

LibRCG

Version 3.1.1

Rui Carlos Gonçalves
rcgoncalves.pt@gmail.com

August 2, 2015

Contents

1 About	1
2 Data Structure Index	2
3 File Index	2
4 Data Structure Documentation	4
5 File Documentation	13
Index	105

1 About

LibRCG is a C library, developed by [Rui Carlos Gonçalves \(rcgoncalves.pt@gmail.com\)](mailto:rcgoncalves.pt@gmail.com). Among others, this library provides:

- Functions to read data from the standard input, to sort data, etc;
- Functions to manipulate strings;
- Linear programming algorithms (simplex and simplex dual);
- Data structures such as list, and functions to manage them;
- Finite maps data structures (balanced binary search trees and hash maps);
- An iterator.

1.1 License

LibRCG, a C library

Copyright (C) 2004–2015 Rui Carlos Gonçalves

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

1.2 Downloads

- [Source Code](#)
- [Documentation \(HTML\)](#)
- [Documentation \(PDF\)](#)

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

SArray		
Array structure		4
SHashMap		
Hash table structure		4
SHashNode		
Hash table node structure		5
SIterator		
Iterator structure		6
SList		
Linked list structure		7
SListNode		
Linked list node structure		8
SQueue		
Queue structure		9
SQueueNode		
Queue node structure		9
SStack		
Stack structure		10
SStackNode		
Stack node structure		11
STreeMap		
Tree structure		11
STreeNode		
Tree node structure		12

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

array.c		
Implementation of a dynamic array		13
array.h		
Implementation of a dynamic array		18

hashmap.c	Implementation of a hash table	22
hashmap.h	Implementation of a hash table	28
iterator.c	Implementation of an iterator	34
iterator.h	Implementation of an iterator	37
list.c	Implementation of a linked list	42
list.h	Implementation of a linked list	48
queue.c	Implementation of a queue as linked list	54
queue.h	Implementation of a queue as linked list	57
rlp.c	Implementation of linear programming functions	61
rlp.h	Implementation of linear programming functions	65
rstring.c	Implementation of functions to manipulate strings	67
rstring.h	Implementation of functions to manipulate strings	70
stack.c	Implementation of a stack as a linked list	72
stack.h	Implementation of a stack as a linked list	76
treemap.c	Implementation of an AVL tree (self-balancing binary search tree)	80
treemap.h	Implementation of an AVL tree (self-balancing binary search tree)	91
util.c	Implementation of some utility functions	98
util.h	Implementation of some utility functions	101

4 Data Structure Documentation

4.1 SArray Struct Reference

Array structure.

Data Fields

- int `capacity`
Current capacity of this array.
- int `size`
Number of element of this array.
- void ** `array`
Array of pointers to this array's elements.

4.1.1 Detailed Description

Array structure.

Definition at line 23 of file array.h.

4.1.2 Field Documentation

4.1.2.1 void** array

Array of pointers to this array's elements.

Definition at line 30 of file array.h.

4.1.2.2 int capacity

Current capacity of this array.

Definition at line 26 of file array.h.

4.1.2.3 int size

Number of element of this array.

Definition at line 28 of file array.h.

4.2 SHashMap Struct Reference

Hash table structure.

Data Fields

- int(* `hash`)(void *)
Hash function of this hash table.
- int(* `equals`)(void *, void *)
Comparison function of this hash table.

- int `size`
Number of elements of this hash table.
- int `length`
Number of buckets of this hash table.
- float `factor`
Load factor.
- `HashNode *` `elems`
Buckets of this hash table.

4.2.1 Detailed Description

Hash table structure.

Definition at line 82 of file `hashmap.h`.

4.2.2 Field Documentation

4.2.2.1 `HashNode*` `elems`

Buckets of this hash table.

Definition at line 95 of file `hashmap.h`.

4.2.2.2 `int(* equals)(void *, void *)`

Comparison function of this hash table.

Definition at line 87 of file `hashmap.h`.

4.2.2.3 float `factor`

Load factor.

Definition at line 93 of file `hashmap.h`.

4.2.2.4 `int(* hash)(void *)`

Hash function of this hash table.

Definition at line 85 of file `hashmap.h`.

4.2.2.5 int `length`

Number of buckets of this hash table.

Definition at line 91 of file `hashmap.h`.

4.2.2.6 int `size`

Number of elements of this hash table.

Definition at line 89 of file `hashmap.h`.

4.3 SHashNode Struct Reference

Hash table node structure.

Data Fields

- void * [key](#)
Node's key.
- void * [value](#)
Node's value.
- struct sHashNode * [next](#)
Next node.

4.3.1 Detailed Description

Hash table node structure.

Definition at line 64 of file hashmap.h.

4.3.2 Field Documentation

4.3.2.1 void* key

Node's key.

Definition at line 67 of file hashmap.h.

4.3.2.2 struct sHashNode* next

Next node.

Definition at line 71 of file hashmap.h.

4.3.2.3 void* value

Node's value.

Definition at line 69 of file hashmap.h.

4.4 SIterator Struct Reference

Iterator structure.

Data Fields

- int [capacity](#)
Capacity of this iterator.
- int [size](#)
Number of elements of this iterator.
- int [pos](#)
Current position of this iterator.
- void ** [values](#)
Elements of this iterator.

4.4.1 Detailed Description

Iterator structure.

Definition at line 17 of file iterator.h.

4.4.2 Field Documentation

4.4.2.1 int capacity

Capacity of this iterator..

Definition at line 20 of file iterator.h.

4.4.2.2 int pos

Current position of this iterator.

Definition at line 24 of file iterator.h.

4.4.2.3 int size

Number of elements of this iterator.

Definition at line 22 of file iterator.h.

4.4.2.4 void** values

Elements of this iterator.

Definition at line 26 of file iterator.h.

4.5 SList Struct Reference

Linked list structure.

Data Fields

- [int size](#)
Number of elements of this linked list.
- [ListNode first](#)
First node.
- [ListNode last](#)
Last node.

4.5.1 Detailed Description

Linked list structure.

Definition at line 37 of file list.h.

4.5.2 Field Documentation

4.5.2.1 ListNode first

First node.

Definition at line 42 of file list.h.

4.5.2.2 ListNode last

Last node.

Definition at line 44 of file list.h.

4.5.2.3 int size

Number of elements of this linked list.

Definition at line 40 of file list.h.

4.6 SListNode Struct Reference

Linked list node structure.

Data Fields

- void * [value](#)
Node's value.
- struct sListNode * [prev](#)
Previous node.
- struct sListNode * [next](#)
Next node.

4.6.1 Detailed Description

Linked list node structure.

Definition at line 19 of file list.h.

4.6.2 Field Documentation

4.6.2.1 struct sListNode* next

Next node.

Definition at line 26 of file list.h.

4.6.2.2 struct sListNode* prev

Previous node.

Definition at line 24 of file list.h.

4.6.2.3 void* value

Node's value.

Definition at line 22 of file list.h.

4.7 SQueue Struct Reference

Queue structure.

Data Fields

- `int size`
Number of elements of this queue.
- `QueueNode head`
Apontador para o início da queue.
- `QueueNode last`
Last node.

4.7.1 Detailed Description

Queue structure.

Definition at line 35 of file queue.h.

4.7.2 Field Documentation

4.7.2.1 QueueNode head

Apontador para o início da queue.

First node.

Definition at line 41 of file queue.h.

4.7.2.2 QueueNode last

Last node.

Definition at line 43 of file queue.h.

4.7.2.3 int size

Number of elements of this queue.

Definition at line 38 of file queue.h.

4.8 SQueueNode Struct Reference

Queue node structure.

Data Fields

- `void * value`
Node's value.
- `struct sQueueNode * next`
Next node.

4.8.1 Detailed Description

Queue node structure.

Definition at line 19 of file queue.h.

4.8.2 Field Documentation

4.8.2.1 struct sQueueNode* next

Next node.

Definition at line 24 of file queue.h.

4.8.2.2 void* value

Node's value.

Definition at line 22 of file queue.h.

4.9 SStack Struct Reference

Stack structure.

Data Fields

- [int size](#)
Number of values of this stack.
- [StackNode top](#)
Top node of this stack.

4.9.1 Detailed Description

Stack structure.

Definition at line 35 of file stack.h.

4.9.2 Field Documentation

4.9.2.1 int size

Number of values of this stack.

Definition at line 38 of file stack.h.

4.9.2.2 StackNode top

Top node of this stack.

Definition at line 40 of file stack.h.

4.10 SStackNode Struct Reference

Stack node structure.

Data Fields

- void * [value](#)
Node's value.
- struct sStackNode * [next](#)
Next node.

4.10.1 Detailed Description

Stack node structure.

Definition at line 19 of file stack.h.

4.10.2 Field Documentation

4.10.2.1 struct sStackNode* next

Next node.

Definition at line 24 of file stack.h.

4.10.2.2 void* value

Node's value.

Definition at line 22 of file stack.h.

4.11 STreeMap Struct Reference

Tree structure.

Data Fields

- int(* [keyComp](#))(void *, void *)
Key comparison function of this tree.
- int [size](#)
Number of elements of this tree.
- [TreeNode](#) [root](#)
Root node of this tree.

4.11.1 Detailed Description

Tree structure.

Definition at line 67 of file treemap.h.

4.11.2 Field Documentation

4.11.2.1 int(* keyComp)(void *, void *)

Key comparison function of this tree.

Definition at line 70 of file treemap.h.

4.11.2.2 TreeNode root

Root node of this tree.

Definition at line 74 of file treemap.h.

4.11.2.3 int size

Number of elements of this tree.

Definition at line 72 of file treemap.h.

4.12 STreeNode Struct Reference

Tree node structure.

Data Fields

- void * [key](#)
Node's key.
- void * [value](#)
Node's value.
- BFactor [bf](#)
Node's balance factor.
- struct sTreeNode * [super](#)
Node's parent.
- struct sTreeNode * [left](#)
Node's left subtree.
- struct sTreeNode * [right](#)
Node's right subtree.

4.12.1 Detailed Description

Tree node structure.

Definition at line 43 of file treemap.h.

4.12.2 Field Documentation

4.12.2.1 BFactor bf

Node's balance factor.

Definition at line 50 of file treemap.h.

4.12.2.2 void* key

Node's key.

Definition at line 46 of file treemap.h.

4.12.2.3 struct sTreeNode* left

Node's left subtree.

Definition at line 54 of file treemap.h.

4.12.2.4 struct sTreeNode* right

Node's right subtree.

Definition at line 56 of file treemap.h.

4.12.2.5 struct sTreeNode* super

Node's parent.

Definition at line 52 of file treemap.h.

4.12.2.6 void* value

Node's value.

Definition at line 48 of file treemap.h.

5 File Documentation

5.1 array.c File Reference

Implementation of a dynamic array.

Functions

- [Array newArray](#) (int size)
Creates an empty array, with the specified initial capacity.
- void [arrayDelete](#) (Array array)
Deletes an array.
- int [arrayInsert](#) (Array array, int index, void *elem, int replace)
Inserts an new element at the specified position of an array.
- int [arrayRemove](#) (Array array, int index, void **elem)
Removes the element at the specified position of an array.
- int [arrayAt](#) (Array array, int index, void **elem)
Provides the element at the specified position of an array.
- int [arrayResize](#) (Array array, int size)
Increases the capacity of an array.
- int [arraySize](#) (Array array)
Returns the size of an array.
- int [arrayCapacity](#) (Array array)

Return the capacity of an array.

- `int arrayMap (Array array, void(*fun)(void *))`

Applies a function to the elements of an array.

- `Iterator arrayIterator (Array array)`

Creates an iterator from an array.

5.1.1 Detailed Description

Implementation of a dynamic array.

Author

Rui Carlos Gonçalves

Version

3.0

Date

10/2011

Definition in file [array.c](#).

5.1.2 Function Documentation

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

Provides the element at the specified position of an array.

If there is no element at the specified position, it will be put the value `NULL` at `elem`.

Attention

This function puts at `elem` a pointer to the element at the specified position. Changes to this element will affect the element in the array.

Parameters

<i>array</i>	the array
<i>index</i>	the index of the element to be provided
<i>elem</i>	pointer where the element at the specified position will be put

Returns

0 if there was an element at the specified position
1 otherwise

Definition at line 95 of file [array.c](#).

5.1.2.2 int arrayCapacity (Array *array*)

Return the capacity of an array.

Parameters

<i>array</i>	the array
--------------	-----------

Returns

the capacity of the array

Definition at line 142 of file array.c.

5.1.2.3 void arrayDelete (Array *array*)

Deletes an array.

Attention

This function only frees the memory used by the array. It does not free the memory used by elements the array contains.

Parameters

<i>array</i>	the array to be deleted
--------------	-------------------------

Definition at line 35 of file array.c.

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

Inserts a new element at the specified position of an array.

The position, specified by argument *index*, must be a non negative integer. If necessary, the capacity of the array will be increased to *index*+1.

If the position is already filled, the *replace* argument specifies whether the new element should be added (it will be added only if *replace*!=0).

Attention

If the new element is NULL, it will not be inserted, and the size of the array will not change.

Parameters

<i>array</i>	the array
<i>index</i>	the index at which the new element is to be inserted
<i>elem</i>	the element to be inserted
<i>replace</i>	specify whether old value will be replaced

Returns

0 if the new element was inserted
 1 if the position was already filled
 2 if the position was not valid

3 if it was not possible increase the array size

Definition at line 43 of file array.c.

5.1.2.5 Iterator `arrayIterator (Array array)`

Creates an iterator from an array.

See Also

Iterator

Parameters

<i>array</i>	the array
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 169 of file array.c.

5.1.2.6 `int arrayMap (Array array, void (*)(void *) fun)`

Applies a function to the elements of an array.

The function to be applied must be of type `void fun(void*)`.

Parameters

<i>array</i>	the array
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 149 of file array.c.

5.1.2.7 `int arrayRemove (Array array, int index, void ** elem)`

Removes the element at the specified position of an array.

Provides the value of the removed element if the value of `elem` is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>array</i>	the array
<i>index</i>	the index of the element to be removed
<i>elem</i>	pointer where the removed element should be put (or NULL)

Returns

0 if an element was removed from the specified position
1 otherwise

Definition at line 71 of file array.c.

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

Increases the capacity of an array.

The new capacity must be greater than the current one.

Parameters

<i>array</i>	the array
<i>size</i>	the new capacity

Returns

0 if the capacity of the array was increased
1 if it was not possible change the array capacity
2 if the new capacity was less than the current one

Definition at line 114 of file array.c.

5.1.2.9 int arraySize (Array *array*)

Returns the size of an array.

Parameters

<i>array</i>	the array
--------------	-----------

Returns

the size of the array

Definition at line 135 of file array.c.

5.1.2.10 Array newArray (int *size*)

Creates an empty array, with the specified initial capacity.

The initial capacity must be a positive number.

Parameters

<i>size</i>	the initial capacity of the array
-------------	-----------------------------------

Returns

NULL if an error occurred
the new array otherwise

Definition at line 12 of file array.c.

5.2 array.h File Reference

Implementation of a dynamic array.

Data Structures

- struct [SArray](#)
Array structure.

Typedefs

- typedef [SArray](#) * [Array](#)
Array definition.

Functions

- [Array newArray](#) (int size)
Creates an empty array, with the specified initial capacity.
- void [arrayDelete](#) ([Array](#) array)
Deletes an array.
- int [arrayInsert](#) ([Array](#) array, int index, void *elem, int replace)
Inserts a new element at the specified position of an array.
- int [arrayRemove](#) ([Array](#) array, int index, void **elem)
Removes the element at the specified position of an array.
- int [arrayAt](#) ([Array](#) array, int index, void **elem)
Provides the element at the specified position of an array.
- int [arrayResize](#) ([Array](#) array, int size)
Increases the capacity of an array.
- int [arraySize](#) ([Array](#) array)
Returns the size of an array.
- int [arrayCapacity](#) ([Array](#) array)
Return the capacity of an array.
- int [arrayMap](#) ([Array](#) array, void(*fun)(void *))
Applies a function to the elements of an array.
- [Iterator arrayIterator](#) ([Array](#) array)
Creates an iterator from an array.

5.2.1 Detailed Description

Implementation of a dynamic array. Provides functions to create and manipulate a dynamic array with a desired length.

The array can resize itself whenever needed, while preserving the previous values.

Author

Rui Carlos Gonçalves

Version

3.0

Date

10/2011

Definition in file [array.h](#).**5.2.2 Typedef Documentation****5.2.2.1 typedef SArray* Array**

Array definition.

Definition at line 36 of file array.h.

5.2.3 Function Documentation**5.2.3.1 int arrayAt (Array array, int index, void ** elem)**

Provides the element at the specified position of an array.

If there is no element at the specified position, it will be put the value NULL at elem.

Attention

This function puts at elem a pointer to the element at the specified position. Changes to this element will affect the element in the array.

Parameters

<i>array</i>	the array
<i>index</i>	the index of the element to be provided
<i>elem</i>	pointer where the element at the specified position will be put

Returns

0 if there was an element at the specified position

1 otherwise

Definition at line 95 of file array.c.

5.2.3.2 int arrayCapacity (Array array)

Return the capacity of an array.

Parameters

<i>array</i>	the array
--------------	-----------

Returns

the capacity of the array

Definition at line 142 of file array.c.

5.2.3.3 void arrayDelete (Array *array*)

Deletes an array.

Attention

This function only frees the memory used by the array. It does not free the memory used by elements the array contains.

Parameters

<i>array</i>	the array to be deleted
--------------	-------------------------

Definition at line 35 of file array.c.

5.2.3.4 int arrayInsert (Array *array*, int *index*, void * *elem*, int *replace*)

Inserts a new element at the specified position of an array.

The position, specified by argument *index*, must be a non negative integer. If necessary, the capacity of the array will be increased to *index*+1.

If the position is already filled, the *replace* argument specifies whether the new element should be added (it will be added only if *replace*!=0).

Attention

If the new element is `NULL`, it will not be inserted, and the size of the array will not change.

Parameters

<i>array</i>	the array
<i>index</i>	the index at which the new element is to be inserted
<i>elem</i>	the element to be inserted
<i>replace</i>	specify whether old value will be replaced

Returns

- 0 if the new element was inserted
- 1 if the position was already filled
- 2 if the position was not valid
- 3 if it was not possible increase the array size

Definition at line 43 of file array.c.

5.2.3.5 Iterator arrayIterator (Array *array*)

Creates an iterator from an array.

See Also

Iterator

Parameters

<i>array</i>	the array
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 169 of file array.c.

5.2.3.6 int arrayMap (Array *array*, void(*)(void *) *fun*)

Applies a function to the elements of an array.

The function to be applied must be of type void fun(void*).

Parameters

<i>array</i>	the array
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 149 of file array.c.

5.2.3.7 int arrayRemove (Array *array*, int *index*, void ** *elem*)

Removes the element at the specified position of an array.

Provides the value of the removed element if the value of *elem* is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>array</i>	the array
<i>index</i>	the index of the element to be removed
<i>elem</i>	pointer where the removed element should be put (or NULL)

Returns

0 if an element was removed from the specified position
1 otherwise

Definition at line 71 of file array.c.

5.2.3.8 int arrayResize (Array *array*, int *size*)

Increases the capacity of an array.

The new capacity must be greater than the current one.

Parameters

<i>array</i>	the array
<i>size</i>	the new capacity

Returns

- 0 if the capacity of the array was increased
- 1 if it was not possible change the array capacity
- 2 if the new capacity was less than the current one

Definition at line 114 of file array.c.

5.2.3.9 int arraySize (Array *array*)

Returns the size of an array.

Parameters

<i>array</i>	the array
--------------	-----------

Returns

the size of the array

Definition at line 135 of file array.c.

5.2.3.10 Array newArray (int *size*)

Creates an empty array, with the specified initial capacity.

The initial capacity must be a positive number.

Parameters

<i>size</i>	the initial capacity of the array
-------------	-----------------------------------

Returns

NULL if an error occurred
the new array otherwise

Definition at line 12 of file array.c.

5.3 hashmap.c File Reference

Implementation of a hash table.

Functions

- static int `reHash` (`HashMap` hmap)
Resizes a hash table.
- `HashMap newHash` (int size, float factor, int(*hash)(void *), int(*equals)(void *, void *))
Creates a hash table.
- int `hashSetHash` (`HashMap` hmap, int(*hash)(void *))
Sets the hash function of a hash table.
- int `hashSetEquals` (`HashMap` hmap, int(*equals)(void *, void *))
Sets the comparison function of a hash table.
- int `hashSetFactor` (`HashMap` hmap, int factor)
Sets the load factor of a hash table.
- void `hashDelete` (`HashMap` hmap)
Deletes a hash table.
- int `hashInsert` (`HashMap` hmap, void *key, void *value, int replace)
Associates a value to a key in a hash table.
- int `hashRemove` (`HashMap` hmap, void *key, void **value, void(*del)(void *))
Removes the mapping for a key from a hash table.
- int `hashGet` (`HashMap` hmap, void *key, void **value)
Provides the mapping for a key from a hash table.
- int `hashSize` (`HashMap` hmap)
Returns the number of elements present in a hash table.
- `Iterator hashKeys` (`HashMap` hmap)
Creates an iterator from the keys of a hash table.
- `Iterator hashValues` (`HashMap` hmap)
Creates an iterator from the values of a hash table.

5.3.1 Detailed Description

Implementation of a hash table.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file `hashmap.c`.

5.3.2 Function Documentation

5.3.2.1 void hashDelete (HashMap *hmap*)

Deletes a hash table.

Attention

This function only free the memory used by the hash table. It does not free the memory used by elements the hash table contains.

Parameters

<i>hmap</i>	the hash table to be deleted
-------------	------------------------------

Definition at line 110 of file hashmap.c.

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

Provides the mapping for a key from a hash table.

If there is no mapping for the specified key, it will be put the value `NULL` at `value`. However, the `NULL` value may also say that the mapping for the specified key was `NULL`. Check the returned value in order to know whether the key was in the hash table.

Attention

This function puts at `value` a pointer to the mapping of the specified key. Changes to this value will affect the value in the hash table.

Parameters

<i>hmap</i>	the hash table
<i>key</i>	key whose mapping is to be provided
<i>value</i>	pointer where the mapping value will be put

Returns

0 if there was a mapping for the specified key
1 otherwise

Definition at line 193 of file hashmap.c.

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

Associates a value to a key in a hash table.

If the key already had a value, the `replace` argument specifies whether the new element should be added (it will be added only if `replace!=0`).

Parameters

<i>hmap</i>	the hash table
<i>key</i>	the key
<i>value</i>	the value to be inserted
<i>replace</i>	specifies whether an old value shall be replaced

Returns

- 0 if the value was inserted
- 1 if the key already had a value
- 2 if an error occurred

Definition at line 129 of file hashmap.c.

5.3.2.4 Iterator hashKeys (HashMap *hmap*)

Creates an iterator from the keys of a hash table.

See Also

Iterator

Parameters

<i>hmap</i>	the hash map
-------------	--------------

Returns

- NULL if an error occurred
- the iterator otherwise

Definition at line 218 of file hashmap.c.

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

Removes the mapping for a key from a hash table.

Provides the value of the removed element if the value of `elem` is not NULL.

Attention

This function does not free the memory used by the removed element. To free the memory used by its key, you have to provide the argument `del`.

Parameters

<i>hmap</i>	the hash table
<i>key</i>	key whose mapping is to be removed
<i>value</i>	pointer where the removed element shall be put (or NULL)
<i>del</i>	function to free the memory used by the key (or NULL)

Returns

- 0 if an element was removed from the specified position
- 1 otherwise

Definition at line 167 of file hashmap.c.

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

Sets the comparison function of a hash table.

Parameters

<i>hmap</i>	the hash table
<i>equals</i>	the new comparison function

Returns

1 if *equals* was equal to NULL (no change was made)
0 otherwise

Definition at line 90 of file hashmap.c.

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

Sets the load factor of a hash table.

The new value must be greater than or equal to 0.1.

Parameters

<i>hmap</i>	the hash table
<i>factor</i>	the new load factor

Returns

1 if *factor* was less than 0.1 (no change was made)
0 otherwise

Definition at line 100 of file hashmap.c.

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

Sets the hash function of a hash table.

Parameters

<i>hmap</i>	the hash table
<i>hash</i>	the new hash function

Returns

1 if *hash* was equal to NULL (no change was made)
0 otherwise

Definition at line 80 of file hashmap.c.

5.3.2.9 int hashSize (HashMap *hmap*)

Returns the number of elements present in a hash table.

Parameters

<i>hmap</i>	the hash table
-------------	----------------

Returns

the number of elements present in the hash table

Definition at line 211 of file hashmap.c.

5.3.2.10 Iterator hashValues (HashMap *hmap*)

Creates an iterator from the values of a hash table.

See Also

Iterator

Parameters

<i>hmap</i>	the hash table
-------------	----------------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 239 of file hashmap.c.

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

Creates a hash table.

The load factor (*factor*) specifies when the number of buckets should be increased (it will be increased when $size > factor * length$). The load factor must be greater than or equal to 0.1 (otherwise value 0.1 will be used).

Parameters

<i>size</i>	the initial number of buckets
<i>factor</i>	the load factor
<i>hash</i>	the hash function
<i>equals</i>	the comparison function

Returns

NULL if an error occurred
the new hash table otherwise

Definition at line 52 of file hashmap.c.

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

Resizes a hash table.

Doubles the number of buckets of a hash table, and updates the positions of the elements.

Parameters

<i>hmap</i>	the hash table
-------------	----------------

Returns

- 1 if an error occurred
- 0 otherwise

Definition at line 24 of file hashmap.c.

5.4 hashmap.h File Reference

Implementation of a hash table.

Data Structures

- struct [SHashNode](#)
Hash table node structure.
- struct [SHashMap](#)
Hash table structure.

Typedefs

- typedef [SHashNode](#) * [HashNode](#)
Hash table node definition.
- typedef [SHashMap](#) * [HashMap](#)
Hash table definition.

Functions

- [HashMap](#) [newHash](#) (int size, float factor, int(*hash)(void *), int(*equals)(void *, void *))
Creates a hash table.
- int [hashSetHash](#) ([HashMap](#) hmap, int(*hash)(void *))
Sets the hash function of a hash table.
- int [hashSetEquals](#) ([HashMap](#) hmap, int(*equals)(void *, void *))
Sets the comparison function of a hash table.
- int [hashSetFactor](#) ([HashMap](#) hmap, int factor)
Sets the load factor of a hash table.
- void [hashDelete](#) ([HashMap](#) hmap)
Deletes a hash table.
- int [hashInsert](#) ([HashMap](#) hmap, void *key, void *value, int replace)
Associates a value to a key in a hash table.
- int [hashRemove](#) ([HashMap](#) hmap, void *key, void **value, void(*del)(void *))
Removes the mapping for a key from a hash table.
- int [hashGet](#) ([HashMap](#) hmap, void *key, void **value)
Provides the mapping for a key from a hash table.
- int [hashSize](#) ([HashMap](#) hmap)
Returns the number of elements present in a hash table.
- [Iterator](#) [hashKeys](#) ([HashMap](#) hmap)
Creates an iterator from the keys of a hash table.
- [Iterator](#) [hashValues](#) ([HashMap](#) hmap)
Creates an iterator from the values of a hash table.

5.4.1 Detailed Description

Implementation of a hash table. Provides functions to create and manipulate a hash table.

Collisions are handled using having a list for each buckets. When the number of elements is greater than a specified ratio of the number of buckets the hash table is resized.

To use this hash table you have to provide the following functions for the data type used for keys:

- `int hash(void *)`

Hash function (used by [hashInsert](#), [hashRemove](#), and [hashGet](#)). Generates a unique hash to a key. This function can be changed using function [hashSetHash](#).

E.g.:

```
\textcolor{keywordtype}{int} hash(\textcolor{keywordtype}{void}* key)
\{
  \textcolor{keywordtype}{int} i,x;
  \textcolor{keywordtype}{char}* aux=key;
  \textcolor{keywordflow}{for} (i=0,x=0;i<32&&aux[i]!=\textcolor{charliteral}{'\(\backslash\)'};x+=a
  \textcolor{keywordflow}{return} x;
\}
```

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

Compares two keys (used by [hashInsert](#), [hashRemove](#), and [hashGet](#)). It should return 0 if `key1` is equal to `key2`, and a value different from 0 otherwise. This function can be changed using function [hashSetEquals](#).

E.g.:

```
\textcolor{keywordtype}{int} keyEquals(\textcolor{keywordtype}{void}* key1,\textcolor{keywordtype}{void}* key2)
\{
  \textcolor{keywordflow}{if} (key1&&key2) \textcolor{keywordflow}{return} strcmp((\textcolor{keywordtype}{void}*)key1,(\textcolor{keywordtype}{void}*)key2);
  \textcolor{keywordflow}{else} !(key1==key2);
\}
```

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [hashmap.h](#).

5.4.2 Typedef Documentation

5.4.2.1 typedef SHashMap* HashMap

Hash table definition.

Definition at line 101 of file [hashmap.h](#).

5.4.2.2 typedef SHashNode* HashNode

Hash table node definition.

Definition at line 77 of file hashmap.h.

5.4.3 Function Documentation

5.4.3.1 void hashDelete (HashMap *hmap*)

Deletes a hash table.

Attention

This function only free the memory used by the hash table. It does not free the memory used by elements the hash table contains.

Parameters

<i>hmap</i>	the hash table to be deleted
-------------	------------------------------

Definition at line 110 of file hashmap.c.

5.4.3.2 int hashGet (HashMap *hmap*, void * *key*, void ** *value*)

Provides the mapping for a key from a hash table.

If there is no mapping for the specified key, it will be put the value `NULL` at `value`. However, the `NULL` value may also say that the mapping for the specified key was `NULL`. Check the returned value in order to know whether the key was in the hash table.

Attention

This function puts at `value` a pointer to the mapping of the specified key. Changes to this value will affect the value in the hash table.

Parameters

<i>hmap</i>	the hash table
<i>key</i>	key whose mapping is to be provided
<i>value</i>	pointer where the mapping value will be put

Returns

0 if there was a mapping for the specified key
1 otherwise

Definition at line 193 of file hashmap.c.

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

Associates a value to a key in a hash table.

If the key already had a value, the `replace` argument specifies whether the new element should be added (it will be added only if `replace != 0`).

Parameters

<i>hmap</i>	the hash table
<i>key</i>	the key
<i>value</i>	the value to be inserted
<i>replace</i>	specifies whether an old value shall be replaced

Returns

- 0 if the value was inserted
- 1 if the key already had a value
- 2 if an error occurred

Definition at line 129 of file hashmap.c.

5.4.3.4 Iterator hashKeys (HashMap *hmap*)

Creates an iterator from the keys of a hash table.

See Also

Iterator

Parameters

<i>hmap</i>	the hash map
-------------	--------------

Returns

- NULL if an error occurred
- the iterator otherwise

Definition at line 218 of file hashmap.c.

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

Removes the mapping for a key from a hash table.

Provides the value of the removed element if the value of `elem` is not NULL.

Attention

This function does not free the memory used by the removed element. To free the memory used by its key, you have to provide the argument `del`.

Parameters

<i>hmap</i>	the hash table
<i>key</i>	key whose mapping is to be removed
<i>value</i>	pointer where the removed element shall be put (or NULL)
<i>del</i>	function to free the memory used by the key (or NULL)

Returns

0 if an element was removed from the specified position
1 otherwise

Definition at line 167 of file hashmap.c.

5.4.3.6 int hashSetEquals (HashMap *hmap*, int(*)(void *, void *) *equals*)

Sets the comparison function of a hash table.

Parameters

<i>hmap</i>	the hash table
<i>equals</i>	the new comparison function

Returns

1 if *equals* was equal to NULL (no change was made)
0 otherwise

Definition at line 90 of file hashmap.c.

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

Sets the load factor of a hash table.

The new value must be greater than or equal to 0.1.

Parameters

<i>hmap</i>	the hash table
<i>factor</i>	the new load factor

Returns

1 if *factor* was less than 0.1 (no change was made)
0 otherwise

Definition at line 100 of file hashmap.c.

5.4.3.8 int hashSetHash (HashMap *hmap*, int(*)(void *) *hash*)

Sets the hash function of a hash table.

Parameters

<i>hmap</i>	the hash table
<i>hash</i>	the new hash function

Returns

1 if *hash* was equal to NULL (no change was made)
0 otherwise

Definition at line 80 of file hashmap.c.

5.4.3.9 int hashSize (HashMap *hmap*)

Returns the number of elements present in a hash table.

Parameters

<i>hmap</i>	the hash table
-------------	----------------

Returns

the number of elements present in the hash table

Definition at line 211 of file hashmap.c.

5.4.3.10 Iterator hashValues (HashMap *hmap*)

Creates an iterator from the values of a hash table.

See Also

Iterator

Parameters

<i>hmap</i>	the hash table
-------------	----------------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 239 of file hashmap.c.

5.4.3.11 HashMap newHash (int *size*, float *factor*, int(*)*(void *) hash*, int(*)*(void *, void *) equals*)

Creates a hash table.

The load factor (*factor*) specifies when the number of buckets should be increased (it will be increased when *size* > *factor***length*). The load factor must be greater than or equal to 0.1 (otherwise value 0.1 will be used).

Parameters

<i>size</i>	the initial number of buckets
<i>factor</i>	the load factor
<i>hash</i>	the hash function
<i>equals</i>	the comparison function

Returns

NULL if an error occurred
the new hash table otherwise

Definition at line 52 of file hashmap.c.

5.5 iterator.c File Reference

Implementation of an iterator.

Functions

- [Iterator newIt](#) (int size)
Creates an iterator.
- void [itDelete](#) ([Iterator](#) it)
Deletes an iterator.
- int [itAdd](#) ([Iterator](#) it, void *val)
Adds an element to an iterator.
- int [itNext](#) ([Iterator](#) it, void **val)
Provides the next element of an iterator.
- int [itHasNext](#) ([Iterator](#) it)
Checks if there is "next".
- int [itPrev](#) ([Iterator](#) it, void **val)
Provides the previous element of an iterator.
- int [itHasPrev](#) ([Iterator](#) it)
Checks if there is "previous".
- int [itAt](#) ([Iterator](#) it, int n, void **val)
Provides the element at the specified position of an iterator.
- int [itSetPos](#) ([Iterator](#) it, int n)
Sets the current position of an iterator.
- int [itGetPos](#) ([Iterator](#) it)
Provides the current position of an iterator.

5.5.1 Detailed Description

Implementation of an iterator.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

08/2015

Definition in file [iterator.c](#).

5.5.2 Function Documentation

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

Adds an element to an iterator.

In case the iterator is full, the element is not added.

Parameters

<i>it</i>	the iterator
<i>val</i>	the value to be added

Returns

- 0 if the value was added
- 1 if the iterator was full

Definition at line 44 of file iterator.c.

5.5.2.2 int itAt (Iterator *it*, int *index*, void ** *elem*)

Provides the element at the specified position of an iterator.

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

Parameters

<i>it</i>	the iterator
<i>index</i>	the position
<i>elem</i>	pointer where the element at the specified position will be put

Returns

- 0 if there was an element at the specified position
- 1 otherwise

Definition at line 110 of file iterator.c.

5.5.2.3 void itDelete (Iterator *it*)

Deletes an iterator.

Parameters

<i>it</i>	the iterator to be deleted
-----------	----------------------------

Definition at line 36 of file iterator.c.

5.5.2.4 int itGetPos (Iterator *it*)

Provides the current position of an iterator.

Parameters

<i>it</i>	the iterator
-----------	--------------

Returns

the current position of the iterator

Definition at line 137 of file iterator.c.

5.5.2.5 int itHasNext (Iterator *it*)

Checks if there is "next".

Parameters

<i>it</i>	the iterator.
-----------	---------------

Returns

1 if there is "next"
0 otherwise

Definition at line 76 of file iterator.c.

5.5.2.6 int itHasPrev (Iterator *it*)

Checks if there is "previous".

Parameters

<i>it</i>	the iterator.
-----------	---------------

Returns

1 if there is "previous"
0 otherwise

Definition at line 102 of file iterator.c.

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

Provides the next element of an iterator.

Parameters

<i>it</i>	the iterator
<i>val</i>	pointer where the next element should be put

Returns

0 if the next element was provided
1 otherwise

Definition at line 58 of file iterator.c.

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

Provides the previous element of an iterator.

Parameters

<i>it</i>	the iterator
<i>val</i>	pointer were the previous element should be put

Returns

0 if the previous element was provided
1 otherwise

Definition at line 84 of file iterator.c.

5.5.2.9 int itSetPos (Iterator *it*, int *n*)

Sets the current position of an iterator.

It changes the value of field `pos` of an iterator (if the value of `n` is valid).

Parameters

<i>it</i>	the iterator
<i>n</i>	the new position

Returns

-1 if the value of the new position is invalid
the old position otherwise

Definition at line 124 of file iterator.c.

5.5.2.10 Iterator newIt (int *size*)

Creates an iterator.

Parameters

<i>size</i>	the capacity
-------------	--------------

Returns

NULL if an error occurred
the new iterator otherwise

Definition at line 12 of file iterator.c.

5.6 iterator.h File Reference

Implementation of an iterator.

Data Structures

- struct [SIterator](#)
Iterator structure.

Typedefs

- typedef [SIterator](#) * [Iterator](#)
Iterator definition.

Functions

- [Iterator newIt](#) (int size)
Creates an iterator.
- void [itDelete](#) ([Iterator](#) it)
Deletes an iterator.
- int [itAdd](#) ([Iterator](#) it, void *val)
Adds an element to an iterator.
- int [itNext](#) ([Iterator](#) it, void **val)
Provides the next element of an iterator.
- int [itHasNext](#) ([Iterator](#) it)
Checks if there is "next".
- int [itPrev](#) ([Iterator](#) it, void **val)
Provides the previous element of an iterator.
- int [itHasPrev](#) ([Iterator](#) it)
Checks if there is "previous".
- int [itAt](#) ([Iterator](#) it, int index, void **elem)
Provides the element at the specified position of an iterator.
- int [itSetPos](#) ([Iterator](#) it, int n)
Sets the current position of an iterator.
- int [itGetPos](#) ([Iterator](#) it)
Provides the current position of an iterator.

5.6.1 Detailed Description

Implementation of an iterator. Iterators contain a sequence of pointers to the elements to be iterated.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

08/2015

Definition in file [iterator.h](#).

5.6.2 Typedef Documentation

5.6.2.1 typedef SIterator* Iterator

Iterator definition.

Definition at line 32 of file iterator.h.

5.6.3 Function Documentation

5.6.3.1 int itAdd (Iterator *it*, void * *val*)

Adds an element to an iterator.

In case the iterator is full, the element is not added.

Parameters

<i>it</i>	the iterator
<i>val</i>	the value to be added

Returns

- 0 if the value was added
- 1 if the iterator was full

Definition at line 44 of file iterator.c.

5.6.3.2 int itAt (Iterator *it*, int *index*, void ** *elem*)

Provides the element at the specified position of an iterator.

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

Parameters

<i>it</i>	the iterator
<i>index</i>	the position
<i>elem</i>	pointer where the element at the specified position will be put

Returns

- 0 if there was an element at the specified position
- 1 otherwise

Definition at line 110 of file iterator.c.

5.6.3.3 void itDelete (Iterator *it*)

Deletes an iterator.

Parameters

<i>it</i>	the iterator to be deleted
-----------	----------------------------

Definition at line 36 of file iterator.c.

5.6.3.4 int itGetPos (Iterator *it*)

Provides the current position of an iterator.

Parameters

<i>it</i>	the iterator
-----------	--------------

Returns

the current position of the iterator

Definition at line 137 of file iterator.c.

5.6.3.5 int itHasNext (Iterator *it*)

Checks if there is "next".

Parameters

<i>it</i>	the iterator.
-----------	---------------

Returns

1 if there is "next"
0 otherwise

Definition at line 76 of file iterator.c.

5.6.3.6 int itHasPrev (Iterator *it*)

Checks if there is "previous".

Parameters

<i>it</i>	the iterator.
-----------	---------------

Returns

1 if there is "previous"
0 otherwise

Definition at line 102 of file iterator.c.

5.6.3.7 int itNext (Iterator *it*, void ** *val*)

Provides the next element of an iterator.

Parameters

<i>it</i>	the iterator
<i>val</i>	pointer were the next element should be put

Returns

0 if the next element was provided
1 otherwise

Definition at line 58 of file iterator.c.

5.6.3.8 int itPrev (Iterator *it*, void ** *val*)

Provides the previous element of an iterator.

Parameters

<i>it</i>	the iterator
<i>val</i>	pointer where the previous element should be put

Returns

0 if the previous element was provided
1 otherwise

Definition at line 84 of file iterator.c.

5.6.3.9 int itSetPos (Iterator *it*, int *n*)

Sets the current position of an iterator.

It changes the value of field `pos` of an iterator (if the value of `n` is valid).

Parameters

<i>it</i>	the iterator
<i>n</i>	the new position

Returns

-1 if the value of the new position is invalid
the old position otherwise

Definition at line 124 of file iterator.c.

5.6.3.10 Iterator newIt (int *size*)

Creates an iterator.

Parameters

<i>size</i>	the capacity
-------------	--------------

Returns

NULL if an error occurred
the new iterator otherwise

Definition at line 12 of file iterator.c.

5.7 list.c File Reference

Implementation of a linked list.

Functions

- [List newList](#) (void)
Creates a list.
- void [listDelete](#) (List list)
Deletes a list.
- int [listInsertFst](#) (List list, void *value)
Inserts an element at the beginning of a list.
- int [listInsertLst](#) (List list, void *value)
Inserts an element at the end of a list.
- int [listInsertAt](#) (List list, int index, void *value)
Inserts a new element at the specified position of a list.
- int [listRemoveFst](#) (List list, void **value)
Removes the first element of a list.
- int [listRemoveLst](#) (List list, void **value)
Removes the last element of a list.
- int [listRemoveAt](#) (List list, int index, void **value)
Removes the element at the specified position of a list.
- int [listFst](#) (List list, void **value)
Provides the value at the first position of a list.
- int [listLst](#) (List list, void **value)
Provides the value at the last position of a list.
- int [listAt](#) (List list, int index, void **value)
Provides the element at the specified position of a list.
- int [listSize](#) (List list)
Returns the size of a list.
- int [listMap](#) (List list, void(*fun)(void *))
Applies a function to the elements of a list.
- [Iterator listIterator](#) (List list)
Creates an iterator from a list.

5.7.1 Detailed Description

Implementation of a linked list.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [list.c](#).

5.7.2 Function Documentation

5.7.2.1 int listAt (List *list*, int *index*, void ** *value*)

Provides the element at the specified position of a list.

If there is no element at the specified position, it will be put the value `NULL` at `value`.

Attention

This function puts at `value` a pointer to the value at the last position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>index</i>	the index of the element to be provided
<i>value</i>	pointer where the value at the specified position will be put

Returns

0 if there was an elements at the specified position
1 otherwise

Definition at line 278 of file list.c.

5.7.2.2 void listDelete (List *list*)

Deletes a list.

Attention

This function only frees the memory used by the list. It does not free the memory used by elements the list contains.

Parameters

<i>list</i>	the list to be deleted
-------------	------------------------

Definition at line 27 of file list.c.

5.7.2.3 int listFst (List *list*, void ** *value*)

Provides the value at the first position of a list.

If the list is empty, it will be put the value `NULL` at `value`.

Attention

This function puts at `value` a pointer to the value at the first position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>value</i>	pointer where the value at the first position will be put

Returns

0 if the list was not empty
1 otherwise

Definition at line 250 of file list.c.

5.7.2.4 int listInsertAt (List *list*, int *index*, void * *value*)

Inserts an new element at the specified position of a list.

The position, specified by argument *index*, must be a non negative integer, and less than the current size of the list.

Parameters

<i>list</i>	the list
<i>index</i>	the index at which the new value is to be inserted
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if the position was not valid
2 if it was not possible to insert the new element

Definition at line 116 of file list.c.

5.7.2.5 int listInsertFst (List *list*, void * *value*)

Inserts an element at the beginning of a list.

Parameters

<i>list</i>	the list
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if it was not possible to insert the new element

Definition at line 42 of file list.c.

5.7.2.6 int listInsertLst (List *list*, void * *value*)

Inserts an element at the end of a list.

Parameters

<i>list</i>	the list
<i>value</i>	the value to be inserted

Returns

- 0 if the new value was inserted
- 1 if it was not possible to insert the new element

Definition at line 79 of file list.c.

5.7.2.7 Iterator listIterator (List *list*)

Creates an iterator from a list.

See Also

Iterator

Parameters

<i>list</i>	the list
-------------	----------

Returns

- NULL if an error occurred
- the iterator otherwise

Definition at line 320 of file list.c.

5.7.2.8 int listLst (List *list*, void ** *value*)

Provides the value at the last position of a list.

If the list is empty, it will be put the value NULL at *value*.

Attention

This function puts at *value* a pointer to the value at the last position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>value</i>	pointer where the value at the last position will be put

Returns

- 0 if the list was not empty
- 1 otherwise

Definition at line 264 of file list.c.

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

Applies a function to the elements of a list.

The function to be applied must be of type void fun(void*).

Parameters

<i>list</i>	the list
<i>fun</i>	the function to be applied

Returns

- 0 if the list was not empty
- 1 otherwise

Definition at line 305 of file list.c.

5.7.2.10 int listRemoveAt (List *list*, int *index*, void ** *value*)

Removes the element at the specified position of a list.

Provides the value of the removed element if the value of `value` is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>index</i>	the index of the element to be removed
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

- 0 if the element was removed
- 1 if the value of `index` was invalid

Definition at line 220 of file list.c.

5.7.2.11 int listRemoveFst (List *list*, void ** *value*)

Removes the first element of a list.

Provides the value of the removed element if the value of `value` is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if the element was removed
1 if the list was empty

Definition at line 158 of file list.c.

5.7.2.12 int listRemoveLst (List *list*, void ** *value*)

Removes the last element of a list.

Provides the value of the removed element if the value of *value* is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if the element was removed
1 if the list was empty

Definition at line 189 of file list.c.

5.7.2.13 int listSize (List *list*)

Returns the size of a list.

Parameters

<i>list</i>	the list
-------------	----------

Returns

the size of the list

Definition at line 298 of file list.c.

5.7.2.14 List newList (void)

Creates a list.

Returns

NULL if an error occurred
the new list otherwise

Definition at line 13 of file list.c.

5.8 list.h File Reference

Implementation of a linked list.

Data Structures

- struct [SListNode](#)
Linked list node structure.
- struct [SList](#)
Linked list structure.

Typedefs

- typedef [SListNode](#) * [ListNode](#)
Linked list node definition.
- typedef [SList](#) * [List](#)
Linked list definition.

Functions

- [List](#) [newList](#) (void)
Creates a list.
- void [listDelete](#) ([List](#) list)
Deletes a list.
- int [listInsertFst](#) ([List](#) list, void *value)
Inserts an element at the beginning of a list.
- int [listInsertLst](#) ([List](#) list, void *value)
Inserts an element at the end of a list.
- int [listInsertAt](#) ([List](#) list, int index, void *value)
Inserts a new element at the specified position of a list.
- int [listRemoveFst](#) ([List](#) list, void **value)
Removes the first element of a list.
- int [listRemoveLst](#) ([List](#) list, void **value)
Removes the last element of a list.
- int [listRemoveAt](#) ([List](#) list, int index, void **value)
Removes the element at the specified position of a list.
- int [listFst](#) ([List](#) list, void **value)
Provides the value at the first position of a list.
- int [listLst](#) ([List](#) list, void **value)
Provides the value at the last position of a list.
- int [listAt](#) ([List](#) list, int index, void **value)
Provides the element at the specified position of a list.
- int [listSize](#) ([List](#) list)
Returns the size of a list.
- int [listMap](#) ([List](#) list, void(*fun)(void *))
Applies a function to the elements of a list.
- Iterator [listIterator](#) ([List](#) list)
Creates an iterator from a list.

5.8.1 Detailed Description

Implementation of a linked list. Provides functions to create and manipulate a linked list.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [list.h](#).

5.8.2 Typedef Documentation

5.8.2.1 typedef SList* List

Linked list definition.

Definition at line 50 of file list.h.

5.8.2.2 typedef SListNode* ListNode

Linked list node definition.

Definition at line 32 of file list.h.

5.8.3 Function Documentation

5.8.3.1 int listAt (List list, int index, void ** value)

Provides the element at the specified position of a list.

If there is no element at the specified position, it will be put the value NULL at value.

Attention

This function puts at value a pointer to the value at the last position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>index</i>	the index of the element to be provided
<i>value</i>	pointer where the value at the specified position will be put

Returns

0 if there was an elements at the specified position
1 otherwise

Definition at line 278 of file list.c.

5.8.3.2 void listDelete (List *list*)

Deletes a list.

Attention

This function only frees the memory used by the list. It does not free the memory used by elements the list contains.

Parameters

<i>list</i>	the list to be deleted
-------------	------------------------

Definition at line 27 of file list.c.

5.8.3.3 int listFst (List *list*, void ** *value*)

Provides the value at the first position of a list.

If the list is empty, it will be put the value NULL at *value*.

Attention

This function puts at *value* a pointer to the value at the first position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>value</i>	pointer where the value at the first position will be put

Returns

0 if the list was not empty
1 otherwise

Definition at line 250 of file list.c.

5.8.3.4 int listInsertAt (List *list*, int *index*, void * *value*)

Inserts an new element at the specified position of a list.

The position, specified by argument *index*, must be a non negative integer, and less than the current size of the list.

Parameters

<i>list</i>	the list
<i>index</i>	the index at which the new value is to be inserted
<i>value</i>	the value to be inserted

Returns

- 0 if the new value was inserted
- 1 if the position was not valid
- 2 if it was not possible to insert the new element

Definition at line 116 of file list.c.

5.8.3.5 int listInsertFst (List *list*, void * *value*)

Inserts an element at the beginning of a list.

Parameters

<i>list</i>	the list
<i>value</i>	the value to be inserted

Returns

- 0 if the new value was inserted
- 1 if it was not possible to insert the new element

Definition at line 42 of file list.c.

5.8.3.6 int listInsertLst (List *list*, void * *value*)

Inserts an element at the end of a list.

Parameters

<i>list</i>	the list
<i>value</i>	the value to be inserted

Returns

- 0 if the new value was inserted
- 1 if it was not possible to insert the new element

Definition at line 79 of file list.c.

5.8.3.7 Iterator listIterator (List *list*)

Creates an iterator from a list.

See Also

Iterator

Parameters

<i>list</i>	the list
-------------	----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 320 of file list.c.

5.8.3.8 int listLst (List *list*, void ** *value*)

Provides the value at the last position of a list.

If the list is empty, it will be put the value NULL at *value*.

Attention

This function puts at *value* a pointer to the value at the last position. Changes to this value will affect the element in the list.

Parameters

<i>list</i>	the list
<i>value</i>	pointer where the value at the last position will be put

Returns

0 if the list was not empty
1 otherwise

Definition at line 264 of file list.c.

5.8.3.9 int listMap (List *list*, void(*)(void *) *fun*)

Applies a function to the elements of a list.

The function to be applied must be of type void fun(void*).

Parameters

<i>list</i>	the list
<i>fun</i>	the function to be applied

Returns

0 if the list was not empty
1 otherwise

Definition at line 305 of file list.c.

5.8.3.10 int listRemoveAt (List *list*, int *index*, void ** *value*)

Removes the element at the specified position of a list.

Provides the value of the removed element if the value of *value* is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>index</i>	the index of the element to be removed
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if the element was removed
1 if the value of `index` was invalid

Definition at line 220 of file list.c.

5.8.3.11 int listRemoveFst (List *list*, void ** *value*)

Removes the first element of a list.

Provides the value of the removed element if the value of `value` is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if the element was removed
1 if the list was empty

Definition at line 158 of file list.c.

5.8.3.12 int listRemoveLst (List *list*, void ** *value*)

Removes the last element of a list.

Provides the value of the removed element if the value of `value` is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>list</i>	the list.
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if the element was removed
1 if the list was empty

Definition at line 189 of file list.c.

5.8.3.13 int listSize (List list)

Returns the size of a list.

Parameters

<i>list</i>	the list
-------------	----------

Returns

the size of the list

Definition at line 298 of file list.c.

5.8.3.14 List newList (void)

Creates a list.

Returns

NULL if an error occurred
the new list otherwise

Definition at line 13 of file list.c.

5.9 queue.c File Reference

Implementation of a queue as linked list.

Functions

- [Queue newListQueue](#) (void)
Creates a queue.
- void [queueDelete](#) (Queue queue)
Deletes a queue.
- int [queueInsert](#) (Queue queue, void *value)
Inserts an element in a queue.
- int [queueRemove](#) (Queue queue, void **value)
Removes an element from a queue.
- int [queueConsult](#) (Queue queue, void **value)
Provides the value at the head of a queue.
- int [queueSize](#) (Queue queue)
Returns the size of a queue.
- int [queueMap](#) (Queue queue, void(*fun)(void *))
Applies a function to the elements of a queue.
- [Iterator queueIterator](#) (Queue queue)
Creates an iterator from a queue.

5.9.1 Detailed Description

Implementation of a queue as linked list.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [queue.c](#).

5.9.2 Function Documentation

5.9.2.1 Queue newQueue (void)

Creates a queue.

Returns

NULL if an error occurred
the new queue otherwise

Definition at line 12 of file queue.c.

5.9.2.2 int queueConsult (Queue *queue*, void ** *value*)

Provides the value at the head of a queue.

If the queue is empty, it will be put the value NULL at *value*.

Attention

This function puts at *value* a pointer to the value at the first position. Changes to this value will affect the element in the list.

Parameters

<i>queue</i>	the queue
<i>value</i>	pointer where the value at the last position will be put

Returns

0 if the queue was not empty
1 otherwise

Definition at line 95 of file queue.c.

5.9.2.3 void queueDelete (Queue *queue*)

Deletes a queue.

Attention

This function only frees the memory used by the queue. It does not free the memory used by elements the queue contains.

Parameters

<i>queue</i>	the queue to be deleted
--------------	-------------------------

Definition at line 26 of file queue.c.

5.9.2.4 int queueInsert (Queue *queue*, void * *value*)

Inserts an element in a queue.

Parameters

<i>queue</i>	the queue
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if it was not possible to insert the new element

Definition at line 44 of file queue.c.

5.9.2.5 Iterator queueIterator (Queue *queue*)

Creates an iterator from a queue.

See Also

Iterator

Parameters

<i>queue</i>	the queue
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 131 of file queue.c.

5.9.2.6 int queueMap (Queue *queue*, void (*)(void *) *fun*)

Applies a function to the elements of a queue.

The function to be applied must be of type void fun(void*).

Parameters

<i>queue</i>	the queue
<i>fun</i>	the function to be applied

Returns

- 0 if the queue was not empty
- 1 otherwise

Definition at line 116 of file queue.c.

5.9.2.7 int queueRemove (Queue *queue*, void ** *value*)

Removes an element from a queue.

Provides the value of the removed element if the value of `value` is not `NULL`.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>queue</i>	the queue
<i>value</i>	pointer where the removed value should be put (or <code>NULL</code>)

Returns

- 0 if an element was removed
- 1 if the queue was empty

Definition at line 64 of file queue.c.

5.9.2.8 int queueSize (Queue *queue*)

Returns the size of a queue.

Parameters

<i>queue</i>	the queue
--------------	-----------

Returns

the size of the queue

Definition at line 109 of file queue.c.

5.10 queue.h File Reference

Implementation of a queue as linked list.

Data Structures

- struct [SQueueNode](#)
Queue node structure.
- struct [SQueue](#)
Queue structure.

Typedefs

- typedef [SQueueNode](#) * [QueueNode](#)
Queue node definition.
- typedef [SQueue](#) * [Queue](#)
Queue definition.

Functions

- [Queue](#) [newQueue](#) (void)
Creates a queue.
- void [queueDelete](#) ([Queue](#) queue)
Deletes a queue.
- int [queueInsert](#) ([Queue](#) queue, void *value)
Inserts an element in a queue.
- int [queueRemove](#) ([Queue](#) queue, void **value)
Removes an element from a queue.
- int [queueConsult](#) ([Queue](#) queue, void **value)
Provides the value at the head of a queue.
- int [queueSize](#) ([Queue](#) queue)
Returns the size of a queue.
- int [queueMap](#) ([Queue](#) queue, void(*fun)(void *))
Applies a function to the elements of a queue.
- [Iterator](#) [queueIterator](#) ([Queue](#) queue)
Creates an iterator from a queue.

5.10.1 Detailed Description

Implementation of a queue as linked list. Provides functions to create and manipulate a queue.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [queue.h](#).

5.10.2 Typedef Documentation

5.10.2.1 typedef SQueue* Queue

Queue definition.

Definition at line 49 of file queue.h.

5.10.2.2 typedef SQueueNode* QueueNode

Queue node definition.

Definition at line 30 of file queue.h.

5.10.3 Function Documentation

5.10.3.1 Queue newQueue (void)

Creates a queue.

Returns

NULL if an error occurred
the new queue otherwise

Definition at line 12 of file queue.c.

5.10.3.2 int queueConsult (Queue *queue*, void ** *value*)

Provides the value at the head of a queue.

If the queue is empty, it will be put the value NULL at *value*.

Attention

This function puts at *value* a pointer to the value at the first position. Changes to this value will affect the element in the list.

Parameters

<i>queue</i>	the queue
<i>value</i>	pointer where the value at the last position will be put

Returns

0 if the queue was not empty
1 otherwise

Definition at line 95 of file queue.c.

5.10.3.3 void queueDelete (Queue *queue*)

Deletes a queue.

Attention

This function only frees the memory used by the queue. It does not free the memory used by elements the queue contains.

Parameters

<i>queue</i>	the queue to be deleted
--------------	-------------------------

Definition at line 26 of file queue.c.

5.10.3.4 int queueInsert (Queue *queue*, void * *value*)

Inserts an element in a queue.

Parameters

<i>queue</i>	the queue
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if it was not possible to insert the new element

Definition at line 44 of file queue.c.

5.10.3.5 Iterator queueIterator (Queue *queue*)

Creates an iterator from a queue.

See Also

Iterator

Parameters

<i>queue</i>	the queue
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 131 of file queue.c.

5.10.3.6 int queueMap (Queue *queue*, void(*)(void *) *fun*)

Applies a function to the elements of a queue.

The function to be applied must be of type void fun(void*).

Parameters

<i>queue</i>	the queue
<i>fun</i>	the function to be applied

Returns

0 if the queue was not empty
1 otherwise

Definition at line 116 of file queue.c.

5.10.3.7 int queueRemove (Queue *queue*, void ** *value*)

Removes an element from a queue.

Provides the value of the removed element if the value of *value* is not NULL.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>queue</i>	the queue
<i>value</i>	pointer where the removed value should be put (or NULL)

Returns

0 if an element was removed
1 if the queue was empty

Definition at line 64 of file queue.c.

5.10.3.8 int queueSize (Queue *queue*)

Returns the size of a queue.

Parameters

<i>queue</i>	the queue
--------------	-----------

Returns

the size of the queue

Definition at line 109 of file queue.c.

5.11 rlp.c File Reference

Implementation of linear programming functions.

Macros

- #define POS(*R*, *C*, *NC*) ((*R*)*(*NC*)+(C))

Given a row (R), a column (C), and the number of columns (NC) of a matrix, computes an equivalent 1D position.

Functions

- static void `fmprint` (double *matrix, int nrows, int ncols, FILE *file)
Prints a matrix associated to an optimization problem.
- static double `minimumc` (double *matrix, int nrows, int ncols, int col, int *row)
Finds the minimum value of a matrix's column.
- static double `minimumr` (double *matrix, int ncols, int row, int *col)
Finds the minimum value of a matrix's row.
- int `simplex` (double *a, int n, int m, FILE *file)
Applies the Simplex Algorithm to an optimization problem.
- int `simplexp` (double *a, int n, int m, int pos, FILE *file)
Applies the Simplex algorithm to a primal optimization problem.
- int `simplexd` (double *a, int n, int m, int pos, FILE *file)
Applies the Simplex algorithm to a dual optimization problem.

5.11.1 Detailed Description

Implementation of linear programming functions.

Author

Rui Carlos Gonçalves

Version

3.0

Date

07/2012

Definition in file `rlp.c`.

5.11.2 Macro Definition Documentation

5.11.2.1 `#define POS(R, C, NC) ((R)*(NC)+(C))`

Given a row (R), a column (C), and the number of columns (NC) of a matrix, computes an equivalent 1D position.

Definition at line 18 of file `rlp.c`.

5.11.3 Function Documentation

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

Prints a matrix associated to an optimization problem.

The file `file`, where the matrix will be printed, must be opened before calling this function.

Parameters

<i>matrix</i>	the matrix to print
<i>nrows</i>	the number of rows
<i>ncols</i>	the number of columns
<i>file</i>	where the tables will be printed

Definition at line 31 of file rlp.c.

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

Finds the minimum value of a matrix's column.

Parameters

<i>matrix</i>	the matrix
<i>nrows</i>	the number of rows
<i>ncols</i>	the number of columns
<i>col</i>	the column index
<i>row</i>	pointer where the row that contains the minimum value will be put

Returns

minimum value of the specified column

Definition at line 72 of file rlp.c.

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

Finds the minimum value of a matrix's row.

Parameters

<i>matrix</i>	the matrix row
<i>ncols</i>	the number of columns
<i>row</i>	the row index
<i>col</i>	pointer where the row that contains the minimum value will be put

Returns

minimum value of the specified row

Definition at line 101 of file rlp.c.

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

Applies the *Simplex Algorithm* to an optimization problem.

Given a problem with n variables (x_1, \dots, x_n) and m conditions ($c_1 = b_1, \dots, c_m = b_m$), the function must receive a matrix a of size $(m+1)*(n+m+2)$, containing:

- at $a[0][i-1]$ (for i in $[1, n]$) the coefficient of variable x_i in the expression to minimize;
- at $a[0][i]$ (for i in $[n, n+m+1]$) the value 0;
- at $a[i][j-1]$ (for i in $[1, m]$, and for j in $[1, n]$) the coefficient of variable x_j in condition c_i ;

- at $a[i][j]$ (for i in $[1, m]$, and for j in $[n, n+m-1]$) the identity matrix;
- at $a[i][n+m]$ (for i in $[1, m]$) the value of b_i ;
- at $a[i][n+m+1]$ (for i in $[1, m]$) the value of $n+i$.

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variable of the function
<i>m</i>	number restrictions
<i>file</i>	file where the tables will be saved (or NULL)

Returns

- 0 if it is possible to solve the problem
- 1 otherwise

Definition at line 118 of file rlp.c.

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

Applies the *Simplex algorithm* to a dual optimization problem.

The input matrix (*a*) must follow the format defined in function [simplex](#).

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variables of the function
<i>m</i>	number restrictions
<i>pos</i>	position of the minimum value of the first row (it must be a negative value)
<i>file</i>	file where the tables will be saved (or NULL)

Returns

- 0 if it is possible to solve the problem
- 1 otherwise

Definition at line 197 of file rlp.c.

5.11.3.6 int simplexp (double * a, int n, int m, int pos, FILE * file)

Applies the *Simplex algorithm* to a primal optimization problem.

The input matrix (*a*) must follow the format defined in function [simplex](#).

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variables of the function
<i>m</i>	number restrictions
<i>pos</i>	position of the minimum value of the restrictions' column (it must be a negative value)
<i>file</i>	file where the tables will be saved (or NULL)

Returns

- 0 if it is possible to solve the problem
- 1 otherwise

Definition at line 136 of file rlp.c.

5.12 rlp.h File Reference

Implementation of linear programming functions.

Functions

- int [simplex](#) (double *a, int n, int m, FILE *file)
Applies the Simplex Algorithm to an optimization problem.
- int [simplexp](#) (double *a, int n, int m, int pos, FILE *file)
Applies the Simplex algorithm to a primal optimization problem.
- int [simplexd](#) (double *a, int n, int m, int pos, FILE *file)
Applies the Simplex algorithm to a dual optimization problem.

5.12.1 Detailed Description

Implementation of linear programming functions.

Author

Rui Carlos Gonçalves

Version

3.0

Date

07/2012

Definition in file [rlp.h](#).

5.12.2 Function Documentation

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

Applies the *Simplex Algorithm* to an optimization problem.

Given a problem with n variables (x_1, \dots, x_n) and m conditions ($c_1 = b_1, \dots, c_m = b_m$), the function must receive a matrix a of size $(m+1)*(n+m+2)$, containing:

- at $a[0][i-1]$ (for i in $[1, n]$) the coefficient of variable x_i in the expression to minimize;
- at $a[0][i]$ (for i in $[n, n+m+1]$) the value 0;
- at $a[i][j-1]$ (for i in $[1, m]$, and for j in $[1, n]$) the coefficient of variable x_j in condition c_i ;
- at $a[i][j]$ (for i in $[1, m]$, and for j in $[n, n+m-1]$) the identity matrix;
- at $a[i][n+m]$ (for i in $[1, m]$) the value of b_i ;
- at $a[i][n+m+1]$ (for i in $[1, m]$) the value of $n+i$.

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variable of the function
<i>m</i>	number restrictions
<i>file</i>	file where the tables will be saved (or NULL)

Returns

- 0 if it is possible to solve the problem
- 1 otherwise

Definition at line 118 of file rlp.c.

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

Applies the *Simplex algorithm* to a dual optimization problem.

The input matrix (a) must follow the format defined in function [simplex](#).

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variables of the function
<i>m</i>	number restrictions
<i>pos</i>	position of the minimum value of the first row (it must be a negative value)
<i>file</i>	file where the tables will be saved (or NULL)

Returns

0 if it is possible to solve the problem
1 otherwise

Definition at line 197 of file rlp.c.

5.12.2.3 int simplexp (double * a, int n, int m, int pos, FILE * file)

Applies the *Simplex algorithm* to a primal optimization problem.

The input matrix (*a*) must follow the format defined in function [simplex](#).

Allows to specify the file where the table resulting from the application of the algorithm (using the parameter *file*).

Parameters

<i>a</i>	matrix that represents the problem
<i>n</i>	number of variables of the function
<i>m</i>	number restrictions
<i>pos</i>	position of the minimum value of the restrictions' column (it must be a negative value)
<i>file</i>	file where the tables will be saved (or NULL)

Returns

0 if it is possible to solve the problem
1 otherwise

Definition at line 136 of file rlp.c.

5.13 rstring.c File Reference

Implementation of functions to manipulate strings.

Functions

- int [trimStart](#) (char *str)
Removes leading whitespaces of a string.
- int [trimEnd](#) (char *str)
Removes trailing whitespaces of a string.
- int [trim](#) (char *str)
Removes leading and trailing whitespaces of a string, as well as consecutive whitespaces in the middle of a string.
- int [charElem](#) (char c, const char *str)
Checks if there is an occurrence of a specific character in a string.
- [List words](#) (const char *str)
Given a string, computes the list of words that the string contains.
- [List strSep](#) (const char *str, const char *delim)
Splits a string.

5.13.1 Detailed Description

Implementation of functions to manipulate strings.

Author

Rui Carlos Gonçalves

Version

3.0

Date

05/2012

Definition in file [rstring.c](#).

5.13.2 Function Documentation

5.13.2.1 `int charElem (char c, const char * str)`

Checks if there is an occurrence of a specific character in a string.

Parameters

<i>c</i>	the character
<i>str</i>	the string

Returns

1 if there is an occurrence of the character in the string
0 otherwise

Definition at line 85 of file `rstring.c`.

5.13.2.2 `List strSep (const char * str, const char * delim)`

Splits a string.

Computes a list of strings, each of which is a substring of the string formed by splitting it on the boundaries defined by the specified delimiters (`delim`).

See Also

List

Parameters

<i>str</i>	the string
<i>delim</i>	the delimiters

Returns

NULL if an error occurred
the list of words resulting from splitting the string at occurrences of the specified delimiters

Definition at line 138 of file rstring.c.

5.13.2.3 int trim (char * *str*)

Removes leading and trailing whitespaces of a string, as well as consecutive whitespaces in the middle of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 54 of file rstring.c.

5.13.2.4 int trimEnd (char * *str*)

Removes trailing whitespaces of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 37 of file rstring.c.

5.13.2.5 int trimStart (char * *str*)

Removes leading whitespaces of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 14 of file rstring.c.

5.13.2.6 List words (const char * *str*)

Given a string, computes the list of words that the string contains.

The original string is not changed.

See Also

List

Parameters

<i>str</i>	the string
------------	------------

Returns

NULL if an error occurred
the list of words otherwise

Definition at line 100 of file rstring.c.

5.14 rstring.h File Reference

Implementation of functions to manipulate strings.

Functions

- int [trimStart](#) (char *str)
Removes leading whitespaces of a string.
- int [trimEnd](#) (char *str)
Removes trailing whitespaces of a string.
- int [trim](#) (char *str)
Removes leading and trailing whitespaces of a string, as well as consecutive whitespaces in the middle of a string.
- int [charElem](#) (char c, const char *str)
Checks if there is an occurrence of a specific character in a string.
- [List words](#) (const char *str)
Given a string, computes the list of words that the string contains.
- [List strSep](#) (const char *str, const char *delim)
Splits a string.

5.14.1 Detailed Description

Implementation of functions to manipulate strings.

Author

Rui Carlos Gonçalves

Version

3.0

Date

05/2012

Definition in file [rstring.h](#).

5.14.2 Function Documentation

5.14.2.1 int charElem (char *c*, const char * *str*)

Checks if there is an occurrence of a specific character in a string.

Parameters

<i>c</i>	the character
<i>str</i>	the string

Returns

1 if there is an occurrence of the character in the string
0 otherwise

Definition at line 85 of file rstring.c.

5.14.2.2 List strSep (const char * *str*, const char * *delim*)

Splits a string.

Computes a list of strings, each of which is a substring of the string formed by splitting it on the boundaries defined by the specified delimiters (*delim*).

See Also

List

Parameters

<i>str</i>	the string
<i>delim</i>	the delimiters

Returns

NULL if an error occurred
the list of words resulting from splitting the string at occurrences of the specified delimiters

Definition at line 138 of file rstring.c.

5.14.2.3 int trim (char * *str*)

Removes leading and trailing whitespaces of a string, as well as consecutive whitespaces in the middle of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 54 of file rstring.c.

5.14.2.4 int trimEnd (char * *str*)

Removes trailing whitespaces of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 37 of file rstring.c.

5.14.2.5 int trimStart (char * *str*)

Removes leading whitespaces of a string.

Parameters

<i>str</i>	the string
------------	------------

Returns

the size of the resulting string

Definition at line 14 of file rstring.c.

5.14.2.6 List words (const char * *str*)

Given a string, computes the list of words that the string contains.

The original string is not changed.

See Also

List

Parameters

<i>str</i>	the string
------------	------------

Returns

NULL if an error occurred
the list of words otherwise

Definition at line 100 of file rstring.c.

5.15 stack.c File Reference

Implementation of a stack as a linked list.

Functions

- [Stack `newStack`](#) (void)
Creates a stack.
- void [stackDelete](#) (Stack stack)
Deletes a stack.
- int [stackPush](#) (Stack stack, void *value)
Inserts an element at the top of the stack.
- int [stackPop](#) (Stack stack, void **value)
Removes an elements from the top of a stack.
- int [stackTop](#) (Stack stack, void **value)
Provides the value of the element at the top of a stack.
- int [stackSize](#) (Stack stack)
Returns the size of a stack.
- int [stackMap](#) (Stack stack, void(*fun)(void *))
Applies a function to the elements of a stack.
- [Iterator `stackIterator`](#) (Stack stack)
Creates an iterator from a stack.

5.15.1 Detailed Description

Implementation of a stack as a linked list.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [stack.c](#).

5.15.2 Function Documentation

5.15.2.1 Stack `newStack` (void)

Creates a stack.

Returns

NULL if an error occurred
the new stack otherwise

Definition at line 13 of file `stack.c`.

5.15.2.2 void stackDelete (Stack *stack*)

Deletes a stack.

Attention

This function only frees the memory used by the stack. It does not free the memory used by elements the stack contains.

Parameters

<i>stack</i>	the stack to be deleted
--------------	-------------------------

Definition at line 26 of file stack.c.

5.15.2.3 Iterator stackIterator (Stack *stack*)

Creates an iterator from a stack.

See Also

Iterator

Parameters

<i>stack</i>	the stack
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 122 of file stack.c.

5.15.2.4 int stackMap (Stack *stack*, void (*)(void *) *fun*)

Applies a function to the elements of a stack.

The function to be applied must be of type void fun(void*).

Parameters

<i>stack</i>	the stack
<i>fun</i>	the function to be applied

Returns

0 if the stack was not empty
1 otherwise

Definition at line 105 of file stack.c.

5.15.2.5 int stackPop (Stack *stack*, void ** *value*)

Removes an elements from the top of a stack.

Provides the value of the removed element if the value of `value` is not `NULL`.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>stack</i>	the stack
<i>value</i>	pointer where the removed value should be put (or <code>NULL</code>)

Returns

0 if the element was removed
1 if the stack was empty

Definition at line 62 of file `stack.c`.

5.15.2.6 `int stackPush (Stack stack, void * value)`

Inserts an element at the top of the stack.

Parameters

<i>stack</i>	the stack
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if it was not possible to insert the new element

Definition at line 44 of file `stack.c`.

5.15.2.7 `int stackSize (Stack stack)`

Returns the size of a stack.

Parameters

<i>stack</i>	the stack
--------------	-----------

Returns

the size of the stack

Definition at line 98 of file `stack.c`.

5.15.2.8 `int stackTop (Stack stack, void ** value)`

Provides the value of the element at the top of a stack.

If the stack is empty, it will be put the value `NULL` at `value`.

Attention

This function puts at `value` a pointer to the value at the top. Changes to this value will affect the element in the stack.

Parameters

<code>stack</code>	the stack.
<code>value</code>	pointer where the value at the first position will be put

Returns

0 if the stack was not empty
1 otherwise

Definition at line 84 of file `stack.c`.

5.16 `stack.h` File Reference

Implementation of a stack as a linked list.

Data Structures

- struct [SStackNode](#)
Stack node structure.
- struct [SStack](#)
Stack structure.

Typedefs

- typedef [SStackNode](#) * [StackNode](#)
Stack node definition.
- typedef [SStack](#) * [Stack](#)
Stack definition.

Functions

- [Stack](#) [newStack](#) (void)
Creates a stack.
- void [stackDelete](#) ([Stack](#) stack)
Deletes a stack.
- int [stackPush](#) ([Stack](#) stack, void *value)
Inserts an element at the top of the stack.
- int [stackPop](#) ([Stack](#) stack, void **value)
Removes an elements from the top of a stack.
- int [stackTop](#) ([Stack](#) stack, void **value)
Provides the value of the element at the top of a stack.
- int [stackSize](#) ([Stack](#) stack)

Returns the size of a stack.

- `int stackMap (Stack stack, void(*fun)(void *))`

Applies a function to the elements of a stack.

- `Iterator stackIterator (Stack stack)`

Creates an iterator from a stack.

5.16.1 Detailed Description

Implementation of a stack as a linked list. Provides functions to create and manipulate stacks.

Author

Rui Carlos Gonçalves

Version

3.0.1

Date

01/2014

Definition in file [stack.h](#).

5.16.2 Typedef Documentation

5.16.2.1 `typedef SStack* Stack`

Stack definition.

Definition at line 46 of file [stack.h](#).

5.16.2.2 `typedef SStackNode* StackNode`

Stack node definition.

Definition at line 30 of file [stack.h](#).

5.16.3 Function Documentation

5.16.3.1 `Stack newStack (void)`

Creates a stack.

Returns

NULL if an error occurred
the new stack otherwise

Definition at line 13 of file [stack.c](#).

5.16.3.2 void stackDelete (Stack *stack*)

Deletes a stack.

Attention

This function only frees the memory used by the stack. It does not free the memory used by elements the stack contains.

Parameters

<i>stack</i>	the stack to be deleted
--------------	-------------------------

Definition at line 26 of file stack.c.

5.16.3.3 Iterator stackIterator (Stack *stack*)

Creates an iterator from a stack.

See Also

Iterator

Parameters

<i>stack</i>	the stack
--------------	-----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 122 of file stack.c.

5.16.3.4 int stackMap (Stack *stack*, void (*)(void *) *fun*)

Applies a function to the elements of a stack.

The function to be applied must be of type void fun(void*).

Parameters

<i>stack</i>	the stack
<i>fun</i>	the function to be applied

Returns

0 if the stack was not empty
1 otherwise

Definition at line 105 of file stack.c.

5.16.3.5 int stackPop (Stack *stack*, void ** *value*)

Removes an elements from the top of a stack.

Provides the value of the removed element if the value of `value` is not `NULL`.

Attention

This function does not free the memory used by the removed element.

Parameters

<i>stack</i>	the stack
<i>value</i>	pointer where the removed value should be put (or <code>NULL</code>)

Returns

0 if the element was removed
1 if the stack was empty

Definition at line 62 of file `stack.c`.

5.16.3.6 int stackPush (Stack *stack*, void * *value*)

Inserts an element at the top of the stack.

Parameters

<i>stack</i>	the stack
<i>value</i>	the value to be inserted

Returns

0 if the new value was inserted
1 if it was not possible to insert the new element

Definition at line 44 of file `stack.c`.

5.16.3.7 int stackSize (Stack *stack*)

Returns the size of a stack.

Parameters

<i>stack</i>	the stack
--------------	-----------

Returns

the size of the stack

Definition at line 98 of file `stack.c`.

5.16.3.8 int stackTop (Stack *stack*, void ** *value*)

Provides the value of the element at the top of a stack.

If the stack is empty, it will be put the value `NULL` at `value`.

Attention

This function puts at `value` a pointer to the value at the top. Changes to this value will affect the element in the stack.

Parameters

<i>stack</i>	the stack.
<i>value</i>	pointer where the value at the first position will be put

Returns

0 if the stack was not empty
1 otherwise

Definition at line 84 of file `stack.c`.

5.17 treemap.c File Reference

Implementation of an AVL tree (self-balancing binary search tree).

Functions

- static `TreeNode leftRotate (TreeNode tree)`
Rotates a tree to left.
- static `TreeNode rightRotate (TreeNode tree)`
Rotates a tree to right.
- static `TreeNode leftBalance (TreeNode tree)`
Rebalances a tree that is balanced to the left as result of an insertion operation.
- static `TreeNode rightBalance (TreeNode tree)`
Rebalances a tree that is balanced to the right as result of an insertion operation.
- static `TreeNode rLeftBalance (TreeNode tree, int *h)`
Rebalances a tree that is balanced to the left as result of a remotion operation.
- static `TreeNode rRightBalance (TreeNode tree, int *h)`
Rebalances a tree that is balanced to the right as result of a remotion operation.
- static `TreeNode upperLeft (TreeNode tree)`
Returns the node with the greatest key on the left subtree of a tree.
- `TreeMap newTree (int(*keyComp)(void *, void *))`
Creates a tree.
- `int treeSetKComp (TreeMap tree, int(*keyComp)(void *, void *))`
Sets the comparison function of this tree.
- static `void treeDelAux (TreeNode tree)`
Deletes a node of a tree and all of its children nodes.
- `void treeDelete (TreeMap tree)`
Deletes a tree.
- static `TreeNode treeInsAux (TreeNode tree, void *key, void *val, int replace, int *h, int(*comp)(void *, void *))`
Insertion auxiliary function.

- int `treeInsert` (`TreeMap` tree, void *key, void *val, int replace)
Associates a value to a key in a tree.
- static `TreeNode` `treeRemAux` (`TreeNode` tree, void *key, void **value, void(*del)(void *), int *h, int(*comp)(void *, void *))
Remotion auxiliary function.
- int `treeRemove` (`TreeMap` tree, void *key, void **value, void(*del)(void *))
Removes the mapping for a key from a tree.
- int `treeGet` (`TreeMap` tree, void *key, void **value)
Provides the mapping for a key from a tree.
- int `treeIsBalancedAux` (`TreeNode` tree)
Checks if a tree is balanced.
- int `treeIsBalanced` (`TreeMap` tree)
Checks if a tree is balanced.
- static int `treeHightAux` (`TreeNode` tree)
Returns the height of a tree.
- int `treeHeight` (`TreeMap` tree)
Returns the height of a tree.
- int `treeSize` (`TreeMap` tree)
Returns the number of elements present in a tree.
- static void `treeInOAux` (`TreeNode` tree, void(*fun)(void *, void *))
Inorder traversal auxiliary function.
- int `treeInOrder` (`TreeMap` tree, void(*fun)(void *, void *))
Applies a function to the elements of an tree (inorder traversal).
- static void `treePreOrderAux` (`TreeNode` tree, void(*fun)(void *, void *))
Preorder traversal auxiliary function.
- int `treePreOrder` (`TreeMap` tree, void(*fun)(void *, void *))
Applies a function to the elements of an tree (preorder traversal).
- static void `treePostOrderAux` (`TreeNode` tree, void(*fun)(void *, void *))
Postorder traversal auxiliary function.
- int `treePostOrder` (`TreeMap` tree, void(*fun)(void *, void *))
Applies a function to the elements of an tree (postorder traversal).
- static int `treeKAux` (`TreeNode` tree, `Iterator` it)
Traverses a tree and adds the keys to an iterator.
- `Iterator` `treeKeys` (`TreeMap` tree)
Creates an iterator from the keys of a tree.
- static int `treeVAux` (`TreeNode` tree, `Iterator` it)
Traverses a tree and adds the values to an iterator.
- `Iterator` `treeValues` (`TreeMap` tree)
Creates an iterator from the values of a tree.

5.17.1 Detailed Description

Implementation of an AVL tree (self-balancing binary search tree).

Author

Rui Carlos Gonçalves

Version

3.0

Date

07/2012

Definition in file [treemap.c](#).**5.17.2 Function Documentation****5.17.2.1 static `TreeNode leftBalance (TreeNode tree)`** [`static`]

Rebalances a tree that is balanced to the left as result of an insertion operation.

Parameters

<i>tree</i>	the root of the tree to rebalance
-------------	-----------------------------------

Returns

the new tree

Definition at line 72 of file `treemap.c`.**5.17.2.2 static `TreeNode leftRotate (TreeNode tree)`** [`static`]

Rotates a tree to left.

Parameters

<i>tree</i>	the root of the tree to rotate
-------------	--------------------------------

Returns

the new tree

Definition at line 20 of file `treemap.c`.**5.17.2.3 `TreeNode newTree (int*(void *, void *) keyComp)`**

Creates a tree.

Parameters

<i>keyComp</i>	the comparison function
----------------	-------------------------

ReturnsNULL if an error occurred
the new tree otherwiseDefinition at line 273 of file `treemap.c`.

5.17.2.4 static TreeNode rightBalance (TreeNode *tree*) [static]

Rebalances a tree that is balanced to the right as result of an insertion operation.

Parameters

<i>tree</i>	the root of the tree to rebalance
-------------	-----------------------------------

Returns

the new tree

Definition at line 121 of file treemap.c.

5.17.2.5 static TreeNode rightRotate (TreeNode *tree*) [static]

Rotates a tree to right.

Parameters

<i>tree</i>	the root of the tree to rotate
-------------	--------------------------------

Returns

the new tree

Definition at line 45 of file treemap.c.

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

Rebalances a tree that is balanced to the left as result of a remotion operation.

Parameters

<i>tree</i>	the root of the tree to rebalance
<i>h</i>	specifies whether the height of the tree changed

Returns

the new tree

Definition at line 171 of file treemap.c.

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

Rebalances a tree that is balanced to the right as result of a remotion operation.

Parameters

<i>tree</i>	the root of the tree to rebalance
<i>h</i>	specifies whether the height of the tree changed

Returns

the new tree

Definition at line 219 of file treemap.c.

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

Deletes a node of a tree and all of its children nodes.

Parameters

<i>tree</i>	the tree
-------------	----------

Definition at line 306 of file treemap.c.

5.17.2.9 void treeDelete (*TreeMap tree*)

Deletes a tree.

Attention

This function only free the memory used by the tree. It does not free the memory used by elements the tree contains.

Parameters

<i>tree</i>	the tree to be deleted
-------------	------------------------

Definition at line 318 of file treemap.c.

5.17.2.10 int treeGet (*TreeMap tree*, void * *key*, void ** *value*)

Provides the mapping for a key from a tree.

If there is no mapping for the specified key, it will be put the value `NULL` at `value`. However, the `NULL` value may also say that the mapping for the specified key was `NULL`. Check the returned value in order to know whether the key was in the tree.

Attention

This function puts at `value` a pointer to the mapping of the specified key. Changes to this value will affect the value in the tree.

Parameters

<i>tree</i>	the tree
<i>key</i>	key whose mapping is to be provided
<i>value</i>	pointer where the mapping value will be put

Returns

0 if there was a mapping for the specified key
1 otherwise

Definition at line 550 of file treemap.c.

5.17.2.11 int treeHeight (TreeMap *tree*)

Returns the height of a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the height of the tree

Definition at line 622 of file treemap.c.

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

Returns the height of a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the height of the tree

Definition at line 608 of file treemap.c.

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

Inorder traversal auxiliary function.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Definition at line 642 of file treemap.c.

5.17.2.14 int treeInOrder (TreeMap *tree*, void(*)(void *, void *) *fun*)

Applies a function to the elements of an tree (inorder traversal).

The function to be applied must be of type void fun(void*, void*).

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 654 of file treemap.c.

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

Insertion auxiliary function.

Parameters

<i>tree</i>	the tree
<i>key</i>	the key
<i>val</i>	the value to be inserted
<i>replace</i>	specifies whether an old value shall be replaced
<i>h</i>	specifies whether the height of the tree changed ($h < 0?$), and whether an error occurred ($h > 0?$)
<i>comp</i>	key comparison function

Returns

new tree

Definition at line 340 of file treemap.c.

5.17.2.16 `int treelnInsert (TreeMap tree, void * key, void * value, int replace)`

Associates a value to a key in a tree.

If the key already had a value, the `replace` argument specifies whether the new element should be added (it will be added only if `replace != 0`).

Parameters

<i>tree</i>	the tree
<i>key</i>	the key
<i>value</i>	the value to be inserted
<i>replace</i>	specifies whether an old value shall be replaced

Returns

0 if the value was inserted
1 if the key already had a value
2 if an error occurred

Definition at line 413 of file treemap.c.

5.17.2.17 `int treelnBalanced (TreeMap tree)`

Checks if a tree is balanced.

A tree is balanced if, for each node of the tree, the difference between the height of the left subtree and the height of right subtree is not greater than 1.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

0 if the tree is not balanced
1 otherwise

Definition at line 593 of file treemap.c.

5.17.2.18 int treelsBalancedAux (*TreeNode tree*)

Checks if a tree is balanced.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

-1 is the tree is not balanced
tree height otherwise

Definition at line 577 of file treemap.c.

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

Traverses a tree and adds the keys to an iterator.

Parameters

<i>tree</i>	the tree
<i>it</i>	the iterator

Returns

1 if an error occurred
0 otherwise

Definition at line 731 of file treemap.c.

5.17.2.20 Iterator treeKeys (*TreeMap tree*)

Creates an iterator from the keys of a tree.

See Also

Iterator

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 745 of file treemap.c.

5.17.2.21 `int treePostOrder (TreeMap tree, void(*)(void *, void *) fun)`

Applies a function to the elements of an tree (postorder traversal).

The function to be applied must be of type `void fun(void*, void*)`.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 711 of file treemap.c.

5.17.2.22 `static void treePostOrderAux (TreeNode tree, void(*)(void *, void *) fun) [static]`

Postorder traversal auxiliary function.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Definition at line 699 of file treemap.c.

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

Applies a function to the elements of an tree (preorder traversal).

The function to be applied must be of type `void fun(void*, void*)`.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 683 of file treemap.c.

5.17.2.24 `static void treePreOrderAux (TreeNode tree, void(*)(void *, void *) fun) [static]`

Preorder traversal auxiliary function.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Definition at line 671 of file treemap.c.

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

Remotion auxiliary function.

Parameters

<i>tree</i>	the tree
<i>key</i>	key whose mapping is to be removed
<i>value</i>	pointer where the removed element shall be put (or NULL)
<i>del</i>	function to free the memory used by the key (or NULL)
<i>h</i>	specifies whether the height of the tree changed
<i>comp</i>	key comparison function

Returns

new tree

Definition at line 438 of file treemap.c.

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

Removes the mapping for a key from a tree.

Provides the value of the removed element if the value of `elem` is not NULL.

Attention

This function does not free the memory used by the removed element. To free the memory used by its key, you have to provide the argument `del`.

Parameters

<i>tree</i>	the tree
<i>key</i>	key whose mapping is to be removed
<i>value</i>	pointer where the removed element shall be put (or NULL)
<i>del</i>	function to free the memory used by the key (or NULL)

Returns

0 if an element was removed from the specified position
1 otherwise

Definition at line 539 of file treemap.c.

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

Sets the comparison function of this tree.

Parameters

<i>tree</i>	the tree
<i>keyComp</i>	the new comparison function

Returns

1 if `keyComp` was equal to `NULL` (no change was made)
0 otherwise

Definition at line 291 of file `treemap.c`.

5.17.2.28 int treeSize (TreeMap *tree*)

Returns the number of elements present in a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the number of elements present in the tree.

Definition at line 629 of file `treemap.c`.

5.17.2.29 Iterator treeValues (TreeMap *tree*)

Creates an iterator from the values of a tree.

See Also

Iterator

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

`NULL` if an error occurred
the iterator otherwise

Definition at line 783 of file `treemap.c`.

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

Traverses a tree and adds the values to an iterator.

Parameters

<i>tree</i>	the tree
<i>it</i>	the iterator

Returns

- 1 if an error occurred
- 0 otherwise

Definition at line 769 of file treemap.c.

5.17.2.31 static TreeNode upperLeft (TreeNode *tree*) [static]

Returns the node with the greatest key on the left subtree of a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the node with the greatest key on the left subtree of the tree

Definition at line 265 of file treemap.c.

5.18 treemap.h File Reference

Implementation of an AVL tree (self-balancing binary search tree).

Data Structures

- struct [STreeNode](#)
Tree node structure.
- struct [STreeMap](#)
Tree structure.

Typedefs

- typedef [STreeNode](#) * [TreeNode](#)
Tree node definition.
- typedef [STreeMap](#) * [TreeMap](#)
Tree definition.

Enumerations

- enum [BFactor](#) { **L**, **E**, **R** }
Type that defines the balance factor of a tree.

Functions

- [TreeMap](#) [newTree](#) (int(*keyComp)(void *, void *))
Creates a tree.
- int [treeSetKComp](#) ([TreeMap](#) tree, int(*keyComp)(void *, void *))

- Sets the comparison function of this tree.*

 - void [treeDelete](#) ([TreeMap](#) tree)

Deletes a tree.
- int [treeInsert](#) ([TreeMap](#) tree, void *key, void *value, int replace)
- Associates a value to a key in a tree.*

 - int [treeRemove](#) ([TreeMap](#) tree, void *key, void **value, void(*del)(void *))

Removes the mapping for a key from a tree.
- int [treeGet](#) ([TreeMap](#) tree, void *key, void **value)
- Provides the mapping for a key from a tree.*

 - int [treeIsBalanced](#) ([TreeMap](#) tree)

Checks if a tree is balanced.
- int [treeHeight](#) ([TreeMap](#) tree)
- Returns the height of a tree.*

 - int [treeSize](#) ([TreeMap](#) tree)

Returns the number of elements present in a tree.
- int [treeInOrder](#) ([TreeMap](#) tree, void(*fun)(void *, void *))
- Applies a function to the elements of an tree (inorder traversal).*

 - int [treePreOrder](#) ([TreeMap](#) tree, void(*fun)(void *, void *))

Applies a function to the elements of an tree (preorder traversal).
- int [treePostOrder](#) ([TreeMap](#) tree, void(*fun)(void *, void *))
- Applies a function to the elements of an tree (postorder traversal).*

 - [Iterator treeKeys](#) ([TreeMap](#) tree)

Creates an iterator from the keys of a tree.
- [Iterator treeValues](#) ([TreeMap](#) tree)
- Creates an iterator from the values of a tree.*

5.18.1 Detailed Description

Implementation of an AVL tree (self-balancing binary search tree). Provides functions to create and manipulate an AVL tree.

To use this tree you have to provide the following function for the date type used for keys:

`int keyComp(void* key1, void* key2)` Compares two keys (used by [treeInsert](#), [treeRemove](#), and [treeGet](#)). It should return 0 if key1 is equal to key2, a value less than 0 if key1 is less than key2, and a value greater than 0 if key1 is greater than 0. This function can be changed using function [treeSetKComp](#).

```
\textcolor{keywordtype}{int} keyComp(\textcolor{keywordtype}{void}* key1, \textcolor{keywordtype}{void}* ke
\{
  \textcolor{keywordflow}{if} (key1&&key2) \textcolor{keywordflow}{return} strcmp((\textcolor{keywordtype}{void})* key1, (\textcolor{keywordtype}{void})* key2);
  \textcolor{keywordflow}{else} \textcolor{keywordflow}{return} 0;
\}
```

Author

Rui Carlos Gonçalves

Version

3.0

Date

07/2012

Definition in file [treemap.h](#).**5.18.2 Typedef Documentation****5.18.2.1 typedef STreeMap* TreeMap**

Tree definition.

Definition at line 80 of file treemap.h.

5.18.2.2 typedef STreeNode* TreeNode

Tree node definition.

Definition at line 62 of file treemap.h.

5.18.3 Enumeration Type Documentation**5.18.3.1 enum BFactor**

Type that defines the balance factor of a tree.

Definition at line 38 of file treemap.h.

5.18.4 Function Documentation**5.18.4.1 TreeMap newTree (int (*)(void *, void *) keyComp)**

Creates a tree.

Parameters

<i>keyComp</i>	the comparison function
----------------	-------------------------

Returns

NULL if an error occurred
the new tree otherwise

Definition at line 273 of file treemap.c.

5.18.4.2 void treeDelete (TreeMap tree)

Deletes a tree.

Attention

This function only free the memory used by the tree. It does not free the memory used by elements the tree contains.

Parameters

<i>tree</i>	the tree to be deleted
-------------	------------------------

Definition at line 318 of file treemap.c.

5.18.4.3 int treeGet (TreeMap *tree*, void * *key*, void ** *value*)

Provides the mapping for a key from a tree.

If there is no mapping for the specified key, it will be put the value `NULL` at `value`. However, the `NULL` value may also say that the mapping for the specified key was `NULL`. Check the returned value in order to know whether the key was in the tree.

Attention

This function puts at `value` a pointer to the mapping of the specified key. Changes to this value will affect the value in the tree.

Parameters

<i>tree</i>	the tree
<i>key</i>	key whose mapping is to be provided
<i>value</i>	pointer where the mapping value will be put

Returns

0 if there was a mapping for the specified key
1 otherwise

Definition at line 550 of file treemap.c.

5.18.4.4 int treeHeight (TreeMap *tree*)

Returns the height of a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the height of the tree

Definition at line 622 of file treemap.c.

5.18.4.5 int treeInOrder (TreeMap *tree*, void (*)(void *, void *) *fun*)

Applies a function to the elements of an tree (inorder traversal).

The function to be applied must be of type `void fun(void*, void*)`.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

- 0 if the array was not empty
- 1 otherwise

Definition at line 654 of file treemap.c.

5.18.4.6 int treemapInsert (TreeMap tree, void * key, void * value, int replace)

Associates a value to a key in a tree.

If the key already had a value, the `replace` argument specifies whether the new element should be added (it will be added only if `replace != 0`).

Parameters

<i>tree</i>	the tree
<i>key</i>	the key
<i>value</i>	the value to be inserted
<i>replace</i>	specifies whether an old value shall be replaced

Returns

- 0 if the value was inserted
- 1 if the key already had a value
- 2 if an error occurred

Definition at line 413 of file treemap.c.

5.18.4.7 int treemapIsBalanced (TreeMap tree)

Checks if a tree is balanced.

A tree is balanced if, for each node of the tree, the difference between the height of the left subtree and the height of right subtree is not greater than 1.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

- 0 if the tree is not balanced
- 1 otherwise

Definition at line 593 of file treemap.c.

5.18.4.8 Iterator treemapKeys (TreeMap tree)

Creates an iterator from the keys of a tree.

See Also

Iterator

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 745 of file treemap.c.

5.18.4.9 int treePostOrder (TreeMap *tree*, void(*)*(void *, void *) fun*)

Applies a function to the elements of an tree (postorder traversal).

The function to be applied must be of type `void fun(void*, void*)`.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 711 of file treemap.c.

5.18.4.10 int treePreOrder (TreeMap *tree*, void(*)*(void *, void *) fun*)

Applies a function to the elements of an tree (preorder traversal).

The function to be applied must be of type `void fun(void*, void*)`.

Parameters

<i>tree</i>	the tree
<i>fun</i>	the function to be applied

Returns

0 if the array was not empty
1 otherwise

Definition at line 683 of file treemap.c.

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

Removes the mapping for a key from a tree.

Provides the value of the removed element if the value of `elem` is not NULL.

Attention

This function does not free the memory used by the removed element. To free the memory used by its key, you have to provide the argument `del`.

Parameters

<i>tree</i>	the tree
<i>key</i>	key whose mapping is to be removed
<i>value</i>	pointer where the removed element shall be put (or <code>NULL</code>)
<i>del</i>	function to free the memory used by the key (or <code>NULL</code>)

Returns

0 if an element was removed from the specified position
1 otherwise

Definition at line 539 of file `treemap.c`.

5.18.4.12 int treeSetKComp (TreeMap tree, int(*) (void *, void *) keyComp)

Sets the comparison function of this tree.

Parameters

<i>tree</i>	the tree
<i>keyComp</i>	the new comparison function

Returns

1 if `keyComp` was equal to `NULL` (no change was made)
0 otherwise

Definition at line 291 of file `treemap.c`.

5.18.4.13 int treeSize (TreeMap tree)

Returns the number of elements present in a tree.

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

the number of elements present in the tree.

Definition at line 629 of file `treemap.c`.

5.18.4.14 Iterator treeValues (TreeMap tree)

Creates an iterator from the values of a tree.

See Also

Iterator

Parameters

<i>tree</i>	the tree
-------------	----------

Returns

NULL if an error occurred
the iterator otherwise

Definition at line 783 of file treemap.c.

5.19 util.c File Reference

Implementation of some utility functions.

Macros

- #define [BUFSIZE](#) 32
Size of the initial buffer to be used in functions `rgets` and `rgetsEOF`.

Functions

- int [rgets](#) (char **str)
Reads a line from the stdin.
- int [rgetsEOF](#) (char **str)
Reads a text from the stdin.
- int [rngets](#) (char *str, int dim)
Reads a line from the stdin.
- int [getRandom](#) (int min, int max)
Generates a random number in a given interval.
- int [merge](#) (void *vals[], int begin, int middle, int end, int(*comp)(void *, void *))
Merges two ordered parts of an array of pointers, based on a given comparison function.
- int [mergeSort](#) (void *vals[], int begin, int end, int(*comp)(void *, void *))
Orders an array into ascending order, based on a given comparison function.

5.19.1 Detailed Description

Implementation of some utility functions.

Author

Rui Carlos Gonçalves

Version

3.1

Date

01/2014

Definition in file [util.c](#).**5.19.2 Macro Definition Documentation****5.19.2.1 #define BUFSIZE 32**Size of the initial buffer to be used in functions `rgets` and `rgetsEOF`.Definition at line 19 of file `util.c`.**5.19.3 Function Documentation****5.19.3.1 int getRandom (int *min*, int *max*)**

Generates a random number in a given interval.

AttentionThe value of `max` must be greater than `min`, otherwise the function will return 0.**Parameters**

<i>min</i>	minimum allowed value
<i>max</i>	maximum allowed value

Returns

0 if $max < min$
 a random number in the given interval otherwise

Definition at line 122 of file `util.c`.**5.19.3.2 int merge (void * *vals*[], int *begin*, int *middle*, int *end*, int(*)*(void *, void *) comp*)**

Merges two ordered parts of an array of pointers, based on a given comparison function.

Parameters

<i>vals</i>	the array
<i>begin</i>	beginning of the first part
<i>middle</i>	beginning of the second part
<i>end</i>	end of the second part
<i>comp</i>	comparison function

Returns

1 if an error occurred
0 otherwise

Definition at line 155 of file util.c.

5.19.3.3 int mergeSort (void * vals[], int begin, int end, int(*)(void *, void *) comp)

Orders an array into ascending order, based on a given comparison function.

Uses the algorithm *merge sort*.

Parameters

<i>vals</i>	array of pointer to the elements to sort
<i>begin</i>	starting position
<i>end</i>	ending position
<i>comp</i>	comparison function

Returns

number of errors detected

Definition at line 188 of file util.c.

5.19.3.4 int rgets (char ** str)

Reads a line from the *stdin*.

Reads all characters until a \n.

Attention

The memory space where the line read will be stored is allocated by this function. Therefore, it shall **not** be previously allocated.

Parameters

<i>str</i>	pointer where a string containing the line read shall be put
------------	--

Returns

-2 if the *str* is invalid
-1 if an error occur
the size of the string read otherwise

Definition at line 21 of file util.c.

5.19.3.5 int rgetsEOF (char ** str)

Reads a text from the *stdin*.

Reads all characters until an *EndOfFile*.

Attention

The memory space where the text read will be stored is allocated by this function. Therefore, it shall **not** be previously allocated.

Parameters

<i>str</i>	pointer where a string containing the text read shall be put
------------	--

Returns

-2 if the `str` is invalid
 -1 if an error occur
 the size of the string read otherwise

Definition at line 61 of file util.c.

5.19.3.6 int rfgets (char * str, int dim)

Reads a line from the *stdin*.

Reads all characters until a `\n`. The line shall not exceed the size *dim-2* (the additional characters will be lost).

Attention

The memory location where the line read will be stored must be previously allocated.

Parameters

<i>str</i>	pointer where the text read shall be put
<i>dim</i>	maximum size for the line to be read (the line shall not exceed the size <i>dim-2</i>)

Returns

-1 if the line exceeded the maximum size allowed
 the size of the line read otherwise

Definition at line 101 of file util.c.

5.20 util.h File Reference

Implementation of some utility functions.

Functions

- [int rgets](#) (char **str)
Reads a line from the stdin.
- [int rgetsEOF](#) (char **str)
Reads a text from the stdin.
- [int rfgets](#) (char *str, int dim)

Reads a line from the stdin.

- int `getRandom` (int min, int max)
Generates a random number in a given interval.
- int `mergeSort` (void *vals[], int begin, int end, int(*comp)(void *, void *))
Orders an array into ascending order, based on a given comparison function.

5.20.1 Detailed Description

Implementation of some utility functions. Provides functions to read text from de *stdin*, to generate random numbers, and to sort arrays.

Author

Rui Carlos Gonçalves

Version

3.1

Date

01/2014

Definition in file [util.h](#).

5.20.2 Function Documentation

5.20.2.1 int getRandom (int min, int max)

Generates a random number in a given interval.

Attention

The value of `max` must be greater than `min`, otherwise the function will return 0.

Parameters

<i>min</i>	minimum allowed value
<i>max</i>	maximum allowed value

Returns

0 if $max < min$
a random number in the given interval otherwise

Definition at line 122 of file util.c.

5.20.2.2 int mergeSort (void * vals[], int begin, int end, int(*)(void *, void *) comp)

Orders an array into ascending order, based on a given comparison function.

Uses the algorithm *merge sort*.

Parameters

<i>vals</i>	array of pointer to the elements to sort
<i>begin</i>	starting position
<i>end</i>	ending position
<i>comp</i>	comparison function

Returns

number of errors detected

Definition at line 188 of file util.c.

5.20.2.3 int rgets (char ** str)

Reads a line from the *stdin*.

Reads all characters until a `\n`.

Attention

The memory space where the line read will be stored is allocated by this function. Therefore, it shall **not** be previously allocated.

Parameters

<i>str</i>	pointer where a string containing the line read shall be put
------------	--

Returns

-2 if the `str` is invalid
-1 if an error occur
the size of the string read otherwise

Definition at line 21 of file util.c.

5.20.2.4 int rgetsEOF (char ** str)

Reads a text from the *stdin*.

Reads all characters until an *EndOfFile*.

Attention

The memory space where the text read will be stored is allocated by this function. Therefore, it shall **not** be previously allocated.

Parameters

<i>str</i>	pointer where a string containing the text read shall be put
------------	--

Returns

-2 if the `str` is invalid
-1 if an error occur
the size of the string read otherwise

Definition at line 61 of file util.c.

5.20.2.5 int rfgets (char * *str*, int *dim*)

Reads a line from the *stdin*.

Reads all characters until a `\n`. The line shall not exceed the size *dim-2* (the additional characters will be lost).

Attention

The memory location where the line read will be stored must be previously allocated.

Parameters

<i>str</i>	pointer where the text read shall be put
<i>dim</i>	maximum size for the line to be read (the line shall not exceed the size <i>dim-2</i>)

Returns

-1 if the line exceeded the maximum size allowed
the size of the line read otherwise

Definition at line 101 of file util.c.

Index

- Array
 - array.h, 19
- array
 - SArray, 4
- array.c, 13
 - arrayAt, 14
 - arrayCapacity, 14
 - arrayDelete, 15
 - arrayInsert, 15
 - arrayIterator, 16
 - arrayMap, 16
 - arrayRemove, 16
 - arrayResize, 17
 - arraySize, 17
 - newArray, 17
- array.h, 18
 - Array, 19
 - arrayAt, 19
 - arrayCapacity, 19
 - arrayDelete, 20
 - arrayInsert, 20
 - arrayIterator, 20
 - arrayMap, 21
 - arrayRemove, 21
 - arrayResize, 21
 - arraySize, 22
 - newArray, 22
- arrayAt
 - array.c, 14
 - array.h, 19
- arrayCapacity
 - array.c, 14
 - array.h, 19
- arrayDelete
 - array.c, 15
 - array.h, 20
- arrayInsert
 - array.c, 15
 - array.h, 20
- arrayIterator
 - array.c, 16
 - array.h, 20
- arrayMap
 - array.c, 16
 - array.h, 21
- arrayRemove
 - array.c, 16
 - array.h, 21
- arrayResize
 - array.c, 17
 - array.h, 21
- arraySize
 - array.c, 17
 - array.h, 22
- BFactor
 - treemap.h, 93
- BUFSIZE
 - util.c, 99
- bf
 - STreeNode, 12
- capacity
 - SArray, 4
 - SIterator, 7
- charElem
 - rstring.c, 68
 - rstring.h, 71
- elems
 - SHashMap, 5
- equals
 - SHashMap, 5
- factor
 - SHashMap, 5
- first
 - SList, 7
- fmprint
 - rlp.c, 62
- getRandom
 - util.c, 99
 - util.h, 102
- hash
 - SHashMap, 5
- hashDelete
 - hashmap.c, 24
 - hashmap.h, 30
- hashGet
 - hashmap.c, 24
 - hashmap.h, 30
- hashInsert
 - hashmap.c, 24
 - hashmap.h, 30
- hashKeys
 - hashmap.c, 25
 - hashmap.h, 31
- HashMap
 - hashmap.h, 29
- HashNode

- hashmap.h, 29
- hashRemove
 - hashmap.c, 25
 - hashmap.h, 31
- hashSetEquals
 - hashmap.c, 25
 - hashmap.h, 32
- hashSetFactor
 - hashmap.c, 26
 - hashmap.h, 32
- hashSetHash
 - hashmap.c, 26
 - hashmap.h, 32
- hashSize
 - hashmap.c, 26
 - hashmap.h, 32
- hashValues
 - hashmap.c, 27
 - hashmap.h, 33
- hashmap.c, 22
 - hashDelete, 24
 - hashGet, 24
 - hashInsert, 24
 - hashKeys, 25
 - hashRemove, 25
 - hashSetEquals, 25
 - hashSetFactor, 26
 - hashSetHash, 26
 - hashSize, 26
 - hashValues, 27
 - newHash, 27
 - reHash, 27
- hashmap.h, 28
 - hashDelete, 30
 - hashGet, 30
 - hashInsert, 30
 - hashKeys, 31
 - HashMap, 29
 - HashNode, 29
 - hashRemove, 31
 - hashSetEquals, 32
 - hashSetFactor, 32
 - hashSetHash, 32
 - hashSize, 32
 - hashValues, 33
 - newHash, 33
- head
 - SQueue, 9
- itAdd
 - iterator.c, 35
 - iterator.h, 39
- itAt
 - iterator.c, 35
- itDelete
 - iterator.h, 39
- itDelete
 - iterator.c, 35
 - iterator.h, 39
- itGetPos
 - iterator.c, 35
 - iterator.h, 40
- itHasNext
 - iterator.c, 36
 - iterator.h, 40
- itHasPrev
 - iterator.c, 36
 - iterator.h, 40
- itNext
 - iterator.c, 36
 - iterator.h, 40
- itPrev
 - iterator.c, 36
 - iterator.h, 41
- itSetPos
 - iterator.c, 37
 - iterator.h, 41
- Iterator
 - iterator.h, 39
- iterator.c, 34
 - itAdd, 35
 - itAt, 35
 - itDelete, 35
 - itGetPos, 35
 - itHasNext, 36
 - itHasPrev, 36
 - itNext, 36
 - itPrev, 36
 - itSetPos, 37
 - newIt, 37
- iterator.h, 37
 - itAdd, 39
 - itAt, 39
 - itDelete, 39
 - itGetPos, 40
 - itHasNext, 40
 - itHasPrev, 40
 - itNext, 40
 - itPrev, 41
 - itSetPos, 41
 - Iterator, 39
 - newIt, 41
- key
 - SHashNode, 6
 - STreeNode, 12
- keyComp
 - STreeMap, 12

- last
 - SList, 8
 - SQueue, 9
- left
 - STreeNode, 13
- leftBalance
 - treemap.c, 82
- leftRotate
 - treemap.c, 82
- length
 - SHashMap, 5
- List
 - list.h, 49
- list.c, 42
 - listAt, 43
 - listDelete, 43
 - listFst, 43
 - listInsertAt, 44
 - listInsertFst, 44
 - listInsertLst, 44
 - listIterator, 45
 - listLst, 45
 - listMap, 45
 - listRemoveAt, 46
 - listRemoveFst, 46
 - listRemoveLst, 47
 - listSize, 47
 - newList, 47
- list.h, 48
 - List, 49
 - listAt, 49
 - listDelete, 50
 - listFst, 50
 - listInsertAt, 50
 - listInsertFst, 51
 - listInsertLst, 51
 - listIterator, 51
 - listLst, 52
 - listMap, 52
 - ListNode, 49
 - listRemoveAt, 52
 - listRemoveFst, 53
 - listRemoveLst, 53
 - listSize, 54
 - newList, 54
- listAt
 - list.c, 43
 - list.h, 49
- listDelete
 - list.c, 43
 - list.h, 50
- listFst
 - list.c, 43
 - list.h, 50
- listInsertAt
 - list.c, 44
 - list.h, 50
- listInsertFst
 - list.c, 44
 - list.h, 51
- listInsertLst
 - list.c, 44
 - list.h, 51
- listIterator
 - list.c, 45
 - list.h, 51
- listLst
 - list.c, 45
 - list.h, 52
- listMap
 - list.c, 45
 - list.h, 52
- ListNode
 - list.h, 49
- listRemoveAt
 - list.c, 46
 - list.h, 52
- listRemoveFst
 - list.c, 46
 - list.h, 53
- listRemoveLst
 - list.c, 47
 - list.h, 53
- listSize
 - list.c, 47
 - list.h, 54
- merge
 - util.c, 99
- mergeSort
 - util.c, 100
 - util.h, 102
- minimumc
 - rlp.c, 63
- minimumr
 - rlp.c, 63
- newArray
 - array.c, 17
 - array.h, 22
- newHash
 - hashmap.c, 27
 - hashmap.h, 33
- newIt
 - iterator.c, 37
 - iterator.h, 41
- newList
 - list.c, 47

- list.h, 54
- newQueue
 - queue.c, 55
 - queue.h, 59
- newStack
 - stack.c, 73
 - stack.h, 77
- newTree
 - treemap.c, 82
 - treemap.h, 93
- next
 - SHashNode, 6
 - SListNode, 8
 - SQueueNode, 10
 - SStackNode, 11
- POS
 - rlp.c, 62
- pos
 - SIterator, 7
- prev
 - SListNode, 8
- Queue
 - queue.h, 59
- queue.c, 54
 - newQueue, 55
 - queueConsult, 55
 - queueDelete, 55
 - queueInsert, 56
 - queueIterator, 56
 - queueMap, 56
 - queueRemove, 57
 - queueSize, 57
- queue.h, 57
 - newQueue, 59
 - Queue, 59
 - queueConsult, 59
 - queueDelete, 59
 - queueInsert, 60
 - queueIterator, 60
 - queueMap, 60
 - QueueNode, 59
 - queueRemove, 61
 - queueSize, 61
- queueConsult
 - queue.c, 55
 - queue.h, 59
- queueDelete
 - queue.c, 55
 - queue.h, 59
- queueInsert
 - queue.c, 56
 - queue.h, 60
- queueIterator
 - queue.c, 56
 - queue.h, 60
- queueMap
 - queue.c, 56
 - queue.h, 60
- QueueNode
 - queue.h, 59
- queueRemove
 - queue.c, 57
 - queue.h, 61
- queueSize
 - queue.c, 57
 - queue.h, 61
- rLeftBalance
 - treemap.c, 83
- rRightBalance
 - treemap.c, 83
- reHash
 - hashmap.c, 27
- rgets
 - util.c, 100
 - util.h, 103
- rgetsEOF
 - util.c, 100
 - util.h, 103
- right
 - STreeNode, 13
- rightBalance
 - treemap.c, 82
- rightRotate
 - treemap.c, 83
- rlp.c, 61
 - fmprint, 62
 - minimumc, 63
 - minimumr, 63
 - POS, 62
 - simplex, 63
 - simplexd, 64
 - simplexp, 64
- rlp.h, 65
 - simplex, 66
 - simplexd, 66
 - simplexp, 67
- rngets
 - util.c, 101
 - util.h, 104
- root
 - STreeMap, 12
- rstring.c, 67
 - charElem, 68
 - strSep, 68
 - trim, 69

- trimEnd, 69
- trimStart, 69
- words, 69
- rstring.h, 70
 - charElem, 71
 - strSep, 71
 - trim, 71
 - trimEnd, 71
 - trimStart, 72
 - words, 72
- SArray, 4
 - array, 4
 - capacity, 4
 - size, 4
- SHashMap, 4
 - elems, 5
 - equals, 5
 - factor, 5
 - hash, 5
 - length, 5
 - size, 5
- SHashNode, 5
 - key, 6
 - next, 6
 - value, 6
- SIterator, 6
 - capacity, 7
 - pos, 7
 - size, 7
 - values, 7
- SList, 7
 - first, 7
 - last, 8
 - size, 8
- SListNode, 8
 - next, 8
 - prev, 8
 - value, 8
- SQueue, 9
 - head, 9
 - last, 9
 - size, 9
- SQueueNode, 9
 - next, 10
 - value, 10
- SStack, 10
 - size, 10
 - top, 10
- SStackNode, 11
 - next, 11
 - value, 11
- STreeMap, 11
 - keyComp, 12
 - root, 12
 - size, 12
- STreeNode, 12
 - bf, 12
 - key, 12
 - left, 13
 - right, 13
 - super, 13
 - value, 13
- simplex
 - rlp.c, 63
 - rlp.h, 66
- simplexd
 - rlp.c, 64
 - rlp.h, 66
- simplexp
 - rlp.c, 64
 - rlp.h, 67
- size
 - SArray, 4
 - SHashMap, 5
 - SIterator, 7
 - SList, 8
 - SQueue, 9
 - SStack, 10
 - STreeMap, 12
- Stack
 - stack.h, 77
- stack.c, 72
 - newStack, 73
 - stackDelete, 73
 - stackIterator, 74
 - stackMap, 74
 - stackPop, 74
 - stackPush, 75
 - stackSize, 75
 - stackTop, 75
- stack.h, 76
 - newStack, 77
 - Stack, 77
 - stackDelete, 77
 - stackIterator, 78
 - stackMap, 78
 - StackNode, 77
 - stackPop, 78
 - stackPush, 79
 - stackSize, 79
 - stackTop, 79
- stackDelete
 - stack.c, 73
 - stack.h, 77
- stackIterator
 - stack.c, 74
 - stack.h, 78

- stackMap
 - stack.c, 74
 - stack.h, 78
- StackNode
 - stack.h, 77
- stackPop
 - stack.c, 74
 - stack.h, 78
- stackPush
 - stack.c, 75
 - stack.h, 79
- stackSize
 - stack.c, 75
 - stack.h, 79
- stackTop
 - stack.c, 75
 - stack.h, 79
- strSep
 - rstring.c, 68
 - rstring.h, 71
- super
 - STreeNode, 13
- top
 - SStack, 10
- treeDelAux
 - treemap.c, 84
- treeDelete
 - treemap.c, 84
 - treemap.h, 93
- treeGet
 - treemap.c, 84
 - treemap.h, 94
- treeHeight
 - treemap.c, 84
 - treemap.h, 94
- treeHightAux
 - treemap.c, 85
- treeInOAux
 - treemap.c, 85
- treeInOrder
 - treemap.c, 85
 - treemap.h, 94
- treeInsAux
 - treemap.c, 86
- treeInsert
 - treemap.c, 86
 - treemap.h, 95
- treeIsBalanced
 - treemap.c, 86
 - treemap.h, 95
- treeIsBalancedAux
 - treemap.c, 87
- treeKAux
 - treemap.c, 87
- treeKeys
 - treemap.c, 87
- treeMap
 - treemap.c, 87
 - treemap.h, 95
- TreeMap
 - treemap.h, 93
- TreeNode
 - treemap.h, 93
- treePostOrder
 - treemap.c, 88
 - treemap.h, 96
- treePostOrderAux
 - treemap.c, 88
- treePreOrder
 - treemap.c, 88
 - treemap.h, 96
- treePreOrderAux
 - treemap.c, 88
- treeRemAux
 - treemap.c, 89
- treeRemove
 - treemap.c, 89
 - treemap.h, 96
- treeSetKComp
 - treemap.c, 89
 - treemap.h, 97
- treeSize
 - treemap.c, 90
 - treemap.h, 97
- treeVAux
 - treemap.c, 90
- treeValues
 - treemap.c, 90
 - treemap.h, 97
- treemap.c, 80
 - leftBalance, 82
 - leftRotate, 82
 - newTree, 82
 - rLeftBalance, 83
 - rRightBalance, 83
 - rightBalance, 82
 - rightRotate, 83
 - treeDelAux, 84
 - treeDelete, 84
 - treeGet, 84
 - treeHeight, 84
 - treeHightAux, 85
 - treeInOAux, 85
 - treeInOrder, 85
 - treeInsAux, 86
 - treeInsert, 86
 - treeIsBalanced, 86
 - treeIsBalancedAux, 87
 - treeKAux, 87

- treeKeys, 87
- treePostOrder, 88
- treePostOrderAux, 88
- treePreOrder, 88
- treePreOrderAux, 88
- treeRemAux, 89
- treeRemove, 89
- treeSetKComp, 89
- treeSize, 90
- treeVAux, 90
- treeValues, 90
- upperLeft, 91
- treemap.h, 91
 - BFactor, 93
 - newTree, 93
 - treeDelete, 93
 - treeGet, 94
 - treeHeight, 94
 - treeInOrder, 94
 - treeInsert, 95
 - treeIsBalanced, 95
 - treeKeys, 95
 - TreeMap, 93
 - TreeNode, 93
 - treePostOrder, 96
 - treePreOrder, 96
 - treeRemove, 96
 - treeSetKComp, 97
 - treeSize, 97
 - treeValues, 97
- trim
 - rstring.c, 69
 - rstring.h, 71
- trimEnd
 - rstring.c, 69
 - rstring.h, 71
- trimStart
 - rstring.c, 69
 - rstring.h, 72
- upperLeft
 - treemap.c, 91
- util.c, 98
 - BUFSIZE, 99
 - getRandom, 99
 - merge, 99
 - mergeSort, 100
 - rgets, 100
 - rgetsEOF, 100
 - rngets, 101
- util.h, 101
 - getRandom, 102
 - mergeSort, 102
 - rgets, 103
 - rgetsEOF, 103
 - rngets, 104
- value
 - SHashNode, 6
 - SListNode, 8
 - SQueueNode, 10
 - SStackNode, 11
 - STreeNode, 13
- values
 - SIterator, 7
- words
 - rstring.c, 69
 - rstring.h, 72