

cdatastruct

Generated by Doxygen 1.8.1.2

Tue Sep 10 2013 20:30:12

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	Detailed Description	5
3.1.2	Field Documentation	5
3.1.2.1	data	5
3.1.2.2	left	5
3.1.2.3	right	6
3.2	bs_tree_t Struct Reference	6
3.2.1	Detailed Description	6
3.2.2	Field Documentation	6
3.2.2.1	cfunc	6
3.2.2.2	free_func	6
3.2.2.3	length	7
3.2.2.4	mutex	7
3.2.2.5	root	7
3.3	da_stack_t Struct Reference	7
3.3.1	Detailed Description	7
3.3.2	Field Documentation	7
3.3.2.1	length	7
3.3.2.2	stack	7
3.3.2.3	top	7
3.4	dl_list_node_t Struct Reference	8
3.4.1	Detailed Description	8
3.4.2	Field Documentation	8
3.4.2.1	data	8
3.4.2.2	next	8

3.4.2.3	prev	8
3.5	dl_list_t Struct Reference	8
3.5.1	Detailed Description	9
3.5.2	Field Documentation	9
3.5.2.1	back	9
3.5.2.2	cfunc	9
3.5.2.3	free_func	9
3.5.2.4	front	9
3.5.2.5	length	10
3.5.2.6	mutex	10
3.6	ia_stack_t Struct Reference	10
3.6.1	Detailed Description	10
3.6.2	Field Documentation	10
3.6.2.1	length	10
3.6.2.2	stack	10
3.6.2.3	top	10
3.7	kvpair_t Struct Reference	10
3.7.1	Detailed Description	11
3.7.2	Field Documentation	11
3.7.2.1	key	11
3.7.2.2	value	11
3.8	sl_list_node_t Struct Reference	11
3.8.1	Detailed Description	11
3.8.2	Field Documentation	12
3.8.2.1	data	12
3.8.2.2	next	12
3.9	sl_list_t Struct Reference	12
3.9.1	Detailed Description	12
3.9.2	Field Documentation	12
3.9.2.1	cfunc	12
3.9.2.2	free_func	13
3.9.2.3	front	13
3.9.2.4	length	13
3.9.2.5	mutex	13
4	File Documentation	15
4.1	bs_tree.c File Reference	15
4.1.1	Detailed Description	17
4.1.2	Function Documentation	17
4.1.2.1	bs_tree_free	17

4.1.2.2	bs_tree_free_subtree	17
4.1.2.3	bs_tree_init	17
4.1.2.4	bs_tree_inorder_left_traverse	18
4.1.2.5	bs_tree_inorder_left_traverse_int	18
4.1.2.6	bs_tree_inorder_right_traverse	18
4.1.2.7	bs_tree_inorder_right_traverse_int	18
4.1.2.8	bs_tree_insert	18
4.1.2.9	bs_tree_insert_search	19
4.1.2.10	bs_tree_insert_subtree	19
4.1.2.11	bs_tree_isempty	19
4.1.2.12	bs_tree_length	20
4.1.2.13	bs_tree_lock	20
4.1.2.14	bs_tree_new_node	20
4.1.2.15	bs_tree_postorder_left_traverse	20
4.1.2.16	bs_tree_postorder_left_traverse_int	20
4.1.2.17	bs_tree_postorder_right_traverse	21
4.1.2.18	bs_tree_postorder_right_traverse_int	21
4.1.2.19	bs_tree_preorder_left_traverse	21
4.1.2.20	bs_tree_preorder_left_traverse_int	21
4.1.2.21	bs_tree_preorder_right_traverse	22
4.1.2.22	bs_tree_preorder_right_traverse_int	22
4.1.2.23	bs_tree_search	22
4.1.2.24	bs_tree_search_data	22
4.1.2.25	bs_tree_search_node	22
4.1.2.26	bs_tree_unlock	23
4.2	bs_tree.h File Reference	23
4.2.1	Detailed Description	25
4.2.2	Function Documentation	25
4.2.2.1	bs_tree_free_subtree	25
4.2.2.2	bs_tree_inorder_left_traverse_int	25
4.2.2.3	bs_tree_inorder_right_traverse_int	26
4.2.2.4	bs_tree_insert_search	26
4.2.2.5	bs_tree_insert_subtree	26
4.2.2.6	bs_tree_new_node	26
4.2.2.7	bs_tree_postorder_left_traverse_int	27
4.2.2.8	bs_tree_postorder_right_traverse_int	27
4.2.2.9	bs_tree_preorder_left_traverse_int	27
4.2.2.10	bs_tree_preorder_right_traverse_int	27
4.2.2.11	bs_tree_search_node	28
4.3	bst_map.c File Reference	28

4.3.1	Detailed Description	29
4.3.2	Function Documentation	29
4.3.2.1	bst_map_free	29
4.3.2.2	bst_map_init	29
4.3.2.3	bst_map_insert	30
4.3.2.4	bst_map_isempty	30
4.3.2.5	bst_map_length	30
4.3.2.6	bst_map_lock	30
4.3.2.7	bst_map_search	31
4.3.2.8	bst_map_search_data	31
4.3.2.9	bst_map_unlock	31
4.4	cdatastruct.h File Reference	31
4.4.1	Detailed Description	32
4.5	cds_bs_tree.h File Reference	32
4.5.1	Detailed Description	34
4.5.2	Function Documentation	34
4.5.2.1	bs_tree_free	34
4.5.2.2	bs_tree_init	34
4.5.2.3	bs_tree_inorder_left_traverse	35
4.5.2.4	bs_tree_inorder_right_traverse	35
4.5.2.5	bs_tree_insert	35
4.5.2.6	bs_tree_isempty	35
4.5.2.7	bs_tree_length	36
4.5.2.8	bs_tree_lock	36
4.5.2.9	bs_tree_postorder_left_traverse	36
4.5.2.10	bs_tree_postorder_right_traverse	36
4.5.2.11	bs_tree_preorder_left_traverse	36
4.5.2.12	bs_tree_preorder_right_traverse	37
4.5.2.13	bs_tree_search	37
4.5.2.14	bs_tree_search_data	37
4.5.2.15	bs_tree_unlock	37
4.6	cds_bst_map.h File Reference	37
4.6.1	Detailed Description	39
4.6.2	Function Documentation	39
4.6.2.1	bst_map_free	39
4.6.2.2	bst_map_init	39
4.6.2.3	bst_map_insert	39
4.6.2.4	bst_map_isempty	40
4.6.2.5	bst_map_length	40
4.6.2.6	bst_map_lock	40

4.6.2.7	bst_map_search	40
4.6.2.8	bst_map_search_data	40
4.6.2.9	bst_map_unlock	41
4.7	cds_common.h File Reference	41
4.7.1	Detailed Description	41
4.7.2	Enumeration Type Documentation	42
4.7.2.1	cds_error	42
4.8	cds_da_stack.h File Reference	42
4.8.1	Detailed Description	43
4.8.2	Function Documentation	44
4.8.2.1	da_stack_free	44
4.8.2.2	da_stack_init	44
4.8.2.3	da_stack_isfull	44
4.8.2.4	da_stack_peek	44
4.8.2.5	da_stack_pop	44
4.8.2.6	da_stack_push	45
4.8.2.7	is_stack_isempty	45
4.9	cds_dl_list.h File Reference	45
4.9.1	Detailed Description	47
4.9.2	Function Documentation	47
4.9.2.1	dl_list_append	47
4.9.2.2	dl_list_data	48
4.9.2.3	dl_list_delete_at	48
4.9.2.4	dl_list_find_index	48
4.9.2.5	dl_list_find_itr	48
4.9.2.6	dl_list_first	49
4.9.2.7	dl_list_free	49
4.9.2.8	dl_list_init	49
4.9.2.9	dl_list_insert_after	49
4.9.2.10	dl_list_insert_at	50
4.9.2.11	dl_list_insert_before	50
4.9.2.12	dl_list_isempty	50
4.9.2.13	dl_list_itr_from_index	50
4.9.2.14	dl_list_last	51
4.9.2.15	dl_list_length	51
4.9.2.16	dl_list_lock	51
4.9.2.17	dl_list_next	51
4.9.2.18	dl_list_prepend	51
4.9.2.19	dl_list_prev	52
4.9.2.20	dl_list_unlock	52

4.10	cds_general.h File Reference	52
4.10.1	Detailed Description	53
4.10.2	Function Documentation	53
4.10.2.1	cds_compare_double	53
4.10.2.2	cds_compare_float	54
4.10.2.3	cds_compare_int	54
4.10.2.4	cds_compare_long	54
4.10.2.5	cds_compare_longlong	54
4.10.2.6	cds_compare_string	55
4.10.2.7	cds_compare_uint	55
4.10.2.8	cds_compare_ulong	55
4.10.2.9	cds_compare_ulonglong	55
4.10.2.10	cds_new_double	56
4.10.2.11	cds_new_float	56
4.10.2.12	cds_new_int	56
4.10.2.13	cds_new_long	56
4.10.2.14	cds_new_longlong	57
4.10.2.15	cds_new_string	57
4.10.2.16	cds_new_uint	57
4.10.2.17	cds_new_ulong	57
4.10.2.18	cds_new_ulonglong	57
4.11	cds_ia_stack.h File Reference	58
4.11.1	Detailed Description	59
4.11.2	Function Documentation	59
4.11.2.1	ia_stack_free	59
4.11.2.2	ia_stack_init	59
4.11.2.3	ia_stack_isfull	59
4.11.2.4	ia_stack_peek	60
4.11.2.5	ia_stack_pop	60
4.11.2.6	ia_stack_push	60
4.11.2.7	is_stack_isempty	60
4.12	cds_queue.h File Reference	61
4.12.1	Detailed Description	62
4.12.2	Function Documentation	62
4.12.2.1	queue_free	62
4.12.2.2	queue_init	62
4.12.2.3	queue_isempty	62
4.12.2.4	queue_length	63
4.12.2.5	queue_lock	63
4.12.2.6	queue_pop	63

4.12.2.7	queue_pushback	63
4.12.2.8	queue_unlock	64
4.13	cds_sl_list.h File Reference	64
4.13.1	Detailed Description	66
4.13.2	Function Documentation	66
4.13.2.1	sl_list_data	66
4.13.2.2	sl_list_delete_at	66
4.13.2.3	sl_list_find_index	66
4.13.2.4	sl_list_find_itr	67
4.13.2.5	sl_list_first	67
4.13.2.6	sl_list_free	67
4.13.2.7	sl_list_init	67
4.13.2.8	sl_list_insert_after	67
4.13.2.9	sl_list_insert_at	68
4.13.2.10	sl_list_isempty	68
4.13.2.11	sl_list_itr_from_index	68
4.13.2.12	sl_list_length	69
4.13.2.13	sl_list_lock	69
4.13.2.14	sl_list_next	69
4.13.2.15	sl_list_prepend	69
4.13.2.16	sl_list_unlock	69
4.14	cds_stack.h File Reference	69
4.14.1	Detailed Description	71
4.14.2	Function Documentation	71
4.14.2.1	stack_free	71
4.14.2.2	stack_init	71
4.14.2.3	stack_isempty	71
4.14.2.4	stack_length	72
4.14.2.5	stack_lock	72
4.14.2.6	stack_peek	72
4.14.2.7	stack_pop	72
4.14.2.8	stack_push	73
4.14.2.9	stack_unlock	73
4.15	da_stack.c File Reference	73
4.15.1	Detailed Description	74
4.15.2	Function Documentation	74
4.15.2.1	da_stack_free	74
4.15.2.2	da_stack_init	74
4.15.2.3	da_stack_isfull	75
4.15.2.4	da_stack_peek	75

4.15.2.5	<code>da_stack_pop</code>	75
4.15.2.6	<code>da_stack_push</code>	75
4.15.2.7	<code>is_stack_isempty</code>	76
4.16	<code>dl_list.c</code> File Reference	76
4.16.1	Detailed Description	78
4.16.2	Function Documentation	78
4.16.2.1	<code>dl_list_append</code>	78
4.16.2.2	<code>dl_list_data</code>	78
4.16.2.3	<code>dl_list_delete_at</code>	78
4.16.2.4	<code>dl_list_find</code>	78
4.16.2.5	<code>dl_list_find_index</code>	79
4.16.2.6	<code>dl_list_find_itr</code>	79
4.16.2.7	<code>dl_list_first</code>	79
4.16.2.8	<code>dl_list_free</code>	79
4.16.2.9	<code>dl_list_free_node</code>	80
4.16.2.10	<code>dl_list_init</code>	80
4.16.2.11	<code>dl_list_insert_after</code>	80
4.16.2.12	<code>dl_list_insert_at</code>	80
4.16.2.13	<code>dl_list_insert_before</code>	81
4.16.2.14	<code>dl_list_insert_node_after_mid</code>	81
4.16.2.15	<code>dl_list_insert_node_back</code>	81
4.16.2.16	<code>dl_list_insert_node_before_mid</code>	81
4.16.2.17	<code>dl_list_insert_node_front</code>	82
4.16.2.18	<code>dl_list_isempty</code>	82
4.16.2.19	<code>dl_list_itr_from_index</code>	82
4.16.2.20	<code>dl_list_last</code>	82
4.16.2.21	<code>dl_list_length</code>	83
4.16.2.22	<code>dl_list_lock</code>	83
4.16.2.23	<code>dl_list_new_node</code>	83
4.16.2.24	<code>dl_list_next</code>	83
4.16.2.25	<code>dl_list_prepend</code>	83
4.16.2.26	<code>dl_list_prev</code>	84
4.16.2.27	<code>dl_list_remove_at</code>	84
4.16.2.28	<code>dl_list_remove_node_back</code>	84
4.16.2.29	<code>dl_list_remove_node_front</code>	84
4.16.2.30	<code>dl_list_remove_node_mid</code>	84
4.16.2.31	<code>dl_list_unlock</code>	85
4.17	<code>dl_list.h</code> File Reference	85
4.17.1	Detailed Description	87
4.17.2	Function Documentation	87

4.17.2.1	dl_list_find	87
4.17.2.2	dl_list_free_node	87
4.17.2.3	dl_list_insert_node_after_mid	87
4.17.2.4	dl_list_insert_node_back	88
4.17.2.5	dl_list_insert_node_before_mid	88
4.17.2.6	dl_list_insert_node_front	88
4.17.2.7	dl_list_new_node	88
4.17.2.8	dl_list_remove_at	88
4.17.2.9	dl_list_remove_node_back	89
4.17.2.10	dl_list_remove_node_front	89
4.17.2.11	dl_list_remove_node_mid	89
4.18	general.c File Reference	89
4.18.1	Detailed Description	91
4.18.2	Function Documentation	91
4.18.2.1	cds_compare_double	91
4.18.2.2	cds_compare_float	91
4.18.2.3	cds_compare_int	91
4.18.2.4	cds_compare_long	92
4.18.2.5	cds_compare_longlong	92
4.18.2.6	cds_compare_string	92
4.18.2.7	cds_compare_uint	93
4.18.2.8	cds_compare_ulong	93
4.18.2.9	cds_compare_ulonglong	93
4.18.2.10	cds_new_double	93
4.18.2.11	cds_new_float	94
4.18.2.12	cds_new_int	94
4.18.2.13	cds_new_long	94
4.18.2.14	cds_new_longlong	94
4.18.2.15	cds_new_string	94
4.18.2.16	cds_new_uint	95
4.18.2.17	cds_new_ulong	95
4.18.2.18	cds_new_ulonglong	95
4.19	ia_stack.c File Reference	95
4.19.1	Detailed Description	96
4.19.2	Function Documentation	97
4.19.2.1	ia_stack_free	97
4.19.2.2	ia_stack_init	97
4.19.2.3	ia_stack_isfull	97
4.19.2.4	ia_stack_peek	97
4.19.2.5	ia_stack_pop	98

4.19.2.6	ia_stack_push	98
4.19.2.7	is_stack_isempty	98
4.20	queue.c File Reference	98
4.20.1	Detailed Description	99
4.20.2	Function Documentation	100
4.20.2.1	queue_free	100
4.20.2.2	queue_init	100
4.20.2.3	queue_isempty	100
4.20.2.4	queue_length	100
4.20.2.5	queue_lock	100
4.20.2.6	queue_pop	101
4.20.2.7	queue_pushback	101
4.20.2.8	queue_unlock	101
4.21	sl_list.c File Reference	101
4.21.1	Detailed Description	103
4.21.2	Function Documentation	103
4.21.2.1	sl_list_data	103
4.21.2.2	sl_list_delete_at	103
4.21.2.3	sl_list_find	103
4.21.2.4	sl_list_find_index	104
4.21.2.5	sl_list_find_itr	104
4.21.2.6	sl_list_first	104
4.21.2.7	sl_list_free	104
4.21.2.8	sl_list_free_node	105
4.21.2.9	sl_list_init	105
4.21.2.10	sl_list_insert_after	105
4.21.2.11	sl_list_insert_at	105
4.21.2.12	sl_list_isempty	106
4.21.2.13	sl_list_itr_from_index	106
4.21.2.14	sl_list_length	106
4.21.2.15	sl_list_lock	106
4.21.2.16	sl_list_new_node	106
4.21.2.17	sl_list_next	107
4.21.2.18	sl_list_prepend	107
4.21.2.19	sl_list_remove_at	107
4.21.2.20	sl_list_unlock	107
4.22	sl_list.h File Reference	107
4.22.1	Detailed Description	109
4.22.2	Function Documentation	109
4.22.2.1	sl_list_find	109

4.22.2.2	sl_list_free_node	109
4.22.2.3	sl_list_new_node	110
4.22.2.4	sl_list_remove_at	110
4.23	stack.c File Reference	110
4.23.1	Detailed Description	111
4.23.2	Function Documentation	111
4.23.2.1	stack_free	111
4.23.2.2	stack_init	111
4.23.2.3	stack_isempty	112
4.23.2.4	stack_length	112
4.23.2.5	stack_lock	112
4.23.2.6	stack_peek	112
4.23.2.7	stack_pop	112
4.23.2.8	stack_push	113
4.23.2.9	stack_unlock	113

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
da_stack_t	Struct to hold an double array stack	7
dl_list_node_t	Struct for double linked list node	8
dl_list_t	Struct to contain a list	8
ia_stack_t	Struct to hold an integer array stack	10
kvpair_t	Key-value pair struct	10
sl_list_node_t	Struct for singly linked list node	11
sl_list_t	Struct to contain a list	12

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	15
bs_tree.h	Developer interface to binary search tree data structure	23
bst_map.c	Implementation of binary search tree map data structure	28
cdatastruct.h	Interface to generic C data structures	31
cds_bs_tree.h	User interface to binary search tree data structure	32
cds_bst_map.h	User interface to binary search tree map data structure	37
cds_common.h	Common data types and data for C data structures library	41
cds_da_stack.h	Interface to double array stack functions	42
cds_dl_list.h	User interface to doubly linked list data structure	45
cds_general.h	Interface to general data structure helper functions	52
cds_ia_stack.h	Interface to integer array stack functions	58
cds_queue.h	User interface to queue data structure	61
cds_sl_list.h	User interface to singly linked list data structure	64
cds_stack.h	User interface to stack data structure	69
da_stack.c	Implementation of doubleeeger array stack functions	73
dl_list.c	Implementation of doubly linked list data structure	76
dl_list.h	Developer interface to double linked list data structure	85
general.c	Implementation of general data structure helper functions	89
ia_stack.c	Implementation of integer array stack functions	95

queue.c	Implementation of queue data structure	98
sl_list.c	Implementation of singly linked list data structure	101
sl_list.h	Developer interface to singly linked list data structure	107
stack.c	Implementation of stack data structure	110

Chapter 3

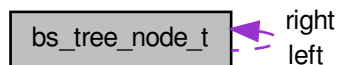
Data Structure Documentation

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 Detailed Description

Struct for binary search tree node.

3.1.2 Field Documentation

3.1.2.1 void* `bs_tree_node_t::data`

Pointer to data

3.1.2.2 struct `bs_tree_node_t`* `bs_tree_node_t::left`

Pointer to left child node

3.1.2.3 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:

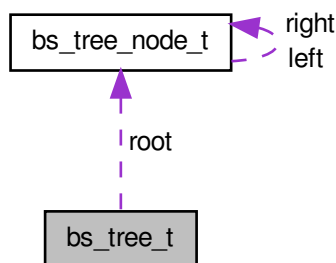
- [cds_bs_tree.h](#)

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

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

3.2.1 Detailed Description

Struct to contain a binary search tree.

3.2.2 Field Documentation

3.2.2.1 int(* bs_tree_t::cfunc)()

Pointer to compare function

3.2.2.2 void(* bs_tree_t::free_func)()

Pointer to node free function

3.2.2.3 size_t bs_tree_t::length

Length of list

3.2.2.4 pthread_mutex_t bs_tree_t::mutex

Mutex

3.2.2.5 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 da_stack_t Struct Reference

Struct to hold an double array stack.

Data Fields

- double * [stack](#)
- size_t [length](#)
- size_t [top](#)

3.3.1 Detailed Description

Struct to hold an double array stack.

3.3.2 Field Documentation

3.3.2.1 size_t da_stack_t::length

The length of the array

3.3.2.2 double* da_stack_t::stack

A pointer to the array

3.3.2.3 size_t da_stack_t::top

The index for the top of the stack

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

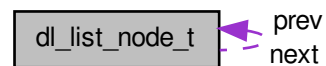
- [da_stack.c](#)

3.4 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](#)

3.4.1 Detailed Description

Struct for double linked list node.

3.4.2 Field Documentation

3.4.2.1 void* dl_list_node_t::data

Pointer to data

3.4.2.2 struct dl_list_node_t* dl_list_node_t::next

Pointer to next node

3.4.2.3 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:

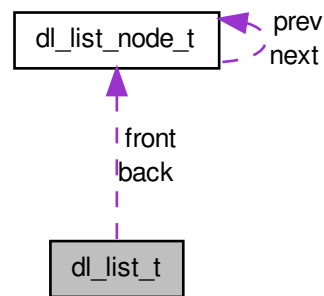
- [cds_dl_list.h](#)

3.5 dl_list_t Struct Reference

Struct to contain a list.

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

Collaboration diagram for dl_list_t:



Data Fields

- pthread_mutex_t [mutex](#)
- struct [dl_list_node_t](#) * [front](#)
- struct [dl_list_node_t](#) * [back](#)
- size_t [length](#)
- int(* [cfunc](#))()
- void(* [free_func](#))()

3.5.1 Detailed Description

Struct to contain a list.

3.5.2 Field Documentation

3.5.2.1 struct [dl_list_node_t](#)* [dl_list_t::back](#)

Pointer to last node

3.5.2.2 int(* [dl_list_t::cfunc](#))()

Pointer to compare function

3.5.2.3 void(* [dl_list_t::free_func](#))()

Pointer to free function

3.5.2.4 struct [dl_list_node_t](#)* [dl_list_t::front](#)

Pointer to first node

3.5.2.5 `size_t dl_list_t::length`

Length of list

3.5.2.6 `pthread_mutex_t dl_list_t::mutex`

Mutex

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

- [dl_list.h](#)

3.6 `ia_stack_t` Struct Reference

Struct to hold an integer array stack.

Data Fields

- `int *` [stack](#)
- `size_t` [length](#)
- `size_t` [top](#)

3.6.1 Detailed Description

Struct to hold an integer array stack.

3.6.2 Field Documentation

3.6.2.1 `size_t ia_stack_t::length`

The length of the array

3.6.2.2 `int* ia_stack_t::stack`

A pointer to the array

3.6.2.3 `size_t ia_stack_t::top`

The index for the top of the stack

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

- [ia_stack.c](#)

3.7 `kvpair_t` Struct Reference

Key-value pair struct.

Data Fields

- `char * key`
- `void * value`

3.7.1 Detailed Description

Key-value pair struct.

3.7.2 Field Documentation

3.7.2.1 `char* kvpair_t::key`

Key string

3.7.2.2 `void* kvpair_t::value`

Pointer to data

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

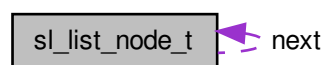
- [bst_map.c](#)

3.8 `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`

3.8.1 Detailed Description

Struct for singly linked list node.

3.8.2 Field Documentation

3.8.2.1 void* sl_list_node_t::data

Pointer to data

3.8.2.2 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:

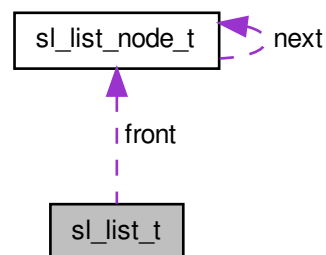
- [cds_sl_list.h](#)

3.9 sl_list_t Struct Reference

Struct to contain a list.

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

Collaboration diagram for sl_list_t:



Data Fields

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

3.9.1 Detailed Description

Struct to contain a list.

3.9.2 Field Documentation

3.9.2.1 int(* sl_list_t::cfunc)()

Pointer to compare function

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

Pointer to free function

3.9.2.3 `struct sl_list_node_t* sl_list_t::front`

Pointer to first node

3.9.2.4 `size_t sl_list_t::length`

Length of list

3.9.2.5 `pthread_mutex_t sl_list_t::mutex`

Mutex

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

- [sl_list.h](#)

Chapter 4

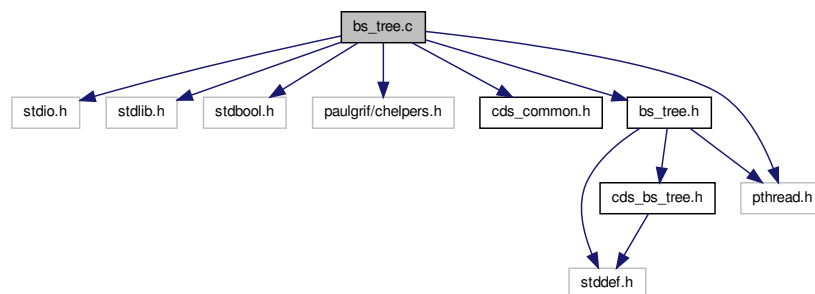
File Documentation

4.1 bs_tree.c File Reference

Implementation of binary search tree data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "bs_tree.h"
#include <pthread.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.
- void [bs_tree_preorder_left_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a preorder left-to-right traversal of a bs_tree.
- void [bs_tree_inorder_left_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs an inorder left-to-right traversal of a bs_tree.
- void [bs_tree_postorder_left_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a postorder left-to-right traversal of a bs_tree.
- void [bs_tree_preorder_right_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a preorder right-to-left traversal of a bs_tree.
- void [bs_tree_inorder_right_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs an inorder right-to-left traversal of a bs_tree.
- void [bs_tree_postorder_right_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a postorder right-to-left traversal of a bs_tree.
- void [bs_tree_lock](#) ([bs_tree](#) tree)
Locks a tree's mutex.
- void [bs_tree_unlock](#) ([bs_tree](#) tree)
Unlocks a tree's mutex.
- [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](#) [bs_tree_insert_search](#) ([bs_tree](#) tree, void *data, bool *found)
Searches a tree for insertion purposes.
- void [bs_tree_preorder_left_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs a preorder left-to-right traversal of a bs_tree.
- void [bs_tree_inorder_left_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs an inorder left-to-right traversal of a bs_tree.
- void [bs_tree_postorder_left_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs a postorder left-to-right traversal of a bs_tree.
- void [bs_tree_preorder_right_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs a preorder right-to-left traversal of a bs_tree.
- void [bs_tree_inorder_right_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs an inorder right-to-left traversal of a bs_tree.
- void [bs_tree_postorder_right_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs a postorder right-to-left traversal of a bs_tree.

4.1.1 Detailed Description

Implementation of binary search tree 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/>

4.1.2 Function Documentation

4.1.2.1 void bs_tree_free (bs_tree tree)

Frees the resources associated with a 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)

Frees the resources associated with a subtree.

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)

Initializes a new binary search tree.

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 void `bs_tree_inorder_left_traverse` (`bs_tree tree`, void(*)(`void *`, `void *arg`) `dfunc`, `void * arg`)

Performs an inorder left-to-right traversal of a `bs_tree`.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.5 void `bs_tree_inorder_left_traverse.int` (`bs_tree tree`, `bs_tree_node node`, void(*)(`void *`, `void *`) `dfunc`, `void * arg`)

Performs an inorder left-to-right traversal of a `bs_tree`.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.6 void `bs_tree_inorder_right_traverse` (`bs_tree tree`, void(*)(`void *`, `void *arg`) `dfunc`, `void * arg`)

Performs an inorder right-to-left traversal of a `bs_tree`.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.7 void `bs_tree_inorder_right_traverse.int` (`bs_tree tree`, `bs_tree_node node`, void(*)(`void *`, `void *`) `dfunc`, `void * arg`)

Performs an inorder right-to-left traversal of a `bs_tree`.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

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

Inserts data into a tree.

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.

4.1.2.9 **bs_tree_node** `bs_tree_insert_search (bs_tree tree, void * data, bool * found)`

Searches a tree for insertion purposes.

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.

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

Inserts a data element into a subtree.

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.

4.1.2.11 **bool** `bs_tree_isempty (const bs_tree tree)`

Checks if a tree is empty.

Parameters

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

Returns

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

4.1.2.12 `size_t bs_tree_length (const bs_tree tree)`

Returns the number of elements in a tree.

Parameters

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

Returns

The number of elements in the tree.

4.1.2.13 `void bs_tree_lock (bs_tree tree)`

Locks a tree's mutex.

Parameters

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

4.1.2.14 `bs_tree_node bs_tree_new_node (void * data)`

Creates and allocates memory for a new node.

Parameters

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

Returns

A pointer to the newly-created node.

4.1.2.15 `void bs_tree_postorder_left_traverse (bs_tree tree, void(*)(void *, void *) dfunc, void * arg)`

Performs a postorder left-to-right traversal of a `bs_tree`.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.16 `void bs_tree_postorder_left_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)`

Performs a postorder left-to-right traversal of a `bs_tree`.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.17 `void bs_tree_postorder_right_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a postorder right-to-left traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.18 `void bs_tree_postorder_right_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)`

Performs a postorder right-to-left traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.19 `void bs_tree_preorder_left_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a preorder left-to-right traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.20 `void bs_tree_preorder_left_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)`

Performs a preorder left-to-right traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.21 `void bs_tree_preorder_right_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a preorder right-to-left traversal of a `bs_tree`.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.1.2.22 `void bs_tree_preorder_right_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)`

Performs a preorder right-to-left traversal of a `bs_tree`.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

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

Determines if a data element is in a tree.

Parameters

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

Returns

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

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

Searches a tree for a piece of data and returns it.

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.

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

Searches a tree for a piece of 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.

4.1.2.26 void bs_tree_unlock (bs_tree tree)

Unlocks a tree's mutex.

Parameters

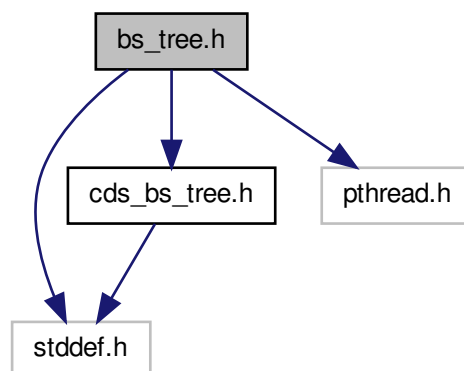
<i>tree</i>	A pointer to the tree.
-------------	------------------------

4.2 bs_tree.h File Reference

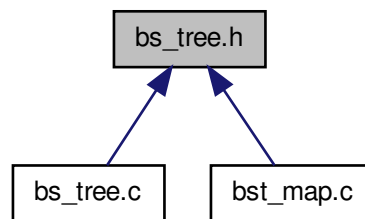
Developer interface to binary search tree data structure.

```
#include <stddef.h>
#include "cds_bs_tree.h"
#include <pthread.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.

Macros

- `#define _POSIX_C_SOURCE 200809L`
Enable POSIX library.

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](#) [bs_tree_insert_search](#) ([bs_tree](#) tree, void *key, bool *found)
Searches a tree for insertion purposes.
- void [bs_tree_preorder_left_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs a preorder left-to-right traversal of a bs_tree.
- void [bs_tree_inorder_left_traverse_int](#) ([bs_tree](#) tree, [bs_tree_node](#) node, void(*dfunc)(void *, void *), void *arg)
Performs an inorder left-to-right traversal of a bs_tree.

- void `bs_tree_postorder_left_traverse_int` (`bs_tree` tree, `bs_tree_node` node, void(*dfunc)(void *, void *), void *arg)
Performs a postorder left-to-right traversal of a bs_tree.
- void `bs_tree_preorder_right_traverse_int` (`bs_tree` tree, `bs_tree_node` node, void(*dfunc)(void *, void *), void *arg)
Performs a preorder right-to-left traversal of a bs_tree.
- void `bs_tree_inorder_right_traverse_int` (`bs_tree` tree, `bs_tree_node` node, void(*dfunc)(void *, void *), void *arg)
Performs an inorder right-to-left traversal of a bs_tree.
- void `bs_tree_postorder_right_traverse_int` (`bs_tree` tree, `bs_tree_node` node, void(*dfunc)(void *, void *), void *arg)
Performs a postorder right-to-left traversal of a bs_tree.

4.2.1 Detailed Description

Developer interface to binary search tree 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/>

4.2.2 Function Documentation

4.2.2.1 void bs_tree_free_subtree (bs_tree tree, bs_tree_node node)

Frees the resources associated with a subtree.

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 void bs_tree_inorder_left_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)

Performs an inorder left-to-right traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.2.2.3 `void bs_tree_inorder_right_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)`

Performs an inorder right-to-left traversal of a `bs_tree`.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.2.2.4 `bs_tree_node bs_tree_insert_search (bs_tree tree, void * data, bool * found)`

Searches a tree for insertion purposes.

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.

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

Inserts a data element into a subtree.

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.

4.2.2.6 `bs_tree_node bs_tree_new_node (void * data)`

Creates and allocates memory for a new node.

Parameters

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

Returns

A pointer to the newly-created node.

4.2.2.7 void bs_tree_postorder_left_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)

Performs a postorder left-to-right traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to dfunc().

4.2.2.8 void bs_tree_postorder_right_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)

Performs a postorder right-to-left traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to dfunc().

4.2.2.9 void bs_tree_preorder_left_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)

Performs a preorder left-to-right traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to dfunc().

4.2.2.10 void bs_tree_preorder_right_traverse_int (bs_tree tree, bs_tree_node node, void(*)(void *, void *) dfunc, void * arg)

Performs a preorder right-to-left traversal of a bs_tree.

This function is called internally by the matching function that the library user calls.

Parameters

<i>tree</i>	A pointer to the tree.
<i>node</i>	A pointer to the current node.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

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

Searches a tree for a piece of data.

Parameters

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

Returns

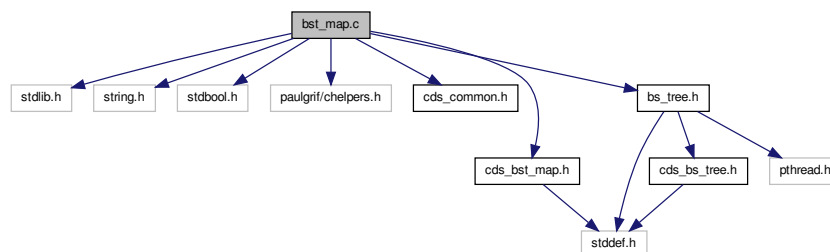
A pointer to the node in which the data was found, or `NULL` if the data was not found.

4.3 `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.
- void `bst_map_lock` (`bst_map` map)
Locks a map's mutex.
- void `bst_map_unlock` (`bst_map` map)
Unlocks a map's mutex.

4.3.1 Detailed Description

Implementation of binary search tree map 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/>

4.3.2 Function Documentation

4.3.2.1 void `bst_map_free` (`bst_map` map)

Frees the resources associated with a BST map.

Parameters

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

4.3.2.2 `bst_map` `bst_map_init` (void)

Initializes a new binary search tree map.

Returns

A pointer to the new map.

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

Inserts a key-value pair into a map.

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)

Checks if a map is empty.

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)

Returns the number of elements in a BST map.

Parameters

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

Returns

The number of elements in the map.

4.3.2.6 void bst_map_lock (bst_map map)

Locks a map's mutex.

Parameters

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

4.3.2.7 bool bst_map_search (const bst_map map, const char * key)

Determines if a key is in a map.

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.

4.3.2.8 void* bst_map_search_data (const bst_map map, const char * key)

Searches a map for a value matching a key and returns it.

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.

4.3.2.9 void bst_map_unlock (bst_map map)

Unlocks a map's mutex.

Parameters

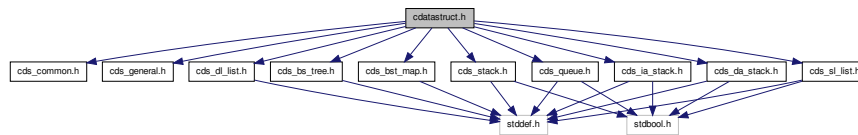
<i>map</i>	A pointer to the map.
------------	-----------------------

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 "cds_ia_stack.h"
#include "cds_da_stack.h"
```

Include dependency graph for `cdatastruct.h`:



4.4.1 Detailed Description

Interface to generic C data structures. Interface to generic C data structures.

Author

Paul Griffiths

Copyright

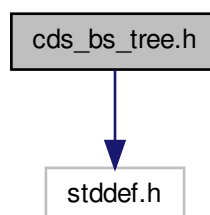
Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

4.5 `cds_bs_tree.h` File Reference

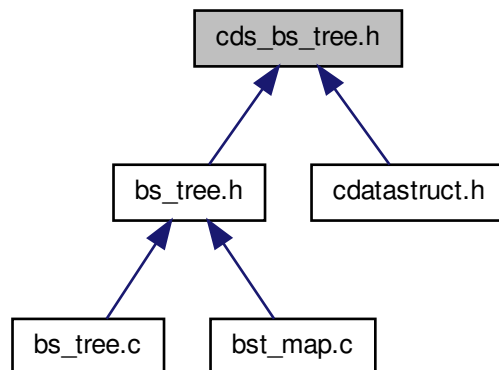
User interface to binary search tree data structure.

```
#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.
- void [bs_tree_preorder_left_traverse](#) ([bs_tree](#) tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a preorder left-to-right traversal of a bs_tree.

- void `bs_tree_inorder_left_traverse` (`bs_tree` tree, void(*dfunc)(void *, void *arg), void *arg)
Performs an inorder left-to-right traversal of a bs_tree.
- void `bs_tree_postorder_left_traverse` (`bs_tree` tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a postorder left-to-right traversal of a bs_tree.
- void `bs_tree_preorder_right_traverse` (`bs_tree` tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a preorder right-to-left traversal of a bs_tree.
- void `bs_tree_inorder_right_traverse` (`bs_tree` tree, void(*dfunc)(void *, void *arg), void *arg)
Performs an inorder right-to-left traversal of a bs_tree.
- void `bs_tree_postorder_right_traverse` (`bs_tree` tree, void(*dfunc)(void *, void *arg), void *arg)
Performs a postorder right-to-left traversal of a bs_tree.
- void `bs_tree_lock` (`bs_tree` tree)
Locks a tree's mutex.
- void `bs_tree_unlock` (`bs_tree` tree)
Unlocks a tree's mutex.

4.5.1 Detailed Description

User interface to binary search tree 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/>

4.5.2 Function Documentation

4.5.2.1 void `bs_tree_free` (`bs_tree` tree)

Frees the resources associated with a 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*)

Initializes a new binary search tree.

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 void bs_tree_inorder_left_traverse (bs_tree tree, void(*) (void *, void *arg) dfunc, void * arg)

Performs an inorder left-to-right traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.5.2.4 void bs_tree_inorder_right_traverse (bs_tree tree, void(*) (void *, void *arg) dfunc, void * arg)

Performs an inorder right-to-left traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.5.2.5 bool bs_tree_insert (bs_tree tree, void * data)

Inserts data into a tree.

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.

4.5.2.6 bool bs_tree_isempty (const bs_tree tree)

Checks if a tree is empty.

Parameters

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

Returns

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

4.5.2.7 `size_t bs_tree_length (const bs_tree tree)`

Returns the number of elements in a tree.

Parameters

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

Returns

The number of elements in the tree.

4.5.2.8 `void bs_tree_lock (bs_tree tree)`

Locks a tree's mutex.

Parameters

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

4.5.2.9 `void bs_tree_postorder_left_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a postorder left-to-right traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.5.2.10 `void bs_tree_postorder_right_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a postorder right-to-left traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.5.2.11 `void bs_tree_preorder_left_traverse (bs_tree tree, void(*)(void *, void *arg) dfunc, void * arg)`

Performs a preorder left-to-right traversal of a bs_tree.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

4.5.2.12 void `bs_tree_preorder_right_traverse` (`bs_tree tree`, void(*)`(void *, void *arg) dfunc`, void * `arg`)

Performs a preorder right-to-left traversal of a `bs_tree`.

Parameters

<i>tree</i>	A pointer to the tree.
<i>dfunc</i>	A pointer to the function to invoke for each node.
<i>arg</i>	A pointer to the argument to pass to <code>dfunc()</code> .

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

Determines if a data element is in a tree.

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.

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

Searches a tree for a piece of data and returns it.

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.

4.5.2.15 void `bs_tree_unlock` (`bs_tree tree`)

Unlocks a tree's mutex.

Parameters

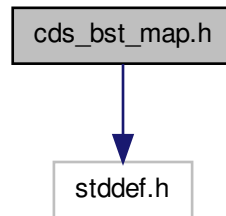
<i>tree</i>	A pointer to the tree.
-------------	------------------------

4.6 cds_bst_map.h File Reference

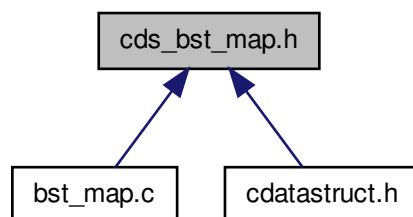
User interface to binary search tree map data structure.

```
#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.
- void [bst_map_lock](#) ([bst_map](#) map)
Locks a map's mutex.
- void [bst_map_unlock](#) ([bst_map](#) map)
Unlocks a map's mutex.

4.6.1 Detailed Description

User interface to binary search tree map 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/>

4.6.2 Function Documentation

4.6.2.1 void [bst_map_free](#) ([bst_map](#) map)

Frees the resources associated with a BST map.

Parameters

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

4.6.2.2 [bst_map](#) [bst_map_init](#) (void)

Initializes a new binary search tree map.

Returns

A pointer to the new map.

4.6.2.3 bool [bst_map_insert](#) ([bst_map](#) map, const char * *key*, void * *value*)

Inserts a key-value pair into a map.

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)`

Checks if a map is empty.

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)`

Returns the number of elements in a BST map.

Parameters

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

Returns

The number of elements in the map.

4.6.2.6 `void bst_map_lock (bst_map map)`

Locks a map's mutex.

Parameters

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

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

Determines if a key is in a map.

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.

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

Searches a map for a value matching a key and returns it.

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.

4.6.2.9 void bst_map_unlock (bst_map map)

Unlocks a map's mutex.

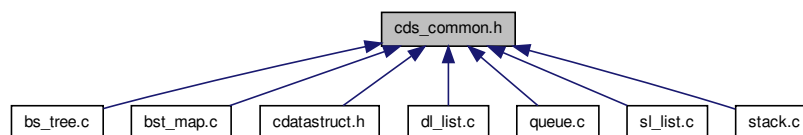
Parameters

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

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

Common data types and data for C data structures library.

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

Enumeration of return error codes.

Enumerator:

CDSERR_ERROR Unspecified error

CDSERR_OUTOFRANGE Index out of range

CDSERR_NOTFOUND Data element not found

CDSERR_BADITERATOR Invalid iterator

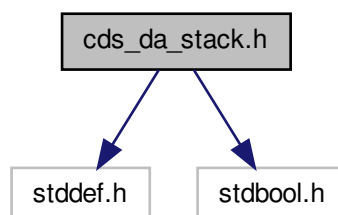
4.8 cds_da_stack.h File Reference

Interface to double array stack functions.

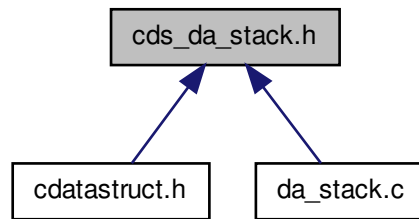
```
#include <stddef.h>
```

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

Include dependency graph for cds_da_stack.h:



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



Typedefs

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

Functions

- [da_stack](#) [da_stack_init](#) (const [size_t](#) size)
Constructs and initializes a new stack.
- void [da_stack_free](#) ([da_stack](#) stk)
Destructs and frees a stack.
- bool [is_stack_isempty](#) (const [da_stack](#) stk)
Checks if a stack is empty.
- bool [da_stack_isfull](#) (const [da_stack](#) stk)
Checks if a stack is full.
- double [da_stack_peek](#) (const [da_stack](#) stk)
Returns the top element of the stack without popping it.
- double [da_stack_pop](#) ([da_stack](#) stk)
Pops the top element of the stack.
- void [da_stack_push](#) ([da_stack](#) stk, const double n)
Pushes an element onto the stack.

4.8.1 Detailed Description

Interface to double array stack functions.

Author

Paul Griffiths

Copyright

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

4.8.2 Function Documentation

4.8.2.1 void da_stack_free (da_stack stk)

Destructs and frees a stack.

Parameters

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

4.8.2.2 da_stack da_stack_init (const size_t size)

Constructs and initializes a new stack.

Parameters

<i>size</i>	The initial size of the new stack.
-------------	------------------------------------

Returns

A pointer to the new stack.

4.8.2.3 bool da_stack_isfull (const da_stack stk)

Checks if a stack is full.

Parameters

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

Returns

`true` is the stack is full, `false` otherwise.

4.8.2.4 double da_stack_peek (const da_stack stk)

Returns the top element of the stack without popping it.

Parameters

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

Returns

The value of the element at the top.

4.8.2.5 double da_stack_pop (da_stack stk)

Pops the top element of the stack.

Parameters

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

Returns

The popped element of the stack.

4.8.2.6 void da_stack_push (da_stack stk, const double f)

Pushes an element onto the stack.

If the stack is full, this function attempts to reallocate memory for the array. Each time the stack is full, the amount of memory requested is double the original amount.

Parameters

<i>stk</i>	A pointer to the stack.
<i>f</i>	The value to push onto the stack.

4.8.2.7 bool is_stack_isempty (const da_stack stk)

Checks if a stack is empty.

Parameters

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

Returns

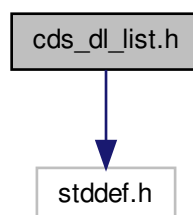
`true` if the stack is empty, `false` otherwise.

4.9 cds_dl_list.h File Reference

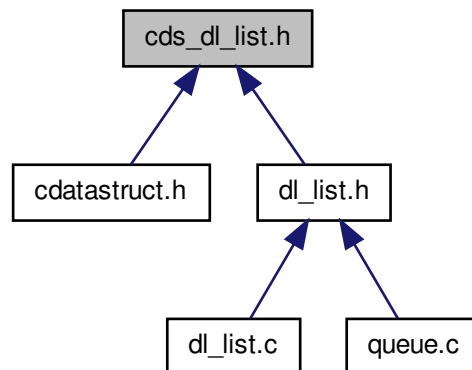
User interface to doubly linked list data structure.

```
#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.
- void [dl_list_lock](#) ([dl_list](#) list)
Locks a list's mutex.
- void [dl_list_unlock](#) ([dl_list](#) list)
Unlocks a list's mutex.

4.9.1 Detailed Description

User interface to doubly linked 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/>

4.9.2 Function Documentation

4.9.2.1 void dl_list_append (dl_list list, void * data)

Inserts an element at the end of a list.

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.

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

Returns a pointer to the data at a specified 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.

4.9.2.3 int dl_list_delete_at (dl_list list, const size_t index)

Deletes a list element at a specified 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.

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

Finds the index of the specified data in a list.

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.

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

Gets an iterator to the specified data in a list.

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.

4.9.2.6 `dl_list_itr dl_list_first (const dl_list list)`

Returns an iterator to the first element of a list.

Parameters

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

Returns

An iterator to the first element.

4.9.2.7 `void dl_list_free (dl_list list)`

Frees the resources associated with a list.

Parameters

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

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

Initializes a new doubly linked list.

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.

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

Inserts an element after a provided iterator.

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.

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

Inserts an element at the specified index of a list.

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.

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

Inserts an element before a provided iterator.

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.

4.9.2.12 `bool dl_list_isempty (const dl_list list)`

Checks if a list is empty.

Parameters

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

Returns

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

4.9.2.13 `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.

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.

4.9.2.14 dl_list_itr dl_list_last (const dl_list list)

Returns an iterator to the last element of a list.

Parameters

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

Returns

An iterator to the first element.

4.9.2.15 size_t dl_list_length (const dl_list list)

Returns the number of elements in a list.

Parameters

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

4.9.2.16 void dl_list_lock (dl_list list)

Locks a list's mutex.

Parameters

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

4.9.2.17 dl_list_itr dl_list_next (const dl_list_itr itr)

Advances a list iterator by one element.

Parameters

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

Returns

The advanced iterator.

4.9.2.18 void dl_list_prepend (dl_list list, void * data)

Inserts an element at the beginning of a list.

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.

4.9.2.19 `dl_list_itr dl_list_prev (const dl_list_itr itr)`

Backs up a list iterator by one element.

Parameters

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

Returns

The backed up iterator.

4.9.2.20 `void dl_list_unlock (dl_list list)`

Unlocks a list's mutex.

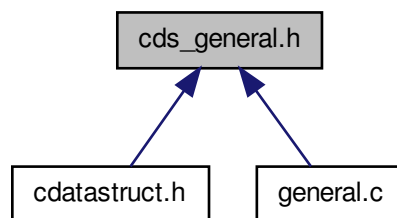
Parameters

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

4.10 `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_longlong` (const long long n)
Dynamically allocates memory for a new `long long`.

- void * [cds_new_ulonglong](#) (const unsigned long long n)
Allocates memory for a new unsigned long long.
- void * [cds_new_float](#) (const float n)
Dynamically allocates memory for a new float.
- void * [cds_new_double](#) (const double n)
Dynamically allocates memory for a new double.
- 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_longlong](#) (const void *data, const void *cmp)
Compares two long long via void pointers.
- int [cds_compare_ulonglong](#) (const void *data, const void *cmp)
Compares two unsigned long long via void pointers.
- int [cds_compare_float](#) (const void *data, const void *cmp)
Compares two float via void pointers.
- int [cds_compare_double](#) (const void *data, const void *cmp)
Compares two double via void pointers.
- int [cds_compare_string](#) (const void *data, const void *cmp)
Compares two strings via void pointers.

4.10.1 Detailed Description

Interface to general data structure helper functions. 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/>

4.10.2 Function Documentation

4.10.2.1 int cds_compare_double (const void * data, const void * cmp)

Compares two double via void pointers.

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.

4.10.2.2 int cds_compare_float (const void * *data*, const void * *cmp*)

Compares two `float` via `void` pointers.

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.

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

Compares two `int` via `void` pointers.

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.

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

Compares two `long` via `void` pointers.

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.

4.10.2.5 int cds_compare_longlong (const void * *data*, const void * *cmp*)

Compares two `long long` via `void` pointers.

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.

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

Compares two strings via `void` pointers.

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.

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

Compares two `unsigned int` via `void` pointers.

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.

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

Compares two `unsigned long` via `void` pointers.

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.

4.10.2.9 `int cds_compare_ulonglong (const void * data, const void * cmp)`

Compares two `unsigned long long` via `void` pointers.

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.

4.10.2.10 void* cds_new_double (const double f)

Dynamically allocates memory for a new `double`.

Parameters

<code>f</code>	The new <code>double</code> for which to allocate.
----------------	--

Returns

A `void` pointer to the allocated memory.

4.10.2.11 void* cds_new_float (const float f)

Dynamically allocates memory for a new `float`.

Parameters

<code>f</code>	The new <code>float</code> for which to allocate.
----------------	---

Returns

A `void` pointer to the allocated memory.

4.10.2.12 void* cds_new_int (const int n)

Dynamically allocates memory for a new `int`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.10.2.13 void* cds_new_long (const long n)

Dynamically allocates memory for a new `long`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.10.2.14 void* cds_new_longlong (const long long *n*)

Dynamically allocates memory for a new long long.

Parameters

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

Returns

A void pointer to the allocated memory.

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

Dynamically allocates memory for a new string.

Parameters

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

Returns

A void pointer to the allocated memory.

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

Dynamically allocates memory for a new unsigned int.

Parameters

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

Returns

A void pointer to the allocated memory.

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

Dynamically allocates memory for a new unsigned long.

Parameters

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

Returns

A void pointer to the allocated memory.

4.10.2.18 void* cds_new_ulonglong (const unsigned long long *n*)

Allocates memory for a new unsigned long long.

Parameters

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

Returns

A void pointer to the allocated memory.

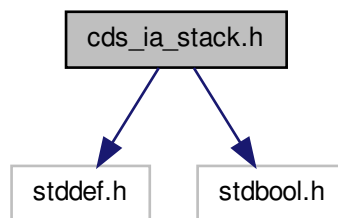
4.11 cds_ia_stack.h File Reference

Interface to integer array stack functions.

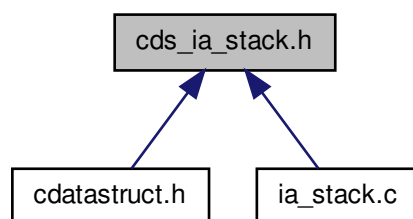
```
#include <stddef.h>
```

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

Include dependency graph for cds_ia_stack.h:



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

**Typedefs**

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

Functions

- [ia_stack](#) [ia_stack_init](#) (const [size_t](#) size)

- Constructs and initializes a new stack.*
- void [ia_stack_free](#) ([ia_stack](#) stk)
- Destructs and frees a stack.*
- bool [is_stack_isempty](#) (const [ia_stack](#) stk)
- Checks if a stack is empty.*
- bool [ia_stack_isfull](#) (const [ia_stack](#) stk)
- Checks if a stack is full.*
- int [ia_stack_peek](#) (const [ia_stack](#) stk)
- Returns the top element of the stack without popping it.*
- int [ia_stack_pop](#) ([ia_stack](#) stk)
- Pops the top element of the stack.*
- void [ia_stack_push](#) ([ia_stack](#) stk, const int n)
- Pushes an element onto the stack.*

4.11.1 Detailed Description

Interface to integer array stack functions.

Author

Paul Griffiths

Copyright

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

4.11.2 Function Documentation

4.11.2.1 void [ia_stack_free](#) ([ia_stack](#) stk)

Destructs and frees a stack.

Parameters

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

4.11.2.2 [ia_stack](#) [ia_stack_init](#) (const [size_t](#) size)

Constructs and initializes a new stack.

Parameters

<i>size</i>	The initial size of the new stack.
-------------	------------------------------------

Returns

A pointer to the new stack.

4.11.2.3 bool [ia_stack_isfull](#) (const [ia_stack](#) stk)

Checks if a stack is full.

Parameters

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

Returns

`true` is the stack is full, `false` otherwise.

4.11.2.4 int ia_stack_peek (const ia_stack stk)

Returns the top element of the stack without popping it.

Parameters

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

Returns

The value of the element at the top.

4.11.2.5 int ia_stack_pop (ia_stack stk)

Pops the top element of the stack.

Parameters

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

Returns

The popped element of the stack.

4.11.2.6 void ia_stack_push (ia_stack stk, const int n)

Pushes an element onto the stack.

If the stack is full, this function attempts to reallocate memory for the array. Each time the stack is full, the amount of memory requested is double the original amount.

Parameters

<i>stk</i>	A pointer to the stack.
<i>n</i>	The value to push onto the stack.

4.11.2.7 bool is_stack_isempty (const ia_stack stk)

Checks if a stack is empty.

Parameters

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

Returns

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

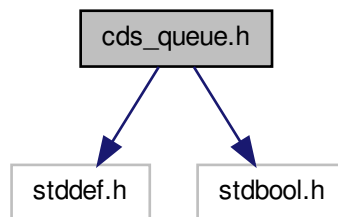
4.12 cds_queue.h File Reference

User interface to queue data structure.

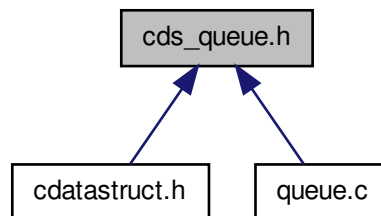
```
#include <stddef.h>
```

```
#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.

- `void queue_lock (queue que)`

Locks a queue's mutex.

- `void queue_unlock (queue que)`

Unlocks a queue's mutex.

4.12.1 Detailed Description

User interface to queue 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/>

4.12.2 Function Documentation

4.12.2.1 `void queue_free (queue que)`

Frees memory and releases resources used by a queue.

Parameters

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

4.12.2.2 `queue queue_init (void(*) (void *) free_func)`

Initializes a new queue.

Parameters

<code>free_func</code>	A pointer to a function to free a queue 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 queue.

4.12.2.3 `bool queue_isempty (const queue que)`

Checks if a queue is empty.

Parameters

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

Returns

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

4.12.2.4 `size_t queue_length (const queue que)`

Gets the number of items in a queue.

Parameters

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

Returns

The number of items in the queue.

4.12.2.5 `void queue_lock (queue que)`

Locks a queue's mutex.

Parameters

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

4.12.2.6 `void* queue_pop (queue que)`

Pops a data item from the queue.

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.

4.12.2.7 `void queue_pushback (queue que, void * data)`

Pushes a data item onto the back of the queue.

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.

4.12.2.8 void queue_unlock (queue *que*)

Unlocks a queue's mutex.

Parameters

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

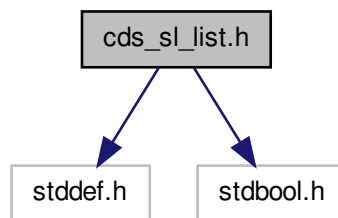
4.13 cds_sl_list.h File Reference

User interface to singly linked list data structure.

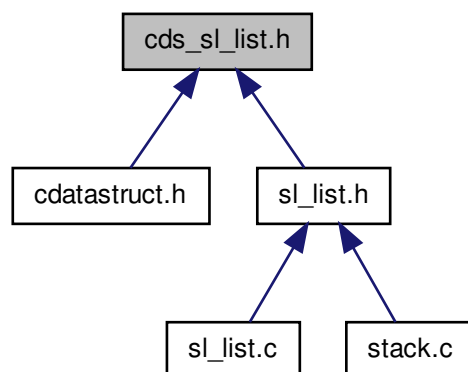
```
#include <stddef.h>
```

```
#include <stdbool.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.
- void [sl_list_lock](#) ([sl_list](#) list)
Locks a list's mutex.
- void [sl_list_unlock](#) ([sl_list](#) list)
Unlocks a list's mutex.

4.13.1 Detailed Description

User interface to singly linked 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/>

4.13.2 Function Documentation

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

Returns a pointer to the data at a specified 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.

4.13.2.2 int sl_list_delete_at (sl_list list, const size_t index)

Deletes a list element at a specified 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.

4.13.2.3 int sl_list_find_index (const sl_list list, const void * data)

Gets an index to the specified data in a list.

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.

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

Gets an iterator to the specified data in a list.

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.

4.13.2.5 `sl_list_itr sl_list_first (const sl_list list)`

Returns an iterator to the first element of a list.

Parameters

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

Returns

An iterator to the first element.

4.13.2.6 `void sl_list_free (sl_list list)`

Frees the resources associated with a list.

Parameters

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

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

Initializes a new singly linked list.

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.

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

Inserts an element after a provided iterator.

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.

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

Inserts an element at the specified index of a list.

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.

4.13.2.10 `bool sl_list_isempty (const sl_list list)`

Checks if a list is empty.

Parameters

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

Returns

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

4.13.2.11 `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.

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.

4.13.2.12 `size_t sl_list_length (const sl_list list)`

Returns the number of elements in a list.

Parameters

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

4.13.2.13 `void sl_list_lock (sl_list list)`

Locks a list's mutex.

Parameters

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

4.13.2.14 `sl_list_itr sl_list_next (const sl_list_itr itr)`

Advances a list iterator by one element.

Parameters

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

Returns

The advanced iterator.

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

Inserts an element at the beginning of a list.

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.

4.13.2.16 `void sl_list_unlock (sl_list list)`

Unlocks a list's mutex.

Parameters

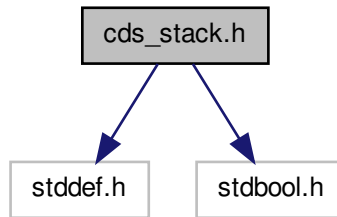
<i>list</i>	A pointer to the list.
-------------	------------------------

4.14 cds_stack.h File Reference

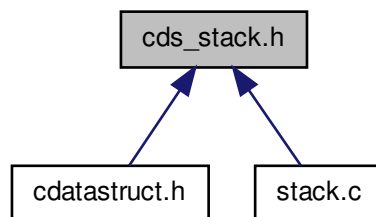
User interface to stack data structure.

```
#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_peek` (`stack` stk)
Peeks at the data for the top element of the stack.

- void `stack_push` (`stack` stk, void *data)
Pushes a data item onto the stack.
- void `stack_lock` (`stack` stk)
Locks a stack's mutex.
- void `stack_unlock` (`stack` stk)
Unlocks a stack's mutex.

4.14.1 Detailed Description

User interface to stack 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/>

4.14.2 Function Documentation

4.14.2.1 void stack_free (stack stk)

Frees memory and releases resources used by a stack.

Parameters

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

4.14.2.2 stack stack_init (void(*) (void *) free_func)

Initializes a new stack.

Parameters

<code>free_func</code>	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.

4.14.2.3 bool stack_isempty (const stack stk)

Checks if a stack is empty.

Parameters

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

Returns

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

4.14.2.4 `size_t stack_length (const stack stk)`

Gets the number of items in a stack.

Parameters

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

Returns

The number of items in the stack.

4.14.2.5 `void stack_lock (stack stk)`

Locks a stack's mutex.

Parameters

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

4.14.2.6 `void* stack_peek (stack stk)`

Peeks at the data for the top element of the stack.

The top item is not popped from the stack, and the user should not `free()` the pointer from this function.

Parameters

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

Returns

A `void` pointer to the popped data item.

4.14.2.7 `void* stack_pop (stack stk)`

Pops a data item from the stack.

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.

4.14.2.8 void stack_push (stack stk, void * data)

Pushes a data item onto the stack.

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.

4.14.2.9 void stack_unlock (stack stk)

Unlocks a stack's mutex.

Parameters

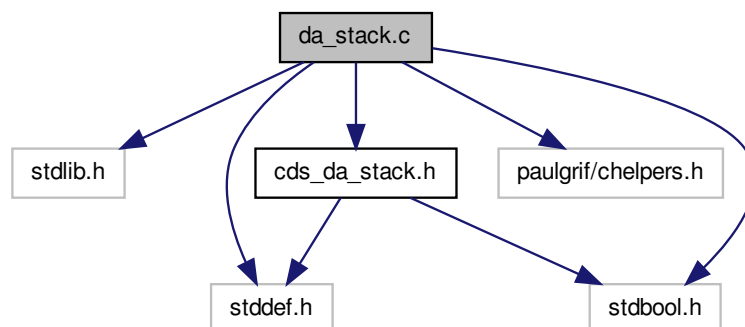
<i>stk</i>	A pointer to the stack.
------------	-------------------------

4.15 da_stack.c File Reference

Implementation of doubleeeger array stack functions.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_da_stack.h"
```

Include dependency graph for da_stack.c:



Data Structures

- struct [da_stack_t](#)

Struct to hold an double array stack.

Typedefs

- typedef struct [da_stack_t](#) [da_stack_t](#)
Struct to hold an double array stack.

Functions

- [da_stack](#) [da_stack_init](#) (const [size_t](#) size)
Constructs and initializes a new stack.
- void [da_stack_free](#) ([da_stack](#) stk)
Destructs and frees a stack.
- bool [is_stack_isempty](#) (const [da_stack](#) stk)
Checks if a stack is empty.
- bool [da_stack_isfull](#) (const [da_stack](#) stk)
Checks if a stack is full.
- double [da_stack_peek](#) (const [da_stack](#) stk)
Returns the top element of the stack without popping it.
- double [da_stack_pop](#) ([da_stack](#) stk)
Pops the top element of the stack.
- void [da_stack_push](#) ([da_stack](#) stk, const double f)
Pushes an element onto the stack.

4.15.1 Detailed Description

Implementation of doubleeager array stack 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 void [da_stack_free](#) ([da_stack](#) *stk*)

Destructs and frees a stack.

Parameters

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

4.15.2.2 [da_stack](#) [da_stack_init](#) (const [size_t](#) *size*)

Constructs and initializes a new stack.

Parameters

<i>size</i>	The initial size of the new stack.
-------------	------------------------------------

Returns

A pointer to the new stack.

4.15.2.3 bool da_stack_isfull (const da_stack stk)

Checks if a stack is full.

Parameters

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

Returns

`true` is the stack is full, `false` otherwise.

4.15.2.4 double da_stack_peek (const da_stack stk)

Returns the top element of the stack without popping it.

Parameters

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

Returns

The value of the element at the top.

4.15.2.5 double da_stack_pop (da_stack stk)

Pops the top element of the stack.

Parameters

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

Returns

The popped element of the stack.

4.15.2.6 void da_stack_push (da_stack stk, const double f)

Pushes an element onto the stack.

If the stack is full, this function attempts to reallocate memory for the array. Each time the stack is full, the amount of memory requested is double the original amount.

Parameters

<i>stk</i>	A pointer to the stack.
<i>f</i>	The value to push onto the stack.

4.15.2.7 `bool is_stack_isempty (const da_stack stk)`

Checks if a stack is empty.

Parameters

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

Returns

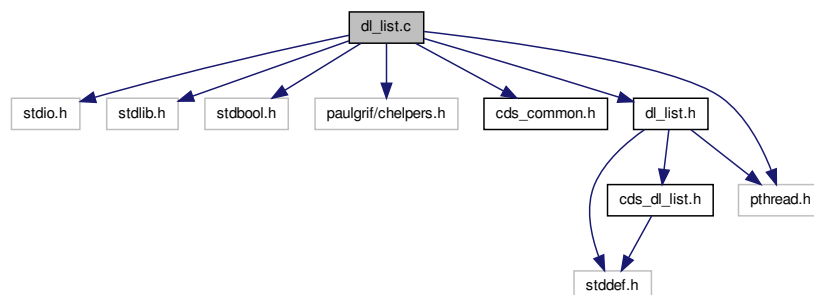
`true` is the stack is empty, `false` otherwise.

4.16 `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 <pthread.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.

 - void [dl_list_lock](#) ([dl_list](#) list)

Locks a list's mutex.

 - void [dl_list_unlock](#) ([dl_list](#) list)

Unlocks a list's mutex.

4.16.1 Detailed Description

Implementation of doubly linked 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 dl_list_append (dl_list list, void * data)

Inserts an element at the end of a list.

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.

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

Returns a pointer to the data at a specified 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.

4.16.2.3 int dl_list_delete_at (dl_list list, const size_t index)

Deletes a list element at a specified 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.

4.16.2.4 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.

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.

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

Finds the index of the specified data in a list.

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.

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

Gets an iterator to the specified data in a list.

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.

4.16.2.7 dl_list_itr dl_list_first (const dl_list *list*)

Returns an iterator to the first element of a list.

Parameters

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

Returns

An iterator to the first element.

4.16.2.8 void dl_list_free (dl_list *list*)

Frees the resources associated with a list.

Parameters

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

4.16.2.9 void dl_list_free_node (dl_list list, dl_list_node node)

Frees resources for a node and any data.

Parameters

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

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

Initializes a new doubly linked list.

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.

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

Inserts an element after a provided iterator.

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.

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

Inserts an element at the specified index of a list.

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.

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

Inserts an element before a provided iterator.

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.

4.16.2.14 `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.

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.

4.16.2.15 `void dl_list_insert_node_back (dl_list list, dl_list_node node)`

Inserts a node at the back of a list.

Parameters

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

4.16.2.16 `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.

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.

4.16.2.17 `void dl_list.insert_node_front (dl_list list, dl_list_node node)`

Inserts a node at the front of a list.

Parameters

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

4.16.2.18 `bool dl_list.isempty (const dl_list list)`

Checks if a list is empty.

Parameters

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

Returns

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

4.16.2.19 `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.

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.

4.16.2.20 `dl_list_itr dl_list_last (const dl_list list)`

Returns an iterator to the last element of a list.

Parameters

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

Returns

An iterator to the first element.

4.16.2.21 `size_t dl_list_length (const dl_list list)`

Returns the number of elements in a list.

Parameters

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

4.16.2.22 `void dl_list_lock (dl_list list)`

Locks a list's mutex.

Parameters

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

4.16.2.23 `dl_list_node dl_list_new_node (void * data)`

Creates a new list node.

Parameters

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

Returns

A pointer to the newly created node.

4.16.2.24 `dl_list_itr dl_list_next (const dl_list_itr itr)`

Advances a list iterator by one element.

Parameters

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

Returns

The advanced iterator.

4.16.2.25 `void dl_list_prepend (dl_list list, void * data)`

Inserts an element at the beginning of a list.

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.

4.16.2.26 `dl_list_itr dl_list_prev (const dl_list_itr itr)`

Backs up a list iterator by one element.

Parameters

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

Returns

The backed up iterator.

4.16.2.27 `dl_list_node dl_list_remove_at (dl_list list, const size_t index)`

Removes, but does not delete, an element at an 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\(\)](#).

4.16.2.28 `dl_list_node dl_list_remove_node_back (dl_list list)`

Removes the last node of a list.

Parameters

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

Returns

A pointer to the removed node.

4.16.2.29 `dl_list_node dl_list_remove_node_front (dl_list list)`

Removes the first node of a list.

Parameters

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

Returns

A pointer to the removed node.

4.16.2.30 `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.

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`.

4.16.2.31 void dl_list_unlock (dl_list list)

Unlocks a list's mutex.

Parameters

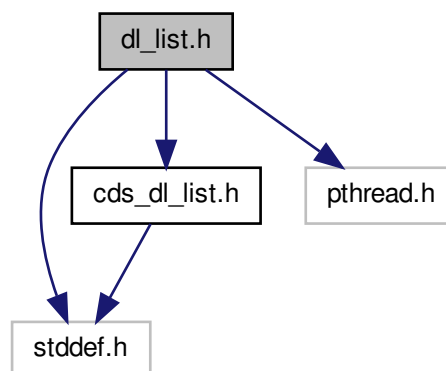
<i>list</i>	A pointer to the list.
-------------	------------------------

4.17 dl_list.h File Reference

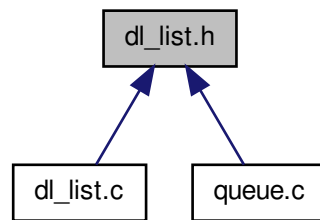
Developer interface to double linked list data structure.

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

Macros

- `#define _POSIX_C_SOURCE 200809L`
Enable POSIX library.

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.

4.17.1 Detailed Description

Developer interface to double linked 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/>

4.17.2 Function Documentation

4.17.2.1 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.

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.

4.17.2.2 void dl_list_free_node (dl_list list, dl_list_node node)

Frees resources for a node and any data.

Parameters

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

4.17.2.3 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.

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.

4.17.2.4 void dl_list_insert_node_back (dl_list list, dl_list_node node)

Inserts a node at the back of a list.

Parameters

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

4.17.2.5 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.

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.

4.17.2.6 void dl_list_insert_node_front (dl_list list, dl_list_node node)

Inserts a node at the front of a list.

Parameters

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

4.17.2.7 dl_list_node dl_list_new_node (void * data)

Creates a new list node.

Parameters

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

Returns

A pointer to the newly created node.

4.17.2.8 dl_list_node dl_list_remove_at (dl_list list, const size_t index)

Removes, but does not delete, an element at an 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()`.

4.17.2.9 `dl_list_node dl_list_remove_node_back (dl_list list)`

Removes the last node of a list.

Parameters

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

Returns

A pointer to the removed node.

4.17.2.10 `dl_list_node dl_list_remove_node_front (dl_list list)`

Removes the first node of a list.

Parameters

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

Returns

A pointer to the removed node.

4.17.2.11 `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.

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.

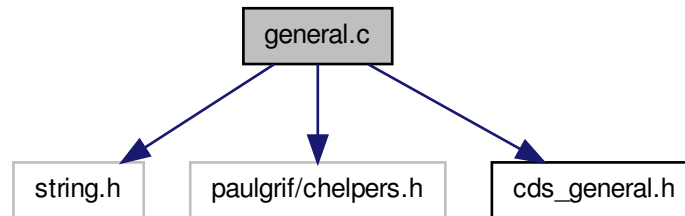
Returns

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

4.18 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_longlong` (const long long n)
Dynamically allocates memory for a new long long.
- void * `cds_new_ulonglong` (const unsigned long long n)
Allocates memory for a new unsigned long long.
- void * `cds_new_float` (const float f)
Dynamically allocates memory for a new float.
- void * `cds_new_double` (const double f)
Dynamically allocates memory for a new double.
- 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_longlong` (const void *data, const void *cmp)
Compares two long long via void pointers.
- int `cds_compare_ulonglong` (const void *data, const void *cmp)
Compares two unsigned long long via void pointers.

- int `cds_compare_float` (const void *data, const void *cmp)
Compares two float via void pointers.
- int `cds_compare_double` (const void *data, const void *cmp)
Compares two double via void pointers.
- int `cds_compare_string` (const void *data, const void *cmp)
Compares two strings via void pointers.

4.18.1 Detailed Description

Implementation of general data structure helper functions. 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/>

4.18.2 Function Documentation

4.18.2.1 int cds_compare_double (const void * data, const void * cmp)

Compares two double via void pointers.

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.

4.18.2.2 int cds_compare_float (const void * data, const void * cmp)

Compares two float via void pointers.

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.

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

Compares two int via void pointers.

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.

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

Compares two `long` via `void` pointers.

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.

4.18.2.5 `int cds_compare_longlong (const void * data, const void * cmp)`

Compares two `long long` via `void` pointers.

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.

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

Compares two strings via `void` pointers.

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.

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

Compares two `unsigned int` via `void` pointers.

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.

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

Compares two `unsigned long` via `void` pointers.

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.

4.18.2.9 `int cds_compare_ulonglong (const void * data, const void * cmp)`

Compares two `unsigned long long` via `void` pointers.

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.

4.18.2.10 `void* cds_new_double (const double f)`

Dynamically allocates memory for a new `double`.

Parameters

<i>f</i>	The new <code>double</code> for which to allocate.
----------	--

Returns

A `void` pointer to the allocated memory.

4.18.2.11 void* cds_new_float (const float *f*)

Dynamically allocates memory for a new `float`.

Parameters

<i>f</i>	The new <code>float</code> for which to allocate.
----------	---

Returns

A `void` pointer to the allocated memory.

4.18.2.12 void* cds_new_int (const int *n*)

Dynamically allocates memory for a new `int`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.18.2.13 void* cds_new_long (const long *n*)

Dynamically allocates memory for a new `long`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.18.2.14 void* cds_new_longlong (const long long *n*)

Dynamically allocates memory for a new `long long`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

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

Dynamically allocates memory for a new string.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.18.2.16 `void* cds_new_uint (const unsigned int n)`

Dynamically allocates memory for a new `unsigned int`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.18.2.17 `void* cds_new_ulong (const unsigned long n)`

Dynamically allocates memory for a new `unsigned long`.

Parameters

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

Returns

A `void` pointer to the allocated memory.

4.18.2.18 `void* cds_new_ulonglong (const unsigned long long n)`

Allocates memory for a new `unsigned long long`.

Parameters

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

Returns

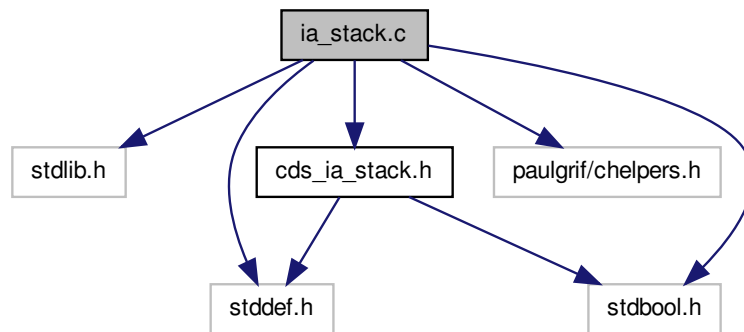
A `void` pointer to the allocated memory.

4.19 ia_stack.c File Reference

Implementation of integer array stack functions.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_ia_stack.h"
```

Include dependency graph for `ia_stack.c`:



Data Structures

- struct [ia_stack_t](#)
Struct to hold an integer array stack.

Typedefs

- typedef struct [ia_stack_t](#) [ia_stack_t](#)
Struct to hold an integer array stack.

Functions

- [ia_stack](#) [ia_stack_init](#) (const [size_t](#) size)
Constructs and initializes a new stack.
- void [ia_stack_free](#) ([ia_stack](#) stk)
Destructs and frees a stack.
- bool [is_stack_isempty](#) (const [ia_stack](#) stk)
Checks if a stack is empty.
- bool [ia_stack_isfull](#) (const [ia_stack](#) stk)
Checks if a stack is full.
- int [ia_stack_peek](#) (const [ia_stack](#) stk)
Returns the top element of the stack without popping it.
- int [ia_stack_pop](#) ([ia_stack](#) stk)
Pops the top element of the stack.
- void [ia_stack_push](#) ([ia_stack](#) stk, const int n)
Pushes an element onto the stack.

4.19.1 Detailed Description

Implementation of integer array stack functions.

Author

Paul Griffiths

Copyright

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

4.19.2 Function Documentation**4.19.2.1 void ia_stack_free (ia_stack *stk*)**

Destructs and frees a stack.

Parameters

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

4.19.2.2 ia_stack ia_stack_init (const size_t *size*)

Constructs and initializes a new stack.

Parameters

<i>size</i>	The initial size of the new stack.
-------------	------------------------------------

Returns

A pointer to the new stack.

4.19.2.3 bool ia_stack_isfull (const ia_stack *stk*)

Checks if a stack is full.

Parameters

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

Returns

`true` if the stack is full, `false` otherwise.

4.19.2.4 int ia_stack_peek (const ia_stack *stk*)

Returns the top element of the stack without popping it.

Parameters

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

Returns

The value of the element at the top.

4.19.2.5 int ia_stack_pop (ia_stack stk)

Pops the top element of the stack.

Parameters

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

Returns

The popped element of the stack.

4.19.2.6 void ia_stack_push (ia_stack stk, const int n)

Pushes an element onto the stack.

If the stack is full, this function attempts to reallocate memory for the array. Each time the stack is full, the amount of memory requested is double the original amount.

Parameters

<i>stk</i>	A pointer to the stack.
<i>n</i>	The value to push onto the stack.

4.19.2.7 bool is_stack_isempty (const ia_stack stk)

Checks if a stack is empty.

Parameters

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

Returns

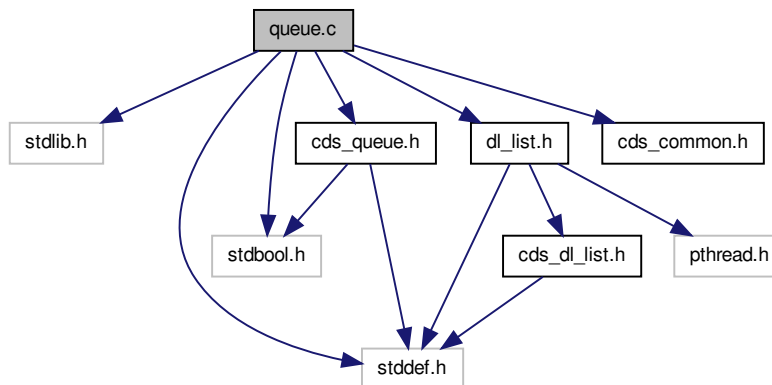
`true` is the stack is empty, `false` otherwise.

4.20 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.
- `void queue_lock (queue que)`
Locks a queue's mutex.
- `void queue_unlock (queue que)`
Unlocks a queue's mutex.

4.20.1 Detailed Description

Implementation of queue data structure. 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/>

4.20.2 Function Documentation

4.20.2.1 void queue_free (queue *que*)

Frees memory and releases resources used by a queue.

Parameters

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

4.20.2.2 queue queue_init (void(*) (void *) *free_func*)

Initializes a new queue.

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.

4.20.2.3 bool queue_isempty (const queue *que*)

Checks if a queue is empty.

Parameters

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

Returns

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

4.20.2.4 size_t queue_length (const queue *que*)

Gets the number of items in a queue.

Parameters

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

Returns

The number of items in the queue.

4.20.2.5 void queue_lock (queue *que*)

Locks a queue's mutex.

Parameters

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

4.20.2.6 void* queue_pop (queue *que*)

Pops a data item from the queue.

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.

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

Pushes a data item onto the back of the queue.

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.

4.20.2.8 void queue_unlock (queue *que*)

Unlocks a queue's mutex.

Parameters

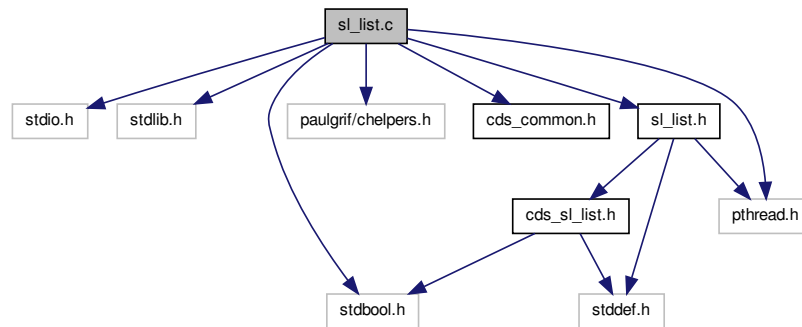
<i>que</i>	A pointer to the queue.
------------	-------------------------

4.21 sl_list.c File Reference

Implementation of singly linked list data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "sl_list.h"
#include <pthread.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.
- `void sl_list_lock (sl_list list)`
Locks a list's mutex.
- `void sl_list_unlock (sl_list list)`
Unlocks a list's mutex.

4.21.1 Detailed Description

Implementation of singly linked 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/>

4.21.2 Function Documentation

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

Returns a pointer to the data at a specified 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.

4.21.2.2 int sl_list_delete_at (sl_list list, const size_t index)

Deletes a list element at a specified 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.

4.21.2.3 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.

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.

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

Gets an index to the specified data in a list.

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.

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

Gets an iterator to the specified data in a list.

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.

4.21.2.6 `sl_list_itr sl_list_first (const sl_list list)`

Returns an iterator to the first element of a list.

Parameters

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

Returns

An iterator to the first element.

4.21.2.7 `void sl_list_free (sl_list list)`

Frees the resources associated with a list.

Parameters

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

4.21.2.8 `void sl_list.free_node (sl_list list, sl_list_node node)`

Frees resources for a node and any data.

Parameters

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

4.21.2.9 `sl_list sl_list.init (int(*)(const void *, const void *) cfunc, void(*)(void *) free_func)`

Initializes a new singly linked list.

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.

4.21.2.10 `int sl_list.insert_after (sl_list list, const sl_list_itr itr, void * data)`

Inserts an element after a provided iterator.

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.

4.21.2.11 `int sl_list.insert_at (sl_list list, const size_t index, void * data)`

Inserts an element at the specified index of a list.

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.

4.21.2.12 `bool sl_list_isempty (const sl_list list)`

Checks if a list is empty.

Parameters

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

Returns

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

4.21.2.13 `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.

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.

4.21.2.14 `size_t sl_list_length (const sl_list list)`

Returns the number of elements in a list.

Parameters

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

4.21.2.15 `void sl_list_lock (sl_list list)`

Locks a list's mutex.

Parameters

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

4.21.2.16 `sl_list_node sl_list_new_node (void * data)`

Creates a new list node.

Parameters

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

Returns

A pointer to the newly created node.

4.21.2.17 `sl_list_itr sl_list.next (const sl_list_itr itr)`

Advances a list iterator by one element.

Parameters

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

Returns

The advanced iterator.

4.21.2.18 `void sl_list.prepend (sl_list list, void * data)`

Inserts an element at the beginning of a list.

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.

4.21.2.19 `sl_list_node sl_list.remove_at (sl_list list, const size_t index)`

Removes, but does not delete, an element at an 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\(\)](#).

4.21.2.20 `void sl_list.unlock (sl_list list)`

Unlocks a list's mutex.

Parameters

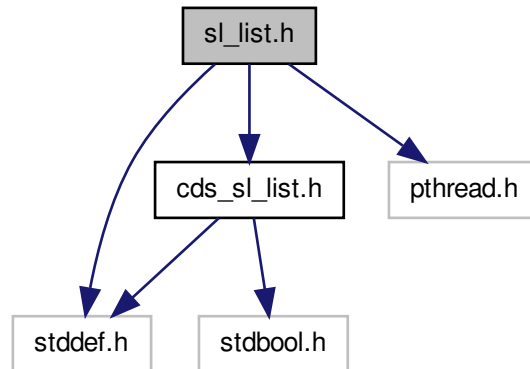
<i>list</i>	A pointer to the list.
-------------	------------------------

4.22 `sl_list.h` File Reference

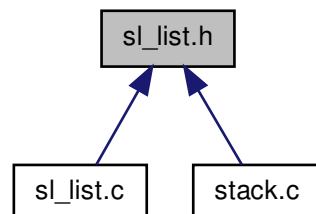
Developer interface to singly linked list data structure.

```
#include <stddef.h>
#include "cds_sl_list.h"
#include <pthread.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.

Macros

- `#define _POSIX_C_SOURCE 200809L`
Enables POSIX library.

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.

4.22.1 Detailed Description

Developer interface to singly linked 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/>

4.22.2 Function Documentation

4.22.2.1 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.

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.

4.22.2.2 void [sl_list_free_node](#) ([sl_list](#) *list*, [sl_list_node](#) *node*)

Frees resources for a node and any data.

Parameters

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

4.22.2.3 `sl_list_node sl_list_new_node (void * data)`

Creates a new list node.

Parameters

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

Returns

A pointer to the newly created node.

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

Removes, but does not delete, an element at an index.

Parameters

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

Returns

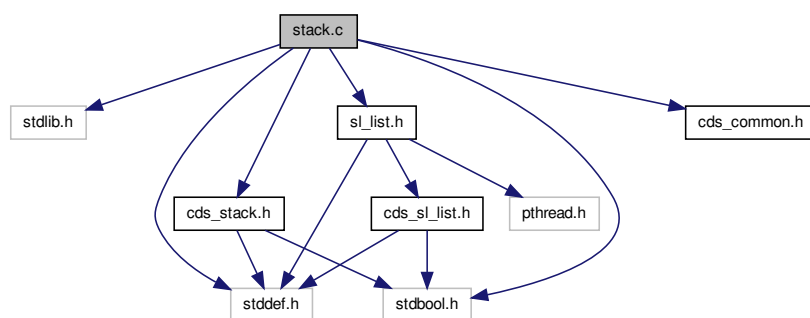
A pointer to the removed node. This should be `free()` d by calling [sl_list_free_node\(\)](#).

4.23 `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_peek](#) ([stack](#) stk)
- Peeks at the data for the top element of the stack.*
- void [stack_push](#) ([stack](#) stk, void *data)
- Pushes a data item onto the stack.*
- void [stack_lock](#) ([stack](#) stk)
- Locks a stack's mutex.*
- void [stack_unlock](#) ([stack](#) stk)
- Unlocks a stack's mutex.*

4.23.1 Detailed Description

Implementation of stack data structure. 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/>

4.23.2 Function Documentation

4.23.2.1 void stack_free ([stack](#) stk)

Frees memory and releases resources used by a stack.

Parameters

stk	A pointer to the stack.
---------------------	-------------------------

4.23.2.2 [stack](#) stack_init (void(*)([void](#) *) [free_func](#))

Initializes a new stack.

Parameters

free_func	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.

4.23.2.3 bool stack_isempty (const stack *stk*)

Checks if a stack is empty.

Parameters

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

Returns

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

4.23.2.4 size_t stack_length (const stack *stk*)

Gets the number of items in a stack.

Parameters

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

Returns

The number of items in the stack.

4.23.2.5 void stack_lock (stack *stk*)

Locks a stack's mutex.

Parameters

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

4.23.2.6 void* stack_peek (stack *stk*)

Peeks at the data for the top element of the stack.

The top item is not popped from the stack, and the user should not `free()` the pointer from this function.

Parameters

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

Returns

A `void` pointer to the popped data item.

4.23.2.7 void* stack_pop (stack *stk*)

Pops a data item from the stack.

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.

4.23.2.8 void stack_push (stack *stk*, void * *data*)

Pushes a data item onto the stack.

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.

4.23.2.9 void stack_unlock (stack *stk*)

Unlocks a stack's mutex.

Parameters

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

Index

- back
 - dl_list_t, 9
- bs_tree.c, 15
 - bs_tree_free, 17
 - bs_tree_free_subtree, 17
 - bs_tree_init, 17
 - bs_tree_inorder_left_traverse, 17
 - bs_tree_inorder_left_traverse_int, 18
 - bs_tree_inorder_right_traverse, 18
 - bs_tree_inorder_right_traverse_int, 18
 - bs_tree_insert, 18
 - bs_tree_insert_search, 19
 - bs_tree_insert_subtree, 19
 - bs_tree_isempty, 19
 - bs_tree_length, 20
 - bs_tree_lock, 20
 - bs_tree_new_node, 20
 - bs_tree_postorder_left_traverse, 20
 - bs_tree_postorder_left_traverse_int, 20
 - bs_tree_postorder_right_traverse, 21
 - bs_tree_postorder_right_traverse_int, 21
 - bs_tree_preorder_left_traverse, 21
 - bs_tree_preorder_left_traverse_int, 21
 - bs_tree_preorder_right_traverse, 21
 - bs_tree_preorder_right_traverse_int, 22
 - bs_tree_search, 22
 - bs_tree_search_data, 22
 - bs_tree_search_node, 22
 - bs_tree_unlock, 23
- bs_tree.h, 23
 - bs_tree_free_subtree, 25
 - bs_tree_inorder_left_traverse_int, 25
 - bs_tree_inorder_right_traverse_int, 25
 - bs_tree_insert_search, 26
 - bs_tree_insert_subtree, 26
 - bs_tree_new_node, 26
 - bs_tree_postorder_left_traverse_int, 27
 - bs_tree_postorder_right_traverse_int, 27
 - bs_tree_preorder_left_traverse_int, 27
 - bs_tree_preorder_right_traverse_int, 27
 - bs_tree_search_node, 28
- bs_tree_free
 - bs_tree.c, 17
 - cds_bs_tree.h, 34
- bs_tree_free_subtree
 - bs_tree.c, 17
 - bs_tree.h, 25
- bs_tree_init
 - bs_tree.c, 17
- bs_tree_inorder_left_traverse
 - bs_tree.c, 17
 - cds_bs_tree.h, 35
- bs_tree_inorder_left_traverse_int
 - bs_tree.c, 18
 - bs_tree.h, 25
- bs_tree_inorder_right_traverse
 - bs_tree.c, 18
 - cds_bs_tree.h, 35
- bs_tree_inorder_right_traverse_int
 - bs_tree.c, 18
 - bs_tree.h, 25
- bs_tree_insert
 - bs_tree.c, 18
 - cds_bs_tree.h, 35
- bs_tree_insert_search
 - bs_tree.c, 19
 - bs_tree.h, 26
- bs_tree_insert_subtree
 - bs_tree.c, 19
 - bs_tree.h, 26
- bs_tree_isempty
 - bs_tree.c, 19
 - cds_bs_tree.h, 35
- bs_tree_length
 - bs_tree.c, 20
 - cds_bs_tree.h, 35
- bs_tree_lock
 - bs_tree.c, 20
 - cds_bs_tree.h, 36
- bs_tree_new_node
 - bs_tree.c, 20
 - bs_tree.h, 26
- bs_tree_node_t, 5
 - data, 5
 - left, 5
 - right, 5
- bs_tree_postorder_left_traverse
 - bs_tree.c, 20
 - cds_bs_tree.h, 36
- bs_tree_postorder_left_traverse_int
 - bs_tree.c, 20
 - bs_tree.h, 27
- bs_tree_postorder_right_traverse
 - bs_tree.c, 21
 - cds_bs_tree.h, 36
- bs_tree_postorder_right_traverse_int
 - bs_tree.c, 21

- bs_tree.h, 27
- bs_tree_preorder_left_traverse
 - bs_tree.c, 21
 - cds_bs_tree.h, 36
- bs_tree_preorder_left_traverse_int
 - bs_tree.c, 21
 - bs_tree.h, 27
- bs_tree_preorder_right_traverse
 - bs_tree.c, 21
 - cds_bs_tree.h, 36
- bs_tree_preorder_right_traverse_int
 - bs_tree.c, 22
 - bs_tree.h, 27
- bs_tree_search
 - bs_tree.c, 22
 - cds_bs_tree.h, 37
- bs_tree_search_data
 - bs_tree.c, 22
 - cds_bs_tree.h, 37
- bs_tree_search_node
 - bs_tree.c, 22
 - bs_tree.h, 28
- bs_tree_t, 6
 - cfunc, 6
 - free_func, 6
 - length, 6
 - mutex, 7
 - root, 7
- bs_tree_unlock
 - bs_tree.c, 23
 - cds_bs_tree.h, 37
- bst_map.c, 28
 - bst_map_free, 29
 - bst_map_init, 29
 - bst_map_insert, 30
 - bst_map_isempty, 30
 - bst_map_length, 30
 - bst_map_lock, 30
 - bst_map_search, 30
 - bst_map_search_data, 31
 - bst_map_unlock, 31
- bst_map_free
 - bst_map.c, 29
 - cds_bst_map.h, 39
- bst_map_init
 - bst_map.c, 29
 - cds_bst_map.h, 39
- bst_map_insert
 - bst_map.c, 30
 - cds_bst_map.h, 39
- bst_map_isempty
 - bst_map.c, 30
 - cds_bst_map.h, 40
- bst_map_length
 - bst_map.c, 30
 - cds_bst_map.h, 40
- bst_map_lock
 - bst_map.c, 30
- cds_bst_map.h, 40
- bst_map_search
 - bst_map.c, 30
 - cds_bst_map.h, 40
- bst_map_search_data
 - bst_map.c, 31
 - cds_bst_map.h, 40
- bst_map_unlock
 - bst_map.c, 31
 - cds_bst_map.h, 41
- CDSERR_BADITERATOR
 - cds_common.h, 42
- CDSERR_ERROR
 - cds_common.h, 42
- CDSERR_NOTFOUND
 - cds_common.h, 42
- CDSERR_OUTOFRANGE
 - cds_common.h, 42
- cdatastruct.h, 31
- cds_common.h
 - CDSERR_BADITERATOR, 42
 - CDSERR_ERROR, 42
 - CDSERR_NOTFOUND, 42
 - CDSERR_OUTOFRANGE, 42
- cds_bs_tree.h, 32
 - bs_tree_free, 34
 - bs_tree_init, 34
 - bs_tree_inorder_left_traverse, 35
 - bs_tree_inorder_right_traverse, 35
 - bs_tree_insert, 35
 - bs_tree_isempty, 35
 - bs_tree_length, 35
 - bs_tree_lock, 36
 - bs_tree_postorder_left_traverse, 36
 - bs_tree_postorder_right_traverse, 36
 - bs_tree_preorder_left_traverse, 36
 - bs_tree_preorder_right_traverse, 36
 - bs_tree_search, 37
 - bs_tree_search_data, 37
 - bs_tree_unlock, 37
- cds_bst_map.h, 37
 - bst_map_free, 39
 - bst_map_init, 39
 - bst_map_insert, 39
 - bst_map_isempty, 40
 - bst_map_length, 40
 - bst_map_lock, 40
 - bst_map_search, 40
 - bst_map_search_data, 40
 - bst_map_unlock, 41
- cds_common.h, 41
 - cds_error, 42
- cds_compare_double
 - cds_general.h, 53
 - general.c, 91
- cds_compare_float
 - cds_general.h, 54
 - general.c, 91

- cds_compare_int
 - cds_general.h, 54
 - general.c, 91
- cds_compare_long
 - cds_general.h, 54
 - general.c, 92
- cds_compare_longlong
 - cds_general.h, 54
 - general.c, 92
- cds_compare_string
 - cds_general.h, 55
 - general.c, 92
- cds_compare_uint
 - cds_general.h, 55
 - general.c, 92
- cds_compare_ulong
 - cds_general.h, 55
 - general.c, 93
- cds_compare_ulonglong
 - cds_general.h, 55
 - general.c, 93
- cds_da_stack.h, 42
 - da_stack_free, 44
 - da_stack_init, 44
 - da_stack_isfull, 44
 - da_stack_peek, 44
 - da_stack_pop, 44
 - da_stack_push, 45
 - is_stack_isempty, 45
- cds_dl_list.h, 45
 - dl_list_append, 47
 - dl_list_data, 47
 - dl_list_delete_at, 48
 - dl_list_find_index, 48
 - dl_list_find_itr, 48
 - dl_list_first, 48
 - dl_list_free, 49
 - dl_list_init, 49
 - dl_list_insert_after, 49
 - dl_list_insert_at, 50
 - dl_list_insert_before, 50
 - dl_list_isempty, 50
 - dl_list_itr_from_index, 50
 - dl_list_last, 51
 - dl_list_length, 51
 - dl_list_lock, 51
 - dl_list_next, 51
 - dl_list_prepend, 51
 - dl_list_prev, 52
 - dl_list_unlock, 52
- cds_error
 - cds_common.h, 42
- cds_general.h, 52
 - cds_compare_double, 53
 - cds_compare_float, 54
 - cds_compare_int, 54
 - cds_compare_long, 54
 - cds_compare_longlong, 54
 - cds_compare_string, 55
 - cds_compare_uint, 55
 - cds_compare_ulong, 55
 - cds_compare_ulonglong, 55
 - cds_new_double, 56
 - cds_new_float, 56
 - cds_new_int, 56
 - cds_new_long, 56
 - cds_new_longlong, 56
 - cds_new_string, 57
 - cds_new_uint, 57
 - cds_new_ulong, 57
 - cds_new_ulonglong, 57
- cds_ia_stack.h, 58
 - ia_stack_free, 59
 - ia_stack_init, 59
 - ia_stack_isfull, 59
 - ia_stack_peek, 60
 - ia_stack_pop, 60
 - ia_stack_push, 60
 - is_stack_isempty, 60
- cds_new_double
 - cds_general.h, 56
 - general.c, 93
- cds_new_float
 - cds_general.h, 56
 - general.c, 93
- cds_new_int
 - cds_general.h, 56
 - general.c, 94
- cds_new_long
 - cds_general.h, 56
 - general.c, 94
- cds_new_longlong
 - cds_general.h, 56
 - general.c, 94
- cds_new_string
 - cds_general.h, 57
 - general.c, 94
- cds_new_uint
 - cds_general.h, 57
 - general.c, 95
- cds_new_ulong
 - cds_general.h, 57
 - general.c, 95
- cds_new_ulonglong
 - cds_general.h, 57
 - general.c, 95
- cds_queue.h, 61
 - queue_free, 62
 - queue_init, 62
 - queue_isempty, 62
 - queue_length, 63
 - queue_lock, 63
 - queue_pop, 63
 - queue_pushback, 63
 - queue_unlock, 63
- cds_sl_list.h, 64

- sl_list_data, 66
- sl_list_delete_at, 66
- sl_list_find_index, 66
- sl_list_find_itr, 66
- sl_list_first, 67
- sl_list_free, 67
- sl_list_init, 67
- sl_list_insert_after, 67
- sl_list_insert_at, 68
- sl_list_isempty, 68
- sl_list_itr_from_index, 68
- sl_list_length, 68
- sl_list_lock, 69
- sl_list_next, 69
- sl_list_prepend, 69
- sl_list_unlock, 69
- cds_stack.h, 69
 - stack_free, 71
 - stack_init, 71
 - stack_isempty, 71
 - stack_length, 72
 - stack_lock, 72
 - stack_peek, 72
 - stack_pop, 72
 - stack_push, 72
 - stack_unlock, 73
- cfunc
 - bs_tree_t, 6
 - dl_list_t, 9
 - sl_list_t, 12
- da_stack.c, 73
 - da_stack_free, 74
 - da_stack_init, 74
 - da_stack_isfull, 75
 - da_stack_peek, 75
 - da_stack_pop, 75
 - da_stack_push, 75
 - is_stack_isempty, 75
- da_stack_free
 - cds_da_stack.h, 44
 - da_stack.c, 74
- da_stack_init
 - cds_da_stack.h, 44
 - da_stack.c, 74
- da_stack_isfull
 - cds_da_stack.h, 44
 - da_stack.c, 75
- da_stack_peek
 - cds_da_stack.h, 44
 - da_stack.c, 75
- da_stack_pop
 - cds_da_stack.h, 44
 - da_stack.c, 75
- da_stack_push
 - cds_da_stack.h, 45
 - da_stack.c, 75
- da_stack_t, 7
 - length, 7
- stack, 7
 - top, 7
- data
 - bs_tree_node_t, 5
 - dl_list_node_t, 8
 - sl_list_node_t, 12
- dl_list.c, 76
 - dl_list_append, 78
 - dl_list_data, 78
 - dl_list_delete_at, 78
 - dl_list_find, 78
 - dl_list_find_index, 79
 - dl_list_find_itr, 79
 - dl_list_first, 79
 - dl_list_free, 79
 - dl_list_free_node, 80
 - dl_list_init, 80
 - dl_list_insert_after, 80
 - dl_list_insert_at, 80
 - dl_list_insert_before, 81
 - dl_list_insert_node_after_mid, 81
 - dl_list_insert_node_back, 81
 - dl_list_insert_node_before_mid, 81
 - dl_list_insert_node_front, 82
 - dl_list_isempty, 82
 - dl_list_itr_from_index, 82
 - dl_list_last, 82
 - dl_list_length, 82
 - dl_list_lock, 83
 - dl_list_new_node, 83
 - dl_list_next, 83
 - dl_list_prepend, 83
 - dl_list_prev, 83
 - dl_list_remove_at, 84
 - dl_list_remove_node_back, 84
 - dl_list_remove_node_front, 84
 - dl_list_remove_node_mid, 84
 - dl_list_unlock, 85
- dl_list.h, 85
 - dl_list_find, 87
 - dl_list_free_node, 87
 - dl_list_insert_node_after_mid, 87
 - dl_list_insert_node_back, 88
 - dl_list_insert_node_before_mid, 88
 - dl_list_insert_node_front, 88
 - dl_list_new_node, 88
 - dl_list_remove_at, 88
 - dl_list_remove_node_back, 89
 - dl_list_remove_node_front, 89
 - dl_list_remove_node_mid, 89
- dl_list_append
 - cds_dl_list.h, 47
 - dl_list.c, 78
- dl_list_data
 - cds_dl_list.h, 47
 - dl_list.c, 78
- dl_list_delete_at
 - cds_dl_list.h, 48

- dl_list.c, 78
- dl_list_find
 - dl_list.c, 78
 - dl_list.h, 87
- dl_list_find_index
 - cds_dl_list.h, 48
 - dl_list.c, 79
- dl_list_find_itr
 - cds_dl_list.h, 48
 - dl_list.c, 79
- dl_list_first
 - cds_dl_list.h, 48
 - dl_list.c, 79
- dl_list_free
 - cds_dl_list.h, 49
 - dl_list.c, 79
- dl_list_free_node
 - dl_list.c, 80
 - dl_list.h, 87
- dl_list_init
 - cds_dl_list.h, 49
 - dl_list.c, 80
- dl_list_insert_after
 - cds_dl_list.h, 49
 - dl_list.c, 80
- dl_list_insert_at
 - cds_dl_list.h, 50
 - dl_list.c, 80
- dl_list_insert_before
 - cds_dl_list.h, 50
 - dl_list.c, 81
- dl_list_insert_node_after_mid
 - dl_list.c, 81
 - dl_list.h, 87
- dl_list_insert_node_back
 - dl_list.c, 81
 - dl_list.h, 88
- dl_list_insert_node_before_mid
 - dl_list.c, 81
 - dl_list.h, 88
- dl_list_insert_node_front
 - dl_list.c, 82
 - dl_list.h, 88
- dl_list_isempty
 - cds_dl_list.h, 50
 - dl_list.c, 82
- dl_list_itr_from_index
 - cds_dl_list.h, 50
 - dl_list.c, 82
- dl_list_last
 - cds_dl_list.h, 51
 - dl_list.c, 82
- dl_list_length
 - cds_dl_list.h, 51
 - dl_list.c, 82
- dl_list_lock
 - cds_dl_list.h, 51
 - dl_list.c, 83
- dl_list_new_node
 - dl_list.c, 83
 - dl_list.h, 88
- dl_list_next
 - cds_dl_list.h, 51
 - dl_list.c, 83
- dl_list_node_t, 8
 - data, 8
 - next, 8
 - prev, 8
- dl_list_prepend
 - cds_dl_list.h, 51
 - dl_list.c, 83
- dl_list_prev
 - cds_dl_list.h, 52
 - dl_list.c, 83
- dl_list_remove_at
 - dl_list.c, 84
 - dl_list.h, 88
- dl_list_remove_node_back
 - dl_list.c, 84
 - dl_list.h, 89
- dl_list_remove_node_front
 - dl_list.c, 84
 - dl_list.h, 89
- dl_list_remove_node_mid
 - dl_list.c, 84
 - dl_list.h, 89
- dl_list_t, 8
 - back, 9
 - cfunc, 9
 - free_func, 9
 - front, 9
 - length, 9
 - mutex, 10
- dl_list_unlock
 - cds_dl_list.h, 52
 - dl_list.c, 85
- free_func
 - bs_tree_t, 6
 - dl_list_t, 9
 - sl_list_t, 12
- front
 - dl_list_t, 9
 - sl_list_t, 13
- general.c, 89
 - cds_compare_double, 91
 - cds_compare_float, 91
 - cds_compare_int, 91
 - cds_compare_long, 92
 - cds_compare_longlong, 92
 - cds_compare_string, 92
 - cds_compare_uint, 92
 - cds_compare_ulong, 93
 - cds_compare_ulonglong, 93
 - cds_new_double, 93
 - cds_new_float, 93

- cds_new_int, 94
 - cds_new_long, 94
 - cds_new_longlong, 94
 - cds_new_string, 94
 - cds_new_uint, 95
 - cds_new_ulong, 95
 - cds_new_ulonglong, 95
- ia_stack.c, 95
 - ia_stack_free, 97
 - ia_stack_init, 97
 - ia_stack_isfull, 97
 - ia_stack_peek, 97
 - ia_stack_pop, 97
 - ia_stack_push, 98
 - is_stack_isempty, 98
- ia_stack_free
 - cds_ia_stack.h, 59
 - ia_stack.c, 97
- ia_stack_init
 - cds_ia_stack.h, 59
 - ia_stack.c, 97
- ia_stack_isfull
 - cds_ia_stack.h, 59
 - ia_stack.c, 97
- ia_stack_peek
 - cds_ia_stack.h, 60
 - ia_stack.c, 97
- ia_stack_pop
 - cds_ia_stack.h, 60
 - ia_stack.c, 97
- ia_stack_push
 - cds_ia_stack.h, 60
 - ia_stack.c, 98
- ia_stack_t, 10
 - length, 10
 - stack, 10
 - top, 10
- is_stack_isempty
 - cds_da_stack.h, 45
 - cds_ia_stack.h, 60
 - da_stack.c, 75
 - ia_stack.c, 98
- key
 - kvpair_t, 11
- kvpair_t, 10
 - key, 11
 - value, 11
- left
 - bs_tree_node_t, 5
- length
 - bs_tree_t, 6
 - da_stack_t, 7
 - dl_list_t, 9
 - ia_stack_t, 10
 - sl_list_t, 13
- mutex
 - bs_tree_t, 7
 - dl_list_t, 10
 - sl_list_t, 13
- next
 - dl_list_node_t, 8
 - sl_list_node_t, 12
- prev
 - dl_list_node_t, 8
- queue.c, 98
 - queue_free, 100
 - queue_init, 100
 - queue_isempty, 100
 - queue_length, 100
 - queue_lock, 100
 - queue_pop, 101
 - queue_pushback, 101
 - queue_unlock, 101
- queue_free
 - cds_queue.h, 62
 - queue.c, 100
- queue_init
 - cds_queue.h, 62
 - queue.c, 100
- queue_isempty
 - cds_queue.h, 62
 - queue.c, 100
- queue_length
 - cds_queue.h, 63
 - queue.c, 100
- queue_lock
 - cds_queue.h, 63
 - queue.c, 100
- queue_pop
 - cds_queue.h, 63
 - queue.c, 101
- queue_pushback
 - cds_queue.h, 63
 - queue.c, 101
- queue_unlock
 - cds_queue.h, 63
 - queue.c, 101
- right
 - bs_tree_node_t, 5
- root
 - bs_tree_t, 7
- sl_list.c, 101
 - sl_list_data, 103
 - sl_list_delete_at, 103
 - sl_list_find, 103
 - sl_list_find_index, 104
 - sl_list_find_itr, 104
 - sl_list_first, 104
 - sl_list_free, 104

- sl_list_free_node, 104
- sl_list_init, 105
- sl_list_insert_after, 105
- sl_list_insert_at, 105
- sl_list_isempty, 106
- sl_list_itr_from_index, 106
- sl_list_length, 106
- sl_list_lock, 106
- sl_list_new_node, 106
- sl_list_next, 107
- sl_list_prepend, 107
- sl_list_remove_at, 107
- sl_list_unlock, 107
- sl_list.h, 107
 - sl_list_find, 109
 - sl_list_free_node, 109
 - sl_list_new_node, 109
 - sl_list_remove_at, 110
- sl_list_data
 - cds_sl_list.h, 66
 - sl_list.c, 103
- sl_list_delete_at
 - cds_sl_list.h, 66
 - sl_list.c, 103
- sl_list_find
 - sl_list.c, 103
 - sl_list.h, 109
- sl_list_find_index
 - cds_sl_list.h, 66
 - sl_list.c, 104
- sl_list_find_itr
 - cds_sl_list.h, 66
 - sl_list.c, 104
- sl_list_first
 - cds_sl_list.h, 67
 - sl_list.c, 104
- sl_list_free
 - cds_sl_list.h, 67
 - sl_list.c, 104
- sl_list_free_node
 - sl_list.c, 104
 - sl_list.h, 109
- sl_list_init
 - cds_sl_list.h, 67
 - sl_list.c, 105
- sl_list_insert_after
 - cds_sl_list.h, 67
 - sl_list.c, 105
- sl_list_insert_at
 - cds_sl_list.h, 68
 - sl_list.c, 105
- sl_list_isempty
 - cds_sl_list.h, 68
 - sl_list.c, 106
- sl_list_itr_from_index
 - cds_sl_list.h, 68
 - sl_list.c, 106
- sl_list_length
 - cds_sl_list.h, 68
 - sl_list.c, 106
- sl_list_lock
 - cds_sl_list.h, 69
 - sl_list.c, 106
- sl_list_new_node
 - sl_list.c, 106
 - sl_list.h, 109
- sl_list_next
 - cds_sl_list.h, 69
 - sl_list.c, 107
- sl_list_node_t, 11
 - data, 12
 - next, 12
- sl_list_prepend
 - cds_sl_list.h, 69
 - sl_list.c, 107
- sl_list_remove_at
 - sl_list.c, 107
 - sl_list.h, 110
- sl_list_t, 12
 - cfunc, 12
 - free_func, 12
 - front, 13
 - length, 13
 - mutex, 13
- sl_list_unlock
 - cds_sl_list.h, 69
 - sl_list.c, 107
- stack
 - da_stack_t, 7
 - ia_stack_t, 10
- stack.c, 110
 - stack_free, 111
 - stack_init, 111
 - stack_isempty, 112
 - stack_length, 112
 - stack_lock, 112
 - stack_peek, 112
 - stack_pop, 112
 - stack_push, 113
 - stack_unlock, 113
- stack_free
 - cds_stack.h, 71
 - stack.c, 111
- stack_init
 - cds_stack.h, 71
 - stack.c, 111
- stack_isempty
 - cds_stack.h, 71
 - stack.c, 112
- stack_length
 - cds_stack.h, 72
 - stack.c, 112
- stack_lock
 - cds_stack.h, 72
 - stack.c, 112
- stack_peek

- [cds_stack.h](#), [72](#)
 - [stack.c](#), [112](#)
- [stack_pop](#)
 - [cds_stack.h](#), [72](#)
 - [stack.c](#), [112](#)
- [stack_push](#)
 - [cds_stack.h](#), [72](#)
 - [stack.c](#), [113](#)
- [stack_unlock](#)
 - [cds_stack.h](#), [73](#)
 - [stack.c](#), [113](#)
- [top](#)
 - [da_stack_t](#), [7](#)
 - [ia_stack_t](#), [10](#)
- [value](#)
 - [kvpair_t](#), [11](#)