LibRCG

Version 3.1.1

Rui Carlos Gonçalves
rcgoncalves.pt@gmail.com

August 2, 2015

Contents

1 About

LibRCG is a C library, developed by Rui Carlos Gonçalves (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

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:

3 File Index

3.1 File List

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

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

array	the array
index	the index of the element to be provided
elem	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

array	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

array	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 an 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

array	the array
index	the index at which the new element is to be inserted
elem	the element to be inserted
replace	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

array	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

array	the array
fun	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

array	the array
index	the index of the element to be removed
elem	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

array	the array
size	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

array	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

size	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 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.

5.2.1 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

array	the array
index	the index of the element to be provided
elem	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

array	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

array	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 an 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

array	the array
index	the index at which the new element is to be inserted
elem	the element to be inserted
replace	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

array	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

array	the array
fun	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

array	the array
index	the index of the element to be removed
elem	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

array	the array
size	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

array	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

size	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

hmap	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

hmap	the hash table
key	key whose mapping is to be provided
value	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

hmap	the hash table
key	the key
value	the value to be inserted
replace	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

hmap	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

hmap	the hash table
key	key whose mapping is to be removed
value	pointer where the removed element shall be put (or `NULL`)
del	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

hmap	the hash table
equals	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

hmap	the hash table
factor	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

hmap	the hash table
hash	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

hmap	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

hmap	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

size	the initial number of buckets
factor	the load factor
hash	the hash function
equals	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

hmap	the hash table

Returns

1 if an error occured
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}{'\(\backslash\)0'};x+=au
  \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}{vo
\{
  \textcolor{keywordflow}{if}(key1&&key2) \textcolor{keywordflow}{return} srtcmp((\textcolor{keywordt
  \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

hmap	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

hmap	the hash table
key	key whose mapping is to be provided
value	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

hmap	the hash table
key	the key
value	the value to be inserted
replace	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

hmap	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

hmap	the hash table
key	key whose mapping is to be removed
value	pointer where the removed element shall be put (or `NULL`)
del	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

hmap	the hash table
equals	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

hmap	the hash table
factor	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

hmap	the hash table
hash	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

hmap	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

hmap	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

size	the initial number of buckets
factor	the load factor
hash	the hash function
equals	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

it	the iterator
val	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

it	the iterator
index	the position
elem	pointer were 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

it	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

it	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

it	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

it	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

it	the iterator
val	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.

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

Provides the previous element of an iterator.

Parameters

it	the iterator
val	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 og `n` is valid).

Parameters

it	the iterator
n	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

size	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

it	the iterator
val	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 iterador.

Parameters

it	the iterator
index	the position
elem	pointer were 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

it	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

it	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

it	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

it	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

it	the iterator
val	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

it	the iterator
val	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.

5.6.3.9 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 og `n` is valid).

Parameters

it	the iterator
n	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

size	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 an 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

list	the list
index	the index of the element to be provided
value	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

list	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

list	the list
value	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

list	the list
index	the index at which the new value is to be inserted
value	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

list	the list
value	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

list	the list
value	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

list	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

list	the list
value	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

list	the list
fun	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

list	the list.
index	the index of the element to be removed
value	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

list	the list.
value	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

list	the list.
value	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

list	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 an 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

list	the list
index	the index of the element to be provided
value	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

list	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

list	the list
value	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

list	the list
index	the index at which the new value is to be inserted
value	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

list	the list
value	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

list	the list
value	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

list	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

list	the list
value	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

list	the list
fun	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

list	the list.
index	the index of the element to be removed
value	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

list	the list.
value	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

list	the list.
value	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

list	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 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.

5.9.1 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

queue	the queue
value	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

queue	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

queue	the queue
value	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

queue	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

queue	the queue
fun	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

queue	the queue
value	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.

5.9.2.8 int queueSize (Queue *queue*)

Returns the size of a queue.

Parameters

queue	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

queue	the queue
value	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

queue	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

queue	the queue
value	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

queue	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

queue	the queue
fun	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

queue	the queue
value	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

queue	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

matrix	the matrix to print
nrows	the number of rows
ncols	the number of columns
file	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

matrix	the matrix
nrows	the number of rows
ncols	the number of columns
col	the column index
row	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

matrix	the matrix row
ncols	the number of columns
row	the row index
col	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 , ..., x_n) and m conditions ($c_1 = b_1$, ..., $c_m = b_m$), 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

a	matrix that represents the problem
n	number of variable of the function
m	number restrictions
file	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

a	matrix that represents the problem
n	number of variables of the function
m	number restrictions
pos	position of the minimum value of the first row (it must be a negative value)
file	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

a	matrix that represents the problem
n	number of variables of the function
m	number restrictions
pos	position of the minimum value of the restrictions' column (it must be a negative value)
file	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 , ..., x_n) and m conditions ($c_1 = b_1$, ..., $c_m = b_m$), 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

a	matrix that represents the problem
n	number of variable of the function
m	number restrictions
file	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

a	matrix that represents the problem
n	number of variables of the function
m	number restrictions
pos	position of the minimum value of the first row (it must be a negative value)
file	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

a	matrix that represents the problem
n	number of variables of the function
m	number restrictions
pos	position of the minimum value of the restrictions' column (it must be a negative value)
file	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

c	the character
str	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

str	the string
delim	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

str	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

str	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

str	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

str	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

c	the character
str	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

str	the string
delim	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

str	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

str	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

str	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

str	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

stack	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

stack	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

stack	the stack
fun	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

stack	the stack
value	pointer where the removed value should be put (or `NULL`)

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

stack	the stack
value	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

stack	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

stack	the stack.
value	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

stack	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

stack	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

stack	the stack
fun	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

stack	the stack
value	pointer where the removed value should be put (or `NULL`)

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

stack	the stack
value	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

stack	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

stack	the stack.
value	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

tree	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

tree	the root of the tree to rotate

Returns

the new tree

Definition at line 20 of file treemap.c.

5.17.2.3 TreeMap newTree (int(*)(void *, void *) *keyComp*)

Creates a tree.

Parameters

keyComp	the comparison function

Returns

NULL if an error occurred
the new tree otherwise

Definition 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

tree	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

tree	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

tree	the root of the tree to rebalance
h	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

tree	the root of the tree to rebalance
h	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

tree	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

tree	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

tree	the tree
key	key whose mapping is to be provided
value	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

tree	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

tree	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

tree	the tree
fun	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

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

Insertion auxiliary function.

Parameters

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

Returns

new tree

Definition at line 340 of file treemap.c.

5.17.2.16 int treeInsert (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

tree	the tree
key	the key
value	the value to be inserted
replace	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 treeIsBalanced (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

tree	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 treeIsBalancedAux (TreeNode *tree*)

Checks if a tree is balanced.

Parameters

tree	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

tree	the tree
it	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

tree	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

tree	the tree
fun	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

tree	the tree
fun	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

tree	the tree
fun	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

tree	the tree
fun	the function to be applied

Definition at line 671 of file treemap.c.

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

Remotion auxiliary function.

Parameters

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

Returns

new tree

Definition at line 438 of file treemap.c.

5.17.2.26 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

tree	the tree
key	key whose mapping is to be removed
value	pointer where the removed element shall be put (or `NULL`)
del	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.

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

Sets the comparison function of this tree.

Parameters

tree	the tree
keyComp	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

tree	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

tree	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

tree	the tree
it	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

tree	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}{
  \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

keyComp	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

tree	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

tree	the tree
key	key whose mapping is to be provided
value	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

tree	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

tree	the tree
fun	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 treeInsert (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

tree	the tree
key	the key
value	the value to be inserted
replace	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 treeIsBalanced (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

tree	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 treeKeys (TreeMap *tree*)

Creates an iterator from the keys of a tree.

See Also

Iterator

Parameters

tree	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

tree	the tree
fun	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

tree	the tree
fun	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

tree	the tree
key	key whose mapping is to be removed
value	pointer where the removed element shall be put (or `NULL`)
del	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.

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

Sets the comparison function of this tree.

Parameters

tree	the tree
keyComp	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

tree	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

tree	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.

Attention

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

Parameters

min	minimum allowed value
max	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

vals	the array
begin	beginning of the first part
middle	beginning of the second part
end	end of the second part
comp	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

vals	array of pointer to the elements to sort
begin	starting position
end	ending position
comp	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

str	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

str	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 rngets (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

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

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 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 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

min	minimum allowed value
max	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

vals	array of pointer to the elements to sort
begin	starting position
end	ending position
comp	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

str	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

str	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 rngets (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

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

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