

cdatastruct

Generated by Doxygen 1.8.1.2

Sat Sep 7 2013 00:05:38

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	3
2.1	File List	3
3	Data Structure Documentation	5
3.1	bs_tree_node_t Struct Reference	5
3.1.1	Field Documentation	5
3.1.1.1	data	5
3.1.1.2	left	5
3.1.1.3	right	5
3.2	bs_tree_t Struct Reference	6
3.2.1	Field Documentation	6
3.2.1.1	cfunc	6
3.2.1.2	free_func	6
3.2.1.3	length	6
3.2.1.4	root	6
3.3	dl_list_node_t Struct Reference	7
3.3.1	Field Documentation	7
3.3.1.1	data	7
3.3.1.2	next	7
3.3.1.3	prev	7
3.4	dl_list_t Struct Reference	7
3.4.1	Field Documentation	8
3.4.1.1	back	8
3.4.1.2	cfunc	8
3.4.1.3	free_func	8
3.4.1.4	front	8
3.4.1.5	length	8
3.5	kvpair_t Struct Reference	9
3.5.1	Field Documentation	9

3.5.1.1	key	9
3.5.1.2	value	9
3.6	sl_list_node_t Struct Reference	9
3.6.1	Field Documentation	9
3.6.1.1	data	9
3.6.1.2	next	10
3.7	sl_list_t Struct Reference	10
3.7.1	Field Documentation	10
3.7.1.1	cfunc	10
3.7.1.2	free_func	10
3.7.1.3	front	10
3.7.1.4	length	11
4	File Documentation	13
4.1	bs_tree.c File Reference	13
4.1.1	Detailed Description	14
4.1.2	Function Documentation	14
4.1.2.1	bs_tree_free	14
4.1.2.2	bs_tree_free_subtree	14
4.1.2.3	bs_tree_init	14
4.1.2.4	bs_tree_insert	15
4.1.2.5	bs_tree_insert_subtree	15
4.1.2.6	bs_tree_isempty	15
4.1.2.7	bs_tree_length	16
4.1.2.8	bs_tree_new_node	16
4.1.2.9	bs_tree_search	16
4.1.2.10	bs_tree_search_data	16
4.1.2.11	bs_tree_search_node	16
4.1.2.12	bst_insert_search	17
4.2	bs_tree.h File Reference	17
4.2.1	Detailed Description	18
4.2.2	Function Documentation	19
4.2.2.1	bs_tree_free_subtree	19
4.2.2.2	bs_tree_insert_subtree	19
4.2.2.3	bs_tree_new_node	19
4.2.2.4	bs_tree_search_node	19
4.2.2.5	bst_insert_search	19
4.3	bst_map.c File Reference	20
4.3.1	Detailed Description	21
4.3.2	Function Documentation	21

4.3.2.1	bst_map_free	21
4.3.2.2	bst_map_init	21
4.3.2.3	bst_map_insert	21
4.3.2.4	bst_map_isempty	22
4.3.2.5	bst_map_length	22
4.3.2.6	bst_map_search	22
4.3.2.7	bst_map_search_data	22
4.4	cdatastruct.h File Reference	22
4.4.1	Detailed Description	23
4.5	cds_bs_tree.h File Reference	23
4.5.1	Detailed Description	25
4.5.2	Function Documentation	25
4.5.2.1	bs_tree_free	25
4.5.2.2	bs_tree_init	25
4.5.2.3	bs_tree_insert	25
4.5.2.4	bs_tree_isempty	25
4.5.2.5	bs_tree_length	26
4.5.2.6	bs_tree_search	26
4.5.2.7	bs_tree_search_data	26
4.6	cds_bst_map.h File Reference	26
4.6.1	Detailed Description	28
4.6.2	Function Documentation	28
4.6.2.1	bst_map_free	28
4.6.2.2	bst_map_init	28
4.6.2.3	bst_map_insert	28
4.6.2.4	bst_map_isempty	28
4.6.2.5	bst_map_length	29
4.6.2.6	bst_map_search	29
4.6.2.7	bst_map_search_data	29
4.7	cds_common.h File Reference	29
4.7.1	Detailed Description	30
4.7.2	Enumeration Type Documentation	30
4.7.2.1	cds_error	30
4.8	cds_dl_list.h File Reference	30
4.8.1	Detailed Description	32
4.8.2	Function Documentation	32
4.8.2.1	dl_list_append	32
4.8.2.2	dl_list_data	32
4.8.2.3	dl_list_delete_at	33
4.8.2.4	dl_list_find_index	33

4.8.2.5	dl_list_find_itr	33
4.8.2.6	dl_list_first	33
4.8.2.7	dl_list_free	33
4.8.2.8	dl_list_init	34
4.8.2.9	dl_list_insert_after	34
4.8.2.10	dl_list_insert_at	34
4.8.2.11	dl_list_insert_before	34
4.8.2.12	dl_list_isempty	35
4.8.2.13	dl_list_itr_from_index	35
4.8.2.14	dl_list_last	35
4.8.2.15	dl_list_length	35
4.8.2.16	dl_list_next	35
4.8.2.17	dl_list_prepend	36
4.8.2.18	dl_list_prev	36
4.9	cds_general.h File Reference	36
4.9.1	Detailed Description	37
4.9.2	Function Documentation	37
4.9.2.1	cds_compare_int	37
4.9.2.2	cds_compare_long	37
4.9.2.3	cds_compare_string	38
4.9.2.4	cds_compare_uint	38
4.9.2.5	cds_compare_ulong	38
4.9.2.6	cds_new_int	38
4.9.2.7	cds_new_long	38
4.9.2.8	cds_new_string	39
4.9.2.9	cds_new_uint	39
4.9.2.10	cds_new_ulong	39
4.10	cds_queue.h File Reference	39
4.10.1	Detailed Description	41
4.10.2	Function Documentation	41
4.10.2.1	queue_free	41
4.10.2.2	queue_init	41
4.10.2.3	queue_isempty	41
4.10.2.4	queue_length	41
4.10.2.5	queue_pop	41
4.10.2.6	queue_pushback	42
4.11	cds_sl_list.h File Reference	42
4.11.1	Detailed Description	44
4.11.2	Function Documentation	44
4.11.2.1	sl_list_data	44

4.11.2.2	sl_list_delete_at	44
4.11.2.3	sl_list_find_index	44
4.11.2.4	sl_list_find_itr	45
4.11.2.5	sl_list_first	45
4.11.2.6	sl_list_free	45
4.11.2.7	sl_list_init	45
4.11.2.8	sl_list_insert_after	45
4.11.2.9	sl_list_insert_at	46
4.11.2.10	sl_list_isempty	46
4.11.2.11	sl_list_itr_from_index	46
4.11.2.12	sl_list_length	46
4.11.2.13	sl_list_next	47
4.11.2.14	sl_list_prepend	47
4.12	cds_stack.h File Reference	47
4.12.1	Detailed Description	48
4.12.2	Function Documentation	48
4.12.2.1	stack_free	48
4.12.2.2	stack_init	49
4.12.2.3	stack_isempty	49
4.12.2.4	stack_length	49
4.12.2.5	stack_pop	49
4.12.2.6	stack_push	49
4.13	dl_list.c File Reference	50
4.13.1	Detailed Description	51
4.13.2	Function Documentation	52
4.13.2.1	dl_list_append	52
4.13.2.2	dl_list_data	52
4.13.2.3	dl_list_delete_at	52
4.13.2.4	dl_list_find	52
4.13.2.5	dl_list_find_index	52
4.13.2.6	dl_list_find_itr	53
4.13.2.7	dl_list_first	53
4.13.2.8	dl_list_free	53
4.13.2.9	dl_list_free_node	53
4.13.2.10	dl_list_init	53
4.13.2.11	dl_list_insert_after	53
4.13.2.12	dl_list_insert_at	54
4.13.2.13	dl_list_insert_before	54
4.13.2.14	dl_list_insert_node_after_mid	54
4.13.2.15	dl_list_insert_node_back	55

4.13.2.16	dl_list_insert_node_before_mid	55
4.13.2.17	dl_list_insert_node_front	55
4.13.2.18	dl_list_isempty	55
4.13.2.19	dl_list_itr_from_index	55
4.13.2.20	dl_list_last	55
4.13.2.21	dl_list_length	56
4.13.2.22	dl_list_new_node	56
4.13.2.23	dl_list_next	56
4.13.2.24	dl_list_prepend	56
4.13.2.25	dl_list_prev	56
4.13.2.26	dl_list_remove_at	56
4.13.2.27	dl_list_remove_node_back	57
4.13.2.28	dl_list_remove_node_front	57
4.13.2.29	dl_list_remove_node_mid	57
4.14	dl_list.h File Reference	57
4.14.1	Detailed Description	59
4.14.2	Function Documentation	59
4.14.2.1	dl_list_find	59
4.14.2.2	dl_list_free_node	59
4.14.2.3	dl_list_insert_node_after_mid	60
4.14.2.4	dl_list_insert_node_back	60
4.14.2.5	dl_list_insert_node_before_mid	60
4.14.2.6	dl_list_insert_node_front	60
4.14.2.7	dl_list_new_node	60
4.14.2.8	dl_list_remove_at	60
4.14.2.9	dl_list_remove_node_back	61
4.14.2.10	dl_list_remove_node_front	61
4.14.2.11	dl_list_remove_node_mid	61
4.15	general.c File Reference	61
4.15.1	Detailed Description	62
4.15.2	Function Documentation	63
4.15.2.1	cds_compare_int	63
4.15.2.2	cds_compare_long	63
4.15.2.3	cds_compare_string	63
4.15.2.4	cds_compare_uint	63
4.15.2.5	cds_compare_ulong	63
4.15.2.6	cds_new_int	64
4.15.2.7	cds_new_long	64
4.15.2.8	cds_new_string	64
4.15.2.9	cds_new_uint	64

4.15.2.10 cds_new_ulong	64
4.16 queue.c File Reference	65
4.16.1 Detailed Description	66
4.16.2 Function Documentation	66
4.16.2.1 queue_free	66
4.16.2.2 queue_init	66
4.16.2.3 queue_isempty	66
4.16.2.4 queue_length	66
4.16.2.5 queue_pop	67
4.16.2.6 queue_pushback	67
4.17 sl_list.c File Reference	67
4.17.1 Detailed Description	68
4.17.2 Function Documentation	68
4.17.2.1 sl_list_data	68
4.17.2.2 sl_list_delete_at	69
4.17.2.3 sl_list_find	69
4.17.2.4 sl_list_find_index	69
4.17.2.5 sl_list_find_itr	69
4.17.2.6 sl_list_first	69
4.17.2.7 sl_list_free	70
4.17.2.8 sl_list_free_node	70
4.17.2.9 sl_list_init	70
4.17.2.10 sl_list_insert_after	70
4.17.2.11 sl_list_insert_at	70
4.17.2.12 sl_list_isempty	71
4.17.2.13 sl_list_itr_from_index	71
4.17.2.14 sl_list_length	71
4.17.2.15 sl_list_new_node	71
4.17.2.16 sl_list_next	71
4.17.2.17 sl_list_prepend	71
4.17.2.18 sl_list_remove_at	72
4.18 sl_list.h File Reference	72
4.18.1 Detailed Description	73
4.18.2 Function Documentation	74
4.18.2.1 sl_list_find	74
4.18.2.2 sl_list_free_node	74
4.18.2.3 sl_list_new_node	74
4.18.2.4 sl_list_remove_at	74
4.19 stack.c File Reference	74
4.19.1 Detailed Description	75

4.19.2	Function Documentation	75
4.19.2.1	stack_free	75
4.19.2.2	stack_init	76
4.19.2.3	stack_isempty	76
4.19.2.4	stack_length	76
4.19.2.5	stack_pop	76
4.19.2.6	stack_push	76

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

bs_tree_node_t	Struct for binary search tree node	5
bs_tree_t	Struct to contain a binary search tree	6
dl_list_node_t	Struct for double linked list node	7
dl_list_t	Struct to contain a list	7
kvpair_t	Key-value pair struct	9
sl_list_node_t	Struct for singly linked list node	9
sl_list_t	Struct to contain a list	10

Chapter 2

File Index

2.1 File List

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

bs_tree.c	Implementation of binary search tree data structure	13
bs_tree.h	Developer interface to binary search tree data structure	17
bst_map.c	Implementation of binary search tree map data structure	20
cdatastruct.h	Interface to generic C data structures	22
cds_bs_tree.h	User interface to binary search tree data structure	23
cds_bst_map.h	User interface to binary search tree map data structure	26
cds_common.h	Common data types and data for C data structures library	29
cds_dl_list.h	User interface to doubly linked list data structure	30
cds_general.h	Interface to general data structure helper functions	36
cds_queue.h	User interface to queue data structure	39
cds_sl_list.h	User interface to singly linked list data structure	42
cds_stack.h	User interface to stack data structure	47
dl_list.c	Implementation of doubly linked list data structure	50
dl_list.h	Developer interface to double linked list data structure	57
general.c	Implementation of general data structure helper functions	61
queue.c	Implementation of queue data structure	65
sl_list.c	Implementation of singly linked list data structure	67
sl_list.h	Developer interface to singly linked list data structure	72
stack.c	Implementation of stack data structure	74

Chapter 3

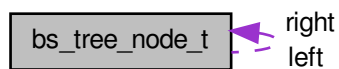
Data Structure Documentation

3.1 bs_tree_node_t Struct Reference

Struct for binary search tree node.

```
#include <cds_bs_tree.h>
```

Collaboration diagram for bs_tree_node_t:



Data Fields

- void * [data](#)
- struct [bs_tree_node_t](#) * [left](#)
- struct [bs_tree_node_t](#) * [right](#)

3.1.1 Field Documentation

3.1.1.1 void* bs_tree_node_t::data

Pointer to data

3.1.1.2 struct bs_tree_node_t* bs_tree_node_t::left

Pointer to left child node

3.1.1.3 struct bs_tree_node_t* bs_tree_node_t::right

Pointer to right child node

The documentation for this struct was generated from the following file:

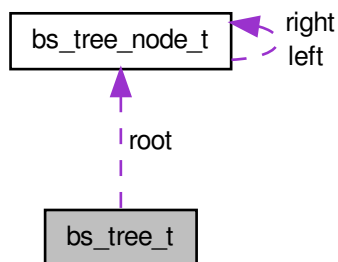
- [cds_bs_tree.h](#)

3.2 bs_tree_t Struct Reference

Struct to contain a binary search tree.

```
#include <bs_tree.h>
```

Collaboration diagram for bs_tree_t:



Data Fields

- struct [bs_tree_node_t](#) * [root](#)
- [size_t](#) [length](#)
- [int](#)(* [cfunc](#))()
- [void](#)(* [free_func](#))()

3.2.1 Field Documentation

3.2.1.1 [int](#)(* [bs_tree_t::cfunc](#))()

Pointer to compare function

3.2.1.2 [void](#)(* [bs_tree_t::free_func](#))()

Pointer to node free function

3.2.1.3 [size_t](#) [bs_tree_t::length](#)

Length of list

3.2.1.4 [struct](#) [bs_tree_node_t](#)* [bs_tree_t::root](#)

Pointer to root node

The documentation for this struct was generated from the following file:

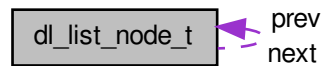
- [bs_tree.h](#)

3.3 dl_list_node_t Struct Reference

Struct for double linked list node.

```
#include <cds_dl_list.h>
```

Collaboration diagram for dl_list_node_t:



Data Fields

- void * [data](#)
- struct [dl_list_node_t](#) * [next](#)
- struct [dl_list_node_t](#) * [prev](#)

3.3.1 Field Documentation

3.3.1.1 void* dl_list_node_t::data

Pointer to data

3.3.1.2 struct dl_list_node_t* dl_list_node_t::next

Pointer to next node

3.3.1.3 struct dl_list_node_t* dl_list_node_t::prev

Pointer to previous node

The documentation for this struct was generated from the following file:

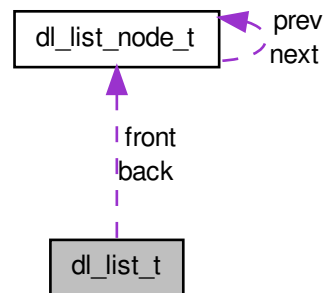
- [cds_dl_list.h](#)

3.4 dl_list_t Struct Reference

Struct to contain a list.

```
#include <dl_list.h>
```

Collaboration diagram for `dl_list_t`:



Data Fields

- struct `dl_list_node_t` * `front`
- struct `dl_list_node_t` * `back`
- `size_t` `length`
- `int(* cfunc)()`
- `void(* free_func)()`

3.4.1 Field Documentation

3.4.1.1 struct `dl_list_node_t`* `dl_list_t::back`

Pointer to last node

3.4.1.2 `int(* dl_list_t::cfunc)()`

Pointer to compare function

3.4.1.3 `void(* dl_list_t::free_func)()`

Pointer to free function

3.4.1.4 struct `dl_list_node_t`* `dl_list_t::front`

Pointer to first node

3.4.1.5 `size_t dl_list_t::length`

Length of list

The documentation for this struct was generated from the following file:

- [dl_list.h](#)

3.5 kvpair_t Struct Reference

Key-value pair struct.

Data Fields

- char * [key](#)
- void * [value](#)

3.5.1 Field Documentation

3.5.1.1 char* kvpair_t::key

Key string

3.5.1.2 void* kvpair_t::value

Pointer to data

The documentation for this struct was generated from the following file:

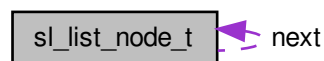
- [bst_map.c](#)

3.6 sl_list_node_t Struct Reference

Struct for singly linked list node.

```
#include <cds_sl_list.h>
```

Collaboration diagram for sl_list_node_t:



Data Fields

- void * [data](#)
- struct [sl_list_node_t](#) * [next](#)

3.6.1 Field Documentation

3.6.1.1 void* sl_list_node_t::data

Pointer to data

3.6.1.2 struct `sl_list_node_t`* `sl_list_node_t::next`

Pointer to next node

The documentation for this struct was generated from the following file:

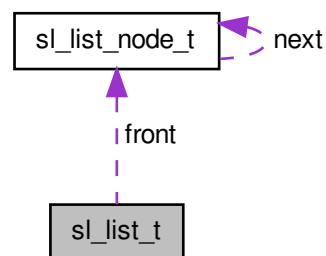
- [cds_sl_list.h](#)

3.7 `sl_list_t` Struct Reference

Struct to contain a list.

```
#include <sl_list.h>
```

Collaboration diagram for `sl_list_t`:



Data Fields

- struct [sl_list_node_t](#) * `front`
- `size_t` `length`
- `int(* cfunc)()`
- `void(* free_func)()`

3.7.1 Field Documentation

3.7.1.1 `int(* sl_list_t::cfunc)()`

Pointer to compare function

3.7.1.2 `void(* sl_list_t::free_func)()`

Pointer to free function

3.7.1.3 struct `sl_list_node_t`* `sl_list_t::front`

Pointer to first node

3.7.1.4 `size_t sl_list_t::length`

Length of list

The documentation for this struct was generated from the following file:

- [sl_list.h](#)

Chapter 4

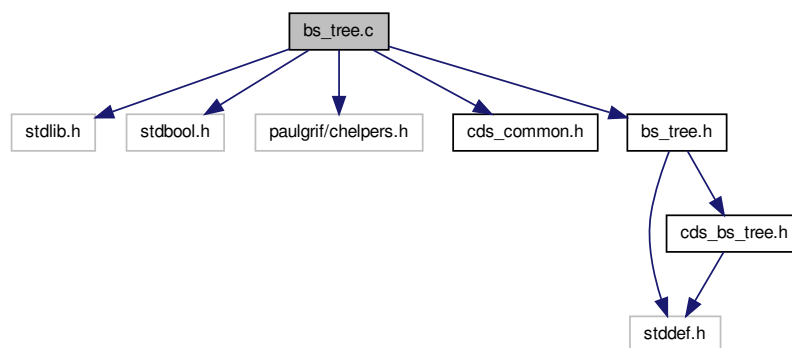
File Documentation

4.1 bs_tree.c File Reference

Implementation of binary search tree data structure.

```
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "bs_tree.h"
```

Include dependency graph for bs_tree.c:



Functions

- `bs_tree bs_tree_init` (`int(*cfunc)(const void *, const void *)`, `void(*free_func)(void *)`)
Initializes a new binary search tree.
- `void bs_tree_free` (`bs_tree tree`)
Frees the resources associated with a tree.
- `size_t bs_tree_length` (`const bs_tree tree`)
Returns the number of elements in a tree.
- `bool bs_tree_isempty` (`const bs_tree tree`)
Checks if a tree is empty.
- `bool bs_tree_search` (`const bs_tree tree`, `const void *data`)
Determines if a data element is in a tree.

- void * [bs_tree_search_data](#) (const [bs_tree](#) tree, const void *data)
Searches a tree for a piece of data and returns it.
- bool [bs_tree_insert](#) ([bs_tree](#) tree, void *data)
Inserts data into a tree.
- [bs_tree_node](#) [bs_tree_new_node](#) (void *data)
Creates and allocates memory for a new node.
- void [bs_tree_free_subtree](#) ([bs_tree](#) tree, [bs_tree_node](#) node)
Frees the resources associated with a subtree.
- [bs_tree_node](#) [bs_tree_search_node](#) (const [bs_tree](#) tree, const void *data)
Searches a tree for a piece of data.
- bool [bs_tree_insert_subtree](#) ([bs_tree](#) tree, [bs_tree_node](#) *p_node, void *data)
Inserts a data element into a subtree.
- [bs_tree_node](#) [bst_insert_search](#) ([bs_tree](#) tree, void *data, bool *found)
Searches a tree for insertion purposes.

4.1.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.1.2 Function Documentation

4.1.2.1 void [bs_tree_free](#) ([bs_tree](#) tree)

Parameters

<i>tree</i>	A pointer to the tree to free.
-------------	--------------------------------

4.1.2.2 void [bs_tree_free_subtree](#) ([bs_tree](#) tree, [bs_tree_node](#) node)

This function frees the node recursively.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the tree node at the root of the subtree.

4.1.2.3 [bs_tree](#) [bs_tree_init](#) (int(*) (const void *, const void *) *cfunc*, void(*) (void *) *free_func*)

Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a free function. The function should return no value, and accept one parameter of type <code>void *</code> . If set to <code>NULL</code> , the standard C <code>free()</code> function is used. This function is useful when the data elements are structs which themselves contain dynamically allocated members, which need to be <code>free()</code> d before the overall struct is <code>free()</code> .

Returns

A pointer to the new tree.

4.1.2.4 `bool bs_tree_insert (bs_tree tree, void * data)`

Duplicated data is replaced. This is a superfluous operation for scalar data, but is necessary for structs, where 'found' may mean only one element of the struct compares equal, and other elements may be different (e.g. a map data structure).

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data to insert.

Returns

`true` if the data was already in the tree and has been replaced, `false` if it was not present and newly added.

4.1.2.5 `bool bs_tree_insert_subtree (bs_tree tree, bs_tree_node * p_node, void * data)`

The data element is replaced if it is found in the tree. This is a superfluous operation for scalar data, but is necessary for structs, where 'found' may mean only one of the struct members compares equal, and other data elements may differ. This function `free()`s the old data when this happens.

Parameters

<i>tree</i>	A pointer to the tree
<i>p_node</i>	A pointer to the pointer to the node at the root of the subtree.
<i>data</i>	A pointer to the data to which to insert.

Returns

`true` if the data was present and duplicated, 'false' if not.

4.1.2.6 `bool bs_tree_isempty (const bs_tree tree)`

Parameters

<i>tree</i>	A pointer to the tree.
-------------	------------------------

Returns

`true` if the tree is empty, otherwise `false`.

4.1.2.7 `size_t bs_tree_length (const bs_tree tree)`**Parameters**

<i>tree</i>	A pointer to the tree.
-------------	------------------------

Returns

The number of elements in the tree.

4.1.2.8 `bs_tree_node bs_tree_new_node (void * data)`**Parameters**

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly-created node.

4.1.2.9 `bool bs_tree_search (const bs_tree tree, const void * data)`**Parameters**

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data for which to search.

Returns

`true` if the data is found, `false` otherwise.

4.1.2.10 `void* bs_tree_search_data (const bs_tree tree, const void * data)`**Parameters**

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data for which to search.

Returns

A pointer to the data if found, `NULL` otherwise.

4.1.2.11 `bs_tree_node bs_tree_search_node (const bs_tree tree, const void * data)`**Parameters**

<i>tree</i>	A pointer to the tree.
<i>data</i>	A pointer to the data for which to search.

Returns

A pointer to the node in which the data was found, or `NULL` if the data was not found.

4.1.2.12 `bs_tree_node bst.insert.search (bs_tree tree, void * data, bool * found)`

The function searches the tree for a piece of data, and if it is not found, returns a pointer to the node under which it should be inserted.

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	A pointer to the data for which to search.
<i>found</i>	A pointer to a <code>bool</code> to populate according to whether the data is already in the tree.

Returns

A pointer to the node in which the data was found, if it was found, or a pointer to the last node tried if it was not. The last tried node is the one under which the new data should be inserted, if it is not already in the tree.

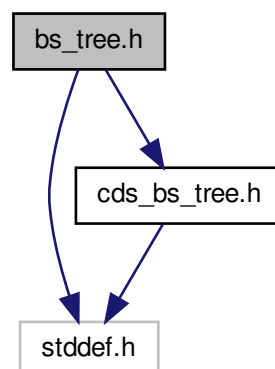
4.2 bs_tree.h File Reference

Developer interface to binary search tree data structure.

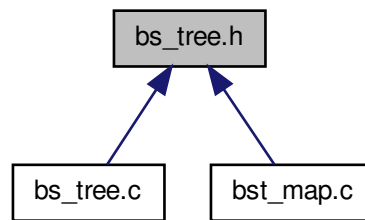
```
#include <stddef.h>
```

```
#include "cds_bs_tree.h"
```

Include dependency graph for `bs_tree.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [bs_tree_t](#)
Struct to contain a binary search tree.

Typedefs

- typedef struct [bs_tree_t](#) [sl_list_t](#)
Struct to contain a binary search tree.
- typedef struct [bs_tree_node_t](#) * [bs_tree_node](#)
Typedef for binary search tree node.

Functions

- [bs_tree_node](#) [bs_tree_new_node](#) (void *data)
Creates and allocates memory for a new node.
- void [bs_tree_free_subtree](#) ([bs_tree](#) tree, [bs_tree_node](#) node)
Frees the resources associated with a subtree.
- [bs_tree_node](#) [bs_tree_search_node](#) (const [bs_tree](#) tree, const void *key)
Searches a tree for a piece of data.
- bool [bs_tree_insert_subtree](#) ([bs_tree](#) tree, [bs_tree_node](#) *p_node, void *data)
Inserts a data element into a subtree.
- [bs_tree_node](#) [bst_insert_search](#) ([bs_tree](#) tree, void *key, bool *found)
Searches a tree for insertion purposes.

4.2.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.2.2 Function Documentation

4.2.2.1 void bs_tree_free_subtree (bs_tree tree, bs_tree_node node)

This function frees the node recursively.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the tree node at the root of the subtree.

4.2.2.2 bool bs_tree_insert_subtree (bs_tree tree, bs_tree_node * p_node, void * data)

The data element is replaced if it is found in the tree. This is a superfluous operation for scalar data, but is necessary for structs, where 'found' may mean only one of the struct members compares equal, and other data elements may differ. This function `free()`s the old data when this happens.

Parameters

<i>tree</i>	A pointer to the tree
<i>p_node</i>	A pointer to the pointer to the node at the root of the subtree.
<i>data</i>	A pointer to the data to which to insert.

Returns

`true` if the data was present and duplicated, 'false' if not.

4.2.2.3 bs_tree_node bs_tree_new_node (void * data)

Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly-created node.

4.2.2.4 bs_tree_node bs_tree_search_node (const bs_tree tree, const void * data)

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	A pointer to the data for which to search.

Returns

A pointer to the node in which the data was found, or `NULL` if the data was not found.

4.2.2.5 bs_tree_node bst_insert_search (bs_tree tree, void * data, bool * found)

The function searches the tree for a piece of data, and if it is not found, returns a pointer to the node under which it should be inserted.

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	A pointer to the data for which to search.
<i>found</i>	A pointer to a <code>bool</code> to populate according to whether the data is already in the tree.

Returns

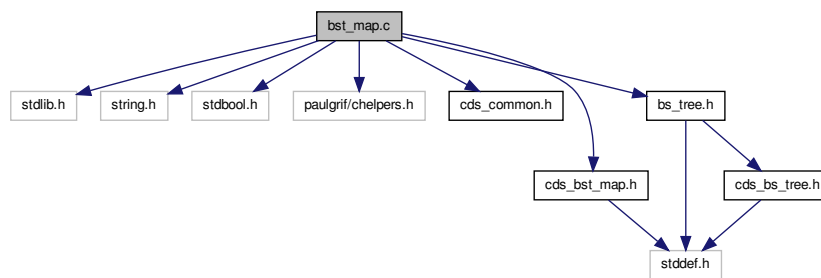
A pointer to the node in which the data was found, if it was found, or a pointer to the last node tried if it was not. The last tried node is the one under which the new data should be inserted, if it is not already in the tree.

4.3 bst_map.c File Reference

Implementation of binary search tree map data structure.

```
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "cds_bst_map.h"
#include "bs_tree.h"
```

Include dependency graph for `bst_map.c`:



Data Structures

- struct `kvpair_t`
Key-value pair struct.

Typedefs

- typedef struct `kvpair_t` `kvpair_t`
Key-value pair struct.
- typedef struct `kvpair_t` * `kvpair`
Typedef for kvpair pointer.

Functions

- `bst_map` `bst_map_init` (void)
Initializes a new binary search tree map.

- void `bst_map_free` (`bst_map` map)
Frees the resources associated with a BST map.
- size_t `bst_map_length` (const `bst_map` map)
Returns the number of elements in a BST map.
- bool `bst_map_isempty` (const `bst_map` map)
Checks if a map is empty.
- bool `bst_map_search` (const `bst_map` map, const char *key)
Determines if a key is in a map.
- void * `bst_map_search_data` (const `bst_map` map, const char *key)
Searches a map for a value matching a key and returns it.
- bool `bst_map_insert` (`bst_map` map, const char *key, void *value)
Inserts a key-value pair into a map.

4.3.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.3.2 Function Documentation

4.3.2.1 void bst_map_free (bst_map map)

Parameters

<i>map</i>	A pointer to the map to free.
------------	-------------------------------

4.3.2.2 bst_map bst_map_init (void)

Returns

A pointer to the new map.

4.3.2.3 bool bst_map_insert (bst_map map, const char * key, void * value)

The value is replaced if the key is already found in the map. Any memory consumed by the old value is automatically `free()`d.

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key of the new value to insert.
<i>value</i>	A pointer to the new value to insert.

Returns

`true` if the key was already in the tree and the value has been replaced, `false` if the key was not present.

4.3.2.4 `bool bst_map_isempty (const bst_map map)`

Parameters

<i>map</i>	A pointer to the map.
------------	-----------------------

Returns

`true` if the map is empty, otherwise `false`.

4.3.2.5 `size_t bst_map_length (const bst_map map)`

Parameters

<i>map</i>	A pointer to the map.
------------	-----------------------

Returns

The number of elements in the map.

4.3.2.6 `bool bst_map_search (const bst_map map, const char * key)`

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key for which to search.

Returns

`true` if the key is found, `false` otherwise.

4.3.2.7 `void* bst_map_search_data (const bst_map map, const char * key)`

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key for which to search.

Returns

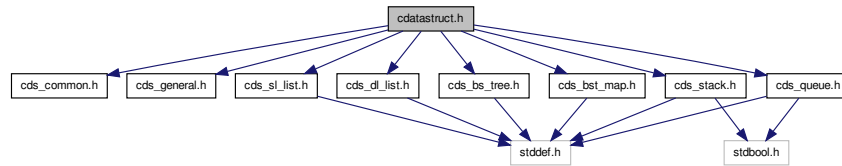
A pointer to the value if found, `NULL` otherwise.

4.4 `cdatastruct.h` File Reference

Interface to generic C data structures.

```
#include "cds_common.h"
#include "cds_general.h"
#include "cds_sl_list.h"
#include "cds_dl_list.h"
#include "cds_stack.h"
#include "cds_queue.h"
#include "cds_bs_tree.h"
#include "cds_bst_map.h"
```


Include dependency graph for cdatastruct.h:



4.4.1 Detailed Description

Interface to generic C data structures.

Author

Paul Griffiths

Copyright

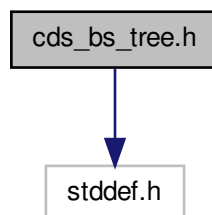
Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.5 cds_bs_tree.h File Reference

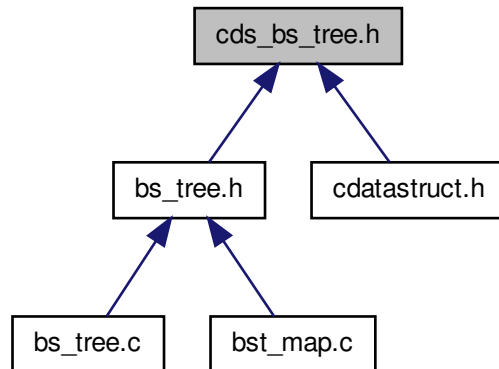
User interface to binary search tree data structure.

```
#include <stddef.h>
```

Include dependency graph for `cds_bs_tree.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [bs_tree_node_t](#)
Struct for binary search tree node.

Typedefs

- typedef struct [bs_tree_node_t](#) [bs_tree_node_t](#)
Struct for binary search tree node.
- typedef struct [bs_tree_t](#) * [bs_tree](#)
Typedef for tree pointer.

Functions

- [bs_tree](#) [bs_tree_init](#) (int(*cfunc)(const void *, const void *), void(*free_func)(void *))
Initializes a new binary search tree.
- void [bs_tree_free](#) ([bs_tree](#) tree)
Frees the resources associated with a tree.
- bool [bs_tree_isempty](#) (const [bs_tree](#) tree)
Checks if a tree is empty.
- size_t [bs_tree_length](#) (const [bs_tree](#) tree)
Returns the number of elements in a tree.
- bool [bs_tree_insert](#) ([bs_tree](#) tree, void *data)
Inserts data into a tree.
- bool [bs_tree_search](#) (const [bs_tree](#) tree, const void *data)
Determines if a data element is in a tree.
- void * [bs_tree_search_data](#) (const [bs_tree](#) tree, const void *data)
Searches a tree for a piece of data and returns it.

4.5.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.5.2 Function Documentation

4.5.2.1 void bs_tree_free (bs_tree tree)

Parameters

<i>tree</i>	A pointer to the tree to free.
-------------	--------------------------------

4.5.2.2 bs_tree bs_tree_init (int (*)(const void *, const void *) cfunc, void (*)(void *) free_func)

Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a free function. The function should return no value, and accept one parameter of type <code>void *</code> . If set to <code>NULL</code> , the standard C <code>free()</code> function is used. This function is useful when the data elements are structs which themselves contain dynamically allocated members, which need to be <code>free()</code> d before the overall struct is <code>free()</code> .

Returns

A pointer to the new tree.

4.5.2.3 bool bs_tree_insert (bs_tree tree, void * data)

Duplicated data is replaced. This is a superfluous operation for scalar data, but is necessary for structs, where 'found' may mean only one element of the struct compares equal, and other elements may be different (e.g. a map data structure).

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data to insert.

Returns

`true` if the data was already in the tree and has been replaced, `false` if it was not present and newly added.

4.5.2.4 bool bs_tree_isempty (const bs_tree tree)

Parameters

<i>tree</i>	A pointer to the tree.
-------------	------------------------

Returns

`true` if the tree is empty, otherwise `false`.

4.5.2.5 `size_t bs_tree_length (const bs_tree tree)`

Parameters

<i>tree</i>	A pointer to the tree.
-------------	------------------------

Returns

The number of elements in the tree.

4.5.2.6 `bool bs_tree_search (const bs_tree tree, const void * data)`

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data for which to search.

Returns

`true` if the data is found, `false` otherwise.

4.5.2.7 `void* bs_tree_search_data (const bs_tree tree, const void * data)`

Parameters

<i>tree</i>	A pointer to the tree.
<i>data</i>	The data for which to search.

Returns

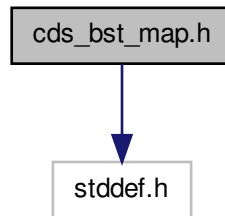
A pointer to the data if found, `NULL` otherwise.

4.6 `cds_bst_map.h` File Reference

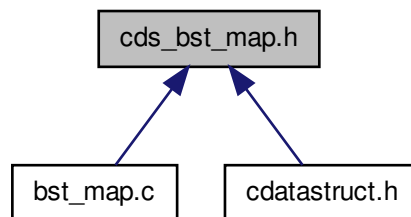
User interface to binary search tree map data structure.

```
#include <stddef.h>
```

Include dependency graph for cds_bst_map.h:



This graph shows which files directly or indirectly include this file:



Typedefs

- typedef struct [bs_tree_t](#) * [bst_map](#)

Typedef for map pointer.

Functions

- [bst_map](#) [bst_map_init](#) (void)
Initializes a new binary search tree map.
- void [bst_map_free](#) ([bst_map](#) map)
Frees the resources associated with a BST map.
- bool [bst_map_isempty](#) (const [bst_map](#) map)
Checks if a map is empty.
- size_t [bst_map_length](#) (const [bst_map](#) map)
Returns the number of elements in a BST map.
- bool [bst_map_insert](#) ([bst_map](#) map, const char *key, void *value)
Inserts a key-value pair into a map.
- bool [bst_map_search](#) (const [bst_map](#) map, const char *key)
Determines if a key is in a map.

- `void * bst_map_search_data (const bst_map map, const char *key)`
Searches a map for a value matching a key and returns it.

4.6.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.6.2 Function Documentation

4.6.2.1 `void bst_map_free (bst_map map)`

Parameters

<i>map</i>	A pointer to the map to free.
------------	-------------------------------

4.6.2.2 `bst_map bst_map_init (void)`

Returns

A pointer to the new map.

4.6.2.3 `bool bst_map_insert (bst_map map, const char * key, void * value)`

The value is replaced if the key is already found in the map. Any memory consumed by the old value is automatically `free()`d.

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key of the new value to insert.
<i>value</i>	A pointer to the new value to insert.

Returns

`true` if the key was already in the tree and the value has been replaced, `false` if the key was not present.

4.6.2.4 `bool bst_map_isempty (const bst_map map)`

Parameters

<i>map</i>	A pointer to the map.
------------	-----------------------

Returns

`true` if the map is empty, otherwise `false`.

4.6.2.5 `size_t bst_map_length (const bst_map map)`

Parameters

<i>map</i>	A pointer to the map.
------------	-----------------------

Returns

The number of elements in the map.

4.6.2.6 `bool bst_map_search (const bst_map map, const char * key)`

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key for which to search.

Returns

`true` is the key is found, `false` otherwise.

4.6.2.7 `void* bst_map_search_data (const bst_map map, const char * key)`

Parameters

<i>map</i>	A pointer to the map.
<i>key</i>	The key for which to search.

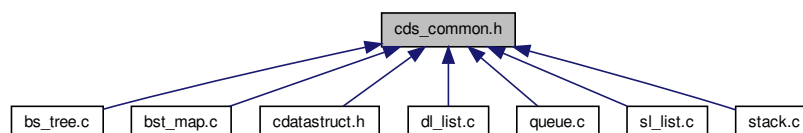
Returns

A pointer to the value if found, `NULL` otherwise.

4.7 cds_common.h File Reference

Common data types and data for C data structures library.

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef enum [cds_error](#) [cds_error](#)
Enumeration of return error codes.

Enumerations

- enum `cds_error` { `CDSERR_ERROR` = -1, `CDSERR_OUTOFRANGE` = -2, `CDSERR_NOTFOUND` = -3, `CDSERR_BADITERATOR` = -4 }

Enumeration of return error codes.

4.7.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.7.2 Enumeration Type Documentation

4.7.2.1 enum `cds_error`

Enumerator:

`CDSERR_ERROR` Unspecified error

`CDSERR_OUTOFRANGE` Index out of range

`CDSERR_NOTFOUND` Data element not found

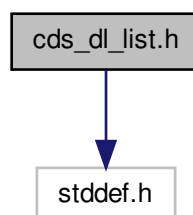
`CDSERR_BADITERATOR` Invalid iterator

4.8 `cds_dl_list.h` File Reference

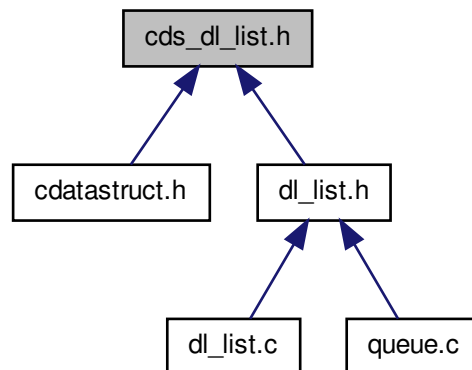
User interface to doubly linked list data structure.

```
#include <stddef.h>
```

Include dependency graph for `cds_dl_list.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [dl_list_node_t](#)
Struct for double linked list node.

Typedefs

- typedef struct [dl_list_node_t](#) [dl_list_node_t](#)
Struct for double linked list node.
- typedef struct [dl_list_t](#) * [dl_list](#)
Typedef for list pointer.
- typedef struct [dl_list_node_t](#) * [dl_list_itr](#)
Typedef for list iterator.

Functions

- [dl_list dl_list_init](#) (int(*cfunc)(const void *, const void *), void(*free_func)(void *))
Initializes a new doubly linked list.
- void [dl_list_free](#) ([dl_list](#) list)
Frees the resources associated with a list.
- size_t [dl_list_length](#) (const [dl_list](#) list)
Returns the number of elements in a list.
- bool [dl_list_isempty](#) (const [dl_list](#) list)
Checks if a list is empty.
- void [dl_list_prepend](#) ([dl_list](#) list, void *data)
Inserts an element at the beginning of a list.
- void [dl_list_append](#) ([dl_list](#) list, void *data)
Inserts an element at the end of a list.
- int [dl_list_insert_before](#) ([dl_list](#) list, const [dl_list_itr](#) itr, void *data)
Inserts an element before a provided iterator.

- int `dl_list_insert_at` (`dl_list` list, const size_t index, void *data)
Inserts an element at the specified index of a list.
- int `dl_list_insert_after` (`dl_list` list, const `dl_list_itr` itr, void *data)
Inserts an element after a provided iterator.
- int `dl_list_delete_at` (`dl_list` list, const size_t index)
Deletes a list element at a specified index.
- int `dl_list_find_index` (const `dl_list` list, const void *data)
Finds the index of the specified data in a list.
- `dl_list_itr` `dl_list_find_itr` (const `dl_list` list, const void *data)
Gets an iterator to the specified data in a list.
- void * `dl_list_data` (const `dl_list` list, const size_t index)
Returns a pointer to the data at a specified index.
- `dl_list_itr` `dl_list_first` (const `dl_list` list)
Returns an iterator to the first element of a list.
- `dl_list_itr` `dl_list_last` (const `dl_list` list)
Returns an iterator to the last element of a list.
- `dl_list_itr` `dl_list_next` (const `dl_list_itr` itr)
Advances a list iterator by one element.
- `dl_list_itr` `dl_list_prev` (const `dl_list_itr` itr)
Backs up a list iterator by one element.
- `dl_list_itr` `dl_list_itr_from_index` (const `dl_list` list, const size_t index)
Return an iterator to a specified element of a list.

4.8.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.8.2 Function Documentation

4.8.2.1 void `dl_list_append` (`dl_list` list, void * data)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.8.2.2 void* `dl_list_data` (const `dl_list` list, const size_t index)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

Returns

A pointer to the data, or NULL if the index is out of range.

4.8.2.3 `int dl_list_delete_at (dl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

Returns

0 on success, CDSERR_OUTOFRANGE if the the index is out of range.

4.8.2.4 `int dl_list_find_index (const dl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

The index of the element, if found, or CDSERR_NOTFOUND if it is not in the list.

4.8.2.5 `dl_list_itr dl_list_find_itr (const dl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

An iterator to the found element, or NULL is the element is not in the list.

4.8.2.6 `dl_list_itr dl_list_first (const dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.8.2.7 `void dl_list_free (dl_list list)`

Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

4.8.2.8 `dl_list dl_list_init (int(*)(const void *, const void *) cfunc, void(*)(void *) free_func)`

Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a function to free a node. The function should return no value, and accept a <code>void</code> pointer to the node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.

Returns

A pointer to the new list.

4.8.2.9 `int dl_list_insert_after (dl_list list, const dl_list_itr itr, void * data)`

Note that `dl_list_first()` may return a `NULL` iterator when the list is empty. One reasonable behavior for this function would be to add a new node to the list in that case. However, an iterator may also become `NULL` when advanced to the end of the list. One possible way to modify this function would be to check the length of this list when the iterator is `NULL`, and if it is zero, add the first node to the list. However, the semantic meaning of adding an element *after* an iterator breaks down if that that iterator does not point to an existing element. Therefore, it is simpler for this function to simply refuse to handle `NULL` iterators. It is unlikely a user would want to call this function unless there are already elements in a list, and a valid iterator has been returned, e.g. through a find function.

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, `CDSERR_BADITERATOR` if `itr` is a `NULL` pointer.

4.8.2.10 `int dl_list_insert_at (dl_list list, const size_t index, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, `CDSERR_OUTOFRANGE` if `index` exceeds the length of the list.

4.8.2.11 `int dl_list_insert_before (dl_list list, const dl_list_itr itr, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_BADITERATOR if `itr` is a NULL pointer.

4.8.2.12 `bool dl_list_isempty (const dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

`true` if the list is empty, otherwise `false`.

4.8.2.13 `dl_list_itr dl_list_itr_from_index (const dl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

Returns

The iterator, or NULL if `index` is out of range.

4.8.2.14 `dl_list_itr dl_list_last (const dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.8.2.15 `size_t dl_list_length (const dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

4.8.2.16 `dl_list_itr dl_list_next (const dl_list_itr itr)`

Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

Returns

The advanced iterator.

4.8.2.17 void dl_list_prepend (dl_list list, void * data)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.8.2.18 dl_list_itr dl_list_prev (const dl_list_itr itr)

Parameters

<i>itr</i>	The iterator to back up.
------------	--------------------------

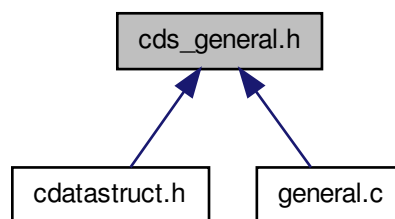
Returns

The backed up iterator.

4.9 cds_general.h File Reference

Interface to general data structure helper functions.

This graph shows which files directly or indirectly include this file:



Functions

- void * [cds_new_int](#) (const int n)
Dynamically allocates memory for a new int.
- void * [cds_new_uint](#) (const unsigned int n)
Dynamically allocates memory for a new unsigned int.
- void * [cds_new_long](#) (const long n)

- Dynamically allocates memory for a new long.*
- void * [cds_new_ulong](#) (const unsigned long n)
- Dynamically allocates memory for a new unsigned long.*
- void * [cds_new_string](#) (const char *str)
- Dynamically allocates memory for a new string.*
- int [cds_compare_int](#) (const void *data, const void *cmp)
- Compares two int via void pointers.*
- int [cds_compare_uint](#) (const void *data, const void *cmp)
- Compares two unsigned int via void pointers.*
- int [cds_compare_long](#) (const void *data, const void *cmp)
- Compares two long via void pointers.*
- int [cds_compare_ulong](#) (const void *data, const void *cmp)
- Compares two unsigned long via void pointers.*
- int [cds_compare_string](#) (const void *data, const void *cmp)
- Compares two strings via void pointers.*

4.9.1 Detailed Description

Interface to general data structure helper functions.

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.9.2 Function Documentation

4.9.2.1 int cds_compare_int (const void * data, const void * cmp)

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.9.2.2 int cds_compare_long (const void * data, const void * cmp)

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.9.2.3 `int cds_compare_string (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.9.2.4 `int cds_compare_uint (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.9.2.5 `int cds_compare_ulong (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.9.2.6 `void* cds_new_int (const int n)`

Parameters

<i>n</i>	The new <code>int</code> for which to allocate.
----------	---

Returns

A `void` pointer to the allocated memory.

4.9.2.7 `void* cds_new_long (const long n)`

Parameters

<i>n</i>	The new <code>long</code> for which to allocate.
----------	--

Returns

A `void` pointer to the allocated memory.

4.9.2.8 void* cds_new_string (const char * str)**Parameters**

<i>str</i>	The new string for which to allocate.
------------	---------------------------------------

Returns

A `void` pointer to the allocated memory.

4.9.2.9 void* cds_new_uint (const unsigned int n)**Parameters**

<i>n</i>	The new unsigned int for which to allocate.
----------	---

Returns

A `void` pointer to the allocated memory.

4.9.2.10 void* cds_new_ulong (const unsigned long n)**Parameters**

<i>n</i>	The new unsigned long for which to allocate.
----------	--

Returns

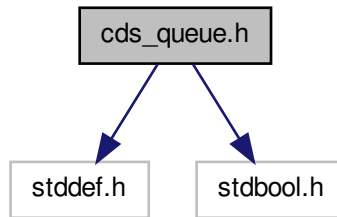
A `void` pointer to the allocated memory.

4.10 cds_queue.h File Reference

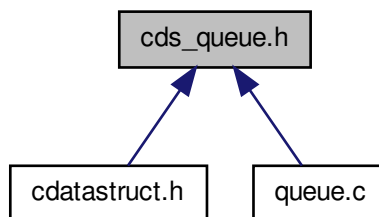
User interface to queue data structure.

```
#include <stddef.h>
#include <stdbool.h>
```

Include dependency graph for cds_queue.h:



This graph shows which files directly or indirectly include this file:



Typedefs

- typedef struct dl_list_t * queue
Typedef for queue pointer.

Functions

- queue queue_init (void(*free_func)(void *))
Initializes a new queue.
- void queue_free (queue que)
Frees memory and releases resources used by a queue.
- size_t queue_length (const queue que)
Gets the number of items in a queue.
- bool queue_isempty (const queue que)
Checks if a queue is empty.
- void * queue_pop (queue que)
Pops a data item from the queue.
- void queue_pushback (queue que, void *data)
Pushes a data item onto the back of the queue.

4.10.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.10.2 Function Documentation

4.10.2.1 void queue_free (queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

4.10.2.2 queue queue_init (void(*) (void *) *free_func*)

Parameters

<i>free_func</i>	A pointer to a function to free a queue node. The function should return no value, and accept a void pointer to a node. If NULL is specified, the standard free () function is used.
------------------	--

Returns

A pointer to the new queue.

4.10.2.3 bool queue_isempty (const queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

true is the queue is empty, false if not.

4.10.2.4 size_t queue_length (const queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

The number of items in the queue.

4.10.2.5 void* queue_pop (queue *que*)

The item returned was previously allocated using malloc (), so the user must free () the returned pointer when done.

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

A `void` pointer to the popped data item.

4.10.2.6 void queue_pushback (queue *que*, void * *data*)

The provided pointer should point to dynamically allocated memory.

Parameters

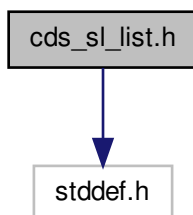
<i>que</i>	A pointer to the queue.
<i>data</i>	A pointer to the data item to be pushed.

4.11 cds_sl_list.h File Reference

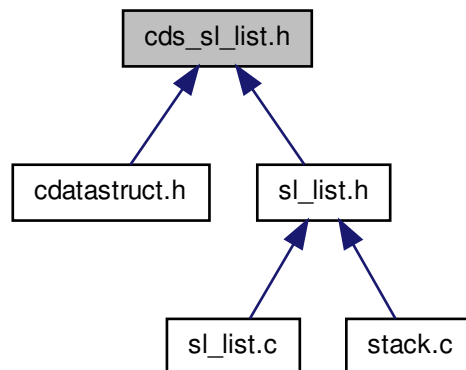
User interface to singly linked list data structure.

```
#include <stddef.h>
```

Include dependency graph for `cds_sl_list.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [sl_list_node_t](#)
Struct for singly linked list node.

Typedefs

- typedef struct [sl_list_node_t](#) [sl_list_node_t](#)
Struct for singly linked list node.
- typedef struct [sl_list_t](#) * [sl_list](#)
Typedef for list pointer.
- typedef struct [sl_list_node_t](#) * [sl_list_itr](#)
Typedef for list iterator.

Functions

- [sl_list sl_list_init](#) (int(*cfunc)(const void *, const void *), void(*free_func)(void *))
Initializes a new singly linked list.
- void [sl_list_free](#) ([sl_list](#) list)
Frees the resources associated with a list.
- size_t [sl_list_length](#) (const [sl_list](#) list)
Returns the number of elements in a list.
- bool [sl_list_isempty](#) (const [sl_list](#) list)
Checks if a list is empty.
- void [sl_list_prepend](#) ([sl_list](#) list, void *data)
Inserts an element at the beginning of a list.
- int [sl_list_insert_at](#) ([sl_list](#) list, const size_t index, void *data)
Inserts an element at the specified index of a list.
- int [sl_list_insert_after](#) ([sl_list](#) list, const [sl_list_itr](#) itr, void *data)
Inserts an element after a provided iterator.

- `int sl_list_delete_at (sl_list list, const size_t index)`
Deletes a list element at a specified index.
- `int sl_list_find_index (const sl_list list, const void *data)`
Gets an index to the specified data in a list.
- `sl_list_itr sl_list_find_itr (const sl_list list, const void *data)`
Gets an iterator to the specified data in a list.
- `void * sl_list_data (const sl_list list, const size_t index)`
Returns a pointer to the data at a specified index.
- `sl_list_itr sl_list_first (const sl_list list)`
Returns an iterator to the first element of a list.
- `sl_list_itr sl_list_next (const sl_list_itr itr)`
Advances a list iterator by one element.
- `sl_list_itr sl_list_itr_from_index (const sl_list list, const size_t index)`
Return an iterator to a specified element of a list.

4.11.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.11.2 Function Documentation

4.11.2.1 `void* sl_list_data (const sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

Returns

A pointer to the data, or NULL if the index is out of range.

4.11.2.2 `int sl_list_delete_at (sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

Returns

0 on success, CDSERR_OUTOFRANGE if the the index is out of range.

4.11.2.3 `int sl_list_find_index (const sl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

The index of the found element, or CDSERR_NOTFOUND if the element is not in the list.

4.11.2.4 `sl_list_itr sl_list_find_itr (const sl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

An iterator to the found element, or NULL if the element is not in the list.

4.11.2.5 `sl_list_itr sl_list_first (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.11.2.6 `void sl_list_free (sl_list list)`

Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

4.11.2.7 `sl_list sl_list_init (int (*)(const void *, const void *) cfunc, void (*)(void *) free_func)`

Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a function for freeing a node. The function should return no value, and accept a <code>void</code> pointer to the node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.

Returns

A pointer to the new list.

4.11.2.8 `int sl_list_insert_after (sl_list list, const sl_list_itr itr, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_BADITERATOR if `itr` is a NULL pointer.

4.11.2.9 `int sl_list_insert_at (sl_list list, const size_t index, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_OUTOFRANGE if `index` exceeds the length of the list.

4.11.2.10 `bool sl_list_isempty (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

`true` if the list is empty, otherwise `false`.

4.11.2.11 `sl_list_itr sl_list_itr_from_index (const sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

Returns

The iterator, or NULL if `index` is out of range.

4.11.2.12 `size_t sl_list_length (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

4.11.2.13 `sl_list_itr sl_list_next (const sl_list_itr itr)`

Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

Returns

The advanced iterator.

4.11.2.14 `void sl_list_prepend (sl_list list, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

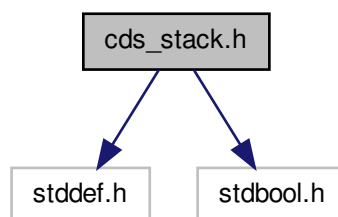
4.12 cds_stack.h File Reference

User interface to stack data structure.

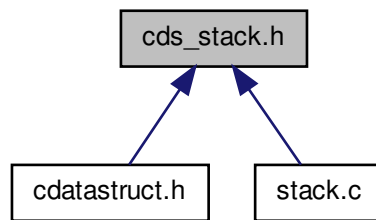
```
#include <stddef.h>
```

```
#include <stdbool.h>
```

Include dependency graph for `cds_stack.h`:



This graph shows which files directly or indirectly include this file:



Typedefs

- typedef struct [sl_list_t](#) * [stack](#)
Typedef for stack pointer.

Functions

- [stack](#) [stack_init](#) (void(*free_func)(void *))
Initializes a new stack.
- void [stack_free](#) ([stack](#) stk)
Frees memory and releases resources used by a stack.
- size_t [stack_length](#) (const [stack](#) stk)
Gets the number of items in a stack.
- bool [stack_isempty](#) (const [stack](#) stk)
Checks if a stack is empty.
- void * [stack_pop](#) ([stack](#) stk)
Pops a data item from the stack.
- void [stack_push](#) ([stack](#) stk, void *data)
Pushes a data item onto the stack.

4.12.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.12.2 Function Documentation

4.12.2.1 void [stack_free](#) ([stack](#) *stk*)

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

4.12.2.2 `stack stack_init (void(*) (void *) free_func)`

Parameters

<i>free_func</i>	A pointer to a function a free a stack node. The function should return no value, and accept a void pointer to a node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.
------------------	--

Returns

A pointer to the new stack.

4.12.2.3 `bool stack_isempty (const stack stk)`

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

`true` is the stack is empty, `false` if not.

4.12.2.4 `size_t stack_length (const stack stk)`

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

The number of items in the stack.

4.12.2.5 `void* stack_pop (stack stk)`

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

A void pointer to the popped data item.

4.12.2.6 `void stack_push (stack stk, void * data)`

The provided pointer should point to dynamically allocated memory.

Parameters

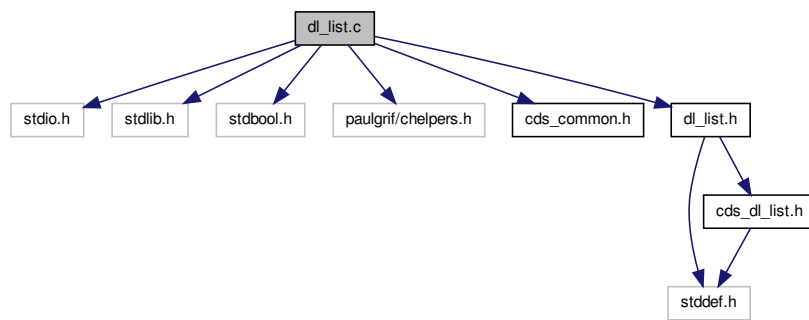
<i>stk</i>	A pointer to the stack.
<i>data</i>	A pointer to the data item to be pushed.

4.13 dl_list.c File Reference

Implementation of doubly linked list data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "dl_list.h"
```

Include dependency graph for dl_list.c:



Functions

- `dl_list dl_list_init (int(*cfunc)(const void *, const void *), void(*free_func)(void *))`
Initializes a new doubly linked list.
- `void dl_list_free (dl_list list)`
Frees the resources associated with a list.
- `size_t dl_list_length (const dl_list list)`
Returns the number of elements in a list.
- `bool dl_list_isempty (const dl_list list)`
Checks if a list is empty.
- `void dl_list_prepend (dl_list list, void *data)`
Inserts an element at the beginning of a list.
- `void dl_list_append (dl_list list, void *data)`
Inserts an element at the end of a list.
- `int dl_list_insert_before (dl_list list, const dl_list_itr itr, void *data)`
Inserts an element before a provided iterator.
- `int dl_list_insert_at (dl_list list, const size_t index, void *data)`
Inserts an element at the specified index of a list.
- `int dl_list_insert_after (dl_list list, const dl_list_itr itr, void *data)`
Inserts an element after a provided iterator.
- `int dl_list_delete_at (dl_list list, const size_t index)`

- Deletes a list element at a specified index.*
- `int dl_list_find_index (const dl_list list, const void *data)`
Finds the index of the specified data in a list.
- `dl_list_itr dl_list_find_itr (const dl_list list, const void *data)`
Gets an iterator to the specified data in a list.
- `void * dl_list_data (const dl_list list, const size_t index)`
Returns a pointer to the data at a specified index.
- `dl_list_itr dl_list_first (const dl_list list)`
Returns an iterator to the first element of a list.
- `dl_list_itr dl_list_last (const dl_list list)`
Returns an iterator to the last element of a list.
- `dl_list_itr dl_list_next (const dl_list_itr itr)`
Advances a list iterator by one element.
- `dl_list_itr dl_list_prev (const dl_list_itr itr)`
Backs up a list iterator by one element.
- `dl_list_itr dl_list_itr_from_index (const dl_list list, const size_t index)`
Return an iterator to a specified element of a list.
- `dl_list_node dl_list_new_node (void *data)`
Creates a new list node.
- `void dl_list_free_node (dl_list list, dl_list_node node)`
Frees resources for a node and any data.
- `void dl_list_insert_node_front (dl_list list, dl_list_node node)`
Inserts a node at the front of a list.
- `void dl_list_insert_node_before_mid (dl_list list, dl_list_itr itr, dl_list_node node)`
Inserts a node in the middle of a list before a specified iterator.
- `void dl_list_insert_node_after_mid (dl_list list, dl_list_itr itr, dl_list_node node)`
Inserts a node in the middle of a list after a specified iterator.
- `void dl_list_insert_node_back (dl_list list, dl_list_node node)`
Inserts a node at the back of a list.
- `dl_list_node dl_list_remove_at (dl_list list, const size_t index)`
Removes, but does not delete, an element at an index.
- `dl_list_node dl_list_remove_node_front (dl_list list)`
Removes the first node of a list.
- `dl_list_node dl_list_remove_node_mid (dl_list list, dl_list_node node)`
Removes a specified node from the middle of a list.
- `dl_list_node dl_list_remove_node_back (dl_list list)`
Removes the last node of a list.
- `void dl_list_find (const dl_list list, const void *data, dl_list_itr *p_itr, int *p_index)`
Finds the index of, and a pointer to, the first node in the list containing the specified data.

4.13.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.13.2 Function Documentation

4.13.2.1 void dl_list_append (dl_list list, void * data)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.13.2.2 void* dl_list_data (const dl_list list, const size_t index)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

Returns

A pointer to the data, or NULL if the index is out of range.

4.13.2.3 int dl_list_delete_at (dl_list list, const size_t index)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

Returns

0 on success, CDSERR_OUTOFRANGE if the the index is out of range.

4.13.2.4 void dl_list_find (const dl_list list, const void * data, dl_list_itr * p_itr, int * p_index)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This is set to CDSERR_NOTFOUND if the data was not found.
<i>p_index</i>	A pointer to an integer the populate with the result. This is set to NULL if the data was not found.

4.13.2.5 int dl_list_find_index (const dl_list list, const void * data)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

The index of the element, if found, or CDSERR_NOTFOUND if it is not in the list.

4.13.2.6 dl_list_itr dl_list_find_itr (const dl_list list, const void * data)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

An iterator to the found element, or NULL if the element is not in the list.

4.13.2.7 dl_list_itr dl_list_first (const dl_list list)

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.13.2.8 void dl_list_free (dl_list list)

Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

4.13.2.9 void dl_list_free_node (dl_list list, dl_list_node node)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to free.

4.13.2.10 dl_list dl_list_init (int (*)(const void *, const void *) cfunc, void (*)(void *) free_func)

Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a function to free a node. The function should return no value, and accept a <code>void</code> pointer to the node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.

Returns

A pointer to the new list.

4.13.2.11 int dl_list_insert_after (dl_list list, const dl_list_itr itr, void * data)

Note that `dl_list_first()` may return a `NULL` iterator when the list is empty. One reasonable behavior for this function would be to add a new node to the list in that case. However, an iterator may also become `NULL` when advanced

to the end of the list. One possible way to modify this function would be to check the length of this list when the iterator is NULL, and if it is zero, add the first node to the list. However, the semantic meaning of adding an element *after* an iterator breaks down if that that iterator does not point to an existing element. Therefore, it is simpler for this function to simply refuse to handle NULL iterators. It is unlikely a user would want to call this function unless there are already elements in a list, and a valid iterator has been returned, e.g. through a find function.

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_BADITERATOR if `itr` is a NULL pointer.

4.13.2.12 `int dl_list.insert_at (dl_list list, const size_t index, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_OUTOFRANGE if `index` exceeds the length of the list.

4.13.2.13 `int dl_list.insert_before (dl_list list, const dl_list_itr itr, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, CDSERR_BADITERATOR if `itr` is a NULL pointer.

4.13.2.14 `void dl_list.insert_node_after_mid (dl_list list, dl_list_itr itr, dl_list_node node)`

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

4.13.2.15 void dl_list.insert_node_back (dl_list *list*, dl_list_node *node*)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

4.13.2.16 void dl_list.insert_node_before_mid (dl_list *list*, dl_list_itr *itr*, dl_list_node *node*)

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator before which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

4.13.2.17 void dl_list.insert_node_front (dl_list *list*, dl_list_node *node*)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

4.13.2.18 bool dl_list.isempty (const dl_list *list*)

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

`true` if the list is empty, otherwise `false`.

4.13.2.19 dl_list_itr dl_list_itr_from_index (const dl_list *list*, const size_t *index*)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

Returns

The iterator, or NULL if `index` is out of range.

4.13.2.20 dl_list_itr dl_list_last (const dl_list *list*)

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.13.2.21 `size_t dl_list_length (const dl_list list)`**Parameters**

<i>list</i>	A pointer to the list.
-------------	------------------------

4.13.2.22 `dl_list_node dl_list_new_node (void * data)`**Parameters**

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly created node.

4.13.2.23 `dl_list_itr dl_list_next (const dl_list_itr itr)`**Parameters**

<i>itr</i>	The iterator to advance
------------	-------------------------

Returns

The advanced iterator.

4.13.2.24 `void dl_list_prepend (dl_list list, void * data)`**Parameters**

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.13.2.25 `dl_list_itr dl_list_prev (const dl_list_itr itr)`**Parameters**

<i>itr</i>	The iterator to back up.
------------	--------------------------

Returns

The backed up iterator.

4.13.2.26 `dl_list_node dl_list_remove_at (dl_list list, const size_t index)`**Parameters**

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

Returns

A pointer to the removed node. This should be `free()` d by calling `dl_list_free_node()`.

4.13.2.27 `dl_list_node dl_list_remove_node_back (dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

A pointer to the removed node.

4.13.2.28 `dl_list_node dl_list_remove_node_front (dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

A pointer to the removed node.

4.13.2.29 `dl_list_node dl_list_remove_node_mid (dl_list list, dl_list_node node)`

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to remove. As this is removing from the middle, this node should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.

Returns

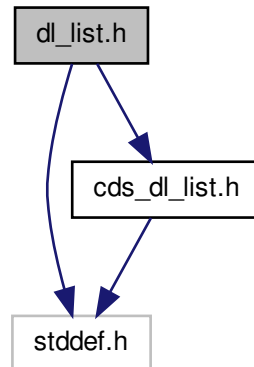
A pointer to the removed node, i.e. equal to `itr`.

4.14 dl_list.h File Reference

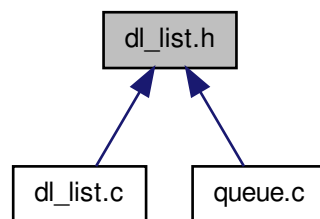
Developer interface to double linked list data structure.

```
#include <stddef.h>
#include "cds_dl_list.h"
```

Include dependency graph for `dl_list.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [dl_list_t](#)
Struct to contain a list.

Typedefs

- typedef struct [dl_list_t](#) [dl_list_t](#)
Struct to contain a list.
- typedef struct [dl_list_node_t](#) * [dl_list_node](#)
Typedef for list node.

Functions

- [dl_list_node](#) [dl_list_new_node](#) (void *data)

- Creates a new list node.*
- void [dl_list_free_node](#) ([dl_list](#) list, [dl_list_node](#) node)
- Frees resources for a node and any data.*
- void [dl_list_insert_node_front](#) ([dl_list](#) list, [dl_list_node](#) node)
- Inserts a node at the front of a list.*
- void [dl_list_insert_node_before_mid](#) ([dl_list](#) list, [dl_list_itr](#) itr, [dl_list_node](#) node)
- Inserts a node in the middle of a list before a specified iterator.*
- void [dl_list_insert_node_after_mid](#) ([dl_list](#) list, [dl_list_itr](#) itr, [dl_list_node](#) node)
- Inserts a node in the middle of a list after a specified iterator.*
- void [dl_list_insert_node_back](#) ([dl_list](#) list, [dl_list_node](#) node)
- Inserts a node at the back of a list.*
- [dl_list_node](#) [dl_list_remove_at](#) ([dl_list](#) list, const [size_t](#) index)
- Removes, but does not delete, an element at an index.*
- [dl_list_node](#) [dl_list_remove_node_front](#) ([dl_list](#) list)
- Removes the first node of a list.*
- [dl_list_node](#) [dl_list_remove_node_mid](#) ([dl_list](#) list, [dl_list_itr](#) itr)
- Removes a specified node from the middle of a list.*
- [dl_list_node](#) [dl_list_remove_node_back](#) ([dl_list](#) list)
- Removes the last node of a list.*
- void [dl_list_find](#) (const [dl_list](#) list, const void *data, [dl_list_itr](#) *p_itr, int *p_index)
- Finds the index of, and a pointer to, the first node in the list containing the specified data.*

4.14.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.14.2 Function Documentation

4.14.2.1 void [dl_list_find](#) (const [dl_list](#) list, const void * data, [dl_list_itr](#) * p_itr, int * p_index)

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This is set to CDSERR_NOTFOUND if the data was not found.
<i>p_index</i>	A pointer to an integer the populate with the result. This is set to NULL if the data was not found.

4.14.2.2 void [dl_list_free_node](#) ([dl_list](#) list, [dl_list_node](#) node)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to free.

4.14.2.3 void dl_list_insert_node_after_mid (dl_list list, dl_list_itr itr, dl_list_node node)

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

4.14.2.4 void dl_list_insert_node_back (dl_list list, dl_list_node node)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

4.14.2.5 void dl_list_insert_node_before_mid (dl_list list, dl_list_itr itr, dl_list_node node)

Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator before which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

4.14.2.6 void dl_list_insert_node_front (dl_list list, dl_list_node node)

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

4.14.2.7 dl_list_node dl_list_new_node (void * data)

Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly created node.

4.14.2.8 dl_list_node dl_list_remove_at (dl_list list, const size_t index)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

Returns

A pointer to the removed node. This should be `free()` d by calling `dl_list_free_node()`.

4.14.2.9 `dl_list_node dl_list_remove_node_back (dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

A pointer to the removed node.

4.14.2.10 `dl_list_node dl_list_remove_node_front (dl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

A pointer to the removed node.

4.14.2.11 `dl_list_node dl_list_remove_node_mid (dl_list list, dl_list_node node)`

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to remove. As this is removing from the middle, this node should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.

Returns

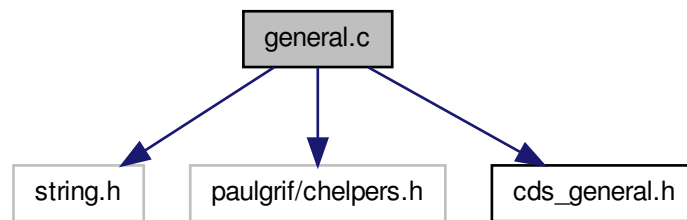
A pointer to the removed node, i.e. equal to `itr`.

4.15 general.c File Reference

Implementation of general data structure helper functions.

```
#include <string.h>
#include <paulgrif/chelpers.h>
#include "cds_general.h"
```

Include dependency graph for general.c:



Functions

- void * `cds_new_int` (const int n)
Dynamically allocates memory for a new int.
- void * `cds_new_uint` (const unsigned int n)
Dynamically allocates memory for a new unsigned int.
- void * `cds_new_long` (const long n)
Dynamically allocates memory for a new long.
- void * `cds_new_ulong` (const unsigned long n)
Dynamically allocates memory for a new unsigned long.
- void * `cds_new_string` (const char *str)
Dynamically allocates memory for a new string.
- int `cds_compare_int` (const void *data, const void *cmp)
Compares two int via void pointers.
- int `cds_compare_uint` (const void *data, const void *cmp)
Compares two unsigned int via void pointers.
- int `cds_compare_long` (const void *data, const void *cmp)
Compares two long via void pointers.
- int `cds_compare_ulong` (const void *data, const void *cmp)
Compares two unsigned long via void pointers.
- int `cds_compare_string` (const void *data, const void *cmp)
Compares two strings via void pointers.

4.15.1 Detailed Description

Implementation of general data structure helper functions.

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.15.2 Function Documentation

4.15.2.1 `int cds_compare_int (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.15.2.2 `int cds_compare_long (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.15.2.3 `int cds_compare_string (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.15.2.4 `int cds_compare_uint (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.15.2.5 `int cds_compare_ulong (const void * data, const void * cmp)`

Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.15.2.6 void* cds_new_int (const int *n*)

Parameters

<i>n</i>	The new int for which to allocate.
----------	------------------------------------

Returns

A void pointer to the allocated memory.

4.15.2.7 void* cds_new_long (const long *n*)

Parameters

<i>n</i>	The new long for which to allocate.
----------	-------------------------------------

Returns

A void pointer to the allocated memory.

4.15.2.8 void* cds_new_string (const char * *str*)

Parameters

<i>str</i>	The new string for which to allocate.
------------	---------------------------------------

Returns

A void pointer to the allocated memory.

4.15.2.9 void* cds_new_uint (const unsigned int *n*)

Parameters

<i>n</i>	The new unsigned int for which to allocate.
----------	---

Returns

A void pointer to the allocated memory.

4.15.2.10 void* cds_new_ulong (const unsigned long *n*)

Parameters

<i>n</i>	The new unsigned long for which to allocate.
----------	--

Returns

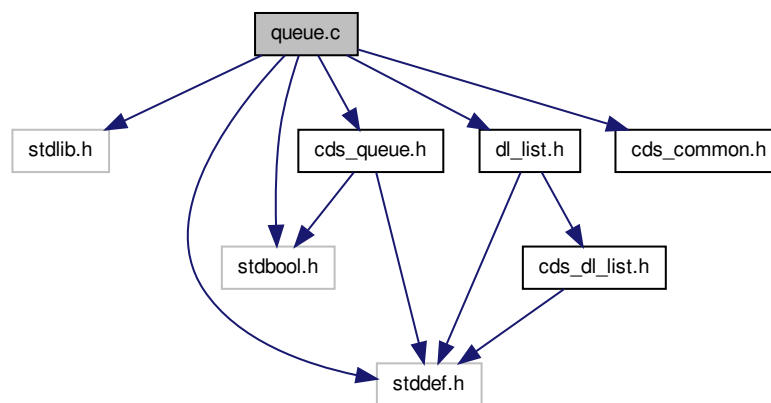
A void pointer to the allocated memory.

4.16 queue.c File Reference

Implementation of queue data structure.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include "cds_queue.h"
#include "dl_list.h"
#include "cds_common.h"
```

Include dependency graph for queue.c:



Functions

- `queue queue_init (void(*free_func)(void *))`
Initializes a new queue.
- `void queue_free (queue que)`
Frees memory and releases resources used by a queue.
- `size_t queue_length (const queue que)`
Gets the number of items in a queue.
- `bool queue_isempty (const queue que)`
Checks if a queue is empty.
- `void * queue_pop (queue que)`
Pops a data item from the queue.
- `void queue_pushback (queue que, void *data)`
Pushes a data item onto the back of the queue.

4.16.1 Detailed Description

Implemented in terms of a doubly linked, double-ended list data structure.

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.16.2 Function Documentation

4.16.2.1 void queue_free (queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

4.16.2.2 queue queue_init (void(*) (void *) *free_func*)

Parameters

<i>free_func</i>	A pointer to a function to free a queue node. The function should return no value, and accept a void pointer to a node. If NULL is specified, the standard <code>free()</code> function is used.
------------------	--

Returns

A pointer to the new queue.

4.16.2.3 bool queue_isempty (const queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

`true` if the queue is empty, `false` if not.

4.16.2.4 size_t queue_length (const queue *que*)

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

The number of items in the queue.

4.16.2.5 void* queue_pop (queue que)

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

Returns

A `void` pointer to the popped data item.

4.16.2.6 void queue_pushback (queue que, void * data)

The provided pointer should point to dynamically allocated memory.

Parameters

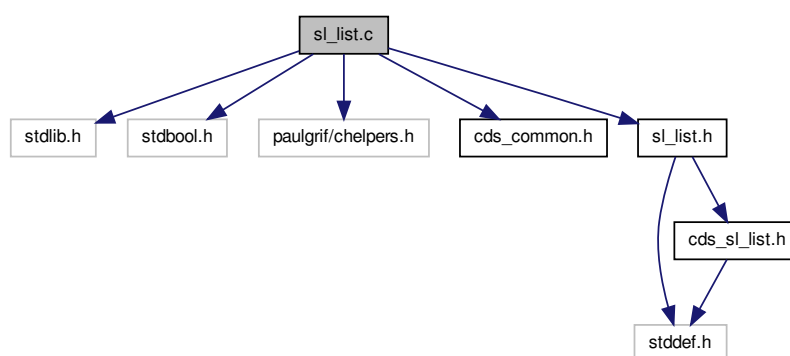
<i>que</i>	A pointer to the queue.
<i>data</i>	A pointer to the data item to be pushed.

4.17 sl_list.c File Reference

Implementation of singly linked list data structure.

```
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "sl_list.h"
```

Include dependency graph for `sl_list.c`:



Functions

- `sl_list sl_list_init` (`int(*cfunc)(const void *, const void *)`, `void(*free_func)(void *)`)
Initializes a new singly linked list.
- `void sl_list_free` (`sl_list list`)

- Frees the resources associated with a list.*
- `size_t sl_list_length` (const `sl_list` list)
- Returns the number of elements in a list.*
- `bool sl_list_isempty` (const `sl_list` list)
- Checks if a list is empty.*
- `void sl_list_prepend` (`sl_list` list, void *data)
- Inserts an element at the beginning of a list.*
- `int sl_list_insert_at` (`sl_list` list, const `size_t` index, void *data)
- Inserts an element at the specified index of a list.*
- `int sl_list_insert_after` (`sl_list` list, const `sl_list_itr` itr, void *data)
- Inserts an element after a provided iterator.*
- `int sl_list_delete_at` (`sl_list` list, const `size_t` index)
- Deletes a list element at a specified index.*
- `int sl_list_find_index` (const `sl_list` list, const void *data)
- Gets an index to the specified data in a list.*
- `sl_list_itr sl_list_find_itr` (const `sl_list` list, const void *data)
- Gets an iterator to the specified data in a list.*
- `void * sl_list_data` (const `sl_list` list, const `size_t` index)
- Returns a pointer to the data at a specified index.*
- `sl_list_itr sl_list_first` (const `sl_list` list)
- Returns an iterator to the first element of a list.*
- `sl_list_itr sl_list_next` (const `sl_list_itr` itr)
- Advances a list iterator by one element.*
- `sl_list_itr sl_list_itr_from_index` (const `sl_list` list, const `size_t` index)
- Return an iterator to a specified element of a list.*
- `sl_list_node sl_list_new_node` (void *data)
- Creates a new list node.*
- `void sl_list_free_node` (`sl_list` list, `sl_list_node` node)
- Frees resources for a node and any data.*
- `sl_list_node sl_list_remove_at` (`sl_list` list, const `size_t` index)
- Removes, but does not delete, an element at an index.*
- `void sl_list_find` (const `sl_list` list, const void *data, `sl_list_itr` *p_itr, int *p_index)
- Gets an index and iterator to a specified piece of data.*

4.17.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.17.2 Function Documentation

4.17.2.1 void* sl_list_data (const sl_list list, const size_t index)

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

Returns

A pointer to the data, or NULL if the index is out of range.

4.17.2.2 `int sl_list_delete_at (sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

Returns

0 on success, CDSERR_OUTOFRANGE if the the index is out of range.

4.17.2.3 `void sl_list_find (const sl_list list, const void * data, sl_list_itr * p_itr, int * p_index)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This parameter is ignored if set to NULL.
<i>p_index</i>	A pointer to an integer index to populate with the result. This parameter is ignored if set to NULL.

4.17.2.4 `int sl_list_find_index (const sl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

The index of the found element, or CDSERR_NOTFOUND if the element is not in the list.

4.17.2.5 `sl_list_itr sl_list_find_itr (const sl_list list, const void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

Returns

An iterator to the found element, or NULL is the element is not in the list.

4.17.2.6 `sl_list_itr sl_list_first (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

An iterator to the first element.

4.17.2.7 void sl_list_free (sl_list list)**Parameters**

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

4.17.2.8 void sl_list_free_node (sl_list list, sl_list_node node)**Parameters**

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to free.

4.17.2.9 sl_list sl_list_init (int(*) (const void *, const void *) cfunc, void(*) (void *) free_func)**Parameters**

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
<i>free_func</i>	A pointer to a function for freeing a node. The function should return no value, and accept a <code>void</code> pointer to the node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.

Returns

A pointer to the new list.

4.17.2.10 int sl_list_insert_after (sl_list list, const sl_list_itr itr, void * data)**Parameters**

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, `CDSERR_BADITERATOR` if `itr` is a `NULL` pointer.

4.17.2.11 int sl_list_insert_at (sl_list list, const size_t index, void * data)**Parameters**

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

Returns

0 on success, `CDSERR_OUTOFRANGE` if `index` exceeds the length of the list.

4.17.2.12 `bool sl_list_isempty (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

`true` if the list is empty, otherwise `false`.

4.17.2.13 `sl_list_itr sl_list_itr_from_index (const sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

Returns

The iterator, or `NULL` if `index` is out of range.

4.17.2.14 `size_t sl_list_length (const sl_list list)`

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

4.17.2.15 `sl_list_node sl_list_new_node (void * data)`

Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly created node.

4.17.2.16 `sl_list_itr sl_list_next (const sl_list_itr itr)`

Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

Returns

The advanced iterator.

4.17.2.17 `void sl_list_prepend (sl_list list, void * data)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.17.2.18 `sl_list_node sl_list_remove_at (sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

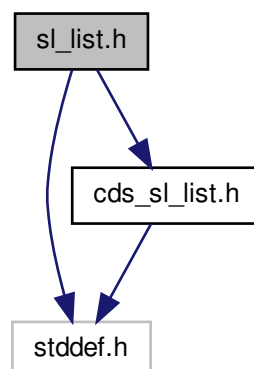
Returns

A pointer to the removed node. This should be `free()` d by calling [sl_list_free_node\(\)](#).

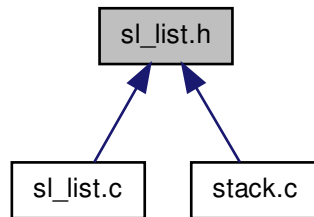
4.18 `sl_list.h` File Reference

Developer interface to singly linked list data structure.

```
#include <stddef.h>
#include "cds_sl_list.h"
Include dependency graph for sl_list.h:
```



This graph shows which files directly or indirectly include this file:



Data Structures

- struct `sl_list_t`
Struct to contain a list.

Typedefs

- typedef struct `sl_list_t` `sl_list_t`
Struct to contain a list.
- typedef struct `sl_list_node_t` * `sl_list_node`
Typedef for list node.

Functions

- `sl_list_node` `sl_list_new_node` (void *data)
Creates a new list node.
- void `sl_list_free_node` (`sl_list` list, `sl_list_node` node)
Frees resources for a node and any data.
- `sl_list_node` `sl_list_remove_at` (`sl_list` list, const size_t index)
Removes, but does not delete, an element at an index.
- void `sl_list_find` (const `sl_list` list, const void *data, `sl_list_itr` *p_itr, int *p_index)
Gets an index and iterator to a specified piece of data.

4.18.1 Detailed Description

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.18.2 Function Documentation

4.18.2.1 `void sl_list_find (const sl_list list, const void * data, sl_list_itr * p_itr, int * p_index)`

Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This parameter is ignored if set to NULL.
<i>p_index</i>	A pointer to an integer index to populate with the result. This parameter is ignored if set to NULL.

4.18.2.2 `void sl_list_free_node (sl_list list, sl_list_node node)`

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to free.

4.18.2.3 `sl_list_node sl_list_new_node (void * data)`

Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

Returns

A pointer to the newly created node.

4.18.2.4 `sl_list_node sl_list_remove_at (sl_list list, const size_t index)`

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

Returns

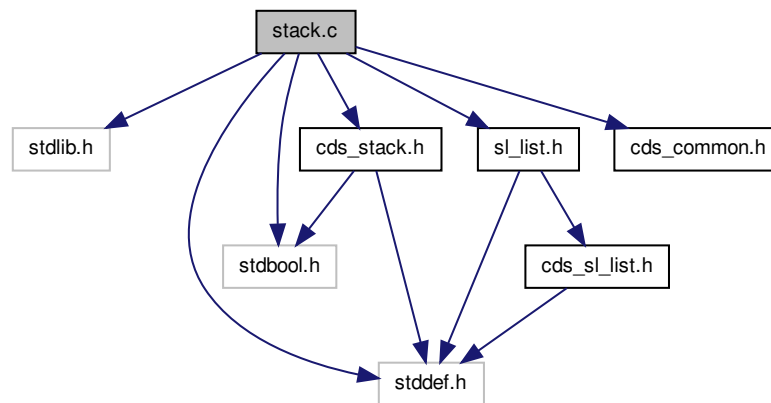
A pointer to the removed node. This should be `free()` d by calling [sl_list_free_node\(\)](#).

4.19 stack.c File Reference

Implementation of stack data structure.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include "cds_stack.h"
#include "sl_list.h"
#include "cds_common.h"
```

Include dependency graph for stack.c:



Functions

- `stack stack_init (void(*free_func)(void *))`
Initializes a new stack.
- `void stack_free (stack stk)`
Frees memory and releases resources used by a stack.
- `size_t stack_length (const stack stk)`
Gets the number of items in a stack.
- `bool stack_isempty (const stack stk)`
Checks if a stack is empty.
- `void * stack_pop (stack stk)`
Pops a data item from the stack.
- `void stack_push (stack stk, void *data)`
Pushes a data item onto the stack.

4.19.1 Detailed Description

Implemented in terms of a singly linked, singled-ended list data structure.

Author

Paul Griffiths

Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.19.2 Function Documentation

4.19.2.1 void stack_free (stack stk)

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

4.19.2.2 `stack stack_init (void(*) (void *) free_func)`

Parameters

<i>free_func</i>	A pointer to a function a free a stack node. The function should return no value, and accept a <code>void</code> pointer to a node. If <code>NULL</code> is specified, the standard <code>free()</code> function is used.
------------------	---

Returns

A pointer to the new stack.

4.19.2.3 `bool stack_isempty (const stack stk)`

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

`true` is the stack is empty, `false` if not.

4.19.2.4 `size_t stack_length (const stack stk)`

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

The number of items in the stack.

4.19.2.5 `void* stack_pop (stack stk)`

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

Returns

A `void` pointer to the popped data item.

4.19.2.6 `void stack_push (stack stk, void * data)`

The provided pointer should point to dynamically allocated memory.

Parameters

<i>stk</i>	A pointer to the stack.
<i>data</i>	A pointer to the data item to be pushed.

Index

- back
 - dl_list_t, 8
- bs_tree.c, 13
 - bs_tree_free, 14
 - bs_tree_free_subtree, 14
 - bs_tree_init, 14
 - bs_tree_insert, 15
 - bs_tree_insert_subtree, 15
 - bs_tree_isempty, 15
 - bs_tree_length, 16
 - bs_tree_new_node, 16
 - bs_tree_search, 16
 - bs_tree_search_data, 16
 - bs_tree_search_node, 16
 - bst_insert_search, 17
- bs_tree.h, 17
 - bs_tree_free_subtree, 19
 - bs_tree_insert_subtree, 19
 - bs_tree_new_node, 19
 - bs_tree_search_node, 19
 - bst_insert_search, 19
- bs_tree_free
 - bs_tree.c, 14
 - cds_bs_tree.h, 25
- bs_tree_free_subtree
 - bs_tree.c, 14
 - bs_tree.h, 19
- bs_tree_init
 - bs_tree.c, 14
 - cds_bs_tree.h, 25
- bs_tree_insert
 - bs_tree.c, 15
 - cds_bs_tree.h, 25
- bs_tree_insert_subtree
 - bs_tree.c, 15
 - bs_tree.h, 19
- bs_tree_isempty
 - bs_tree.c, 15
 - cds_bs_tree.h, 25
- bs_tree_length
 - bs_tree.c, 16
 - cds_bs_tree.h, 26
- bs_tree_new_node
 - bs_tree.c, 16
 - bs_tree.h, 19
- bs_tree_node_t, 5
 - data, 5
 - left, 5
 - right, 5
- bs_tree_search
 - bs_tree.c, 16
 - cds_bs_tree.h, 26
- bs_tree_search_data
 - bs_tree.c, 16
 - cds_bs_tree.h, 26
- bs_tree_search_node
 - bs_tree.c, 16
 - bs_tree.h, 19
- bs_tree_t, 6
 - cfunc, 6
 - free_func, 6
 - length, 6
 - root, 6
- bst_insert_search
 - bs_tree.c, 17
 - bs_tree.h, 19
- bst_map.c, 20
 - bst_map_free, 21
 - bst_map_init, 21
 - bst_map_insert, 21
 - bst_map_isempty, 21
 - bst_map_length, 22
 - bst_map_search, 22
 - bst_map_search_data, 22
- bst_map_free
 - bst_map.c, 21
 - cds_bst_map.h, 28
- bst_map_init
 - bst_map.c, 21
 - cds_bst_map.h, 28
- bst_map_insert
 - bst_map.c, 21
 - cds_bst_map.h, 28
- bst_map_isempty
 - bst_map.c, 21
 - cds_bst_map.h, 28
- bst_map_length
 - bst_map.c, 22
 - cds_bst_map.h, 28
- bst_map_search
 - bst_map.c, 22
 - cds_bst_map.h, 29
- bst_map_search_data
 - bst_map.c, 22
 - cds_bst_map.h, 29
- CDSERR_BADITERATOR
 - cds_common.h, 30
- CDSERR_ERROR

- [cds_common.h](#), [30](#)
- [CDSERR_NOTFOUND](#)
 - [cds_common.h](#), [30](#)
- [CDSERR_OUTOFRANGE](#)
 - [cds_common.h](#), [30](#)
- [cdatastruct.h](#), [22](#)
- [cds_common.h](#)
 - [CDSERR_BADITERATOR](#), [30](#)
 - [CDSERR_ERROR](#), [30](#)
 - [CDSERR_NOTFOUND](#), [30](#)
 - [CDSERR_OUTOFRANGE](#), [30](#)
- [cds_bs_tree.h](#), [23](#)
 - [bs_tree_free](#), [25](#)
 - [bs_tree_init](#), [25](#)
 - [bs_tree_insert](#), [25](#)
 - [bs_tree_isempty](#), [25](#)
 - [bs_tree_length](#), [26](#)
 - [bs_tree_search](#), [26](#)
 - [bs_tree_search_data](#), [26](#)
- [cds_bst_map.h](#), [26](#)
 - [bst_map_free](#), [28](#)
 - [bst_map_init](#), [28](#)
 - [bst_map_insert](#), [28](#)
 - [bst_map_isempty](#), [28](#)
 - [bst_map_length](#), [28](#)
 - [bst_map_search](#), [29](#)
 - [bst_map_search_data](#), [29](#)
- [cds_common.h](#), [29](#)
 - [cds_error](#), [30](#)
- [cds_compare_int](#)
 - [cds_general.h](#), [37](#)
 - [general.c](#), [63](#)
- [cds_compare_long](#)
 - [cds_general.h](#), [37](#)
 - [general.c](#), [63](#)
- [cds_compare_string](#)
 - [cds_general.h](#), [38](#)
 - [general.c](#), [63](#)
- [cds_compare_uint](#)
 - [cds_general.h](#), [38](#)
 - [general.c](#), [63](#)
- [cds_compare_ulong](#)
 - [cds_general.h](#), [38](#)
 - [general.c](#), [63](#)
- [cds_dl_list.h](#), [30](#)
 - [dl_list_append](#), [32](#)
 - [dl_list_data](#), [32](#)
 - [dl_list_delete_at](#), [33](#)
 - [dl_list_find_index](#), [33](#)
 - [dl_list_find_itr](#), [33](#)
 - [dl_list_first](#), [33](#)
 - [dl_list_free](#), [33](#)
 - [dl_list_init](#), [33](#)
 - [dl_list_insert_after](#), [34](#)
 - [dl_list_insert_at](#), [34](#)
 - [dl_list_insert_before](#), [34](#)
 - [dl_list_isempty](#), [35](#)
 - [dl_list_itr_from_index](#), [35](#)
 - [dl_list_last](#), [35](#)
 - [dl_list_length](#), [35](#)
 - [dl_list_next](#), [35](#)
 - [dl_list_prepend](#), [36](#)
 - [dl_list_prev](#), [36](#)
- [cds_error](#)
 - [cds_common.h](#), [30](#)
- [cds_general.h](#), [36](#)
 - [cds_compare_int](#), [37](#)
 - [cds_compare_long](#), [37](#)
 - [cds_compare_string](#), [38](#)
 - [cds_compare_uint](#), [38](#)
 - [cds_compare_ulong](#), [38](#)
 - [cds_new_int](#), [38](#)
 - [cds_new_long](#), [38](#)
 - [cds_new_string](#), [39](#)
 - [cds_new_uint](#), [39](#)
 - [cds_new_ulong](#), [39](#)
- [cds_new_int](#)
 - [cds_general.h](#), [38](#)
 - [general.c](#), [64](#)
- [cds_new_long](#)
 - [cds_general.h](#), [38](#)
 - [general.c](#), [64](#)
- [cds_new_string](#)
 - [cds_general.h](#), [39](#)
 - [general.c](#), [64](#)
- [cds_new_uint](#)
 - [cds_general.h](#), [39](#)
 - [general.c](#), [64](#)
- [cds_new_ulong](#)
 - [cds_general.h](#), [39](#)
 - [general.c](#), [64](#)
- [cds_queue.h](#), [39](#)
 - [queue_free](#), [41](#)
 - [queue_init](#), [41](#)
 - [queue_isempty](#), [41](#)
 - [queue_length](#), [41](#)
 - [queue_pop](#), [41](#)
 - [queue_pushback](#), [42](#)
- [cds_sl_list.h](#), [42](#)
 - [sl_list_data](#), [44](#)
 - [sl_list_delete_at](#), [44](#)
 - [sl_list_find_index](#), [44](#)
 - [sl_list_find_itr](#), [45](#)
 - [sl_list_first](#), [45](#)
 - [sl_list_free](#), [45](#)
 - [sl_list_init](#), [45](#)
 - [sl_list_insert_after](#), [45](#)
 - [sl_list_insert_at](#), [46](#)
 - [sl_list_isempty](#), [46](#)
 - [sl_list_itr_from_index](#), [46](#)
 - [sl_list_length](#), [46](#)
 - [sl_list_next](#), [46](#)
 - [sl_list_prepend](#), [47](#)
- [cds_stack.h](#), [47](#)
 - [stack_free](#), [48](#)
 - [stack_init](#), [49](#)

- stack_isempty, 49
- stack_length, 49
- stack_pop, 49
- stack_push, 49
- cfunc
 - bs_tree_t, 6
 - dl_list_t, 8
 - sl_list_t, 10
- data
 - bs_tree_node_t, 5
 - dl_list_node_t, 7
 - sl_list_node_t, 9
- dl_list.c, 50
 - dl_list_append, 52
 - dl_list_data, 52
 - dl_list_delete_at, 52
 - dl_list_find, 52
 - dl_list_find_index, 52
 - dl_list_find_itr, 52
 - dl_list_first, 53
 - dl_list_free, 53
 - dl_list_free_node, 53
 - dl_list_init, 53
 - dl_list_insert_after, 53
 - dl_list_insert_at, 54
 - dl_list_insert_before, 54
 - dl_list_insert_node_after_mid, 54
 - dl_list_insert_node_back, 54
 - dl_list_insert_node_before_mid, 55
 - dl_list_insert_node_front, 55
 - dl_list_isempty, 55
 - dl_list_itr_from_index, 55
 - dl_list_last, 55
 - dl_list_length, 56
 - dl_list_new_node, 56
 - dl_list_next, 56
 - dl_list_prepend, 56
 - dl_list_prev, 56
 - dl_list_remove_at, 56
 - dl_list_remove_node_back, 57
 - dl_list_remove_node_front, 57
 - dl_list_remove_node_mid, 57
- dl_list.h, 57
 - dl_list_find, 59
 - dl_list_free_node, 59
 - dl_list_insert_node_after_mid, 59
 - dl_list_insert_node_back, 60
 - dl_list_insert_node_before_mid, 60
 - dl_list_insert_node_front, 60
 - dl_list_new_node, 60
 - dl_list_remove_at, 60
 - dl_list_remove_node_back, 61
 - dl_list_remove_node_front, 61
 - dl_list_remove_node_mid, 61
- dl_list_append
 - cds_dl_list.h, 32
 - dl_list.c, 52
- dl_list_data
 - cds_dl_list.h, 32
 - dl_list.c, 52
- dl_list_delete_at
 - cds_dl_list.h, 33
 - dl_list.c, 52
- dl_list_find
 - dl_list.c, 52
 - dl_list.h, 59
- dl_list_find_index
 - cds_dl_list.h, 33
 - dl_list.c, 52
- dl_list_find_itr
 - cds_dl_list.h, 33
 - dl_list.c, 52
- dl_list_first
 - cds_dl_list.h, 33
 - dl_list.c, 53
- dl_list_free
 - cds_dl_list.h, 33
 - dl_list.c, 53
- dl_list_free_node
 - dl_list.c, 53
 - dl_list.h, 59
- dl_list_init
 - cds_dl_list.h, 33
 - dl_list.c, 53
- dl_list_insert_after
 - cds_dl_list.h, 34
 - dl_list.c, 53
- dl_list_insert_at
 - cds_dl_list.h, 34
 - dl_list.c, 54
- dl_list_insert_before
 - cds_dl_list.h, 34
 - dl_list.c, 54
- dl_list_insert_node_after_mid
 - dl_list.c, 54
 - dl_list.h, 59
- dl_list_insert_node_back
 - dl_list.c, 54
 - dl_list.h, 60
- dl_list_insert_node_before_mid
 - dl_list.c, 55
 - dl_list.h, 60
- dl_list_insert_node_front
 - dl_list.c, 55
 - dl_list.h, 60
- dl_list_isempty
 - cds_dl_list.h, 35
 - dl_list.c, 55
- dl_list_itr_from_index
 - cds_dl_list.h, 35
 - dl_list.c, 55
- dl_list_last
 - cds_dl_list.h, 35
 - dl_list.c, 55
- dl_list_length
 - cds_dl_list.h, 35

- dl_list.c, 56
- dl_list_new_node
 - dl_list.c, 56
 - dl_list.h, 60
- dl_list_next
 - cds_dl_list.h, 35
 - dl_list.c, 56
- dl_list_node_t, 7
 - data, 7
 - next, 7
 - prev, 7
- dl_list_prepend
 - cds_dl_list.h, 36
 - dl_list.c, 56
- dl_list_prev
 - cds_dl_list.h, 36
 - dl_list.c, 56
- dl_list_remove_at
 - dl_list.c, 56
 - dl_list.h, 60
- dl_list_remove_node_back
 - dl_list.c, 57
 - dl_list.h, 61
- dl_list_remove_node_front
 - dl_list.c, 57
 - dl_list.h, 61
- dl_list_remove_node_mid
 - dl_list.c, 57
 - dl_list.h, 61
- dl_list_t, 7
 - back, 8
 - cfunc, 8
 - free_func, 8
 - front, 8
 - length, 8
- free_func
 - bs_tree_t, 6
 - dl_list_t, 8
 - sl_list_t, 10
- front
 - dl_list_t, 8
 - sl_list_t, 10
- general.c, 61
 - cds_compare_int, 63
 - cds_compare_long, 63
 - cds_compare_string, 63
 - cds_compare_uint, 63
 - cds_compare_ulong, 63
 - cds_new_int, 64
 - cds_new_long, 64
 - cds_new_string, 64
 - cds_new_uint, 64
 - cds_new_ulong, 64
- key
 - key, 9
 - value, 9
- key
 - keypair_t, 9
- keypair_t, 9
- key, 9
- value, 9
- left
 - bs_tree_node_t, 5
- length
 - bs_tree_t, 6
 - dl_list_t, 8
 - sl_list_t, 10
- next
 - dl_list_node_t, 7
 - sl_list_node_t, 9
- prev
 - dl_list_node_t, 7
- queue.c, 65
 - queue_free, 66
 - queue_init, 66
 - queue_isempty, 66
 - queue_length, 66
 - queue_pop, 66
 - queue_pushback, 67
- queue_free
 - cds_queue.h, 41
 - queue.c, 66
- queue_init
 - cds_queue.h, 41
 - queue.c, 66
- queue_isempty
 - cds_queue.h, 41
 - queue.c, 66
- queue_length
 - cds_queue.h, 41
 - queue.c, 66
- queue_pop
 - cds_queue.h, 41
 - queue.c, 66
- queue_pushback
 - cds_queue.h, 42
 - queue.c, 67
- right
 - bs_tree_node_t, 5
- root
 - bs_tree_t, 6
- sl_list.c, 67
 - sl_list_data, 68
 - sl_list_delete_at, 69
 - sl_list_find, 69
 - sl_list_find_index, 69
 - sl_list_find_itr, 69
 - sl_list_first, 69
 - sl_list_free, 70
 - sl_list_free_node, 70
 - sl_list_init, 70
 - sl_list_insert_after, 70
 - sl_list_insert_at, 70

- sl_list_isempty, 71
- sl_list_itr_from_index, 71
- sl_list_length, 71
- sl_list_new_node, 71
- sl_list_next, 71
- sl_list_prepend, 71
- sl_list_remove_at, 72
- sl_list.h, 72
 - sl_list_find, 74
 - sl_list_free_node, 74
 - sl_list_new_node, 74
 - sl_list_remove_at, 74
- sl_list_data
 - cds_sl_list.h, 44
 - sl_list.c, 68
- sl_list_delete_at
 - cds_sl_list.h, 44
 - sl_list.c, 69
- sl_list_find
 - sl_list.c, 69
 - sl_list.h, 74
- sl_list_find_index
 - cds_sl_list.h, 44
 - sl_list.c, 69
- sl_list_find_itr
 - cds_sl_list.h, 45
 - sl_list.c, 69
- sl_list_first
 - cds_sl_list.h, 45
 - sl_list.c, 69
- sl_list_free
 - cds_sl_list.h, 45
 - sl_list.c, 70
- sl_list_free_node
 - sl_list.c, 70
 - sl_list.h, 74
- sl_list_init
 - cds_sl_list.h, 45
 - sl_list.c, 70
- sl_list_insert_after
 - cds_sl_list.h, 45
 - sl_list.c, 70
- sl_list_insert_at
 - cds_sl_list.h, 46
 - sl_list.c, 70
- sl_list_isempty
 - cds_sl_list.h, 46
 - sl_list.c, 71
- sl_list_itr_from_index
 - cds_sl_list.h, 46
 - sl_list.c, 71
- sl_list_length
 - cds_sl_list.h, 46
 - sl_list.c, 71
- sl_list_new_node
 - sl_list.c, 71
 - sl_list.h, 74
- sl_list_next
 - cds_sl_list.h, 46
 - sl_list.c, 71
- sl_list_node_t, 9
 - data, 9
 - next, 9
- sl_list_prepend
 - cds_sl_list.h, 47
 - sl_list.c, 71
- sl_list_remove_at
 - sl_list.c, 72
 - sl_list.h, 74
- sl_list_t, 10
 - cfunc, 10
 - free_func, 10
 - front, 10
 - length, 10
- stack.c, 74
 - stack_free, 75
 - stack_init, 76
 - stack_isempty, 76
 - stack_length, 76
 - stack_pop, 76
 - stack_push, 76
- stack_free
 - cds_stack.h, 48
 - stack.c, 75
- stack_init
 - cds_stack.h, 49
 - stack.c, 76
- stack_isempty
 - cds_stack.h, 49
 - stack.c, 76
- stack_length
 - cds_stack.h, 49
 - stack.c, 76
- stack_pop
 - cds_stack.h, 49
 - stack.c, 76
- stack_push
 - cds_stack.h, 49
 - stack.c, 76
- value
 - kvpair_t, 9