elemento num iterador.

**Parâmetros:**

- it* iterador.

**Retorna:**

- 1 se existir;
- 0 se não existir.

Definido na linha 74 do ficheiro `iterator.c`.

### 5.5.2.6 int itHasPrev (Iterator *it*)

Verifica se existe elemento "anterior" num iterador.

**Parâmetros:**

*it* iterador.

**Retorna:**

1 se existir;  
0 se não existir.

Definido na linha 99 do ficheiro iterator.c.

### 5.5.2.7 int itNext (Iterator *it*, void \*\* *val*)

Devolve o próximo elemento de um iterador.

Devolve o elemento que está imediatamente a seguir a *pos* e incrementa o seu valor. Caso ocorra algum erro é colocado o valor NULL em *val*.

**Parâmetros:**

*it* iterador.

*val* endereço onde será colocado o resultado

**Retorna:**

0 se for devolvido um elemento;  
1 se não existir próximo elemento.

Definido na linha 57 do ficheiro iterator.c.

### 5.5.2.8 int itPrev (Iterator *it*, void \*\* *val*)

Devolve o anterior elemento de um iterador.

Devolve o elemento que está imediatamente antes de *pos* e decrementa o seu valor. Caso ocorra algum erro é colocado o valor NULL em *val*.

**Parâmetros:**

*it* iterador.

*val* endereço onde será colocado o resultado.

**Retorna:**

0 se for devolvido um elemento;  
1 se não existir próximo elemento.

Definido na linha 82 do ficheiro iterator.c.

### 5.5.2.9 `int itSetPos (Iterator it, int n)`

Altera a posição actual do iterador.

É alterado o valor do campo *pos* do iterador (se o valor de *index* for válido).

**Parâmetros:**

*it* iterador.

*n* novo valor para a posição.

**Retorna:**

-1 se o valor de *n* não for válido;

0 caso contrário.

Definido na linha 123 do ficheiro `iterator.c`.

### 5.5.2.10 `Iterator newIt (int size)`

Cria um iterador.

Se não for possível criar o iterador devolve NULL.

**Parâmetros:**

*size* capacidade do iterador.

**Retorna:**

iterador inicializado ou NULL.

Definido na linha 13 do ficheiro `iterator.c`.

## 5.6 Referência ao Ficheiro `iterator.h`

### 5.6.1 Descrição Detalhada

Implementação de um iterador.

O iterador será um array de apontadores. O campo *pos* de um iterador indica uma posição entre o elemento devolvido pela função `itNext` e a função `itPrev`.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro [iterator.h](#).

### Estruturas de Dados

- struct `SIterator`  
*Estrutura do iterador.*

### Definições de Tipos

- typedef `SIterator * Iterator`  
*Definição do iterador.*

### Funções

- `Iterator newIt` (int size)  
*Cria um iterador.*
- void `itDelete` (`Iterator` it)  
*Elimina um iterador.*
- int `itAdd` (`Iterator` it, void \*val)  
*Adiciona um elemento a um iterador.*
- int `itNext` (`Iterator` it, void \*\*val)  
*Devolve o próximo elemento de um iterador.*
- int `itHasNext` (`Iterator` it)  
*Verifica se existe "próximo" elemento num iterador.*
- int `itPrev` (`Iterator` it, void \*\*val)  
*Devolve o anterior elemento de um iterador.*
- int `itHasPrev` (`Iterator` it)  
*Verifica se existe elemento "anterior" num iterador.*
- int `itAt` (`Iterator` it, int index, void \*\*val)  
*Verifica qual o elemento numa determinada posição do array de valores de um iterador.*
- int `itSetPos` (`Iterator` it, int n)  
*Altera a posição actual do iterador.*
- int `itGetPos` (`Iterator` it)  
*Determina a posição actual do iterador.*

## 5.6.2 Documentação das Funções

### 5.6.2.1 int itAdd (Iterator *it*, void \* *val*)

Adiciona um elemento a um iterador.

Verifica se a capacidade do iterador já foi atingida, em caso afirmativo o elemento não é adicionado e é devolvido o valor 1.

#### Parâmetros:

*it* iterador.

*val* valor a inserir.

#### Retorna:

0 se o valor for adicionado;

1 se o iterador não tiver espaço livre.

Definido na linha 44 do ficheiro iterator.c.

### 5.6.2.2 int itAt (Iterator *it*, int *index*, void \*\* *val*)

Verifica qual o elemento numa determinada posição do array de valores de um iterador.

Se o valor de *index* não for válido é colocado o valor NULL em *val*.

#### Parâmetros:

*it* iterador.

*index* posição pretendida.

*val* local onde será colocado o resultado.

#### Retorna:

0 se o elemento existir;

1 se o valor de *index* não for válido.

Definido na linha 107 do ficheiro iterator.c.

### 5.6.2.3 void itDelete (Iterator *it*)

Elimina um iterador.

Esta função limita-se a libertar o espaço ocupado pelo array de apontadores e pela estrutura do iterador.

#### Parâmetros:

*it* iterador.

Definido na linha 36 do ficheiro iterator.c.

### 5.6.2.4 int itGetPos (Iterator *it*)

Determina a posição actual do iterador.

Devolve o valor do campo *pos* do iterador.

**Parâmetros:**

*it* iterador.

**Retorna:**

posição do iterador.

Definido na linha 138 do ficheiro iterator.c.

**5.6.2.5 int itHasNext (Iterator *it*)**

Verifica se existe "próximo" elemento num iterador.

**Parâmetros:**

*it* iterador.

**Retorna:**

1 se existir;  
0 se não existir.

Definido na linha 74 do ficheiro iterator.c.

**5.6.2.6 int itHasPrev (Iterator *it*)**

Verifica se existe elemento "anterior" num iterador.

**Parâmetros:**

*it* iterador.

**Retorna:**

1 se existir;  
0 se não existir.

Definido na linha 99 do ficheiro iterator.c.

**5.6.2.7 int itNext (Iterator *it*, void \*\* *val*)**

Devolve o próximo elemento de um iterador.

Devolve o elemento que está imediatamente a seguir a *pos* e incrementa o seu valor. Caso ocorra algum erro é colocado o valor NULL em *val*.

**Parâmetros:**

*it* iterador.

*val* endereço onde será colocado o resultado

**Retorna:**

0 se for devolvido um elemento;  
1 se não existir próximo elemento.

Definido na linha 57 do ficheiro iterator.c.

**5.6.2.8 `int itPrev (Iterator it, void ** val)`**

Devolve o anterior elemento de um iterador.

Devolve o elemento que está imediatamente antes de *pos* e decrementa o seu valor. Caso ocorra algum erro é colocado o valor NULL em *val*.

**Parâmetros:**

*it* iterador.

*val* endereço onde será colocado o resultado.

**Retorna:**

0 se for devolvido um elemento;  
1 se não existir próximo elemento.

Definido na linha 82 do ficheiro `iterator.c`.

**5.6.2.9 `int itSetPos (Iterator it, int n)`**

Altera a posição actual do iterador.

É alterado o valor do campo *pos* do iterador (se o valor de *index* for válido).

**Parâmetros:**

*it* iterador.

*n* novo valor para a posição.

**Retorna:**

-1 se o valor de *n* não for válido;  
0 caso contrário.

Definido na linha 123 do ficheiro `iterator.c`.

**5.6.2.10 `Iterator newIt (int size)`**

Cria um iterador.

Se não for possível criar o iterador devolve NULL.

**Parâmetros:**

*size* capacidade do iterador.

**Retorna:**

iterador inicializado ou NULL.

Definido na linha 13 do ficheiro `iterator.c`.

## 5.7 Referência ao Ficheiro list.c

### 5.7.1 Descrição Detalhada

Implementação de uma lista (duplamente) ligada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [list.c](#).

### Funções

- [List newList](#) ()  
*Cria uma lista.*
- void [listDelete](#) ([List](#) list)  
*Elimina uma lista.*
- int [listInsertFst](#) ([List](#) list, void \*inf)  
*Insera um elemento no início de uma lista.*
- int [listInsertLst](#) ([List](#) list, void \*inf)  
*Insera um elemento no fim de uma lista.*
- int [listInsertAt](#) ([List](#) list, int index, void \*inf)  
*Insera um elemento numa determinada posição de uma lista.*
- int [listRemoveFst](#) ([List](#) list, void \*\*inf)  
*Remove o primeiro elemento de uma lista.*
- int [listRemoveLst](#) ([List](#) list, void \*\*inf)  
*Remove o último elemento de uma lista.*
- int [listRemoveAt](#) ([List](#) list, int index, void \*\*inf)  
*Remove o elemento de uma determinada posição de uma lista.*
- int [listFst](#) ([List](#) list, void \*\*inf)  
*Verifica qual o primeiro elemento de uma lista.*
- int [listLst](#) ([List](#) list, void \*\*inf)  
*Verifica qual o último elemento de uma lista.*

- `int listAt (List list, int index, void **inf)`  
*Verifica qual o elemento numa determinada posição de uma lista.*
- `int listSize (List list)`  
*Determina o tamanho de uma lista.*
- `int listMap (List list, void(*fun)(void *))`  
*Aplica uma função aos elementos de uma lista.*
- `Iterator listIterator (List list)`  
*Cria um iterador a partir de uma lista.*

## 5.7.2 Documentação das Funções

### 5.7.2.1 `int listAt (List list, int index, void ** inf)`

Verifica qual o elemento numa determinada posição de uma lista.

A posição, especificada pelo argumento *index*, tem que estar entre 0 e o tamanho da lista menos 1. Se isto não acontecer é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*list* lista.

*index* posição do elemento que procuramos.

*inf* endereço onde será colocado o resultado.

#### Retorna:

0 se o valor de *index* for válido;

1 se o valor de *index* não for válido.

Definido na linha 321 do ficheiro list.c.

### 5.7.2.2 `void listDelete (List list)`

Elimina uma lista.

#### Atenção:

apenas liberta a memória referente à estrutura da lista; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*list* lista.

Definido na linha 28 do ficheiro list.c.

**5.7.2.3 int listFst (List list, void \*\* inf)**

Verifica qual o primeiro elemento de uma lista.

Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a lista não estiver vazia;

1 se a lista estiver vazia.

Definido na linha 289 do ficheiro list.c.

**5.7.2.4 int listInsertAt (List list, int index, void \* inf)**

Inserir um elemento numa determinada posição de uma lista.

A posição, especificada pelo argumento *index*, tem que estar entre 0 e o tamanho da lista. O (n+1)-ésimo elemento e todos os seguintes avançam uma posição.

**Parâmetros:**

*list* lista.

*index* posição em que será inserido.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 se o valor de *index* não for válido;

2 se não for possível alocar memória para o novo elemento.

Definido na linha 133 do ficheiro list.c.

**5.7.2.5 int listInsertFst (List list, void \* inf)**

Inserir um elemento no início de uma lista.

Verifica se é possível inserir o elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*list* lista.

*inf* endereço do elementos que queremos inserir.

**Retorna:**

- 0 se o elemento for inserido;
- 1 se não for possível alocar memória para o novo elemento.

Definido na linha 45 do ficheiro list.c.

**5.7.2.6 int listInsertLst (List list, void \* inf)**

Inserir um elemento no fim de uma lista.

Verifica se é possível inserir o elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

- list* lista.
- inf* endereço do elemento que queremos inserir.

**Retorna:**

- 0 se o elemento for inserido;
- 1 se não for possível alocar memória para o novo elemento.

Definido na linha 89 do ficheiro list.c.

**5.7.2.7 Iterator listIterator (List list)**

Cria um iterador a partir de uma lista.

Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

- list* lista.

**Retorna:**

- iterador criado ou NULL.

Definido na linha 366 do ficheiro list.c.

**5.7.2.8 int listLst (List list, void \*\* inf)**

Verifica qual o último elemento de uma lista.

Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a lista não estiver vazia;

1 se a lista estiver vazia.

Definido na linha 305 do ficheiro list.c.

**5.7.2.9 int listMap (List list, void(\*)(void \*) fun)**

Aplica uma função aos elementos de uma lista.

A função *fun* tem que ser do tipo: `void fun(void*)`.

**Parâmetros:**

*list* lista.

*fun* função a ser aplicada.

**Retorna:**

0 se a lista não estiver vazia;

1 se a lista estiver vazia.

Definido na linha 350 do ficheiro list.c.

**5.7.2.10 int listRemoveAt (List list, int index, void \*\* inf)**

Remove o elemento de uma determinada posição de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

A posição, especificada pelo argumento *n*, tem que estar entre 0 e o tamanho da lista menos 1. O *n*-ésimo elemento e todos os seguintes recuam uma posição.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*index* posição do elemento que queremos remover.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se o valor de *n* não for válido.

Definido na linha 253 do ficheiro list.c.

**5.7.2.11 int listRemoveFst (List list, void \*\* inf)**

Remove o primeiro elemento de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;  
1 se a lista estiver vazia.

Definido na linha 185 do ficheiro list.c.

**5.7.2.12 int listRemoveLst (List list, void \*\* inf)**

Remove o último elemento de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;  
1 se a lista estiver vazia.

Definido na linha 219 do ficheiro list.c.

**5.7.2.13 int listSize (List list)**

Determina o tamanho de uma lista.

Devolve o valor do campo *size* da lista.

**Parâmetros:**

*list* lista.

**Retorna:**

número de elementos da lista.

Definido na linha 343 do ficheiro list.c.

#### 5.7.2.14 List newList ()

Cria uma lista.

Se não for possível criar a lista devolve NULL.

**Retorna:**

lista inicializada ou NULL.

Definido na linha 13 do ficheiro list.c.

## 5.8 Referência ao Ficheiro list.h

### 5.8.1 Descrição Detalhada

Implementação de uma lista (duplamente) ligada.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma lista (duplamente) ligada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [list.h](#).

### Estruturas de Dados

- struct [SListNode](#)  
*Estrutura do nodo da lista.*
- struct [SList](#)  
*Estrutura da lista.*

### Definições de Tipos

- typedef [SListNode](#) \* [ListNode](#)  
*Definição do apontador para os nodos da lista.*
- typedef [SList](#) \* [List](#)  
*Definição da lista.*

### Funções

- `List newList ()`  
*Cria uma lista.*
- `void listDelete (List list)`  
*Elimina uma lista.*
- `int listInsertFst (List list, void *inf)`  
*Insere um elemento no início de uma lista.*
- `int listInsertLst (List list, void *inf)`  
*Insere um elemento no fim de uma lista.*
- `int listInsertAt (List list, int index, void *inf)`  
*Insere um elemento numa determinada posição de uma lista.*
- `int listRemoveFst (List list, void **inf)`  
*Remove o primeiro elemento de uma lista.*
- `int listRemoveLst (List list, void **inf)`  
*Remove o último elemento de uma lista.*
- `int listRemoveAt (List list, int index, void **inf)`  
*Remove o elemento de uma determinada posição de uma lista.*
- `int listFst (List list, void **inf)`  
*Verifica qual o primeiro elemento de uma lista.*
- `int listLst (List list, void **inf)`  
*Verifica qual o último elemento de uma lista.*
- `int listAt (List list, int index, void **inf)`  
*Verifica qual o elemento numa determinada posição de uma lista.*
- `int listSize (List list)`  
*Determina o tamanho de uma lista.*
- `int listMap (List list, void(*fun)(void *))`  
*Aplica uma função aos elementos de uma lista.*
- `Iterator listIterator (List list)`  
*Cria um iterador a partir de uma lista.*

## 5.8.2 Documentação das Funções

### 5.8.2.1 int listAt (List list, int index, void \*\* inf)

Verifica qual o elemento numa determinada posição de uma lista.

A posição, especificada pelo argumento *index*, tem que estar entre 0 e o tamanho da lista menos 1. Se isto não acontecer é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*list* lista.

*index* posição do elemento que procuramos.

*inf* endereço onde será colocado o resultado.

#### Retorna:

0 se o valor de *index* for válido;

1 se o valor de *index* não for válido.

Definido na linha 321 do ficheiro list.c.

### 5.8.2.2 void listDelete (List list)

Elimina uma lista.

#### Atenção:

apenas liberta a memória referente à estrutura da lista; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*list* lista.

Definido na linha 28 do ficheiro list.c.

### 5.8.2.3 int listFst (List list, void \*\* inf)

Verifica qual o primeiro elemento de uma lista.

Se a lista estiver vazia é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*list* lista.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a lista não estiver vazia;  
1 se a lista estiver vazia.

Definido na linha 289 do ficheiro list.c.

**5.8.2.4 int listInsertAt (List list, int index, void \* inf)**

Inserir um elemento numa determinada posição de uma lista.

A posição, especificada pelo argumento *index*, tem que estar entre 0 e o tamanho da lista. O (n+1)-ésimo elemento e todos os seguintes avançam uma posição.

**Parâmetros:**

*list* lista.  
*index* posição em que será inserido.  
*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;  
1 se o valor de *index* não for válido;  
2 se não for possível alocar memória para o novo elemento.

Definido na linha 133 do ficheiro list.c.

**5.8.2.5 int listInsertFst (List list, void \* inf)**

Inserir um elemento no início de uma lista.

Verifica se é possível inserir o elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*list* lista.  
*inf* endereço do elementos que queremos inserir.

**Retorna:**

0 se o elemento for inserido;  
1 se não for possível alocar memória para o novo elemento.

Definido na linha 45 do ficheiro list.c.

**5.8.2.6 int listInsertLst (List list, void \* inf)**

Inserir um elemento no fim de uma lista.

Verifica se é possível inserir o elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*list* lista.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;  
1 se não for possível alocar memória para o novo elemento.

Definido na linha 89 do ficheiro list.c.

**5.8.2.7 Iterator listIterator (List list)**

Cria um iterador a partir de uma lista.

Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*list* lista.

**Retorna:**

iterador criado ou NULL.

Definido na linha 366 do ficheiro list.c.

**5.8.2.8 int listLst (List list, void \*\* inf)**

Verifica qual o último elemento de uma lista.

Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço do elemento pretendido; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a lista não estiver vazia;  
1 se a lista estiver vazia.

Definido na linha 305 do ficheiro list.c.

**5.8.2.9 int listMap (List list, void(\*)(void \*) fun)**

Aplica uma função aos elementos de uma lista.

A função *fun* tem que ser do tipo: `void fun(void*)`.

**Parâmetros:**

*list* lista.

*fun* função a ser aplicada.

**Retorna:**

0 se a lista não estiver vazia;

1 se a lista estiver vazia.

Definido na linha 350 do ficheiro list.c.

**5.8.2.10 int listRemoveAt (List list, int index, void \*\* inf)**

Remove o elemento de uma determinada posição de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

A posição, especificada pelo argumento *n*, tem que estar entre 0 e o tamanho da lista menos 1. O *n*-ésimo elemento e todos os seguintes recuam uma posição.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*index* posição do elemento que queremos remover.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se o valor de *n* não for válido.

Definido na linha 253 do ficheiro list.c.

**5.8.2.11 int listRemoveFst (List list, void \*\* inf)**

Remove o primeiro elemento de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a lista estiver vazia.

Definido na linha 185 do ficheiro list.c.

**5.8.2.12 int listRemoveLst (List list, void \*\* inf)**

Remove o último elemento de uma lista.

Permite devolver a informação do elemento removido, caso o valor de *inf* seja diferente de NULL. Se a lista estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*list* lista.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a lista estiver vazia.

Definido na linha 219 do ficheiro list.c.

**5.8.2.13 int listSize (List list)**

Determina o tamanho de uma lista.

Devolve o valor do campo *size* da lista.

**Parâmetros:**

*list* lista.

**Retorna:**

número de elementos da lista.

Definido na linha 343 do ficheiro list.c.

**5.8.2.14 List newList ()**

Cria uma lista.

Se não for possível criar a lista devolve NULL.

**Retorna:**

lista inicializada ou NULL.

Definido na linha 13 do ficheiro list.c.

## 5.9 Referência ao Ficheiro pqueue.c

### 5.9.1 Descrição Detalhada

Implementação de uma pqueue como lista ligada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

1.0.1

**Data:**

02/2009

Definido no ficheiro [pqueue.c](#).

**Funções**

- [PQueue newPQueue](#) (int(\*comp)(void \*, void \*))  
*Cria uma pqueue.*
- int [pqueueSetComp](#) (PQueue pqueue, int(\*comp)(void \*, void \*))  
*Altera a função que compara os elementos de uma árvore.*
- void [pqueueDelete](#) (PQueue pqueue)  
*Elimina uma pqueue.*
- int [pqueueInsert](#) (PQueue pqueue, void \*inf)  
*Insere um elemento numa pqueue.*
- int [pqueueRemove](#) (PQueue pqueue, void \*\*inf)  
*Remove um elemento de uma pqueue.*
- int [pqueueConsult](#) (PQueue pqueue, void \*\*inf)  
*Verifica qual o elemento na cabeça de uma pqueue.*
- int [pqueueSize](#) (PQueue pqueue)  
*Determina o tamanho de uma pqueue.*
- int [pqueueMap](#) (PQueue pqueue, void(\*fun)(void \*))  
*Aplica uma função aos elementos de uma pqueue começando pela cabeça.*
- [Iterator pqueueIterator](#) (PQueue pqueue)  
*Cria um iterador a partir da pqueue.*

## 5.9.2 Documentação das Funções

### 5.9.2.1 PQueue newPQueue (int(\*))(void \*, void \*) *comp*)

Cria uma pqueue.

Se não for possível criar a pqueue devolve NULL. Tem que ser especificada a função que compara os elementos da pqueue. Esta função pode ser alterada a qualquer momento, utilizando a função [pqueueSetComp](#).

#### Parâmetros:

*comp* função que compara dois elementos.

#### Retorna:

pqueue inicializada ou NULL.

Definido na linha 13 do ficheiro pqueue.c.

### 5.9.2.2 int pqueueConsult (PQueue *pqueue*, void \*\* *inf*)

Verifica qual o elemento na cabeça de uma pqueue.

Se a pqueue estiver vazia é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço da informação que está na cabeça da pqueue; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*pqueue* pqueue.

*inf* endereço onde é colocado o resultado.

#### Retorna:

0 se a pqueue não estiver vazia;

1 se a pqueue estiver vazia.

Definido na linha 134 do ficheiro pqueue.c.

### 5.9.2.3 void pqueueDelete (PQueue *pqueue*)

Elimina uma pqueue.

#### Atenção:

apenas liberta a memória ocupada pela estrutura da pqueue; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*pqueue* pqueue.

Definido na linha 41 do ficheiro pqueue.c.

**5.9.2.4 int pqueueInsert (PQueue pqueue, void \* inf)**

Insere um elemento numa pqueue.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*pqueue* pqueue.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 se não for possível alocar espaço para o novo elemento.

Definido na linha 61 do ficheiro pqueue.c.

**5.9.2.5 Iterator pqueueIterator (PQueue pqueue)**

Cria um iterador a partir da pqueue.

Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*pqueue* pqueue.

**Retorna:**

iterador criado ou NULL.

Definido na linha 173 do ficheiro pqueue.c.

**5.9.2.6 int pqueueMap (PQueue pqueue, void(\*)(void \*) fun)**

Aplica uma função aos elementos de uma pqueue começando pela cabeça.

A função *fun* tem que ser do tipo: `void fun (void*)`.

**Parâmetros:**

*pqueue* pqueue.

*fun* função a ser aplicada.

**Retorna:**

0 se a pqueue não estiver vazia;

1 se a pqueue estiver vazia.

Definido na linha 157 do ficheiro pqueue.c.

**5.9.2.7 int pqueueRemove (PQueue pqueue, void \*\* inf)**

Remove um elemento de uma pqueue.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a pqueue estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*pqueue* pqueue.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a pqueue estiver vazia.

Definido na linha 102 do ficheiro pqueue.c.

**5.9.2.8 int pqueueSetComp (PQueue pqueue, int(\*) (void \*, void \*) comp)**

Altera a função que compara os elementos de uma árvore.

O valor de *comp* não pode ser NULL.

**Parâmetros:**

*pqueue* pqueue.

*comp* nova função.

**Retorna:**

1 se *comp* for NULL (não é efectuada qualquer alteração);

0 caso contrário.

Definido na linha 29 do ficheiro pqueue.c.

**5.9.2.9 int pqueueSize (PQueue pqueue)**

Determina o tamanho de uma pqueue.

Devolve o valor do campo *size* da pqueue.

**Parâmetros:**

*pqueue* pqueue.

**Retorna:**

número de elementos da pqueue.

Definido na linha 150 do ficheiro pqueue.c.

## 5.10 Referência ao Ficheiro pqueue.h

### 5.10.1 Descrição Detalhada

Implementação de uma queue com prioridades.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma queue onde os elementos são inseridos ordenadamente.

Na criação de uma pqueue é necessário especificar a função que compara os elementos.

```
int comp(void* key1, void* key2)
```

(usada pela função [pqueueInsert](#)); esta função permite indicar qual o elemento a ser removido: será o menor valor contido na pqueue; pode ser alterada através da função [pqueueSetComp](#).

```
int comp(void* x1, void* x2)
{
    if(x1&&x2) return strcmp((char*)x1, (char*)x2);
    else return 0;
}
```

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

1.0.1

**Data:**

02/2009

Definido no ficheiro [pqueue.h](#).

**Estruturas de Dados**

- struct [SPQueueNode](#)  
*Estrutura do nodo da pqueue.*
- struct [SPQueue](#)  
*Estrutura da pqueue.*

**Definições de Tipos**

- typedef [SPQueueNode](#) \* [PQueueNode](#)  
*Definição do apontador para os nodos da pqueue.*
- typedef [SPQueue](#) \* [PQueue](#)  
*Definição da pqueue.*

## Funções

- `PQueue newPQueue (int(*comp)(void *, void *))`  
*Cria uma pqueue.*
- `int pqueueSetComp (PQueue pqueue, int(*comp)(void *, void *))`  
*Altera a função que compara os elementos de uma árvore.*
- `void pqueueDelete (PQueue pqueue)`  
*Elimina uma pqueue.*
- `int pqueueInsert (PQueue pqueue, void *inf)`  
*Insere um elemento numa pqueue.*
- `int pqueueRemove (PQueue pqueue, void **inf)`  
*Remove um elemento de uma pqueue.*
- `int pqueueConsult (PQueue pqueue, void **inf)`  
*Verifica qual o elemento na cabeça de uma pqueue.*
- `int pqueueSize (PQueue pqueue)`  
*Determina o tamanho de uma pqueue.*
- `int pqueueMap (PQueue pqueue, void(*fun)(void *))`  
*Aplica uma função aos elementos de uma pqueue começando pela cabeça.*
- `Iterator pqueueIterator (PQueue pqueue)`  
*Cria um iterador a partir da pqueue.*

### 5.10.2 Documentação das Funções

#### 5.10.2.1 `PQueue newPQueue (int(*) (void *, void *) comp)`

Cria uma pqueue.

Se não for possível criar a pqueue devolve NULL. Tem que ser especificada a função que compara os elementos da pqueue. Esta função pode ser alterada a qualquer momento, utilizando a função `pqueueSetComp`.

#### Parâmetros:

**comp** função que compara dois elementos.

#### Retorna:

pqueue inicializada ou NULL.

Definido na linha 13 do ficheiro pqueue.c.

**5.10.2.2 int pqueueConsult (PQueue pqueue, void \*\* inf)**

Verifica qual o elemento na cabeça de uma pqueue.

Se a pqueue estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço da informação que está na cabeça da pqueue; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*pqueue* pqueue.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a pqueue não estiver vazia;

1 se a pqueue estiver vazia.

Definido na linha 134 do ficheiro pqueue.c.

**5.10.2.3 void pqueueDelete (PQueue pqueue)**

Elimina uma pqueue.

**Atenção:**

apenas liberta a memória ocupada pela estrutura da pqueue; não liberta o espaço ocupado pelos elementos nela contidos.

**Parâmetros:**

*pqueue* pqueue.

Definido na linha 41 do ficheiro pqueue.c.

**5.10.2.4 int pqueueInsert (PQueue pqueue, void \* inf)**

Inserir um elemento numa pqueue.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*pqueue* pqueue.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 se não for possível alocar espaço para o novo elemento.

Definido na linha 61 do ficheiro pqueue.c.

#### 5.10.2.5 Iterator pqueueIterator (PQueue pqueue)

Cria um iterador a partir da pqueue.

Se ocorrer algum erro a função devolve NULL.

##### Ver Também:

[Iterator](#)

##### Parâmetros:

*pqueue* pqueue.

##### Retorna:

iterador criado ou NULL.

Definido na linha 173 do ficheiro pqueue.c.

#### 5.10.2.6 int pqueueMap (PQueue pqueue, void (\*)(void \*) fun)

Aplica uma função aos elementos de uma pqueue começando pela cabeça.

A função *fun* tem que ser do tipo: `void fun (void*)`.

##### Parâmetros:

*pqueue* pqueue.

*fun* função a ser aplicada.

##### Retorna:

0 se a pqueue não estiver vazia;

1 se a pqueue estiver vazia.

Definido na linha 157 do ficheiro pqueue.c.

#### 5.10.2.7 int pqueueRemove (PQueue pqueue, void \*\* inf)

Remove um elemento de uma pqueue.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a pqueue estiver vazia é colocado o valor NULL em *inf*.

##### Atenção:

esta função não liberta o espaço ocupado pelo elemento removido.

##### Parâmetros:

*pqueue* pqueue.

*inf* endereço onde é colocado o elemento removido (ou NULL).

##### Retorna:

0 se o elemento for removido;

1 se a pqueue estiver vazia.

Definido na linha 102 do ficheiro pqueue.c.

**5.10.2.8 int pqueueSetComp (PQueue *pqueue*, int(\*)(void \*, void \*) *comp*)**

Altera a função que compara os elementos de uma árvore.

O valor de *comp* não pode ser NULL.

**Parâmetros:**

*pqueue* pqueue.

*comp* nova função.

**Retorna:**

1 se *comp* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 29 do ficheiro pqueue.c.

**5.10.2.9 int pqueueSize (PQueue *pqueue*)**

Determina o tamanho de uma pqueue.

Devolve o valor do campo *size* da pqueue.

**Parâmetros:**

*pqueue* pqueue.

**Retorna:**

número de elementos da pqueue.

Definido na linha 150 do ficheiro pqueue.c.

**5.11 Referência ao Ficheiro queue.c****5.11.1 Descrição Detalhada**

Implementação de uma queue como lista ligada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

08/2007

Definido no ficheiro [queue.c](#).

## Funções

- `Queue newQueue ()`  
*Cria uma queue.*
- `void queueDelete (Queue queue)`  
*Elimina uma queue.*
- `int queueInsert (Queue queue, void *inf)`  
*Insere um elemento numa queue.*
- `int queueRemove (Queue queue, void **inf)`  
*Remove um elemento de uma queue.*
- `int queueConsult (Queue queue, void **inf)`  
*Verifica qual o elemento na cabeça de uma queue.*
- `int queueSize (Queue queue)`  
*Determina o tamanho de uma queue.*
- `int queueMap (Queue queue, void(*fun)(void *))`  
*Aplica uma função aos elementos de uma queue começando pela cabeça.*
- `Iterator queueIterator (Queue queue)`  
*Cria um iterador a partir da queue.*

### 5.11.2 Documentação das Funções

#### 5.11.2.1 Queue newQueue ()

Cria uma queue.

Se não for possível criar a queue devolve NULL.

#### Retorna:

queue inicializada ou NULL.

Definido na linha 13 do ficheiro queue.c.

#### 5.11.2.2 int queueConsult (Queue queue, void \*\* inf)

Verifica qual o elemento na cabeça de uma queue.

Se a queue estiver vazia é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço da informação que está na cabeça da queue; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*queue* queue.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a queue não estiver vazia;

1 se a queue estiver vazia.

Definido na linha 104 do ficheiro queue.c.

**5.11.2.3 void queueDelete (Queue *queue*)**

Elimina uma queue.

**Atenção:**

apenas liberta a memória ocupada pela estrutura da queue; não liberta o espaço ocupado pelos elementos nela contidos.

**Parâmetros:**

*queue* queue.

Definido na linha 29 do ficheiro queue.c.

**5.11.2.4 int queueInsert (Queue *queue*, void \* *inf*)**

Insere um elemento numa queue.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*queue* queue.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 se não for possível alocar espaço para o novo elemento.

Definido na linha 49 do ficheiro queue.c.

**5.11.2.5 Iterator queueIterator (Queue *queue*)**

Cria um iterador a partir da queue.

Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*queue* queue.

**Retorna:**

iterador criado ou NULL.

Definido na linha 143 do ficheiro queue.c.

**5.11.2.6 int queueMap (Queue *queue*, void(\*)(void \*) *fun*)**

Aplica uma função aos elementos de uma queue começando pela cabeça.

A função *fun* tem que ser do tipo: void fun(void\*).

**Parâmetros:**

*queue* queue.

*fun* função a ser aplicada.

**Retorna:**

0 se a queue não estiver vazia;

1 se a queue estiver vazia.

Definido na linha 127 do ficheiro queue.c.

**5.11.2.7 int queueRemove (Queue *queue*, void \*\* *inf*)**

Remove um elemento de uma queue.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a queue estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*queue* queue.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a queue estiver vazia.

Definido na linha 71 do ficheiro queue.c.

**5.11.2.8 int queueSize (Queue *queue*)**

Determina o tamanho de uma queue.

Devolve o valor do campo *size* da queue.

**Parâmetros:**

*queue* queue.

**Retorna:**

número de elementos da queue.

Definido na linha 120 do ficheiro queue.c.

## 5.12 Referência ao Ficheiro queue.h

### 5.12.1 Descrição Detalhada

Implementação de uma queue como lista ligada.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma queue.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [queue.h](#).

**Estruturas de Dados**

- struct [SQueueNode](#)  
*Estrutura do nodo da queue.*
- struct [SQueue](#)  
*Estrutura da queue.*

**Definições de Tipos**

- typedef [SQueueNode](#) \* [QueueNode](#)  
*Definição do apontador para os nodos da queue.*
- typedef [SQueue](#) \* [Queue](#)  
*Definição da queue.*

**Funções**

- [Queue newQueue](#) ()  
*Cria uma queue.*
- void [queueDelete](#) ([Queue](#) queue)

*Elimina uma queue.*

- int `queueInsert` (`Queue queue`, void \**inf*)  
*Insere um elemento numa queue.*
- int `queueRemove` (`Queue queue`, void \*\**inf*)  
*Remove um elemento de uma queue.*
- int `queueConsult` (`Queue queue`, void \*\**inf*)  
*Verifica qual o elemento na cabeça de uma queue.*
- int `queueSize` (`Queue queue`)  
*Determina o tamanho de uma queue.*
- int `queueMap` (`Queue queue`, void(\*fun)(void \*))  
*Aplica uma função aos elementos de uma queue começando pela cabeça.*
- Iterator `queueIterator` (`Queue queue`)  
*Cria um iterador a partir da queue.*

## 5.12.2 Documentação das Funções

### 5.12.2.1 Queue newQueue ()

Cria uma queue.

Se não for possível criar a queue devolve NULL.

#### Retorna:

queue inicializada ou NULL.

Definido na linha 13 do ficheiro queue.c.

### 5.12.2.2 int queueConsult (Queue queue, void \*\* inf)

Verifica qual o elemento na cabeça de uma queue.

Se a queue estiver vazia é colocado o valor NULL em *inf*.

#### Atenção:

esta função coloca em *inf* o endereço da informação que está na cabeça da queue; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

#### Parâmetros:

*queue* queue.

*inf* endereço onde é colocado o resultado.

**Retorna:**

- 0 se a queue não estiver vazia;
- 1 se a queue estiver vazia.

Definido na linha 104 do ficheiro queue.c.

**5.12.2.3 void queueDelete (Queue queue)**

Elimina uma queue.

**Atenção:**

apenas liberta a memória ocupada pela estrutura da queue; não liberta o espaço ocupado pelos elementos nela contidos.

**Parâmetros:**

*queue* queue.

Definido na linha 29 do ficheiro queue.c.

**5.12.2.4 int queueInsert (Queue queue, void \* inf)**

Inserir um elemento numa queue.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*queue* queue.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

- 0 se o elemento for inserido;
- 1 se não for possível alocar espaço para o novo elemento.

Definido na linha 49 do ficheiro queue.c.

**5.12.2.5 Iterator queueIterator (Queue queue)**

Cria um iterador a partir da queue.

Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*queue* queue.

**Retorna:**

iterador criado ou NULL.

Definido na linha 143 do ficheiro queue.c.

**5.12.2.6 int queueMap (Queue *queue*, void(\*) (void \*) *fun*)**

Aplica uma função aos elementos de uma queue começando pela cabeça.

A função *fun* tem que ser do tipo: void *fun* (void\*).

**Parâmetros:**

*queue* queue.

*fun* função a ser aplicada.

**Retorna:**

0 se a queue não estiver vazia;

1 se a queue estiver vazia.

Definido na linha 127 do ficheiro queue.c.

**5.12.2.7 int queueRemove (Queue *queue*, void \*\* *inf*)**

Remove um elemento de uma queue.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a queue estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*queue* queue.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a queue estiver vazia.

Definido na linha 71 do ficheiro queue.c.

**5.12.2.8 int queueSize (Queue *queue*)**

Determina o tamanho de uma queue.

Devolve o valor do campo *size* da queue.

**Parâmetros:**

*queue* queue.

**Retorna:**

número de elementos da queue.

Definido na linha 120 do ficheiro queue.c.

## 5.13 Referência ao Ficheiro rlp.c

### 5.13.1 Descrição Detalhada

Implementação de funções de programação linear.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.1.1

**Data:**

02/2009

Definido no ficheiro [rlp.c](#).

**Macros**

- #define [POS](#)(L, C, NC) ((L)\*(NC)+(C))

*Dado a linha, a coluna e o número de colunas de um array de duas dimensões obtém o índice dessa posição assumindo que o array é de uma dimensão.*

**Funções**

- static void [fmprint](#) (double \*matrix, int nrows, int ncols, FILE \*file)  
*Imprime uma matriz associada a um problema de programação linear.*
- static double [minimumc](#) (double \*matrix, int nrows, int ncols, int col, int \*row)  
*Determina o menor valor de uma coluna de uma matriz.*
- static double [minimumr](#) (double \*matrix, int ncols, int row, int \*col)  
*Determina o menor valor de uma linha de uma matriz.*
- int [simplex](#) (double \*a, int n, int m, FILE \*file)  
*Aplica o Algoritmo Simplex a um problema de optimização (programação linear).*
- int [simplexp](#) (double \*a, int n, int m, int pos, FILE \*file)  
*Aplica o Algoritmo Simplex primal a um problema de programação linear.*
- int [simplexd](#) (double \*a, int n, int m, int pos, FILE \*file)  
*Aplica o Algoritmo Simplex dual a um problema de programação linear.*

### 5.13.2 Documentação das Funções

#### 5.13.2.1 `static void fmprint (double * matrix, int nrows, int ncols, FILE *file)` [static]

Imprime uma matriz associada a um problema de programação linear.

A matriz é impressa no ficheiro *file* (que deverá ter sido previamente aberto).

##### Parâmetros:

*matrix* matriz que vamos imprimir.

*nrows* número de linhas.

*ncols* número de colunas.

*file* ficheiro onde a matriz será impressa.

Definido na linha 30 do ficheiro rlp.c.

#### 5.13.2.2 `static double minimumc (double * matrix, int nrows, int ncols, int col, int * row)` [static]

Determina o menor valor de uma coluna de uma matriz.

##### Parâmetros:

*matrix* matriz onde procuramos o valor.

*nrows* número de linhas da matriz.

*ncols* número de colunas da matriz.

*col* coluna onde vamos procurar.

*row* local onde é colocado o índice da linha em que o elemento ocorreu.

##### Retorna:

mínimo valor encontrado na coluna *col* da matriz *matrix*.

Definido na linha 70 do ficheiro rlp.c.

#### 5.13.2.3 `static double minimumr (double * matrix, int ncols, int row, int * col)` [static]

Determina o menor valor de uma linha de uma matriz.

##### Parâmetros:

*matrix* matriz onde procuramos o valor.

*ncols* número de colunas da matriz.

*row* linha onde vamos procurar.

*col* local onde é colocado o índice da coluna em que o elemento ocorreu.

##### Retorna:

mínimo valor encontrado na linha *row* da matriz *matrix*.

Definido na linha 98 do ficheiro rlp.c.

**5.13.2.4 int simplex (double \* a, int n, int m, FILE \* file)**

Aplica o *Algoritmo Simplex* a um problema de optimização (programação linear).

Dado um problema de  $n$  variáveis ( $x_1, \dots, x_n$ ) e  $m$  condições ( $c_1=b_1, \dots, c_m=b_m$ ), a função deve receber uma matriz  $a$  de dimensão  $(m+1)*(n+m+2)$  contendo:

- em  $a[0][i-1]$  (para  $i$  de 1 até  $n$ ) o coeficiente da variável  $x_i$  na expressão a minimizar;
- em  $a[0][i]$  (para  $i$  de  $n$  até  $n+m+1$ ) o valor 0;
- em  $a[i][j-1]$  (para  $i$  de 1 até  $m$  e  $j$  de 1 até  $n$ ) o coeficiente da variável  $x_j$  na condição  $c_i$ ;
- em  $a[i][j]$  (para  $i$  de 1 até  $m$  e  $j$  de  $n$  até  $n+m-1$ ) a matriz identidade;
- em  $a[i][n+m]$  (para  $i$  de 1 até  $m$ ) o valor  $b_i$ ;
- em  $a[i][n+m+1]$  (para  $i$  de 1 até  $m$ ) o valor  $n+i$ .

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

**Parâmetros:**

- $a$  matriz que representa o problema (conforme a descrição acima).
- $n$  número de variáveis da função objectivo.
- $m$  número de condições.
- file* ficheiro onde as tabelas serão impressas (ou NULL).

**Retorna:**

- 0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 115 do ficheiro rlp.c.

**5.13.2.5 int simplexd (double \* a, int n, int m, int pos, FILE \* file)**

Aplica o *Algoritmo Simplex* dual a um problema de programação linear.

A matriz de entrada ( $a$ ) segue o formato da matriz de entrada da função [simplex](#).

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

**Parâmetros:**

- $a$  matriz que representa o problema.
- $n$  número de variáveis da função objectivo.
- $m$  número de condições.
- pos* posição em que se encontra o menor valor da primeira linha (sendo que o menor valor terá que ser obrigatoriamente negativo).
- file* ficheiro onde as tabelas serão impressas (ou NULL).

**Retorna:**

- 0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 207 do ficheiro rlp.c.

### 5.13.2.6 int simplexp (double \*a, int n, int m, int pos, FILE \*file)

Aplica o *Algoritmo Simplex* primal a um problema de programação linear.

A matriz de entrada (*a*) segue o formato da matriz de entrada da função `simplex`.

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

#### Parâmetros:

*a* matriz que representa o problema.

*n* número de variáveis da função objectivo.

*m* número de condições.

*pos* posição em que se encontra o menor valor da última coluna das restrições (sendo que o menor valor terá que ser obrigatoriamente negativo).

*file* ficheiro onde as tabelas serão impressas (ou NULL).

#### Retorna:

0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 143 do ficheiro rlp.c.

## 5.14 Referência ao Ficheiro rlp.h

### 5.14.1 Descrição Detalhada

Implementação de funções de programação linear.

#### Autor:

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

#### Versão:

2.1.1

#### Data:

02/2009

Definido no ficheiro `rlp.h`.

#### Funções

- int `simplex` (double \*a, int n, int m, FILE \*file)  
*Aplica o Algoritmo Simplex a um problema de optimização (programação linear).*
- int `simplexp` (double \*a, int n, int m, int pos, FILE \*file)  
*Aplica o Algoritmo Simplex primal a um problema de programação linear.*
- int `simplexd` (double \*a, int n, int m, int pos, FILE \*file)  
*Aplica o Algoritmo Simplex dual a um problema de programação linear.*

### 5.14.2 Documentação das Funções

#### 5.14.2.1 `int simplex (double * a, int n, int m, FILE * file)`

Aplica o *Algoritmo Simplex* a um problema de optimização (programação linear).

Dado um problema de  $n$  variáveis ( $x_1, \dots, x_n$ ) e  $m$  condições ( $c_1=b_1, \dots, c_m=b_m$ ), a função deve receber uma matriz  $a$  de dimensão  $(m+1)*(n+m+2)$  contendo:

- em  $a[0][i-1]$  (para  $i$  de 1 até  $n$ ) o coeficiente da variável  $x_i$  na expressão a minimizar;
- em  $a[0][i]$  (para  $i$  de  $n$  até  $n+m+1$ ) o valor 0;
- em  $a[i][j-1]$  (para  $i$  de 1 até  $m$  e  $j$  de 1 até  $n$ ) o coeficiente da variável  $x_j$  na condição  $c_i$ ;
- em  $a[i][j]$  (para  $i$  de 1 até  $m$  e  $j$  de  $n$  até  $n+m-1$ ) a matriz identidade;
- em  $a[i][n+m]$  (para  $i$  de 1 até  $m$ ) o valor  $b_i$ ;
- em  $a[i][n+m+1]$  (para  $i$  de 1 até  $m$ ) o valor  $n+i$ .

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

#### Parâmetros:

- a* matriz que representa o problema (conforme a descrição acima).
- n* número de variáveis da função objectivo.
- m* número de condições.
- file* ficheiro onde as tabelas serão impressas (ou NULL).

#### Retorna:

- 0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 115 do ficheiro rlp.c.

#### 5.14.2.2 `int simplexd (double * a, int n, int m, int pos, FILE * file)`

Aplica o *Algoritmo Simplex* dual a um problema de programação linear.

A matriz de entrada (*a*) segue o formato da matriz de entrada da função `simplex`.

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

#### Parâmetros:

- a* matriz que representa o problema.
- n* número de variáveis da função objectivo.
- m* número de condições.
- pos* posição em que se encontra o menor valor da primeira linha (sendo que o menor valor terá que ser obrigatoriamente negativo).
- file* ficheiro onde as tabelas serão impressas (ou NULL).

#### Retorna:

- 0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 207 do ficheiro rlp.c.

### 5.14.2.3 int simplexp (double \* a, int n, int m, int pos, FILE \* file)

Aplica o *Algoritmo Simplex* primal a um problema de programação linear.

A matriz de entrada (*a*) segue o formato da matriz de entrada da função [simplex](#).

Permite definir um ficheiro onde são colocadas as várias tabelas resultantes da aplicação do algoritmo (através da variável *file*).

#### Parâmetros:

*a* matriz que representa o problema.

*n* número de variáveis da função objectivo.

*m* número de condições.

*pos* posição em que se encontra o menor valor da última coluna das restrições (sendo que o menor valor terá que ser obrigatoriamente negativo).

*file* ficheiro onde as tabelas serão impressas (ou NULL).

#### Retorna:

0 se for possível determinar um resultado. 1 caso contrário.

Definido na linha 143 do ficheiro rlp.c.

## 5.15 Referência ao Ficheiro rstring.c

### 5.15.1 Descrição Detalhada

Implementação de funções que manipulam strings.

#### Autor:

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

#### Versão:

2.0.1

#### Data:

02/2009

Definido no ficheiro [rstring.c](#).

#### Funções

- int [delISpaces](#) (char \*str)  
*Remove espaços de uma string.*
- int [delCSpaces](#) (char \*str)  
*Remove espaços de uma string.*
- int [delESpaces](#) (char \*str)

*Remove espaços de uma string.*

- `int delSpaces` (`char *str`)  
*Remove espaços de uma string.*
- `int charElem` (`char c`, `const char *str`)  
*Verifica se um carácter ocorre numa string.*
- `List words` (`const char *str`)  
*Dada uma string, obtém a lista de palavras que a compõe.*
- `List strSep` (`const char *str`, `const char *delim`)  
*Divide uma string.*

## 5.15.2 Documentação das Funções

### 5.15.2.1 `int charElem` (`char c`, `const char * str`)

Verifica se um carácter ocorre numa string.

#### Parâmetros:

- `c` carácter que procuramos.
- `str` string onde vamos procurar.

#### Retorna:

- 0 se o carácter não ocorrer na string;
- 1 caso contrário.

Definido na linha 107 do ficheiro rstring.c.

### 5.15.2.2 `int delCSpaces` (`char * str`)

Remove espaços de uma string.

Remove espaço consecutivos de uma string.

#### Parâmetros:

- `str` string na qual vamos remover os espaços.

#### Retorna:

- tamanho da string resultante.

Definido na linha 38 do ficheiro rstring.c.

### 5.15.2.3 `int delESpaces` (`char * str`)

Remove espaços de uma string.

Remove os espaços no fim de uma string.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 60 do ficheiro rstring.c.

**5.15.2.4 int delSpaces (char \* *str*)**

Remove espaços de uma string.

Remove os espaços no início de uma string.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 14 do ficheiro rstring.c.

**5.15.2.5 int delSpaces (char \* *str*)**

Remove espaços de uma string.

Remove os espaços no início e no fim de uma string e ainda espaços consecutivos entre palavras.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 76 do ficheiro rstring.c.

**5.15.2.6 List strSep (const char \* *str*, const char \* *delim*)**

Divide uma string.

Cria uma lista de strings com as substrings da original delimitadas pelos caracteres pertencentes a *delim*. A string original não é alterada. Caso *str* seja NULL ou ocorra algum erro é devolvido NULL.

**Ver Também:**

[List](#)

**Parâmetros:**

*str* string a separar.

*delim* delimitadores.

**Retorna:**

lista de substrings.

Definido na linha 165 do ficheiro rstring.c.

**5.15.2.7 List words (const char \* str)**

Dada uma string, obtem a lista de palavras que a compõe.

A string original não é alterada. Caso *str* seja NULL ou ocorra algum erro é devolvido NULL.

**Ver Também:**

[List](#)

**Parâmetros:**

*str* string a separar.

**Retorna:**

lista de palavras ou NULL.

Definido na linha 123 do ficheiro rstring.c.

## 5.16 Referência ao Ficheiro rstring.h

### 5.16.1 Descrição Detalhada

Implementação de funções que manipulam strings.

Esta biblioteca disponibiliza funções que removem espaços em determinadas posições de uma string e que separam a string em função de determinados caracteres.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [rstring.h](#).

**Funções**

- int [delISpaces](#) (char \*str)  
*Remove espaços de uma string.*
- int [delCSpaces](#) (char \*str)

*Remove espaços de uma string.*

- `int delESpaces (char *str)`  
*Remove espaços de uma string.*
- `int delSpaces (char *str)`  
*Remove espaços de uma string.*
- `int charElem (char c, const char *str)`  
*Verifica se um carácter ocorre numa string.*
- `List words (const char *str)`  
*Dada uma string, obtem a lista de palavras que a compõe.*
- `List strSep (const char *str, const char *delim)`  
*Divide uma string.*

## 5.16.2 Documentação das Funções

### 5.16.2.1 `int charElem (char c, const char * str)`

Verifica se um carácter ocorre numa string.

#### Parâmetros:

- c* carácter que procuramos.
- str* string onde vamos procurar.

#### Retorna:

- 0 se o carácter não ocorrer na string;
- 1 caso contrário.

Definido na linha 107 do ficheiro rstring.c.

### 5.16.2.2 `int delCSpaces (char * str)`

Remove espaços de uma string.

Remove espaço consecutivos de uma string.

#### Parâmetros:

- str* string na qual vamos remover os espaços.

#### Retorna:

- tamanho da string resultante.

Definido na linha 38 do ficheiro rstring.c.

**5.16.2.3 int delESpaces (char \* str)**

Remove espaços de uma string.

Remove os espaços no fim de uma string.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 60 do ficheiro rstring.c.

**5.16.2.4 int delISpaces (char \* str)**

Remove espaços de uma string.

Remove os espaços no início de uma string.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 14 do ficheiro rstring.c.

**5.16.2.5 int delSpaces (char \* str)**

Remove espaços de uma string.

Remove os espaços no início e no fim de uma string e ainda espaços consecutivos entre palavras.

**Parâmetros:**

*str* string na qual vamos remover os espaços.

**Retorna:**

tamanho da string resultante.

Definido na linha 76 do ficheiro rstring.c.

**5.16.2.6 List strSep (const char \* str, const char \* delim)**

Divide uma string.

Cria uma lista de strings com as substrings da original delimitadas pelos caracteres pertencentes a *delim*. A string original não é alterada. Caso *str* seja NULL ou ocorra algum erro é devolvido NULL.

**Ver Também:**

[List](#)

**Parâmetros:**

*str* string a separar.  
*delim* delimitadores.

**Retorna:**

lista de substrings.

Definido na linha 165 do ficheiro `rstring.c`.

**5.16.2.7 List words (const char \* str)**

Dada uma string, obtem a lista de palavras que a compõe.

A string original não é alterada. Caso *str* seja NULL ou ocorra algum erro é devolvido NULL.

**Ver Também:**

[List](#)

**Parâmetros:**

*str* string a separar.

**Retorna:**

lista de palavras ou NULL.

Definido na linha 123 do ficheiro `rstring.c`.

**5.17 Referência ao Ficheiro `stack.c`****5.17.1 Descrição Detalhada**

Implementação de uma stack como lista ligada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [stack.c](#).

**Funções**

- [Stack newStack \(\)](#)  
*Cria uma stack.*

- void `stackDelete` (`Stack stack`)  
*Elimina uma stack.*
- int `stackPush` (`Stack stack`, void \*`inf`)  
*Insere um elemento numa stack.*
- int `stackPop` (`Stack stack`, void \*\*`inf`)  
*Remove o elemento que está no topo de uma stack.*
- int `stackTop` (`Stack stack`, void \*\*`inf`)  
*Verifica qual o elemento no topo de uma stack.*
- int `stackSize` (`Stack stack`)  
*Determina o tamanho de uma stack.*
- int `stackMap` (`Stack stack`, void(\*`fun`)(void \*))  
*Aplica uma função aos elementos de uma stack começando no topo.*
- Iterator `stackIterator` (`Stack stack`)  
*Cria um iterador a partir de uma stack.*

## 5.17.2 Documentação das Funções

### 5.17.2.1 Stack `newStack` ()

Cria uma stack.

Se não for possível criar a stack devolve NULL.

#### Retorna:

stack inicializada ou NULL.

Definido na linha 13 do ficheiro stack.c.

### 5.17.2.2 void `stackDelete` (`Stack stack`)

Elimina uma stack.

#### Atenção:

apenas liberta a memória referente à estrutura da stack; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*stack* stack.

Definido na linha 28 do ficheiro stack.c.

### 5.17.2.3 Iterator stackIterator (Stack *stack*)

Cria um iterador a partir de uma *stack*.

Se ocorrer algum erro a função devolve NULL.

#### Ver Também:

[Iterator](#)

#### Parâmetros:

*stack* *stack*.

#### Retorna:

iterador criado ou NULL.

Definido na linha 129 do ficheiro stack.c.

### 5.17.2.4 int stackMap (Stack *stack*, void(\*) (void \*) *fun*)

Aplica uma função aos elementos de uma *stack* começando no topo.

A função *fun* tem que ser do tipo: void *fun*(void\*).

#### Parâmetros:

*stack* *stack*.

*fun* função a ser aplicada.

#### Retorna:

0 se a *stack* não estiver vazia;

1 se a *stack* estiver vazia.

Definido na linha 113 do ficheiro stack.c.

### 5.17.2.5 int stackPop (Stack *stack*, void \*\* *inf*)

Remove o elemento que está no topo de uma *stack*.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a *stack* estiver vazia é colocado o valor NULL em *inf*.

#### Atenção:

esta função não liberta o espaço ocupado pelo elemento removido.

#### Parâmetros:

*stack* *stack*.

*inf* endereço onde é colocado o elemento removido (ou NULL).

#### Retorna:

0 se o elemento for removido;

1 se a *stack* estiver vazia.

Definido na linha 67 do ficheiro stack.c.

**5.17.2.6 int stackPush (Stack stack, void \* inf)**

Insere um elemento numa stack.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*stack* stack.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 caso não seja possível alocar memória para o novo elemento.

Definido na linha 47 do ficheiro stack.c.

**5.17.2.7 int stackSize (Stack stack)**

Determina o tamanho de uma stack.

Devolve o valor do campo *size* da stack.

**Parâmetros:**

*stack* stack.

**Retorna:**

número de elementos da stack.

Definido na linha 106 do ficheiro stack.c.

**5.17.2.8 int stackTop (Stack stack, void \*\* inf)**

Verifica qual o elemento no topo de uma stack.

Se a stack estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço da informação que está no topo da stack; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*stack* stack.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a stack não estiver vazia;

1 se a stack estiver vazia.

Definido na linha 90 do ficheiro stack.c.

## 5.18 Referência ao Ficheiro stack.h

### 5.18.1 Descrição Detalhada

Implementação de uma stack como lista ligada.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma stack.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.1

**Data:**

02/2009

Definido no ficheiro [stack.h](#).

### Estruturas de Dados

- struct [SStackNode](#)  
*Estrutura do nodo de uma stack.*
- struct [SStack](#)  
*Estrutura de uma stack.*

### Definições de Tipos

- typedef [SStackNode](#) \* [StackNode](#)  
*Definição do apontador para os nodos da stack.*
- typedef [SStack](#) \* [Stack](#)  
*Definição da stack.*

### Funções

- [Stack](#) [newStack](#) ()  
*Cria uma stack.*
- void [stackDelete](#) ([Stack](#) stack)  
*Elimina uma stack.*
- int [stackPush](#) ([Stack](#) stack, void \*inf)  
*Insere um elemento numa stack.*
- int [stackPop](#) ([Stack](#) stack, void \*\*inf)

*Remove o elemento que está no topo de uma stack.*

- `int stackTop (Stack stack, void **inf)`  
*Verifica qual o elemento no topo de uma stack.*
- `int stackSize (Stack stack)`  
*Determina o tamanho de uma stack.*
- `int stackMap (Stack stack, void(*fun)(void *))`  
*Aplica uma função aos elementos de uma stack começando no topo.*
- `Iterator stackIterator (Stack stack)`  
*Cria um iterador a partir de uma stack.*

## 5.18.2 Documentação das Funções

### 5.18.2.1 Stack newStack ()

Cria uma stack.

Se não for possível criar a stack devolve NULL.

#### Retorna:

stack inicializada ou NULL.

Definido na linha 13 do ficheiro stack.c.

### 5.18.2.2 void stackDelete (Stack stack)

Elimina uma stack.

#### Atenção:

apenas liberta a memória referente à estrutura da stack; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*stack* stack.

Definido na linha 28 do ficheiro stack.c.

### 5.18.2.3 Iterator stackIterator (Stack stack)

Cria um iterador a partir de uma stack.

Se ocorrer algum erro a função devolve NULL.

#### Ver Também:

[Iterator](#)

**Parâmetros:**

*stack* stack.

**Retorna:**

iterador criado ou NULL.

Definido na linha 129 do ficheiro stack.c.

**5.18.2.4 int stackMap (Stack *stack*, void(\*)(void \*) *fun*)**

Aplica uma função aos elementos de uma stack começando no topo.

A função *fun* tem que ser do tipo: void fun(void\*).

**Parâmetros:**

*stack* stack.

*fun* função a ser aplicada.

**Retorna:**

0 se a stack não estiver vazia;

1 se a stack estiver vazia.

Definido na linha 113 do ficheiro stack.c.

**5.18.2.5 int stackPop (Stack *stack*, void \*\* *inf*)**

Remove o elemento que está no topo de uma stack.

Permite devolver o elemento removido, caso o valor de *inf* seja diferente de NULL. Se a stack estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função não liberta o espaço ocupado pelo elemento removido.

**Parâmetros:**

*stack* stack.

*inf* endereço onde é colocado o elemento removido (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a stack estiver vazia.

Definido na linha 67 do ficheiro stack.c.

**5.18.2.6 int stackPush (Stack *stack*, void \* *inf*)**

Insere um elemento numa stack.

Verifica se é possível inserir o novo elemento, devolvendo 1 caso não seja possível.

**Parâmetros:**

*stack* stack.

*inf* endereço do elemento que queremos inserir.

**Retorna:**

0 se o elemento for inserido;

1 caso não seja possível alocar memória para o novo elemento.

Definido na linha 47 do ficheiro `stack.c`.

**5.18.2.7 int stackSize (Stack stack)**

Determina o tamanho de uma stack.

Devolve o valor do campo *size* da stack.

**Parâmetros:**

*stack* stack.

**Retorna:**

número de elementos da stack.

Definido na linha 106 do ficheiro `stack.c`.

**5.18.2.8 int stackTop (Stack stack, void \*\* inf)**

Verifica qual o elemento no topo de uma stack.

Se a stack estiver vazia é colocado o valor NULL em *inf*.

**Atenção:**

esta função coloca em *inf* o endereço da informação que está no topo da stack; depois de executar esta função é aconselhável fazer uma cópia da informação e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*stack* stack.

*inf* endereço onde é colocado o resultado.

**Retorna:**

0 se a stack não estiver vazia;

1 se a stack estiver vazia.

Definido na linha 90 do ficheiro `stack.c`.

**5.19 Referência ao Ficheiro `treemap.c`****5.19.1 Descrição Detalhada**

Implementação de uma árvore binária de pesquisa equilibrada.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro `treemap.c`.

**Funções**

- static `TreeNode leftRotate (TreeNode tree)`  
*Efectua uma rotação à esquerda.*
- static `TreeNode rightRotate (TreeNode tree)`  
*Efectua uma rotação à direita.*
- static `TreeNode leftBalance (TreeNode tree)`  
*Efectua as rotações necessária a uma árvore que está balanceada para a esquerda.*
- static `TreeNode rightBalance (TreeNode tree)`  
*Efectua as rotações necessária a uma árvore que está balanceada para a direita.*
- static `TreeNode rLeftBalance (TreeNode tree, int *h)`  
*Efectua as rotações necessária a uma árvore que está balanceada para a esquerda.*
- static `TreeNode rRightBalance (TreeNode tree, int *h)`  
*Efectua as rotações necessária a uma árvore que está balanceada para a direita.*
- static `TreeNode upperLeft (TreeNode tree)`  
*Determina o nodo com maior chave na subárvores esquerda da árvore dada.*
- `TreeNode newTree (int(*keyComp)(void *, void *))`  
*Cria uma árvore.*
- int `treeSetKComp (TreeNode tree, int(*keyComp)(void *, void *))`  
*Altera a função que compara as chaves de uma árvore.*
- static void `treeDelAux (TreeNode tree)`  
*Elimina um nodo de uma (sub)árvore e todos os seus descendentes.*
- void `treeDelete (TreeNode tree)`  
*Elimina uma árvore.*
- static `TreeNode treeInsAux (TreeNode tree, void *key, void *val, int replace, int *h, int(*comp)(void *, void *))`  
*Função auxiliar da função de inserção.*

- `int treeInsert (TreeMap tree, void *key, void *val, int replace)`  
*Inserir um par chave/valor numa árvore.*
- `static TreeNode treeRemAux (TreeNode tree, void *key, void **value, void(*del)(void *), int *h, int(*comp)(void *, void *))`  
*Função auxiliar da função de remoção.*
- `int treeRemove (TreeMap tree, void *key, void **value, void(*del)(void *))`  
*Remove um elemento de uma árvore.*
- `int treeGet (TreeMap tree, void *key, void **value)`  
*Procura um elemento numa árvore.*
- `int treeIsAVLAux (TreeNode tree)`  
*Verifica se uma (sub)árvore está equilibrada na raiz e se todas as suas subárvores também estão.*
- `int treeIsAVL (TreeMap tree)`  
*Verifica se uma árvore está equilibrada.*
- `static int treeHightAux (TreeNode tree)`  
*Determina a altura de uma (sub)árvore.*
- `int treeHeight (TreeMap tree)`  
*Determina a altura de uma árvore.*
- `int treeSize (TreeMap tree)`  
*Determina o número de elementos de uma árvore.*
- `static void treeInOAux (TreeNode tree, void(*fun)(void *, void *))`  
*Função auxiliar da função que efectua uma travessia In-Order da árvore.*
- `int treeInOrder (TreeMap tree, void(*fun)(void *, void *))`  
*Efectua uma travessia In-Order de uma árvore.*
- `static void treePrOAux (TreeNode tree, void(*fun)(void *, void *))`  
*Função auxiliar da função que efectua uma travessia Pre-Order da árvore.*
- `int treePreOrder (TreeMap tree, void(*fun)(void *, void *))`  
*Efectua uma travessia Pre-Order de uma árvore.*
- `static void treePsOAux (TreeNode tree, void(*fun)(void *, void *))`  
*Função auxiliar da função que efectua uma travessia Pos-Order da árvore.*
- `int treePosOrder (TreeMap tree, void(*fun)(void *, void *))`  
*Efectua uma travessia Pos-Order de uma árvore.*
- `static int treeKAux (TreeNode tree, Iterator it)`  
*Percorre uma (sub)árvore e adiciona as chaves a um iterador.*

- **Iterator** `treeKeys` (`TreeMap tree`)  
*Cria um iterador a partir das chaves de uma árvore.*
- `static int` `treeVAux` (`TreeNode tree`, `Iterator it`)  
*Percorre uma (sub)árvore e adiciona os valores associados às chaves a um iterador.*
- **Iterator** `treeValues` (`TreeMap tree`)  
*Cria um iterador a partir dos valores associados às chaves de uma árvore.*

## 5.19.2 Documentação das Funções

### 5.19.2.1 `static TreeNode leftBalance (TreeNode tree)` [`static`]

Efectua as rotações necessária a uma árvore que está balanceada para a esquerda.

Esta função destina-se a (sub)árvores cujo desequilíbrio resulte do processo de inserção de um novo elemento.

#### Parâmetros:

*tree* raiz da (sub)árvore que vamos equilibrar.

#### Retorna:

árvore equilibrada.

Definido na linha 76 do ficheiro treemap.c.

### 5.19.2.2 `static TreeNode leftRotate (TreeNode tree)` [`static`]

Efectua uma rotação à esquerda.

#### Parâmetros:

*tree* raiz da (sub)árvore que vamos rodar.

#### Retorna:

nova árvore.

Definido na linha 19 do ficheiro treemap.c.

### 5.19.2.3 `TreeMap newTree (int(*)(void *, void *) keyComp)`

Cria uma árvore.

Se não for possível criar a árvore devolve NULL. Tem que ser especificada a função que compara chaves. Esta função pode ser alterada a qualquer momento, utilizando a função `treeSetKComp`.

#### Parâmetros:

*keyComp* função que compara duas chaves.

#### Retorna:

árvore inicializada ou NULL.

Definido na linha 281 do ficheiro treemap.c.

**5.19.2.4 static `TreeNode rightBalance (TreeNode tree)` [static]**

Efectua as rotações necessária a uma árvore que está balanceada para a direita.

Esta função destina-se a (sub)árvores cujo desequilíbrio resulte do processo de inserção de um novo elemento.

**Parâmetros:**

*tree* raiz da (sub)árvore que vamos equilibrar.

**Retorna:**

árvore equilibrada.

Definido na linha 126 do ficheiro `treemap.c`.

**5.19.2.5 static `TreeNode rightRotate (TreeNode tree)` [static]**

Efectua uma rotação à direita.

**Parâmetros:**

*tree* raiz da (sub)árvore que vamos rodar.

**Retorna:**

nova árvore.

Definido na linha 46 do ficheiro `treemap.c`.

**5.19.2.6 static `TreeNode rLeftBalance (TreeNode tree, int *h)` [static]**

Efectua as rotações necessária a uma árvore que está balanceada para a esquerda.

Esta função destina-se a (sub)árvores cujo desequilíbrio resulte do processo de remoção de um novo elemento.

**Parâmetros:**

*tree* raiz da (sub)árvore que vamos equilibrar.

*h* indica se a altura da árvore foi alterada ou não.

**Retorna:**

árvore equilibrada.

Definido na linha 177 do ficheiro `treemap.c`.

**5.19.2.7 static `TreeNode rRightBalance (TreeNode tree, int *h)` [static]**

Efectua as rotações necessária a uma árvore que está balanceada para a direita.

Esta função destina-se a (sub)árvores cujo desequilíbrio resulte do processo de remoção de um novo elemento.

**Parâmetros:**

*tree* raiz da (sub)árvore que vamos equilibrar.

*h* indica se a altura da árvore foi ou não alterada.

**Retorna:**

árvore equilibrada.

Definido na linha 226 do ficheiro treemap.c.

**5.19.2.8 static void treeDelAux (TreeNode *tree*) [static]**

Elimina um nodo de uma (sub)árvore e todos os seus descendentes.

**Parâmetros:**

*tree* raiz da árvore.

Definido na linha 317 do ficheiro treemap.c.

**5.19.2.9 void treeDelete (TreeMap *tree*)**

Elimina uma árvore.

**Atenção:**

apenas liberta a memória referente à estrutura da árvore; não liberta o espaço ocupado pelos elementos nela contidos.

**Parâmetros:**

*tree* árvore.

Definido na linha 329 do ficheiro treemap.c.

**5.19.2.10 int treeGet (TreeMap *tree*, void \* *key*, void \*\* *value*)**

Procura um elemento numa árvore.

Devolve o valor associado a uma chave, se esta existir. Se a chave não existir é colocado o valor NULL em *value*.

**Atenção:**

esta função coloca em *value* o endereço do valor pretendido; depois de executar esta função é aconselhável fazer uma cópia do mesmo e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*tree* árvore.

*key* chave que procuramos.

*value* endereço onde é colocado o resultado.

**Retorna:**

0 se o elemento existir;

1 se o elemento não existir.

Definido na linha 591 do ficheiro treemap.c.

**5.19.2.11 int treeHeight (TreeMap tree)**

Determina a altura de uma árvore.

**Parâmetros:**

*tree* árvore.

**Retorna:**

altura da árvore.

Definido na linha 671 do ficheiro treemap.c.

**5.19.2.12 static int treeHightAux (TreeNode tree) [static]**

Determina a altura de uma (sub)árvore.

**Parâmetros:**

*tree* árvore.

**Retorna:**

altura da árvore.

Definido na linha 655 do ficheiro treemap.c.

**5.19.2.13 static void treeInOAux (TreeNode tree, void(\*)(void \*, void \*) fun) [static]**

Função auxiliar da função que efectua uma travessia *In-Order* da árvore.

**Parâmetros:**

*tree* árvore.

*fun* função que é aplicada aos elementos da árvore.

Definido na linha 691 do ficheiro treemap.c.

**5.19.2.14 int treeInOrder (TreeMap tree, void(\*)(void \*, void \*) fun)**

Efectua uma travessia *In-Order* de uma árvore.

Aplica a função *fun* a todos os elementos da árvore.

A função *fun* tem que ser do tipo: void fun(void\*, void\*).

**Parâmetros:**

*tree* árvore.

*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;

1 se a árvore estiver vazia.

Definido na linha 703 do ficheiro treemap.c.

**5.19.2.15** `static TreeNode treeInsAux (TreeNode tree, void * key, void * val, int replace, int * h, int(*)(void *, void *) comp) [static]`

Função auxiliar da função de inserção.

**Parâmetros:**

*tree* árvore onde vamos fazer a inserção.

*key* chave do elemento a inserir.

*val* valor associado à chave.

*replace* variável que indica se valores já existentes são ou não substituídos.

*h* variável que permite saber se a inserção aumentou o tamanho da árvore.

*comp* função que permite comparar duas chaves.

**Retorna:**

nova árvore.

Definido na linha 351 do ficheiro treemap.c.

**5.19.2.16** `int treeInsert (TreeMap tree, void * key, void * value, int replace)`

Insere um par chave/valor numa árvore.

Caso a chave já exista, a variável *replace* determina se o valor antigo é ou não substituído (caso seja não há substituição, caso tenha outro valor o novo elemento é inserido).

**Parâmetros:**

*tree* árvore.

*key* chave.

*value* valor associado à chave.

*replace* variável que determina se os elementos já existentes são ou não substituídos.

**Retorna:**

0 se o elemento for inserido;

1 se a árvore já possuía a chave indicada;

2 caso não seja possível alocar memória para o novo elemento.

Definido na linha 435 do ficheiro treemap.c.

**5.19.2.17** `int treeIsAVL (TreeMap tree)`

Verifica se uma árvore está equilibrada.

Considera-se que a árvore está equilibrada se a diferença entre a altura das subárvores esquerda e direita (em todos os nodos) não for superior a 1.

**Parâmetros:**

*tree* árvore.

**Retorna:**

0 se a árvore não estiver equilibrada;

1 caso contrário.

Definido na linha 641 do ficheiro treemap.c.

**5.19.2.18 int treeIsAVLAux (TreeNode tree)**

Verifica se uma (sub)árvore está equilibrada na raiz e se todas as suas subárvores também estão.

**Parâmetros:**

*tree* árvore.

**Retorna:**

-1 se não está equilibrada;  
altura da árvore caso contrário.

Definido na linha 623 do ficheiro treemap.c.

**5.19.2.19 static int treeKAux (TreeNode tree, Iterator it) [static]**

Percorre uma (sub)árvore e adiciona as chaves a um iterador.

**Parâmetros:**

*tree* árvore.

*it* iterador onde são colocadas as chaves.

**Retorna:**

1 se ocorrer algum erro;  
0 caso contrário.

Definido na linha 784 do ficheiro treemap.c.

**5.19.2.20 Iterator treeKeys (TreeMap tree)**

Cria um iterador a partir das chaves de uma árvore.

Faz uma travessia *In-Order* da árvore e "coloca" as referências para as chaves num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*tree* árvore.

**Retorna:**

iterador criado ou NULL.

Definido na linha 799 do ficheiro treemap.c.

**5.19.2.21 int treePosOrder (TreeMap tree, void(\*)(void \*, void \*) fun)**

Efectua uma travessia *Pos-Order* de uma árvore.

Aplica a função *fun* a todos os elementos da árvore.

A função *fun* tem que ser do tipo: `void fun(void*, void*)`.

**Parâmetros:**

*tree* árvore.

*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;

1 se a árvore estiver vazia.

Definido na linha 763 do ficheiro `treemap.c`.

**5.19.2.22 `int treePreOrder (TreeMap tree, void(*) (void *, void *) fun)`**

Efectua uma travessia *Pre-Order* de uma árvore.

Aplica a função `fun` a todos os elementos da árvore.

A função `fun` tem que ser do tipo: `void fun(void*, void*)`.

**Parâmetros:**

*tree* árvore.

*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;

1 se a árvore estiver vazia.

Definido na linha 733 do ficheiro `treemap.c`.

**5.19.2.23 `static void treePrOAux (TreeNode tree, void(*) (void *, void *) fun)` [static]**

Função auxiliar da função que efectua uma travessia *Pre-Order* da árvore.

**Parâmetros:**

*tree* árvore.

*fun* função que é aplicada aos elementos da árvore.

Definido na linha 721 do ficheiro `treemap.c`.

**5.19.2.24 `static void treePsOAux (TreeNode tree, void(*) (void *, void *) fun)` [static]**

Função auxiliar da função que efectua uma travessia *Pos-Order* da árvore.

**Parâmetros:**

*tree* árvore.

*fun* função que é aplicada aos elementos da árvore.

Definido na linha 751 do ficheiro `treemap.c`.

**5.19.2.25** `static TreeNode treeRemAux (TreeNode tree, void * key, void ** value, void(*) (void *) del, int * h, int(*) (void *, void *) comp)` [static]

Função auxiliar da função de remoção.

**Parâmetros:**

*tree* árvore onde vamos fazer a inserção.

*key* chave do elemento a inserir.

*value* local onde será colocada a informação associada ao elemento removido.

*del* função que elimina um chave.

*h* variável que permite saber se a remoção aumentou o tamanho da árvore.

*comp* função que permite comparar duas chaves.

**Retorna:**

nova árvore.

Definido na linha 464 do ficheiro treemap.c.

**5.19.2.26** `int treeRemove (TreeMap tree, void * key, void ** value, void(*) (void *) del)`

Remove um elemento de uma árvore.

Permite devolver o valor removido, caso o valor de *value* seja diferente de NULL. Se a chave não existir ou o elemento não for removido é colocado o valor NULL em *value*.

**Atenção:**

esta função não liberta o espaço ocupado pelo valor associado à chave; já o espaço ocupado pela chave removida, se *del* for diferente de NULL, será libertado.

**Parâmetros:**

*tree* árvore.

*key* chave que queremos remover.

*value* endereço onde é colocado o elemento removido (ou NULL).

*del* função que elimina uma chave (ou NULL).

**Retorna:**

0 se o elemento for removido;

1 se a chave não existir;

Definido na linha 576 do ficheiro treemap.c.

**5.19.2.27** `int treeSetKComp (TreeMap tree, int(*) (void *, void *) keyComp)`

Altera a função que compara as chaves de uma árvore.

O valor de *keyComp* não pode ser NULL.

**Parâmetros:**

*tree* árvore.

*keyComp* nova função.

**Retorna:**

1 se *keyComp* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 300 do ficheiro treemap.c.

**5.19.2.28 int treeSize (TreeMap tree)**

Determina o número de elementos de uma árvore.

Devolve o valor do campo *size* da árvore.

**Parâmetros:**

*tree* árvore.

**Retorna:**

número de elementos da árvore.

Definido na linha 679 do ficheiro treemap.c.

**5.19.2.29 Iterator treeValues (TreeMap tree)**

Cria um iterador a partir dos valores associados às chaves de uma árvore.

Faz uma travessia *In-Order* da árvore e "coloca" as referências para os "valores" num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*tree* árvore.

**Retorna:**

iterador criado ou NULL.

Definido na linha 838 do ficheiro treemap.c.

**5.19.2.30 static int treeVAux (TreeNode tree, Iterator it) [static]**

Percorre uma (sub)árvore e adiciona os valores associados às chaves a um iterador.

**Parâmetros:**

*tree* árvore.

*it* iterador onde são colocadas os valores associados às chaves.

**Retorna:**

1 se ocorrer algum erro;  
0 caso contrário.

Definido na linha 823 do ficheiro treemap.c.

**5.19.2.31** `static TreeNode upperLeft (TreeNode tree)` [`static`]

Determina o nodo com maior chave na subárvores esquerda da árvore dada.

**Parâmetros:**

*tree* árvore.

**Retorna:**

maior elemento da esquerda.

Definido na linha 271 do ficheiro `treemap.c`.

## 5.20 Referência ao Ficheiro `treemap.h`

### 5.20.1 Descrição Detalhada

Implementação de uma árvore binária de pesquisa equilibrada.

Esta biblioteca disponibiliza um conjunto de funções que permitem manipular uma árvore binária de pesquisa equilibrada.

Na criação de uma árvore é necessário especificar a função que compara as chaves.

```
int keyComp(void* key1, void* key2)
```

(usada pelas funções `treeInsert`, `treeRemove` e `treeGet`); deve devolver um valor menor do que 0 se *key1* for menor do que *key2*, um valor maior do que 0 se *key1* for maior do que *key2* e 0 caso contrário; pode ser alterada através da função `treeSetKComp`.

```
int keyComp(void* key1, void* key2)
{
    if(key1&&key2) return strcmp((char*)key1, (char*)key2);
    else return 0;
}
```

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.0.2

**Data:**

02/2009

Definido no ficheiro `treemap.h`.

**Estruturas de Dados**

- struct `STreeNode`  
*Estrutura do nodo da árvore.*
- struct `STreeMap`  
*Estrutura de uma árvore.*

### Definições de Tipos

- typedef `STreeNode` \* `TreeNode`  
*Definição do apontador para os nodos da árvore.*
- typedef `STreeMap` \* `TreeMap`  
*Definição da árvore.*

### Enumerações

- enum `BFactor` { `L`, `E`, `R` }  
*Tipo que indica para onde uma árvore está balanceada.*

### Funções

- `TreeMap newTree` (`int(*keyComp)(void *, void *)`)  
*Cria uma árvore.*
- `int treeSetKComp` (`TreeMap tree`, `int(*keyComp)(void *, void *)`)  
*Altera a função que compara as chaves de uma árvore.*
- `void treeDelete` (`TreeMap tree`)  
*Elimina uma árvore.*
- `int treeInsert` (`TreeMap tree`, `void *key`, `void *value`, `int replace`)  
*Insere um par chave/valor numa árvore.*
- `int treeRemove` (`TreeMap tree`, `void *key`, `void **value`, `void(*del)(void *)`)  
*Remove um elemento de uma árvore.*
- `int treeGet` (`TreeMap tree`, `void *key`, `void **value`)  
*Procura um elemento numa árvore.*
- `int treeIsAVL` (`TreeMap tree`)  
*Verifica se uma árvore está equilibrada.*
- `int treeHeight` (`TreeMap tree`)  
*Determina a altura de uma árvore.*
- `int treeSize` (`TreeMap tree`)  
*Determina o número de elementos de uma árvore.*
- `int treeInOrder` (`TreeMap tree`, `void(*fun)(void *, void *)`)  
*Efectua uma travessia In-Order de uma árvore.*
- `int treePreOrder` (`TreeMap tree`, `void(*fun)(void *, void *)`)  
*Efectua uma travessia Pre-Order de uma árvore.*

- `int treePosOrder (TreeMap tree, void(*fun)(void *, void *))`  
*Efectua uma travessia Pos-Order de uma árvore.*
- `Iterator treeKeys (TreeMap tree)`  
*Cria um iterador a partir das chaves de uma árvore.*
- `Iterator treeValues (TreeMap tree)`  
*Cria um iterador a partir dos valores associados às chaves de uma árvore.*

## 5.20.2 Documentação das Funções

### 5.20.2.1 `TreeMap newTree (int(*)(void *, void *) keyComp)`

Cria uma árvore.

Se não for possível criar a árvore devolve NULL. Tem que ser especificada a função que compara chaves. Esta função pode ser alterada a qualquer momento, utilizando a função `treeSetKComp`.

#### Parâmetros:

*keyComp* função que compara duas chaves.

#### Retorna:

árvore inicializada ou NULL.

Definido na linha 281 do ficheiro `treemap.c`.

### 5.20.2.2 `void treeDelete (TreeMap tree)`

Elimina uma árvore.

#### Atenção:

apenas liberta a memória referente à estrutura da árvore; não liberta o espaço ocupado pelos elementos nela contidos.

#### Parâmetros:

*tree* árvore.

Definido na linha 329 do ficheiro `treemap.c`.

### 5.20.2.3 `int treeGet (TreeMap tree, void * key, void ** value)`

Procura um elemento numa árvore.

Devolve o valor associado a uma chave, se esta existir. Se a chave não existir é colocado o valor NULL em *value*.

#### Atenção:

esta função coloca em *value* o endereço do valor pretendido; depois de executar esta função é aconselhável fazer uma cópia do mesmo e passar a trabalhar com a cópia para que não haja problemas de partilha de referências.

**Parâmetros:**

*tree* árvore.  
*key* chave que procuramos.  
*value* endereço onde é colocado o resultado.

**Retorna:**

0 se o elemento existir;  
1 se o elemento não existir.

Definido na linha 591 do ficheiro `treemap.c`.

**5.20.2.4 int treeHeight (TreeMap tree)**

Determina a altura de uma árvore.

**Parâmetros:**

*tree* árvore.

**Retorna:**

altura da árvore.

Definido na linha 671 do ficheiro `treemap.c`.

**5.20.2.5 int treeInOrder (TreeMap tree, void(\*) (void \*, void \*) fun)**

Efectua uma travessia *In-Order* de uma árvore.

Aplica a função `fun` a todos os elementos da árvore.

A função `fun` tem que ser do tipo: `void fun(void*, void*)`.

**Parâmetros:**

*tree* árvore.  
*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;  
1 se a árvore estiver vazia.

Definido na linha 703 do ficheiro `treemap.c`.

**5.20.2.6 int treeInsert (TreeMap tree, void \* key, void \* value, int replace)**

Insere um par chave/valor numa árvore.

Caso a chave já exista, a variável `replace` determina se o valor antigo é ou não substituído (caso seja não há substituição, caso tenha outro valor o novo elemento é inserido).

**Parâmetros:**

*tree* árvore.

*key* chave.

*value* valor associado à chave.

*replace* variável que determina se os elementos já existentes são ou não substituídos.

**Retorna:**

0 se o elemento for inserido;

1 se a árvore já possuía a chave indicada;

2 caso não seja possível alocar memória para o novo elemento.

Definido na linha 435 do ficheiro `treemap.c`.

**5.20.2.7 `int treeIsAVL (TreeMap tree)`**

Verifica se uma árvore está equilibrada.

Considera-se que a árvore está equilibrada se a diferença entre a altura das subárvores esquerda e direita (em todos os nodos) não for superior a 1.

**Parâmetros:**

*tree* árvore.

**Retorna:**

0 se a árvore não estiver equilibrada;

1 caso contrário.

Definido na linha 641 do ficheiro `treemap.c`.

**5.20.2.8 `Iterator treeKeys (TreeMap tree)`**

Cria um iterador a partir das chaves de uma árvore.

Faz uma travessia *In-Order* da árvore e "coloca" as referências para as chaves num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*tree* árvore.

**Retorna:**

iterador criado ou NULL.

Definido na linha 799 do ficheiro `treemap.c`.

**5.20.2.9 `int treePosOrder (TreeMap tree, void (*)(void *, void *) fun)`**

Efectua uma travessia *Pos-Order* de uma árvore.

Aplica a função `fun` a todos os elementos da árvore.

A função `fun` tem que ser do tipo: `void fun(void*, void*)`.

**Parâmetros:**

*tree* árvore.

*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;

1 se a árvore estiver vazia.

Definido na linha 763 do ficheiro `treemap.c`.

**5.20.2.10 `int treePreOrder (TreeMap tree, void(*) (void *, void *) fun)`**

Efectua uma travessia *Pre-Order* de uma árvore.

Aplica a função `fun` a todos os elementos da árvore.

A função `fun` tem que ser do tipo: `void fun (void*, void*)`.

**Parâmetros:**

*tree* árvore.

*fun* função a ser aplicada.

**Retorna:**

0 se a árvore não estiver vazia;

1 se a árvore estiver vazia.

Definido na linha 733 do ficheiro `treemap.c`.

**5.20.2.11 `int treeRemove (TreeMap tree, void * key, void ** value, void(*) (void *) del)`**

Remove um elemento de uma árvore.

Permite devolver o valor removido, caso o valor de *value* seja diferente de `NULL`. Se a chave não existir ou o elemento não for removido é colocado o valor `NULL` em *value*.

**Atenção:**

esta função não liberta o espaço ocupado pelo valor associado à chave; já o espaço ocupado pela chave removida, se *del* for diferente de `NULL`, será libertado.

**Parâmetros:**

*tree* árvore.

*key* chave que queremos remover.

*value* endereço onde é colocado o elemento removido (ou `NULL`).

*del* função que elimina uma chave (ou `NULL`).

**Retorna:**

0 se o elemento for removido;

1 se a chave não existir;

Definido na linha 576 do ficheiro `treemap.c`.

**5.20.2.12 int treeSetKComp (TreeMap tree, int(\*) (void \*, void \*) keyComp)**

Altera a função que compara as chaves de uma árvore.

O valor de *keyComp* não pode ser NULL.

**Parâmetros:**

*tree* árvore.

*keyComp* nova função.

**Retorna:**

1 se *keyComp* for NULL (não é efectuada qualquer alteração);  
0 caso contrário.

Definido na linha 300 do ficheiro treemap.c.

**5.20.2.13 int treeSize (TreeMap tree)**

Determina o número de elementos de uma árvore.

Devolve o valor do campo *size* da árvore.

**Parâmetros:**

*tree* árvore.

**Retorna:**

número de elementos da árvore.

Definido na linha 679 do ficheiro treemap.c.

**5.20.2.14 Iterator treeValues (TreeMap tree)**

Cria um iterador a partir dos valores associados às chaves de uma árvore.

Faz uma travessia *In-Order* da árvore e "coloca" as referências para os "valores" num iterador. Se ocorrer algum erro a função devolve NULL.

**Ver Também:**

[Iterator](#)

**Parâmetros:**

*tree* árvore.

**Retorna:**

iterador criado ou NULL.

Definido na linha 838 do ficheiro treemap.c.

## 5.21 Referência ao Ficheiro util.c

### 5.21.1 Descrição Detalhada

Implementação de funções auxiliares básicas.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.1.1

**Data:**

02/2009

Definido no ficheiro [util.c](#).

**Macros**

- #define [BUFSIZE](#) 32  
*Tamanho do buffer inicial usado nas funções [rgets](#) e [rgetsEOF](#).*

**Funções**

- int [rgets](#) (char \*\*str)  
*Lê uma frase do stdin.*
- int [rgetsEOF](#) (char \*\*str)  
*Lê uma frase do stdin.*
- int [rngets](#) (char \*str, int dim)  
*Lê uma frase do stdin.*
- int [getRandom](#) (int min, int max)  
*Gera um número aleatório entre dois valores.*
- static int [merge](#) (void \*vals[ ], int begin, int middle, int end, int(\*comp)(void \*, void \*))  
*Efectua a junção de duas partes (ordenadas) de um array.*
- int [mergeSort](#) (void \*vals[ ], int begin, int end, int(\*comp)(void \*, void \*))  
*Ordena um array.*

**Variáveis**

- long [seed](#) = 0  
*Semente da função [getRandom](#).*

### 5.21.2 Documentação das Funções

#### 5.21.2.1 int getRandom (int min, int max)

Gera um número aleatório entre dois valores.

**Atenção:**

o valor de min deve ser menor do que max.

**Parâmetros:**

*min* valor mínimo que pode ser gerado.

*max* valor máximo que pode ser gerado.

**Retorna:**

valor gerado.

Definido na linha 136 do ficheiro util.c.

#### 5.21.2.2 static int merge (void \* vals[], int begin, int midle, int end, int(\*) (void \*, void \*) comp) [static]

Efectua a junção de duas partes (ordenadas) de um array.

**Parâmetros:**

*vals* array com os valores.

*begin* início da primeira parte do array.

*midle* início da segunda parte do array.

*end* fim da segunda parte do array.

*comp* função que compara os elementos do array.

**Retorna:**

1 se ocorrer algum erro;

0 caso contrário.

Definido na linha 165 do ficheiro util.c.

#### 5.21.2.3 int mergeSort (void \* vals[], int begin, int end, int(\*) (void \*, void \*) comp)

Ordena um array.

Utiliza o algoritmo **Mergesort**.

Ordena os elementos do array no intervalo [*begin*,*end*], usando como função de comparação *comp*.

A função *comp* tem que ser do tipo: `int comp(void*, void*)`.

**Parâmetros:**

*vals* array de apontadores para os elementos a ordenar.

*begin* posição onde começamos a ordenação.

*end* posição onde termina a ordenação.

*comp* função de comparação.

**Retorna:**

número de erros detectados.

Definido na linha 201 do ficheiro util.c.

**5.21.2.4 int rgets (char \*\* str)**

Lê uma frase do stdin.

Lê todos o caracteres inseridos até encontrar o carácter `\n`. Esta função, ao contrário da `rngets`, recebe como argumento o endereço para uma variável do tipo `char*`, e não a própria variável. Se ocorrer algum erro é colocado o valor `NULL` em *str*.

**Atenção:**

o espaço para a string onde será colocado o resultado é reservado pela função `rgets`, como tal **não** deve ser previamente alocado, pois se isso acontecer será perdido.

**Parâmetros:**

*str* endereço da string onde será colocado o resultado.

**Retorna:**

-2 se o endereço do resultado não for válido;  
-1 se ocorrer algum erro;  
tamanho da string caso contrário.

Definido na linha 26 do ficheiro util.c.

**5.21.2.5 int rgetsEOF (char \*\* str)**

Lê uma frase do stdin.

Lê todos o caracteres inseridos até encontrar um `EndOfFile`. Esta função, ao contrário da `rngets`, recebe como argumento o endereço para uma variável do tipo `char*`, e não a própria variável. Se ocorrer algum erro é colocado o valor `NULL` em *str*.

**Atenção:**

o espaço para a string onde será colocado o resultado é reservado pela função `rgetsEOF`, como tal **não** deve ser previamente alocado, pois se isso acontecer será perdido.

**Parâmetros:**

*str* endereço da string onde será colocado o resultado.

**Retorna:**

-2 se o endereço do resultado não for válido;  
-1 se ocorrer algum erro;  
tamanho da string caso contrário.

Definido na linha 70 do ficheiro util.c.

**5.21.2.6 int rngets (char \* str, int dim)**

Lê uma frase do stdin.

Verifica se a frase introduzida não excede a tamanho máximo (dim-2):

- caso isto aconteça a frase é perdida;
- caso contrário a frase é colocada na variável *str* (as frases lidas nunca têm o carácter `\n` no fim).

**Atenção:**

o espaço referenciado por *str* tem que ser previamente alocado.

**Parâmetros:**

*str* local onde é guardada a frase lida.

*dim* tamanho máximo da expressão é dim-2.

**Retorna:**

tamanho da frase lida (ou -1 caso não seja lida nenhuma frase).

Definido na linha 114 do ficheiro util.c.

**5.22 Referência ao Ficheiro util.h****5.22.1 Descrição Detalhada**

Implementação de funções auxiliares básicas.

Esta biblioteca disponibiliza funções que permitem ler frases do teclado, gerar números aleatórios e ordenar vectores.

**Autor:**

Rui Carlos A. Gonçalves <[rcgoncalves.pt@gmail.com](mailto:rcgoncalves.pt@gmail.com)>

**Versão:**

2.1.1

**Data:**

02/2009

Definido no ficheiro [util.h](#).

**Funções**

- [int rgets](#) (char \*\*str)  
*Lê uma frase do stdin.*
- [int rgetsEOF](#) (char \*\*str)  
*Lê uma frase do stdin.*

- `int mgets` (`char *str`, `int dim`)  
*Lê uma frase do stdin.*
- `int getRandom` (`int min`, `int max`)  
*Gera um número aleatório entre dois valores.*
- `int mergeSort` (`void *vals[]`, `int begin`, `int end`, `int(*comp)(void *, void *)`)  
*Ordena um array.*

### 5.22.2 Documentação das Funções

#### 5.22.2.1 `int getRandom (int min, int max)`

Gera um número aleatório entre dois valores.

##### Atenção:

o valor de `min` deve ser menor do que `max`.

##### Parâmetros:

*min* valor mínimo que pode ser gerado.

*max* valor máximo que pode ser gerado.

##### Retorna:

valor gerado.

Definido na linha 136 do ficheiro util.c.

#### 5.22.2.2 `int mergeSort (void * vals[], int begin, int end, int(*)(void *, void *) comp)`

Ordena um array.

Utiliza o algoritmo **Mergesort**.

Ordena os elementos do array no intervalo [*begin*,*end*], usando como função de comparação *comp*.

A função *comp* tem que ser do tipo: `int comp(void*, void*)`.

##### Parâmetros:

*vals* array de apontadores para os elementos a ordenar.

*begin* posição onde começamos a ordenação.

*end* posição onde termina a ordenação.

*comp* função de comparação.

##### Retorna:

número de erros detectados.

Definido na linha 201 do ficheiro util.c.

### 5.22.2.3 int rgets (char \*\* str)

Lê uma frase do stdin.

Lê todos os caracteres inseridos até encontrar o carácter `\n`. Esta função, ao contrário da `rgets`, recebe como argumento o endereço para uma variável do tipo `char*`, e não a própria variável. Se ocorrer algum erro é colocado o valor `NULL` em `str`.

#### Atenção:

o espaço para a string onde será colocado o resultado é reservado pela função `rgets`, como tal **não** deve ser previamente alocado, pois se isso acontecer será perdido.

#### Parâmetros:

`str` endereço da string onde será colocado o resultado.

#### Retorna:

-2 se o endereço do resultado não for válido;  
-1 se ocorrer algum erro;  
tamanho da string caso contrário.

Definido na linha 26 do ficheiro `util.c`.

### 5.22.2.4 int rgetsEOF (char \*\* str)

Lê uma frase do stdin.

Lê todos os caracteres inseridos até encontrar um `EndOfFile`. Esta função, ao contrário da `rgets`, recebe como argumento o endereço para uma variável do tipo `char*`, e não a própria variável. Se ocorrer algum erro é colocado o valor `NULL` em `str`.

#### Atenção:

o espaço para a string onde será colocado o resultado é reservado pela função `rgetsEOF`, como tal **não** deve ser previamente alocado, pois se isso acontecer será perdido.

#### Parâmetros:

`str` endereço da string onde será colocado o resultado.

#### Retorna:

-2 se o endereço do resultado não for válido;  
-1 se ocorrer algum erro;  
tamanho da string caso contrário.

Definido na linha 70 do ficheiro `util.c`.

### 5.22.2.5 int rngets (char \* str, int dim)

Lê uma frase do stdin.

Verifica se a frase introduzida não excede a tamanho máximo (`dim-2`):

- caso isto aconteça a frase é perdida;

- caso contrário a frase é colocada na variável *str* (as frases lidas nunca têm o carácter  $\backslash n$  no fim).

**Atenção:**

o espaço referenciado por *str* tem que ser previamente alocado.

**Parâmetros:**

*str* local onde é guardada a frase lida.

*dim* tamanho máximo da expressão é  $dim-2$ .

**Retorna:**

tamanho da frase lida (ou -1 caso não seja lida nenhuma frase).

Definido na linha 114 do ficheiro util.c.

# Índice

- array.c, 9
  - arrayAt, 10
  - arrayCapacity, 10
  - arrayDelete, 11
  - arrayInsert, 11
  - arrayIterator, 11
  - arrayMap, 12
  - arrayRemove, 12
  - arrayResize, 13
  - arraySize, 13
  - newArray, 13
- array.h, 14
  - arrayAt, 15
  - arrayCapacity, 15
  - arrayDelete, 16
  - arrayInsert, 16
  - arrayIterator, 16
  - arrayMap, 17
  - arrayRemove, 17
  - arrayResize, 18
  - arraySize, 18
  - newArray, 18
- arrayAt
  - array.c, 10
  - array.h, 15
- arrayCapacity
  - array.c, 10
  - array.h, 15
- arrayDelete
  - array.c, 11
  - array.h, 16
- arrayInsert
  - array.c, 11
  - array.h, 16
- arrayIterator
  - array.c, 11
  - array.h, 16
- arrayMap
  - array.c, 12
  - array.h, 17
- arrayRemove
  - array.c, 12
  - array.h, 17
- arrayResize
  - array.c, 13
  - array.h, 18
- arraySize
  - array.c, 13
  - array.h, 18
- charElem
  - rstring.c, 74
  - rstring.h, 77
- delCSpaces
  - rstring.c, 74
  - rstring.h, 77
- delESpaces
  - rstring.c, 74
  - rstring.h, 77
- delISpaces
  - rstring.c, 75
  - rstring.h, 78
- delSpaces
  - rstring.c, 75
  - rstring.h, 78
- fmprint
  - rlp.c, 69
- getRandom
  - util.c, 106
  - util.h, 109
- hashDelete
  - hashmap.c, 20
  - hashmap.h, 26
- hashGet
  - hashmap.c, 20
  - hashmap.h, 26
- hashInsert
  - hashmap.c, 20
  - hashmap.h, 27
- hashKeys
  - hashmap.c, 21
  - hashmap.h, 27
- hashmap.c, 19
  - hashDelete, 20
  - hashGet, 20
  - hashInsert, 20
  - hashKeys, 21
  - hashRemove, 21
  - hashSetEquals, 21
  - hashSetFactor, 22
  - hashSetHash, 22
  - hashSize, 22
  - hashValues, 23
  - newHash, 23
  - reHash, 24
- hashmap.h, 24
  - hashDelete, 26
  - hashGet, 26

- hashInsert, 27
- hashKeys, 27
- hashRemove, 27
- hashSetEquals, 28
- hashSetFactor, 28
- hashSetHash, 29
- hashSize, 29
- hashValues, 29
- newHash, 29
- hashRemove
  - hashmap.c, 21
  - hashmap.h, 27
- hashSetEquals
  - hashmap.c, 21
  - hashmap.h, 28
- hashSetFactor
  - hashmap.c, 22
  - hashmap.h, 28
- hashSetHash
  - hashmap.c, 22
  - hashmap.h, 29
- hashSize
  - hashmap.c, 22
  - hashmap.h, 29
- hashValues
  - hashmap.c, 23
  - hashmap.h, 29
- itAdd
  - iterator.c, 31
  - iterator.h, 36
- itAt
  - iterator.c, 31
  - iterator.h, 36
- itDelete
  - iterator.c, 32
  - iterator.h, 36
- iterator.c, 30
  - itAdd, 31
  - itAt, 31
  - itDelete, 32
  - itGetPos, 32
  - itHasNext, 32
  - itHasPrev, 32
  - itNext, 33
  - itPrev, 33
  - itSetPos, 33
  - newIt, 34
- iterator.h, 34
  - itAdd, 36
  - itAt, 36
  - itDelete, 36
  - itGetPos, 36
  - itHasNext, 37
  - itHasPrev, 37
  - itNext, 37
  - itPrev, 37
  - itSetPos, 38
  - newIt, 38
- itGetPos
  - iterator.c, 32
  - iterator.h, 36
- itHasNext
  - iterator.c, 32
  - iterator.h, 37
- itHasPrev
  - iterator.c, 32
  - iterator.h, 37
- itNext
  - iterator.c, 33
  - iterator.h, 37
- itPrev
  - iterator.c, 33
  - iterator.h, 37
- itSetPos
  - iterator.c, 33
  - iterator.h, 38
- leftBalance
  - treemap.c, 89
- leftRotate
  - treemap.c, 89
- list.c, 39
  - listAt, 40
  - listDelete, 40
  - listFst, 40
  - listInsertAt, 41
  - listInsertFst, 41
  - listInsertLst, 42
  - listIterator, 42
  - listLst, 42
  - listMap, 43
  - listRemoveAt, 43
  - listRemoveFst, 43
  - listRemoveLst, 44
  - listSize, 44
  - newList, 44
- list.h, 45
  - listAt, 47
  - listDelete, 47
  - listFst, 47
  - listInsertAt, 48
  - listInsertFst, 48
  - listInsertLst, 48
  - listIterator, 49
  - listLst, 49
  - listMap, 49
  - listRemoveAt, 50

- listRemoveFst, 50
- listRemoveLst, 51
- listSize, 51
- newList, 51
- listAt
  - list.c, 40
  - list.h, 47
- listDelete
  - list.c, 40
  - list.h, 47
- listFst
  - list.c, 40
  - list.h, 47
- listInsertAt
  - list.c, 41
  - list.h, 48
- listInsertFst
  - list.c, 41
  - list.h, 48
- listInsertLst
  - list.c, 42
  - list.h, 48
- listIterator
  - list.c, 42
  - list.h, 49
- listLst
  - list.c, 42
  - list.h, 49
- listMap
  - list.c, 43
  - list.h, 49
- listRemoveAt
  - list.c, 43
  - list.h, 50
- listRemoveFst
  - list.c, 43
  - list.h, 50
- listRemoveLst
  - list.c, 44
  - list.h, 51
- listSize
  - list.c, 44
  - list.h, 51
- merge
  - util.c, 106
- mergeSort
  - util.c, 106
  - util.h, 109
- minimumc
  - rlp.c, 69
- minimumr
  - rlp.c, 69
- newArray
  - array.c, 13
  - array.h, 18
- newHash
  - hashmap.c, 23
  - hashmap.h, 29
- newIt
  - iterator.c, 34
  - iterator.h, 38
- newList
  - list.c, 44
  - list.h, 51
- newPQueue
  - pqueue.c, 53
  - pqueue.h, 57
- newQueue
  - queue.c, 61
  - queue.h, 65
- newStack
  - stack.c, 80
  - stack.h, 84
- newTree
  - treemap.c, 89
  - treemap.h, 100
- pqueue.c, 52
  - newPQueue, 53
  - pqueueConsult, 53
  - pqueueDelete, 53
  - pqueueInsert, 53
  - pqueueIterator, 54
  - pqueueMap, 54
  - pqueueRemove, 54
  - pqueueSetComp, 55
  - pqueueSize, 55
- pqueue.h, 56
  - newPQueue, 57
  - pqueueConsult, 57
  - pqueueDelete, 58
  - pqueueInsert, 58
  - pqueueIterator, 58
  - pqueueMap, 59
  - pqueueRemove, 59
  - pqueueSetComp, 59
  - pqueueSize, 60
- pqueueConsult
  - pqueue.c, 53
  - pqueue.h, 57
- pqueueDelete
  - pqueue.c, 53
  - pqueue.h, 58
- pqueueInsert
  - pqueue.c, 53
  - pqueue.h, 58

- pqueueIterator
  - pqueue.c, 54
  - pqueue.h, 58
- pqueueMap
  - pqueue.c, 54
  - pqueue.h, 59
- pqueueRemove
  - pqueue.c, 54
  - pqueue.h, 59
- pqueueSetComp
  - pqueue.c, 55
  - pqueue.h, 59
- pqueueSize
  - pqueue.c, 55
  - pqueue.h, 60
- queue.c, 60
  - newQueue, 61
  - queueConsult, 61
  - queueDelete, 62
  - queueInsert, 62
  - queueIterator, 62
  - queueMap, 63
  - queueRemove, 63
  - queueSize, 63
- queue.h, 64
  - newQueue, 65
  - queueConsult, 65
  - queueDelete, 66
  - queueInsert, 66
  - queueIterator, 66
  - queueMap, 66
  - queueRemove, 67
  - queueSize, 67
- queueConsult
  - queue.c, 61
  - queue.h, 65
- queueDelete
  - queue.c, 62
  - queue.h, 66
- queueInsert
  - queue.c, 62
  - queue.h, 66
- queueIterator
  - queue.c, 62
  - queue.h, 66
- queueMap
  - queue.c, 63
  - queue.h, 66
- queueRemove
  - queue.c, 63
  - queue.h, 67
- queueSize
  - queue.c, 63
  - queue.h, 67
- reHash
  - hashmap.c, 24
- rgets
  - util.c, 107
  - util.h, 109
- rgetsEOF
  - util.c, 107
  - util.h, 110
- rightBalance
  - treemap.c, 89
- rightRotate
  - treemap.c, 90
- rLeftBalance
  - treemap.c, 90
- rlp.c, 68
  - fmprint, 69
  - minimumc, 69
  - minimumr, 69
  - simplex, 69
  - simplexd, 70
  - simplexp, 70
- rlp.h, 71
  - simplex, 72
  - simplexd, 72
  - simplexp, 72
- rngets
  - util.c, 107
  - util.h, 110
- rRightBalance
  - treemap.c, 90
- rstring.c, 73
  - charElem, 74
  - delCSpaces, 74
  - delESpaces, 74
  - delISpaces, 75
  - delSpaces, 75
  - strSep, 75
  - words, 76
- rstring.h, 76
  - charElem, 77
  - delCSpaces, 77
  - delESpaces, 77
  - delISpaces, 78
  - delSpaces, 78
  - strSep, 78
  - words, 79
- SArray, 3
- SHashMap, 3
- SHashNode, 4
- simplex
  - rlp.c, 69

- rlp.h, 72
- simplexd
  - rlp.c, 70
  - rlp.h, 72
- simplexp
  - rlp.c, 70
  - rlp.h, 72
- SIterator, 4
- SList, 5
- SListNode, 5
- SPQueue, 6
- SPQueueNode, 6
- SQueue, 6
- SQueueNode, 7
- SStack, 7
- SStackNode, 8
- stack.c, 79
  - newStack, 80
  - stackDelete, 80
  - stackIterator, 80
  - stackMap, 81
  - stackPop, 81
  - stackPush, 81
  - stackSize, 82
  - stackTop, 82
- stack.h, 83
  - newStack, 84
  - stackDelete, 84
  - stackIterator, 84
  - stackMap, 85
  - stackPop, 85
  - stackPush, 85
  - stackSize, 86
  - stackTop, 86
- stackDelete
  - stack.c, 80
  - stack.h, 84
- stackIterator
  - stack.c, 80
  - stack.h, 84
- stackMap
  - stack.c, 81
  - stack.h, 85
- stackPop
  - stack.c, 81
  - stack.h, 85
- stackPush
  - stack.c, 81
  - stack.h, 85
- stackSize
  - stack.c, 82
  - stack.h, 86
- stackTop
  - stack.c, 82
  - stack.h, 86
- STreeMap, 8
- STreeNode, 8
- strSep
  - rstring.c, 75
  - rstring.h, 78
- treeDelAux
  - treemap.c, 91
- treeDelete
  - treemap.c, 91
  - treemap.h, 100
- treeGet
  - treemap.c, 91
  - treemap.h, 100
- treeHeight
  - treemap.c, 91
  - treemap.h, 101
- treeHightAux
  - treemap.c, 92
- treeInOAux
  - treemap.c, 92
- treeInOrder
  - treemap.c, 92
  - treemap.h, 101
- treeInsAux
  - treemap.c, 92
- treeInsert
  - treemap.c, 93
  - treemap.h, 101
- treeIsAVL
  - treemap.c, 93
  - treemap.h, 102
- treeIsAVLAux
  - treemap.c, 93
- treeKAux
  - treemap.c, 94
- treeKeys
  - treemap.c, 94
  - treemap.h, 102
- treemap.c, 86
  - leftBalance, 89
  - leftRotate, 89
  - newTree, 89
  - rightBalance, 89
  - rightRotate, 90
  - rLeftBalance, 90
  - rRightBalance, 90
  - treeDelAux, 91
  - treeDelete, 91
  - treeGet, 91
  - treeHeight, 91
  - treeHightAux, 92
  - treeInOAux, 92

- treeInOrder, 92
- treeInsAux, 92
- treeInsert, 93
- treeIsAVL, 93
- treeIsAVLAux, 93
- treeKAux, 94
- treeKeys, 94
- treePosOrder, 94
- treePreOrder, 95
- treePrOAux, 95
- treePsOAux, 95
- treeRemAux, 95
- treeRemove, 96
- treeSetKComp, 96
- treeSize, 97
- treeValues, 97
- treeVAux, 97
- upperLeft, 97
- treemap.h, 98
  - newTree, 100
  - treeDelete, 100
  - treeGet, 100
  - treeHeight, 101
  - treeInOrder, 101
  - treeInsert, 101
  - treeIsAVL, 102
  - treeKeys, 102
  - treePosOrder, 102
  - treePreOrder, 103
  - treeRemove, 103
  - treeSetKComp, 103
  - treeSize, 104
  - treeValues, 104
- treePosOrder
  - treemap.c, 94
  - treemap.h, 102
- treePreOrder
  - treemap.c, 95
  - treemap.h, 103
- treePrOAux
  - treemap.c, 95
- treePsOAux
  - treemap.c, 95
- treeRemAux
  - treemap.c, 95
- treeRemove
  - treemap.c, 96
  - treemap.h, 103
- treeSetKComp
  - treemap.c, 96
  - treemap.h, 103
- treeSize
  - treemap.c, 97
  - treemap.h, 104
- treeValues
  - treemap.c, 97
  - treemap.h, 104
- treeVAux
  - treemap.c, 97
- upperLeft
  - treemap.c, 97
- util.c, 105
  - getRandom, 106
  - merge, 106
  - mergeSort, 106
  - rgets, 107
  - rgetsEOF, 107
  - rngets, 107
- util.h, 108
  - getRandom, 109
  - mergeSort, 109
  - rgets, 109
  - rgetsEOF, 110
  - rngets, 110
- words
  - rstring.c, 76
  - rstring.h, 79