

cdatastruct

Generated by Doxygen 1.8.1.2

Sun Sep 8 2013 12:39:03



# Contents

<b>1</b>	<b>Data Structure Index</b>	<b>1</b>
1.1	Data Structures . . . . .	1
<b>2</b>	<b>File Index</b>	<b>3</b>
2.1	File List . . . . .	3
<b>3</b>	<b>Data Structure Documentation</b>	<b>5</b>
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	dl_list_node_t Struct Reference . . . . .	7
3.3.1	Detailed Description . . . . .	7
3.3.2	Field Documentation . . . . .	7
3.3.2.1	data . . . . .	7
3.3.2.2	next . . . . .	7
3.3.2.3	prev . . . . .	8
3.4	dl_list_t Struct Reference . . . . .	8
3.4.1	Detailed Description . . . . .	8
3.4.2	Field Documentation . . . . .	8
3.4.2.1	back . . . . .	8
3.4.2.2	cfunc . . . . .	9

3.4.2.3	<a href="#">free_func</a>	9
3.4.2.4	<a href="#">front</a>	9
3.4.2.5	<a href="#">length</a>	9
3.4.2.6	<a href="#">mutex</a>	9
3.5	<a href="#">kvpair_t Struct Reference</a>	9
3.5.1	<a href="#">Detailed Description</a>	9
3.5.2	<a href="#">Field Documentation</a>	9
3.5.2.1	<a href="#">key</a>	9
3.5.2.2	<a href="#">value</a>	9
3.6	<a href="#">sl_list_node_t Struct Reference</a>	10
3.6.1	<a href="#">Detailed Description</a>	10
3.6.2	<a href="#">Field Documentation</a>	10
3.6.2.1	<a href="#">data</a>	10
3.6.2.2	<a href="#">next</a>	10
3.7	<a href="#">sl_list_t Struct Reference</a>	10
3.7.1	<a href="#">Detailed Description</a>	11
3.7.2	<a href="#">Field Documentation</a>	11
3.7.2.1	<a href="#">cfunc</a>	11
3.7.2.2	<a href="#">free_func</a>	11
3.7.2.3	<a href="#">front</a>	11
3.7.2.4	<a href="#">length</a>	11
3.7.2.5	<a href="#">mutex</a>	11
<b>4</b>	<b><a href="#">File Documentation</a></b>	<b>13</b>
4.1	<a href="#">bs_tree.c File Reference</a>	13
4.1.1	<a href="#">Detailed Description</a>	15
4.1.2	<a href="#">Function Documentation</a>	15
4.1.2.1	<a href="#">bs_tree_free</a>	15
4.1.2.2	<a href="#">bs_tree_free_subtree</a>	15
4.1.2.3	<a href="#">bs_tree_init</a>	15
4.1.2.4	<a href="#">bs_tree_inorder_left_traverse</a>	16
4.1.2.5	<a href="#">bs_tree_inorder_left_traverse_int</a>	16
4.1.2.6	<a href="#">bs_tree_inorder_right_traverse</a>	16
4.1.2.7	<a href="#">bs_tree_inorder_right_traverse_int</a>	16
4.1.2.8	<a href="#">bs_tree_insert</a>	16
4.1.2.9	<a href="#">bs_tree_insert_search</a>	17
4.1.2.10	<a href="#">bs_tree_insert_subtree</a>	17
4.1.2.11	<a href="#">bs_tree_isempty</a>	17
4.1.2.12	<a href="#">bs_tree_length</a>	18
4.1.2.13	<a href="#">bs_tree_lock</a>	18

4.1.2.14	<a href="#">bs_tree_new_node</a>	18
4.1.2.15	<a href="#">bs_tree_postorder_left_traverse</a>	18
4.1.2.16	<a href="#">bs_tree_postorder_left_traverse_int</a>	18
4.1.2.17	<a href="#">bs_tree_postorder_right_traverse</a>	19
4.1.2.18	<a href="#">bs_tree_postorder_right_traverse_int</a>	19
4.1.2.19	<a href="#">bs_tree_preorder_left_traverse</a>	19
4.1.2.20	<a href="#">bs_tree_preorder_left_traverse_int</a>	19
4.1.2.21	<a href="#">bs_tree_preorder_right_traverse</a>	20
4.1.2.22	<a href="#">bs_tree_preorder_right_traverse_int</a>	20
4.1.2.23	<a href="#">bs_tree_search</a>	20
4.1.2.24	<a href="#">bs_tree_search_data</a>	20
4.1.2.25	<a href="#">bs_tree_search_node</a>	20
4.1.2.26	<a href="#">bs_tree_unlock</a>	21
4.2	<a href="#">bs_tree.h File Reference</a>	21
4.2.1	<a href="#">Detailed Description</a>	23
4.2.2	<a href="#">Function Documentation</a>	23
4.2.2.1	<a href="#">bs_tree_free_subtree</a>	23
4.2.2.2	<a href="#">bs_tree_inorder_left_traverse_int</a>	23
4.2.2.3	<a href="#">bs_tree_inorder_right_traverse_int</a>	24
4.2.2.4	<a href="#">bs_tree_insert_search</a>	24
4.2.2.5	<a href="#">bs_tree_insert_subtree</a>	24
4.2.2.6	<a href="#">bs_tree_new_node</a>	24
4.2.2.7	<a href="#">bs_tree_postorder_left_traverse_int</a>	25
4.2.2.8	<a href="#">bs_tree_postorder_right_traverse_int</a>	25
4.2.2.9	<a href="#">bs_tree_preorder_left_traverse_int</a>	25
4.2.2.10	<a href="#">bs_tree_preorder_right_traverse_int</a>	25
4.2.2.11	<a href="#">bs_tree_search_node</a>	26
4.3	<a href="#">bst_map.c File Reference</a>	26
4.3.1	<a href="#">Detailed Description</a>	27
4.3.2	<a href="#">Function Documentation</a>	27
4.3.2.1	<a href="#">bst_map_free</a>	27
4.3.2.2	<a href="#">bst_map_init</a>	27
4.3.2.3	<a href="#">bst_map_insert</a>	28
4.3.2.4	<a href="#">bst_map_isempty</a>	28
4.3.2.5	<a href="#">bst_map_length</a>	28
4.3.2.6	<a href="#">bst_map_lock</a>	28
4.3.2.7	<a href="#">bst_map_search</a>	29
4.3.2.8	<a href="#">bst_map_search_data</a>	29
4.3.2.9	<a href="#">bst_map_unlock</a>	29
4.4	<a href="#">cdatastruct.h File Reference</a>	29

4.4.1	Detailed Description	30
4.5	cds_bs_tree.h File Reference	30
4.5.1	Detailed Description	32
4.5.2	Function Documentation	32
4.5.2.1	bs_tree_free	32
4.5.2.2	bs_tree_init	32
4.5.2.3	bs_tree_inorder_left_traverse	33
4.5.2.4	bs_tree_inorder_right_traverse	33
4.5.2.5	bs_tree_insert	33
4.5.2.6	bs_tree_isempty	33
4.5.2.7	bs_tree_length	34
4.5.2.8	bs_tree_lock	34
4.5.2.9	bs_tree_postorder_left_traverse	34
4.5.2.10	bs_tree_postorder_right_traverse	34
4.5.2.11	bs_tree_preorder_left_traverse	34
4.5.2.12	bs_tree_preorder_right_traverse	35
4.5.2.13	bs_tree_search	35
4.5.2.14	bs_tree_search_data	35
4.5.2.15	bs_tree_unlock	35
4.6	cds_bst_map.h File Reference	35
4.6.1	Detailed Description	37
4.6.2	Function Documentation	37
4.6.2.1	bst_map_free	37
4.6.2.2	bst_map_init	37
4.6.2.3	bst_map_insert	37
4.6.2.4	bst_map_isempty	38
4.6.2.5	bst_map_length	38
4.6.2.6	bst_map_lock	38
4.6.2.7	bst_map_search	38
4.6.2.8	bst_map_search_data	38
4.6.2.9	bst_map_unlock	39
4.7	cds_common.h File Reference	39
4.7.1	Detailed Description	39
4.7.2	Enumeration Type Documentation	40
4.7.2.1	cds_error	40
4.8	cds_dl_list.h File Reference	40
4.8.1	Detailed Description	42
4.8.2	Function Documentation	42
4.8.2.1	dl_list_append	42
4.8.2.2	dl_list_data	43

4.8.2.3	<a href="#">dl_list_delete_at</a>	43
4.8.2.4	<a href="#">dl_list_find_index</a>	43
4.8.2.5	<a href="#">dl_list_find_itr</a>	43
4.8.2.6	<a href="#">dl_list_first</a>	44
4.8.2.7	<a href="#">dl_list_free</a>	44
4.8.2.8	<a href="#">dl_list_init</a>	44
4.8.2.9	<a href="#">dl_list_insert_after</a>	44
4.8.2.10	<a href="#">dl_list_insert_at</a>	45
4.8.2.11	<a href="#">dl_list_insert_before</a>	45
4.8.2.12	<a href="#">dl_list_isempty</a>	45
4.8.2.13	<a href="#">dl_list_itr_from_index</a>	45
4.8.2.14	<a href="#">dl_list_last</a>	46
4.8.2.15	<a href="#">dl_list_length</a>	46
4.8.2.16	<a href="#">dl_list_lock</a>	46
4.8.2.17	<a href="#">dl_list_next</a>	46
4.8.2.18	<a href="#">dl_list_prepend</a>	46
4.8.2.19	<a href="#">dl_list_prev</a>	47
4.8.2.20	<a href="#">dl_list_unlock</a>	47
4.9	<a href="#">cds_general.h File Reference</a>	47
4.9.1	<a href="#">Detailed Description</a>	48
4.9.2	<a href="#">Function Documentation</a>	48
4.9.2.1	<a href="#">cds_compare_double</a>	48
4.9.2.2	<a href="#">cds_compare_float</a>	49
4.9.2.3	<a href="#">cds_compare_int</a>	49
4.9.2.4	<a href="#">cds_compare_long</a>	49
4.9.2.5	<a href="#">cds_compare_longlong</a>	49
4.9.2.6	<a href="#">cds_compare_string</a>	50
4.9.2.7	<a href="#">cds_compare_uint</a>	50
4.9.2.8	<a href="#">cds_compare_ulong</a>	50
4.9.2.9	<a href="#">cds_compare_ulonglong</a>	50
4.9.2.10	<a href="#">cds_new_double</a>	51
4.9.2.11	<a href="#">cds_new_float</a>	51
4.9.2.12	<a href="#">cds_new_int</a>	51
4.9.2.13	<a href="#">cds_new_long</a>	51
4.9.2.14	<a href="#">cds_new_longlong</a>	52
4.9.2.15	<a href="#">cds_new_string</a>	52
4.9.2.16	<a href="#">cds_new_uint</a>	52
4.9.2.17	<a href="#">cds_new_ulong</a>	52
4.9.2.18	<a href="#">cds_new_ulonglong</a>	52
4.10	<a href="#">cds_queue.h File Reference</a>	53

4.10.1 Detailed Description . . . . .	54
4.10.2 Function Documentation . . . . .	54
4.10.2.1 queue_free . . . . .	54
4.10.2.2 queue_init . . . . .	54
4.10.2.3 queue_isempty . . . . .	55
4.10.2.4 queue_length . . . . .	55
4.10.2.5 queue_lock . . . . .	55
4.10.2.6 queue_pop . . . . .	55
4.10.2.7 queue_pushback . . . . .	55
4.10.2.8 queue_unlock . . . . .	56
4.11 cds_sl_list.h File Reference . . . . .	56
4.11.1 Detailed Description . . . . .	58
4.11.2 Function Documentation . . . . .	58
4.11.2.1 sl_list_data . . . . .	58
4.11.2.2 sl_list_delete_at . . . . .	58
4.11.2.3 sl_list_find_index . . . . .	59
4.11.2.4 sl_list_find_itr . . . . .	59
4.11.2.5 sl_list_first . . . . .	59
4.11.2.6 sl_list_free . . . . .	59
4.11.2.7 sl_list_init . . . . .	60
4.11.2.8 sl_list_insert_after . . . . .	60
4.11.2.9 sl_list_insert_at . . . . .	60
4.11.2.10 sl_list_isempty . . . . .	60
4.11.2.11 sl_list_itr_from_index . . . . .	61
4.11.2.12 sl_list_length . . . . .	61
4.11.2.13 sl_list_lock . . . . .	61
4.11.2.14 sl_list_next . . . . .	61
4.11.2.15 sl_list_prepend . . . . .	61
4.11.2.16 sl_list_unlock . . . . .	62
4.12 cds_stack.h File Reference . . . . .	62
4.12.1 Detailed Description . . . . .	63
4.12.2 Function Documentation . . . . .	63
4.12.2.1 stack_free . . . . .	63
4.12.2.2 stack_init . . . . .	63
4.12.2.3 stack_isempty . . . . .	64
4.12.2.4 stack_length . . . . .	64
4.12.2.5 stack_lock . . . . .	64
4.12.2.6 stack_pop . . . . .	64
4.12.2.7 stack_push . . . . .	65
4.12.2.8 stack_unlock . . . . .	65



4.13 dl_list.c File Reference	65
4.13.1 Detailed Description	67
4.13.2 Function Documentation	67
4.13.2.1 dl_list_append	67
4.13.2.2 dl_list_data	67
4.13.2.3 dl_list_delete_at	67
4.13.2.4 dl_list_find	68
4.13.2.5 dl_list_find_index	68
4.13.2.6 dl_list_find_itr	68
4.13.2.7 dl_list_first	68
4.13.2.8 dl_list_free	69
4.13.2.9 dl_list_free_node	69
4.13.2.10 dl_list_init	69
4.13.2.11 dl_list_insert_after	69
4.13.2.12 dl_list_insert_at	70
4.13.2.13 dl_list_insert_before	70
4.13.2.14 dl_list_insert_node_after_mid	70
4.13.2.15 dl_list_insert_node_back	71
4.13.2.16 dl_list_insert_node_before_mid	71
4.13.2.17 dl_list_insert_node_front	71
4.13.2.18 dl_list_isempty	71
4.13.2.19 dl_list_itr_from_index	71
4.13.2.20 dl_list_last	72
4.13.2.21 dl_list_length	72
4.13.2.22 dl_list_lock	72
4.13.2.23 dl_list_new_node	72
4.13.2.24 dl_list_next	72
4.13.2.25 dl_list_prepend	73
4.13.2.26 dl_list_prev	73
4.13.2.27 dl_list_remove_at	73
4.13.2.28 dl_list_remove_node_back	73
4.13.2.29 dl_list_remove_node_front	74
4.13.2.30 dl_list_remove_node_mid	74
4.13.2.31 dl_list_unlock	74
4.14 dl_list.h File Reference	74
4.14.1 Detailed Description	76
4.14.2 Function Documentation	76
4.14.2.1 dl_list_find	76
4.14.2.2 dl_list_free_node	77
4.14.2.3 dl_list_insert_node_after_mid	77

4.14.2.4	<a href="#">dl_list_insert_node_back</a>	77
4.14.2.5	<a href="#">dl_list_insert_node_before_mid</a>	77
4.14.2.6	<a href="#">dl_list_insert_node_front</a>	77
4.14.2.7	<a href="#">dl_list_new_node</a>	78
4.14.2.8	<a href="#">dl_list_remove_at</a>	78
4.14.2.9	<a href="#">dl_list_remove_node_back</a>	78
4.14.2.10	<a href="#">dl_list_remove_node_front</a>	78
4.14.2.11	<a href="#">dl_list_remove_node_mid</a>	78
4.15	<a href="#">general.c File Reference</a>	79
4.15.1	<a href="#">Detailed Description</a>	80
4.15.2	<a href="#">Function Documentation</a>	80
4.15.2.1	<a href="#">cds_compare_double</a>	80
4.15.2.2	<a href="#">cds_compare_float</a>	80
4.15.2.3	<a href="#">cds_compare_int</a>	81
4.15.2.4	<a href="#">cds_compare_long</a>	81
4.15.2.5	<a href="#">cds_compare_longlong</a>	81
4.15.2.6	<a href="#">cds_compare_string</a>	82
4.15.2.7	<a href="#">cds_compare_uint</a>	82
4.15.2.8	<a href="#">cds_compare_ulong</a>	82
4.15.2.9	<a href="#">cds_compare_ulonglong</a>	82
4.15.2.10	<a href="#">cds_new_double</a>	83
4.15.2.11	<a href="#">cds_new_float</a>	83
4.15.2.12	<a href="#">cds_new_int</a>	83
4.15.2.13	<a href="#">cds_new_long</a>	83
4.15.2.14	<a href="#">cds_new_longlong</a>	84
4.15.2.15	<a href="#">cds_new_string</a>	84
4.15.2.16	<a href="#">cds_new_uint</a>	84
4.15.2.17	<a href="#">cds_new_ulong</a>	84
4.15.2.18	<a href="#">cds_new_ulonglong</a>	84
4.16	<a href="#">queue.c File Reference</a>	85
4.16.1	<a href="#">Detailed Description</a>	86
4.16.2	<a href="#">Function Documentation</a>	86
4.16.2.1	<a href="#">queue_free</a>	86
4.16.2.2	<a href="#">queue_init</a>	86
4.16.2.3	<a href="#">queue_isempty</a>	86
4.16.2.4	<a href="#">queue_length</a>	86
4.16.2.5	<a href="#">queue_lock</a>	87
4.16.2.6	<a href="#">queue_pop</a>	87
4.16.2.7	<a href="#">queue_pushback</a>	87
4.16.2.8	<a href="#">queue_unlock</a>	87

4.17	sl_list.c File Reference	87
4.17.1	Detailed Description	89
4.17.2	Function Documentation	89
4.17.2.1	sl_list_data	89
4.17.2.2	sl_list_delete_at	89
4.17.2.3	sl_list_find	90
4.17.2.4	sl_list_find_index	90
4.17.2.5	sl_list_find_itr	90
4.17.2.6	sl_list_first	90
4.17.2.7	sl_list_free	91
4.17.2.8	sl_list_free_node	91
4.17.2.9	sl_list_init	91
4.17.2.10	sl_list_insert_after	91
4.17.2.11	sl_list_insert_at	91
4.17.2.12	sl_list_isempty	92
4.17.2.13	sl_list_itr_from_index	92
4.17.2.14	sl_list_length	92
4.17.2.15	sl_list_lock	92
4.17.2.16	sl_list_new_node	93
4.17.2.17	sl_list_next	93
4.17.2.18	sl_list_prepend	93
4.17.2.19	sl_list_remove_at	93
4.17.2.20	sl_list_unlock	93
4.18	sl_list.h File Reference	94
4.18.1	Detailed Description	95
4.18.2	Function Documentation	95
4.18.2.1	sl_list_find	95
4.18.2.2	sl_list_free_node	95
4.18.2.3	sl_list_new_node	96
4.18.2.4	sl_list_remove_at	96
4.19	stack.c File Reference	96
4.19.1	Detailed Description	97
4.19.2	Function Documentation	97
4.19.2.1	stack_free	97
4.19.2.2	stack_init	97
4.19.2.3	stack_isempty	98
4.19.2.4	stack_length	98
4.19.2.5	stack_lock	98
4.19.2.6	stack_pop	98
4.19.2.7	stack_push	98

4.19.2.8 <code>stack_unlock</code> . . . . .	99
--	----

# Chapter 1

## Data Structure Index

### 1.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">bs_tree_node_t</a>	Struct for binary search tree node . . . . .	5
<a href="#">bs_tree_t</a>	Struct to contain a binary search tree . . . . .	6
<a href="#">dl_list_node_t</a>	Struct for double linked list node . . . . .	7
<a href="#">dl_list_t</a>	Struct to contain a list . . . . .	8
<a href="#">kvpair_t</a>	Key-value pair struct . . . . .	9
<a href="#">sl_list_node_t</a>	Struct for singly linked list node . . . . .	10
<a href="#">sl_list_t</a>	Struct to contain a list . . . . .	10



## Chapter 2

# File Index

### 2.1 File List

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

<a href="#">bs_tree.c</a>	Implementation of binary search tree data structure . . . . .	13
<a href="#">bs_tree.h</a>	Developer interface to binary search tree data structure . . . . .	21
<a href="#">bst_map.c</a>	Implementation of binary search tree map data structure . . . . .	26
<a href="#">cdatastruct.h</a>	Interface to generic C data structures . . . . .	29
<a href="#">cds_bs_tree.h</a>	User interface to binary search tree data structure . . . . .	30
<a href="#">cds_bst_map.h</a>	User interface to binary search tree map data structure . . . . .	35
<a href="#">cds_common.h</a>	Common data types and data for C data structures library . . . . .	39
<a href="#">cds_dl_list.h</a>	User interface to doubly linked list data structure . . . . .	40
<a href="#">cds_general.h</a>	Interface to general data structure helper functions . . . . .	47
<a href="#">cds_queue.h</a>	User interface to queue data structure . . . . .	53
<a href="#">cds_sl_list.h</a>	User interface to singly linked list data structure . . . . .	56
<a href="#">cds_stack.h</a>	User interface to stack data structure . . . . .	62
<a href="#">dl_list.c</a>	Implementation of doubly linked list data structure . . . . .	65
<a href="#">dl_list.h</a>	Developer interface to double linked list data structure . . . . .	74
<a href="#">general.c</a>	Implementation of general data structure helper functions . . . . .	79
<a href="#">queue.c</a>	Implementation of queue data structure . . . . .	85
<a href="#">sl_list.c</a>	Implementation of singly linked list data structure . . . . .	87
<a href="#">sl_list.h</a>	Developer interface to singly linked list data structure . . . . .	94
<a href="#">stack.c</a>	Implementation of stack data structure . . . . .	96





## Chapter 3

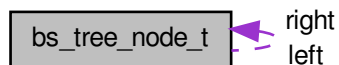
# Data Structure Documentation

### 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:

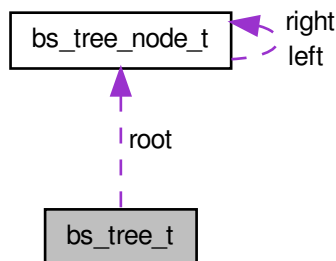
- [cds\\_bs\\_tree.h](#)

## 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:

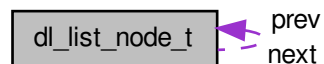
- [bs\\_tree.h](#)

### 3.3 dl\_list\_node\_t Struct Reference

Struct for double linked list node.

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

Collaboration diagram for dl\_list\_node\_t:



#### Data Fields

- void \* [data](#)
- struct [dl\\_list\\_node\\_t](#) \* [next](#)
- struct [dl\\_list\\_node\\_t](#) \* [prev](#)

#### 3.3.1 Detailed Description

Struct for double linked list node.

#### 3.3.2 Field Documentation

## 3.3.2.1 void\* dl\_list\_node\_t::data

Pointer to data

## 3.3.2.2 struct dl\_list\_node\_t\* dl\_list\_node\_t::next

Pointer to next node

### 3.3.2.3 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:

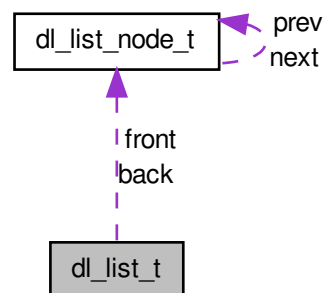
- [cds\\_dl\\_list.h](#)

## 3.4 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](#) )()

### 3.4.1 Detailed Description

Struct to contain a list.

### 3.4.2 Field Documentation

#### 3.4.2.1 struct dl\_list\_node\_t\* dl\_list\_t::back

Pointer to last node

#### 3.4.2.2 int(\* dl\_list\_t::cfunc)()

Pointer to compare function

#### 3.4.2.3 void(\* dl\_list\_t::free\_func)()

Pointer to free function

#### 3.4.2.4 struct dl\_list\_node\_t\* dl\_list\_t::front

Pointer to first node

#### 3.4.2.5 size\_t dl\_list\_t::length

Length of list

#### 3.4.2.6 pthread\_mutex\_t dl\_list\_t::mutex

Mutex

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

- [dl\\_list.h](#)

## 3.5 kvpair\_t Struct Reference

Key-value pair struct.

### Data Fields

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

### 3.5.1 Detailed Description

Key-value pair struct.

### 3.5.2 Field Documentation

#### 3.5.2.1 char\* kvpair\_t::key

Key string

#### 3.5.2.2 void\* kvpair\_t::value

Pointer to data

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

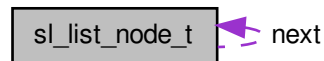
- [bst\\_map.c](#)

### 3.6 `sl_list_node_t` Struct Reference

Struct for singly linked list node.

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

Collaboration diagram for `sl_list_node_t`:



#### Data Fields

- void \* [data](#)
- struct [sl\\_list\\_node\\_t](#) \* [next](#)

#### 3.6.1 Detailed Description

Struct for singly linked list node.

#### 3.6.2 Field Documentation

##### 3.6.2.1 void\* `sl_list_node_t::data`

Pointer to data

##### 3.6.2.2 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:

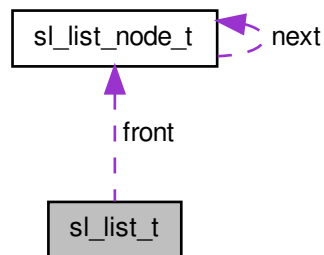
- [cds\\_sl\\_list.h](#)

### 3.7 `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 )()`

#### 3.7.1 Detailed Description

Struct to contain a list.

#### 3.7.2 Field Documentation

##### 3.7.2.1 `int(* sl_list_t::cfunc)()`

Pointer to compare function

##### 3.7.2.2 `void(* sl_list_t::free_func)()`

Pointer to free function

##### 3.7.2.3 `struct sl_list_node_t* sl_list_t::front`

Pointer to first node

##### 3.7.2.4 `size_t sl_list_t::length`

Length of list

##### 3.7.2.5 `pthread_mutex_t sl_list_t::mutex`

Mutex

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

- [sl\\_list.h](#)



## Chapter 4

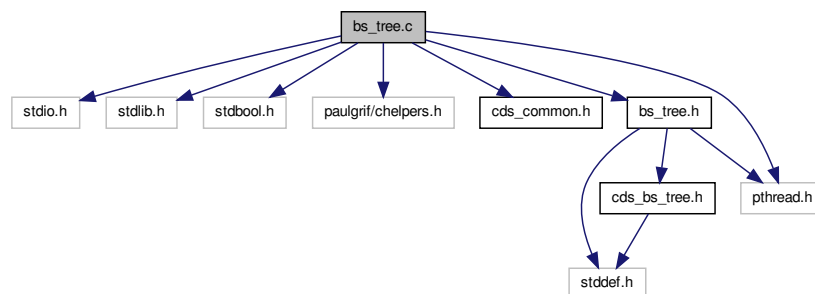
# File Documentation

### 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> d.

##### 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

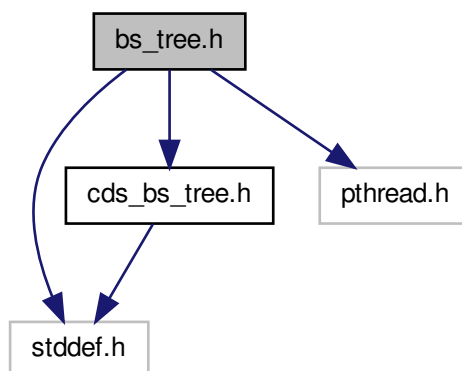
Developer interface to binary search tree data structure.

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

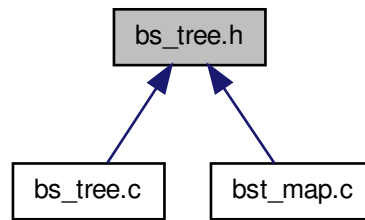
```
#include "cds_bs_tree.h"
```

```
#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

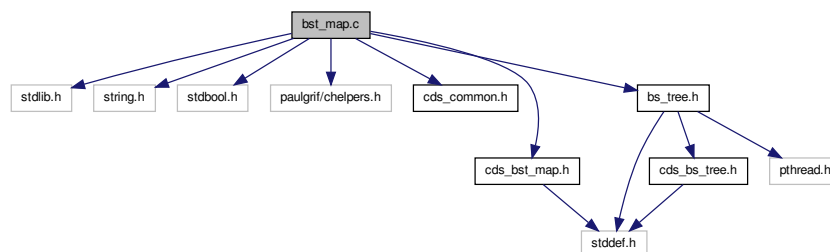
A pointer to the node in which the data was found, or `NULL` if the data was not found.

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

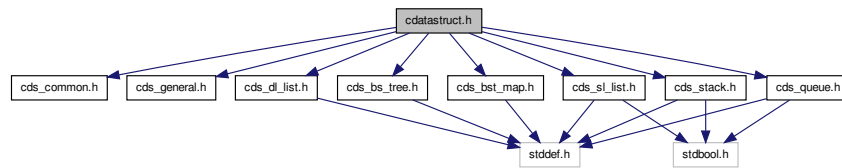
<i>map</i>	A pointer to the map.
------------	-----------------------

## 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 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

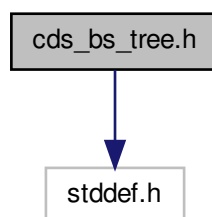
Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.5 `cds_bs_tree.h` File Reference

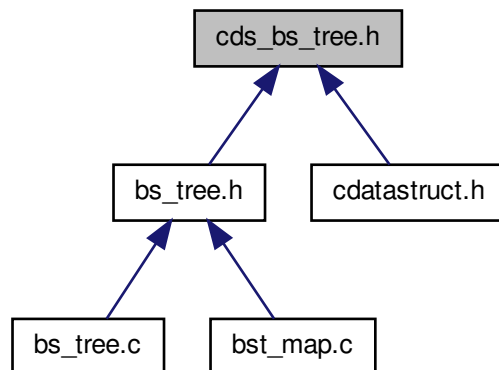
User interface to binary search tree data structure.

```
#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 NULL, 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

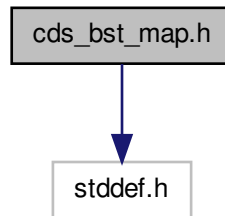
<i>tree</i>	A pointer to the tree.
-------------	------------------------

## 4.6 cds\_bst\_map.h File Reference

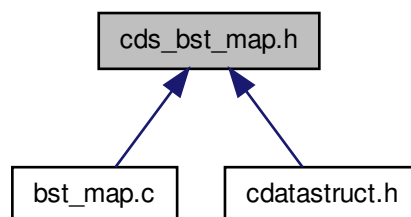
User interface to binary search tree map data structure.

```
#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.

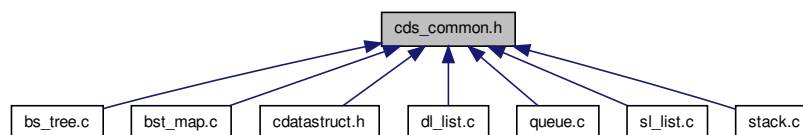
## Parameters

<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

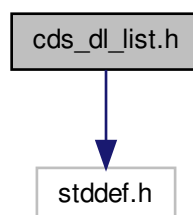
***CDSERR\_BADITERATOR*** Invalid iterator

## 4.8 cds\_dl\_list.h File Reference

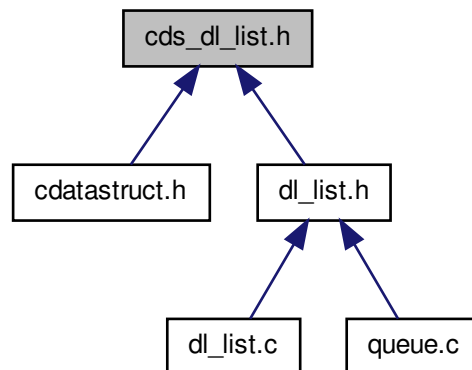
User interface to doubly linked list data structure.

```
#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.*

### 4.8.1 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/>

### 4.8.2 Function Documentation

#### 4.8.2.1 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.

#### 4.8.2.2 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.

#### 4.8.2.3 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.

#### 4.8.2.4 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.

#### 4.8.2.5 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.

#### 4.8.2.6 `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.

#### 4.8.2.7 `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.
-------------	--------------------------------

#### 4.8.2.8 `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.

#### 4.8.2.9 `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.

4.8.2.10 `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.

4.8.2.11 `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.

4.8.2.12 `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`.

4.8.2.13 `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.

**4.8.2.14 `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.

**4.8.2.15 `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.
-------------	------------------------

**4.8.2.16 `void dl_list_lock ( dl_list list )`**

Locks a list's mutex.

**Parameters**

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

**4.8.2.17 `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.

**4.8.2.18 `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.

4.8.2.19 `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.

4.8.2.20 `void dl_list_unlock ( dl_list list )`

Unlocks a list's mutex.

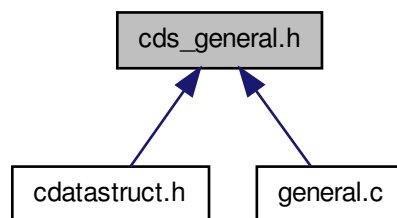
## Parameters

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

## 4.9 cds\_general.h File Reference

Interface to general data structure helper functions.

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



### Functions

- void \* `cds_new_int` (const int n)  
*Dynamically allocates memory for a new int.*
- void \* `cds_new_uint` (const unsigned int n)  
*Dynamically allocates memory for a new unsigned int.*
- void \* `cds_new_long` (const long n)  
*Dynamically allocates memory for a new long.*
- void \* `cds_new_ulong` (const unsigned long n)  
*Dynamically allocates memory for a new unsigned long.*
- void \* `cds_new_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.*

#### 4.9.1 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/>

#### 4.9.2 Function Documentation

##### 4.9.2.1 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.

**4.9.2.2 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.

**4.9.2.3 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.

**4.9.2.4 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.

**4.9.2.5 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.

**4.9.2.6 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.

**4.9.2.7 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.

**4.9.2.8 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.

**4.9.2.9 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.

**4.9.2.10 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.

**4.9.2.11 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.

**4.9.2.12 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.

**4.9.2.13 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.

**4.9.2.14 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.

**4.9.2.15 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.

**4.9.2.16 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.

**4.9.2.17 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.

**4.9.2.18 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.

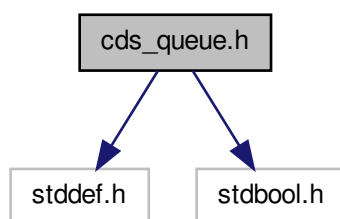
## 4.10 cds\_queue.h File Reference

User interface to queue data structure.

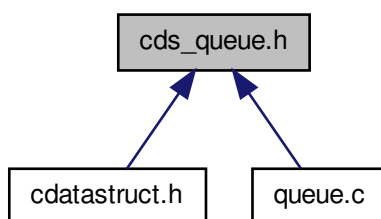
```
#include <stddef.h>
```

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

Include dependency graph for cds\_queue.h:



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



### Typedefs

- typedef struct dl\_list\_t \* [queue](#)  
*Typedef for queue pointer.*

### Functions

- [queue](#) [queue\\_init](#) (void(\*free\_func)(void \*))

- Initializes a new queue.*
- void `queue_free` (`queue` `que`)  
*Frees memory and releases resources used by a queue.*
- size\_t `queue_length` (const `queue` `que`)  
*Gets the number of items in a queue.*
- bool `queue_isempty` (const `queue` `que`)  
*Checks if a queue is empty.*
- void \* `queue_pop` (`queue` `que`)  
*Pops a data item from the queue.*
- void `queue_pushback` (`queue` `que`, void \*`data`)  
*Pushes a data item onto the back of the queue.*
- void `queue_lock` (`queue` `que`)  
*Locks a queue's mutex.*
- void `queue_unlock` (`queue` `que`)  
*Unlocks a queue's mutex.*

#### 4.10.1 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/>

#### 4.10.2 Function Documentation

##### 4.10.2.1 void `queue_free` ( `queue` `que` )

Frees memory and releases resources used by a queue.

##### Parameters

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

##### 4.10.2.2 `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 void pointer to a node. If NULL is specified, the standard <code>free()</code> function is used.
------------------------	--

##### Returns

A pointer to the new queue.

#### 4.10.2.3 `bool queue_isempty ( const queue que )`

Checks if a queue is empty.

##### Parameters

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

##### Returns

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

#### 4.10.2.4 `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.

#### 4.10.2.5 `void queue_lock ( queue que )`

Locks a queue's mutex.

##### Parameters

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

#### 4.10.2.6 `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.

#### 4.10.2.7 `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.

4.10.2.8 void queue\_unlock ( queue *que* )

Unlocks a queue's mutex.

## Parameters

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

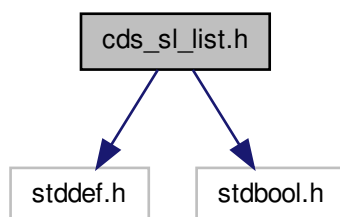
## 4.11 cds\_sl\_list.h File Reference

User interface to singly linked list data structure.

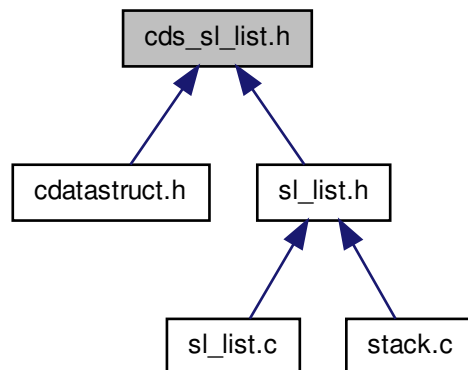
```
#include <stddef.h>
```

```
#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.*

#### 4.11.1 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/>

#### 4.11.2 Function Documentation

##### 4.11.2.1 `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.

##### 4.11.2.2 `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.

4.11.2.3 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.

4.11.2.4 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 is the element is not in the list.

4.11.2.5 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.

4.11.2.6 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.
-------------	--------------------------------

#### 4.11.2.7 `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.

#### 4.11.2.8 `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.

#### 4.11.2.9 `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.

#### 4.11.2.10 `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`.

**4.11.2.11 `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.

**4.11.2.12 `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.
-------------	------------------------

**4.11.2.13 `void sl_list_lock ( sl_list list )`**

Locks a list's mutex.

## Parameters

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

**4.11.2.14 `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.

**4.11.2.15 `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.

4.11.2.16 `void sl_list.unlock ( sl_list list )`

Unlocks a list's mutex.

## Parameters

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

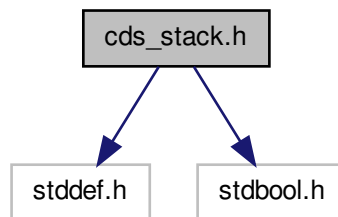
4.12 `cds_stack.h` File Reference

User interface to stack data structure.

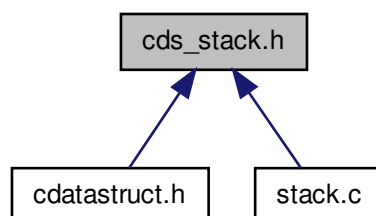
```
#include <stddef.h>
```

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

Include dependency graph for `cds_stack.h`:



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



## Typedefs

- typedef struct [sl\\_list\\_t](#) \* [stack](#)  
*Typedef for stack pointer.*

## Functions

- [stack](#) [stack\\_init](#) (void(\*free\_func)(void \*))  
*Initializes a new stack.*
- void [stack\\_free](#) ([stack](#) stk)  
*Frees memory and releases resources used by a stack.*
- size\_t [stack\\_length](#) (const [stack](#) stk)  
*Gets the number of items in a stack.*
- bool [stack\\_isempty](#) (const [stack](#) stk)  
*Checks if a stack is empty.*
- void \* [stack\\_pop](#) ([stack](#) stk)  
*Pops a data item from the stack.*
- void [stack\\_push](#) ([stack](#) stk, void \*data)  
*Pushes a data item onto the stack.*
- void [stack\\_lock](#) ([stack](#) stk)  
*Locks a stack's mutex.*
- void [stack\\_unlock](#) ([stack](#) stk)  
*Unlocks a stack's mutex.*

### 4.12.1 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/>

### 4.12.2 Function Documentation

#### 4.12.2.1 void stack\_free ( [stack](#) stk )

Frees memory and releases resources used by a stack.

#### Parameters

<a href="#">stk</a>	A pointer to the stack.
---------------------	-------------------------

#### 4.12.2.2 [stack](#) stack\_init ( void(\*) (void \*) [free\\_func](#) )

Initializes a new stack.

## Parameters

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

## Returns

A pointer to the new stack.

4.12.2.3 `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.

4.12.2.4 `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.

4.12.2.5 `void stack_lock ( stack stk )`

Locks a stack's mutex.

## Parameters

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

4.12.2.6 `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.

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

Pushes a data item onto the stack.

The provided pointer should point to dynamically allocated memory.

## Parameters

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

4.12.2.8 `void stack_unlock ( stack stk )`

Unlocks a stack's mutex.

## Parameters

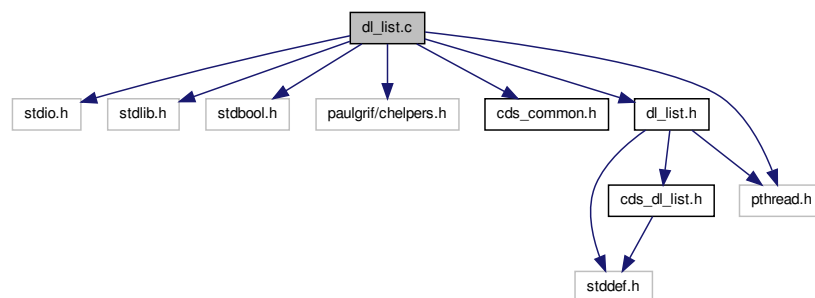
<code>stk</code>	A pointer to the stack.
------------------	-------------------------

## 4.13 dl\_list.c File Reference

Implementation of doubly linked list data structure.

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

### 4.13.1 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/>

### 4.13.2 Function Documentation

#### 4.13.2.1 `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.

#### 4.13.2.2 `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.

#### 4.13.2.3 `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.

#### 4.13.2.4 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.

#### 4.13.2.5 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.

#### 4.13.2.6 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.

#### 4.13.2.7 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.

## 4.13.2.8 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.
-------------	--------------------------------

## 4.13.2.9 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.

## 4.13.2.10 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.

## 4.13.2.11 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.

4.13.2.12 `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.

4.13.2.13 `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.

4.13.2.14 `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.

## 4.13.2.15 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.

## 4.13.2.16 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.

## 4.13.2.17 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.

## 4.13.2.18 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`.

## 4.13.2.19 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.

**4.13.2.20 `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.

**4.13.2.21 `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.
-------------	------------------------

**4.13.2.22 `void dl_list.lock ( dl_list list )`**

Locks a list's mutex.

**Parameters**

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

**4.13.2.23 `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.

**4.13.2.24 `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.

## 4.13.2.25 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.

## 4.13.2.26 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.

## 4.13.2.27 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\(\)](#).

## 4.13.2.28 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.

**4.13.2.29 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.

**4.13.2.30 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`.

**4.13.2.31 void dl\_list\_unlock ( dl\_list list )**

Unlocks a list's mutex.

**Parameters**

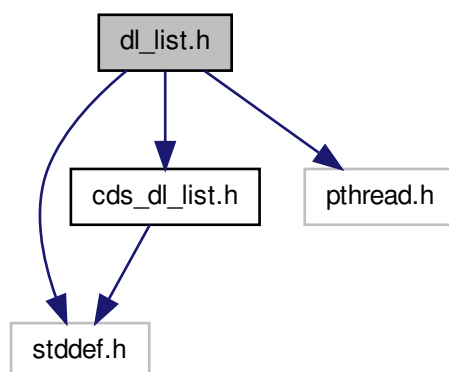
<i>list</i>	A pointer to the list.
-------------	------------------------

**4.14 dl\_list.h File Reference**

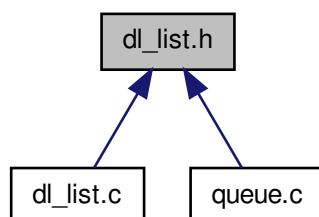
Developer interface to double linked list data structure.

```
#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.*

### 4.14.1 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/>

### 4.14.2 Function Documentation

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

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.



#### 4.14.2.2 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.

#### 4.14.2.3 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.

#### 4.14.2.4 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.

#### 4.14.2.5 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.

#### 4.14.2.6 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.

#### 4.14.2.7 `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.

#### 4.14.2.8 `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\(\)](#).

#### 4.14.2.9 `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.

#### 4.14.2.10 `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.

#### 4.14.2.11 `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.

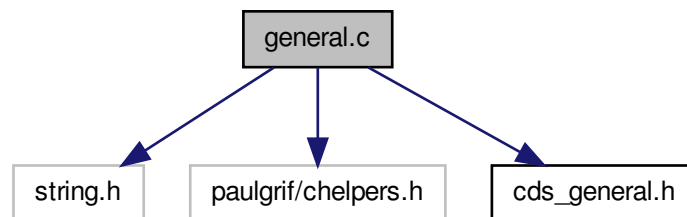
## Returns

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

## 4.15 general.c File Reference

Implementation of general data structure helper functions.

```
#include <string.h>
#include <paulgrif/chelpers.h>
#include "cds_general.h"
Include dependency graph for general.c:
```



## Functions

- void \* [cds\\_new\\_int](#) (const int n)  
*Dynamically allocates memory for a new int.*
- void \* [cds\\_new\\_uint](#) (const unsigned int n)  
*Dynamically allocates memory for a new unsigned int.*
- void \* [cds\\_new\\_long](#) (const long n)  
*Dynamically allocates memory for a new long.*
- void \* [cds\\_new\\_ulong](#) (const unsigned long n)  
*Dynamically allocates memory for a new unsigned long.*
- void \* [cds\\_new\\_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.*

#### 4.15.1 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/>

#### 4.15.2 Function Documentation

##### 4.15.2.1 `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.

##### 4.15.2.2 `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.

4.15.2.3 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.

4.15.2.4 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.

4.15.2.5 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.

#### 4.15.2.6 `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.

#### 4.15.2.7 `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.

#### 4.15.2.8 `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.

#### 4.15.2.9 `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.

**4.15.2.10 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.

**4.15.2.11 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.

**4.15.2.12 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.

**4.15.2.13 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.

**4.15.2.14 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.

**4.15.2.15 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.

**4.15.2.16 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.

**4.15.2.17 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.

**4.15.2.18 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

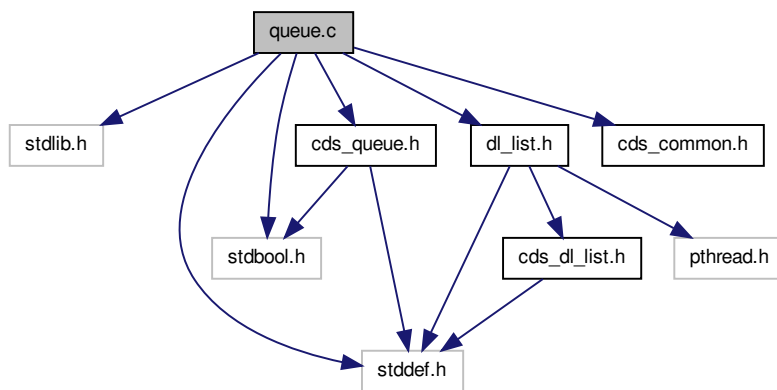
A `void` pointer to the allocated memory.

## 4.16 queue.c File Reference

Implementation of queue data structure.

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

Include dependency graph for queue.c:



## Functions

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

### 4.16.1 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/>

### 4.16.2 Function Documentation

#### 4.16.2.1 void queue\_free ( queue *que* )

Frees memory and releases resources used by a queue.

##### Parameters

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

#### 4.16.2.2 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.

#### 4.16.2.3 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.

#### 4.16.2.4 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.

4.16.2.5 `void queue_lock ( queue que )`

Locks a queue's mutex.

## Parameters

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

4.16.2.6 `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.

4.16.2.7 `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.

4.16.2.8 `void queue_unlock ( queue que )`

Unlocks a queue's mutex.

## Parameters

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

## 4.17 `sl_list.c` File Reference

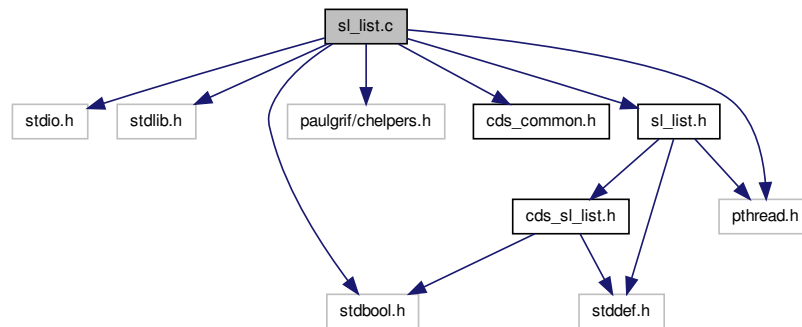
Implementation of singly linked list data structure.

```

#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.*

### 4.17.1 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/>

### 4.17.2 Function Documentation

#### 4.17.2.1 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.

#### 4.17.2.2 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.

**4.17.2.3 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.

**4.17.2.4 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.

**4.17.2.5 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 is the element is not in the list.

**4.17.2.6 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.

4.17.2.7 `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.
-------------	--------------------------------

4.17.2.8 `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.

4.17.2.9 `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.

4.17.2.10 `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.

4.17.2.11 `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.

4.17.2.12 `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`.

4.17.2.13 `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.

4.17.2.14 `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.
-------------	------------------------

4.17.2.15 `void sl_list_lock ( sl_list list )`

Locks a list's mutex.

## Parameters

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



**4.17.2.16** `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.

**4.17.2.17** `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.

**4.17.2.18** `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.

**4.17.2.19** `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\(\)](#).

**4.17.2.20** `void sl_list_unlock ( sl_list list )`

Unlocks a list's mutex.

**Parameters**

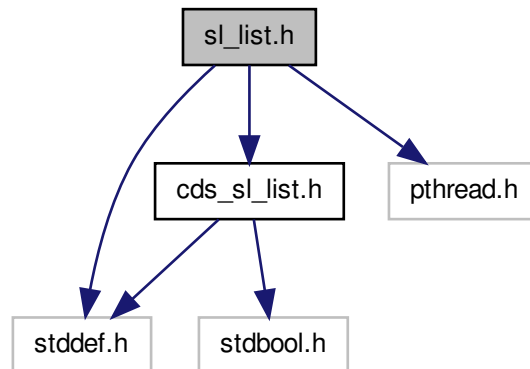
<i>list</i>	A pointer to the list.
-------------	------------------------

## 4.18 sl\_list.h File Reference

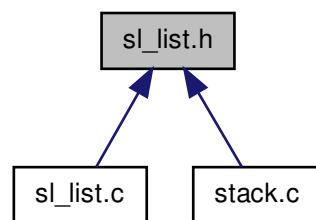
Developer interface to singly linked list data structure.

```
#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.*

### 4.18.1 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/>

### 4.18.2 Function Documentation

#### 4.18.2.1 void [sl\\_list\\_find](#) ( const [sl\\_list](#) list, const void \* data, [sl\\_list\\_itr](#) \* p\_itr, int \* p\_index )

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.

#### 4.18.2.2 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.

#### 4.18.2.3 `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.

#### 4.18.2.4 `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

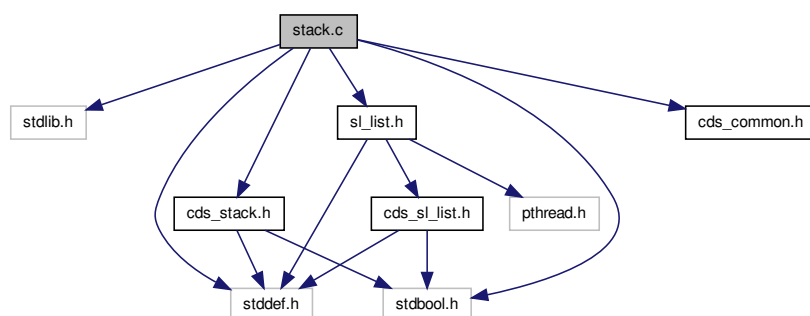
A pointer to the removed node. This should be `free()` d by calling [sl\\_list\\_free\\_node\(\)](#).

## 4.19 `stack.c` File Reference

Implementation of stack data structure.

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

Include dependency graph for `stack.c`:



## Functions

- [stack stack\\_init](#) (void(\*free\_func)(void \*))

- Initializes a new stack.*
- void `stack_free` (`stack` `stk`)  
*Frees memory and releases resources used by a stack.*
- size\_t `stack_length` (const `stack` `stk`)  
*Gets the number of items in a stack.*
- bool `stack_isempty` (const `stack` `stk`)  
*Checks if a stack is empty.*
- void \* `stack_pop` (`stack` `stk`)  
*Pops a data item from the stack.*
- void `stack_push` (`stack` `stk`, void \*`data`)  
*Pushes a data item onto the stack.*
- void `stack_lock` (`stack` `stk`)  
*Locks a stack's mutex.*
- void `stack_unlock` (`stack` `stk`)  
*Unlocks a stack's mutex.*

### 4.19.1 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/>

### 4.19.2 Function Documentation

#### 4.19.2.1 void `stack_free` ( `stack` `stk` )

Frees memory and releases resources used by a stack.

#### Parameters

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

#### 4.19.2.2 `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.

#### 4.19.2.3 `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.

#### 4.19.2.4 `size_t stack_length ( const stack stk )`

Gets the number of items in a stack.

##### Parameters

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

##### Returns

The number of items in the stack.

#### 4.19.2.5 `void stack_lock ( stack stk )`

Locks a stack's mutex.

##### Parameters

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

#### 4.19.2.6 `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

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

##### Returns

A `void` pointer to the popped data item.

#### 4.19.2.7 `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.

4.19.2.8 void stack\_unlock ( stack *stk* )

Unlocks a stack's mutex.

## Parameters

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

# Index

- back
  - dl\_list\_t, 8
- bs\_tree.c, 13
  - bs\_tree\_free, 15
  - bs\_tree\_free\_subtree, 15
  - bs\_tree\_init, 15
  - bs\_tree\_inorder\_left\_traverse, 15
  - bs\_tree\_inorder\_left\_traverse\_int, 16
  - bs\_tree\_inorder\_right\_traverse, 16
  - bs\_tree\_inorder\_right\_traverse\_int, 16
  - bs\_tree\_insert, 16
  - bs\_tree\_insert\_search, 17
  - bs\_tree\_insert\_subtree, 17
  - bs\_tree\_isempty, 17
  - bs\_tree\_length, 18
  - bs\_tree\_lock, 18
  - bs\_tree\_new\_node, 18
  - bs\_tree\_postorder\_left\_traverse, 18
  - bs\_tree\_postorder\_left\_traverse\_int, 18
  - bs\_tree\_postorder\_right\_traverse, 19
  - bs\_tree\_postorder\_right\_traverse\_int, 19
  - bs\_tree\_preorder\_left\_traverse, 19
  - bs\_tree\_preorder\_left\_traverse\_int, 19
  - bs\_tree\_preorder\_right\_traverse, 19
  - bs\_tree\_preorder\_right\_traverse\_int, 20
  - bs\_tree\_search, 20
  - bs\_tree\_search\_data, 20
  - bs\_tree\_search\_node, 20
  - bs\_tree\_unlock, 21
- bs\_tree.h, 21
  - bs\_tree\_free\_subtree, 23
  - bs\_tree\_inorder\_left\_traverse\_int, 23
  - bs\_tree\_inorder\_right\_traverse\_int, 23
  - bs\_tree\_insert\_search, 24
  - bs\_tree\_insert\_subtree, 24
  - bs\_tree\_new\_node, 24
  - bs\_tree\_postorder\_left\_traverse\_int, 25
  - bs\_tree\_postorder\_right\_traverse\_int, 25
  - bs\_tree\_preorder\_left\_traverse\_int, 25
  - bs\_tree\_preorder\_right\_traverse\_int, 25
  - bs\_tree\_search\_node, 26
- bs\_tree\_free
  - bs\_tree.c, 15
  - cds\_bs\_tree.h, 32
- bs\_tree\_free\_subtree
  - bs\_tree.c, 15
  - bs\_tree.h, 23
- bs\_tree\_init
  - bs\_tree.c, 15
- bs\_tree\_inorder\_left\_traverse
  - cds\_bs\_tree.h, 32
- bs\_tree\_inorder\_left\_traverse\_int
  - bs\_tree.c, 16
  - bs\_tree.h, 23
- bs\_tree\_inorder\_right\_traverse
  - bs\_tree.c, 16
  - cds\_bs\_tree.h, 33
- bs\_tree\_inorder\_right\_traverse\_int
  - bs\_tree.c, 16
  - bs\_tree.h, 23
- bs\_tree\_insert
  - bs\_tree.c, 16
  - cds\_bs\_tree.h, 33
- bs\_tree\_insert\_search
  - bs\_tree.c, 17
  - bs\_tree.h, 24
- bs\_tree\_insert\_subtree
  - bs\_tree.c, 17
  - bs\_tree.h, 24
- bs\_tree\_isempty
  - bs\_tree.c, 17
  - cds\_bs\_tree.h, 33
- bs\_tree\_length
  - bs\_tree.c, 18
  - cds\_bs\_tree.h, 33
- bs\_tree\_lock
  - bs\_tree.c, 18
  - cds\_bs\_tree.h, 34
- bs\_tree\_new\_node
  - bs\_tree.c, 18
  - bs\_tree.h, 24
- bs\_tree\_node\_t, 5
  - data, 5
  - left, 5
  - right, 5
- bs\_tree\_postorder\_left\_traverse
  - bs\_tree.c, 18
  - cds\_bs\_tree.h, 34
- bs\_tree\_postorder\_left\_traverse\_int
  - bs\_tree.c, 18
  - bs\_tree.h, 25
- bs\_tree\_postorder\_right\_traverse
  - bs\_tree.c, 19
  - cds\_bs\_tree.h, 34
- bs\_tree\_postorder\_right\_traverse\_int
  - bs\_tree.c, 19



- bs\_tree.h, 25
- bs\_tree\_preorder\_left\_traverse
  - bs\_tree.c, 19
  - cds\_bs\_tree.h, 34
- bs\_tree\_preorder\_left\_traverse\_int
  - bs\_tree.c, 19
  - bs\_tree.h, 25
- bs\_tree\_preorder\_right\_traverse
  - bs\_tree.c, 19
  - cds\_bs\_tree.h, 34
- bs\_tree\_preorder\_right\_traverse\_int
  - bs\_tree.c, 20
  - bs\_tree.h, 25
- bs\_tree\_search
  - bs\_tree.c, 20
  - cds\_bs\_tree.h, 35
- bs\_tree\_search\_data
  - bs\_tree.c, 20
  - cds\_bs\_tree.h, 35
- bs\_tree\_search\_node
  - bs\_tree.c, 20
  - bs\_tree.h, 26
- bs\_tree\_t, 6
  - cfunc, 6
  - free\_func, 6
  - length, 6
  - mutex, 7
  - root, 7
- bs\_tree\_unlock
  - bs\_tree.c, 21
  - cds\_bs\_tree.h, 35
- bst\_map.c, 26
  - bst\_map\_free, 27
  - bst\_map\_init, 27
  - bst\_map\_insert, 28
  - bst\_map\_isempty, 28
  - bst\_map\_length, 28
  - bst\_map\_lock, 28
  - bst\_map\_search, 28
  - bst\_map\_search\_data, 29
  - bst\_map\_unlock, 29
- bst\_map\_free
  - bst\_map.c, 27
  - cds\_bst\_map.h, 37
- bst\_map\_init
  - bst\_map.c, 27
  - cds\_bst\_map.h, 37
- bst\_map\_insert
  - bst\_map.c, 28
  - cds\_bst\_map.h, 37
- bst\_map\_isempty
  - bst\_map.c, 28
  - cds\_bst\_map.h, 38
- bst\_map\_length
  - bst\_map.c, 28
  - cds\_bst\_map.h, 38
- bst\_map\_lock
  - bst\_map.c, 28
- cds\_bst\_map.h, 38
- bst\_map\_search
  - bst\_map.c, 28
  - cds\_bst\_map.h, 38
- bst\_map\_search\_data
  - bst\_map.c, 29
  - cds\_bst\_map.h, 38
- bst\_map\_unlock
  - bst\_map.c, 29
  - cds\_bst\_map.h, 39
- CDSERR\_BADITERATOR
  - cds\_common.h, 40
- CDSERR\_ERROR
  - cds\_common.h, 40
- CDSERR\_NOTFOUND
  - cds\_common.h, 40
- CDSERR\_OUTOFRANGE
  - cds\_common.h, 40
- cdatastruct.h, 29
- cds\_common.h
  - CDSERR\_BADITERATOR, 40
  - CDSERR\_ERROR, 40
  - CDSERR\_NOTFOUND, 40
  - CDSERR\_OUTOFRANGE, 40
- cds\_bs\_tree.h, 30
  - bs\_tree\_free, 32
  - bs\_tree\_init, 32
  - bs\_tree\_inorder\_left\_traverse, 33
  - bs\_tree\_inorder\_right\_traverse, 33
  - bs\_tree\_insert, 33
  - bs\_tree\_isempty, 33
  - bs\_tree\_length, 33
  - bs\_tree\_lock, 34
  - bs\_tree\_postorder\_left\_traverse, 34
  - bs\_tree\_postorder\_right\_traverse, 34
  - bs\_tree\_preorder\_left\_traverse, 34
  - bs\_tree\_preorder\_right\_traverse, 34
  - bs\_tree\_search, 35
  - bs\_tree\_search\_data, 35
  - bs\_tree\_unlock, 35
- cds\_bst\_map.h, 35
  - bst\_map\_free, 37
  - bst\_map\_init, 37
  - bst\_map\_insert, 37
  - bst\_map\_isempty, 38
  - bst\_map\_length, 38
  - bst\_map\_lock, 38
  - bst\_map\_search, 38
  - bst\_map\_search\_data, 38
  - bst\_map\_unlock, 39
- cds\_common.h, 39
  - cds\_error, 40
- cds\_compare\_double
  - cds\_general.h, 48
  - general.c, 80
- cds\_compare\_float
  - cds\_general.h, 49
  - general.c, 80

- cds\_compare\_int
  - cds\_general.h, 49
  - general.c, 81
- cds\_compare\_long
  - cds\_general.h, 49
  - general.c, 81
- cds\_compare\_longlong
  - cds\_general.h, 49
  - general.c, 81
- cds\_compare\_string
  - cds\_general.h, 50
  - general.c, 81
- cds\_compare\_uint
  - cds\_general.h, 50
  - general.c, 82
- cds\_compare\_ulong
  - cds\_general.h, 50
  - general.c, 82
- cds\_compare\_ulonglong
  - cds\_general.h, 50
  - general.c, 82
- cds\_dl\_list.h, 40
  - dl\_list\_append, 42
  - dl\_list\_data, 42
  - dl\_list\_delete\_at, 43
  - dl\_list\_find\_index, 43
  - dl\_list\_find\_itr, 43
  - dl\_list\_first, 43
  - dl\_list\_free, 44
  - dl\_list\_init, 44
  - dl\_list\_insert\_after, 44
  - dl\_list\_insert\_at, 45
  - dl\_list\_insert\_before, 45
  - dl\_list\_isempty, 45
  - dl\_list\_itr\_from\_index, 45
  - dl\_list\_last, 46
  - dl\_list\_length, 46
  - dl\_list\_lock, 46
  - dl\_list\_next, 46
  - dl\_list\_prepend, 46
  - dl\_list\_prev, 47
  - dl\_list\_unlock, 47
- cds\_error
  - cds\_common.h, 40
- cds\_general.h, 47
  - cds\_compare\_double, 48
  - cds\_compare\_float, 49
  - cds\_compare\_int, 49
  - cds\_compare\_long, 49
  - cds\_compare\_longlong, 49
  - cds\_compare\_string, 50
  - cds\_compare\_uint, 50
  - cds\_compare\_ulong, 50
  - cds\_compare\_ulonglong, 50
  - cds\_new\_double, 51
  - cds\_new\_float, 51
  - cds\_new\_int, 51
  - cds\_new\_long, 51
  - cds\_new\_longlong, 51
  - cds\_new\_string, 52
  - cds\_new\_uint, 52
  - cds\_new\_ulong, 52
  - cds\_new\_ulonglong, 52
- cds\_new\_double
  - cds\_general.h, 51
  - general.c, 83
- cds\_new\_float
  - cds\_general.h, 51
  - general.c, 83
- cds\_new\_int
  - cds\_general.h, 51
  - general.c, 83
- cds\_new\_long
  - cds\_general.h, 51
  - general.c, 83
- cds\_new\_longlong
  - cds\_general.h, 51
  - general.c, 83
- cds\_new\_string
  - cds\_general.h, 52
  - general.c, 84
- cds\_new\_uint
  - cds\_general.h, 52
  - general.c, 84
- cds\_new\_ulong
  - cds\_general.h, 52
  - general.c, 84
- cds\_new\_ulonglong
  - cds\_general.h, 52
  - general.c, 84
- cds\_queue.h, 53
  - queue\_free, 54
  - queue\_init, 54
  - queue\_isempty, 54
  - queue\_length, 55
  - queue\_lock, 55
  - queue\_pop, 55
  - queue\_pushback, 55
  - queue\_unlock, 56
- cds\_sl\_list.h, 56
  - sl\_list\_data, 58
  - sl\_list\_delete\_at, 58
  - sl\_list\_find\_index, 59
  - sl\_list\_find\_itr, 59
  - sl\_list\_first, 59
  - sl\_list\_free, 59
  - sl\_list\_init, 60
  - sl\_list\_insert\_after, 60
  - sl\_list\_insert\_at, 60
  - sl\_list\_isempty, 60
  - sl\_list\_itr\_from\_index, 61
  - sl\_list\_length, 61
  - sl\_list\_lock, 61
  - sl\_list\_next, 61
  - sl\_list\_prepend, 61
  - sl\_list\_unlock, 62

- cds\_stack.h, 62
  - stack\_free, 63
  - stack\_init, 63
  - stack\_isempty, 64
  - stack\_length, 64
  - stack\_lock, 64
  - stack\_pop, 64
  - stack\_push, 65
  - stack\_unlock, 65
- cfunc
  - bs\_tree\_t, 6
  - dl\_list\_t, 8
  - sl\_list\_t, 11
- data
  - bs\_tree\_node\_t, 5
  - dl\_list\_node\_t, 7
  - sl\_list\_node\_t, 10
- dl\_list.c, 65
  - dl\_list\_append, 67
  - dl\_list\_data, 67
  - dl\_list\_delete\_at, 67
  - dl\_list\_find, 68
  - dl\_list\_find\_index, 68
  - dl\_list\_find\_itr, 68
  - dl\_list\_first, 68
  - dl\_list\_free, 69
  - dl\_list\_free\_node, 69
  - dl\_list\_init, 69
  - dl\_list\_insert\_after, 69
  - dl\_list\_insert\_at, 70
  - dl\_list\_insert\_before, 70
  - dl\_list\_insert\_node\_after\_mid, 70
  - dl\_list\_insert\_node\_back, 70
  - dl\_list\_insert\_node\_before\_mid, 71
  - dl\_list\_insert\_node\_front, 71
  - dl\_list\_isempty, 71
  - dl\_list\_itr\_from\_index, 71
  - dl\_list\_last, 72
  - dl\_list\_length, 72
  - dl\_list\_lock, 72
  - dl\_list\_new\_node, 72
  - dl\_list\_next, 72
  - dl\_list\_prepend, 73
  - dl\_list\_prev, 73
  - dl\_list\_remove\_at, 73
  - dl\_list\_remove\_node\_back, 73
  - dl\_list\_remove\_node\_front, 73
  - dl\_list\_remove\_node\_mid, 74
  - dl\_list\_unlock, 74
- dl\_list.h, 74
  - dl\_list\_find, 76
  - dl\_list\_free\_node, 76
  - dl\_list\_insert\_node\_after\_mid, 77
  - dl\_list\_insert\_node\_back, 77
  - dl\_list\_insert\_node\_before\_mid, 77
  - dl\_list\_insert\_node\_front, 77
  - dl\_list\_new\_node, 77
  - dl\_list\_remove\_at, 78
  - dl\_list\_remove\_node\_back, 78
  - dl\_list\_remove\_node\_front, 78
  - dl\_list\_remove\_node\_mid, 78
- dl\_list\_append
  - cds\_dl\_list.h, 42
  - dl\_list.c, 67
- dl\_list\_data
  - cds\_dl\_list.h, 42
  - dl\_list.c, 67
- dl\_list\_delete\_at
  - cds\_dl\_list.h, 43
  - dl\_list.c, 67
- dl\_list\_find
  - dl\_list.c, 68
  - dl\_list.h, 76
- dl\_list\_find\_index
  - cds\_dl\_list.h, 43
  - dl\_list.c, 68
- dl\_list\_find\_itr
  - cds\_dl\_list.h, 43
  - dl\_list.c, 68
- dl\_list\_first
  - cds\_dl\_list.h, 43
  - dl\_list.c, 68
- dl\_list\_free
  - cds\_dl\_list.h, 44
  - dl\_list.c, 69
- dl\_list\_free\_node
  - dl\_list.c, 69
  - dl\_list.h, 76
- dl\_list\_init
  - cds\_dl\_list.h, 44
  - dl\_list.c, 69
- dl\_list\_insert\_after
  - cds\_dl\_list.h, 44
  - dl\_list.c, 69
- dl\_list\_insert\_at
  - cds\_dl\_list.h, 45
  - dl\_list.c, 70
- dl\_list\_insert\_before
  - cds\_dl\_list.h, 45
  - dl\_list.c, 70
- dl\_list\_insert\_node\_after\_mid
  - dl\_list.c, 70
  - dl\_list.h, 77
- dl\_list\_insert\_node\_back
  - dl\_list.c, 70
  - dl\_list.h, 77
- dl\_list\_insert\_node\_before\_mid
  - dl\_list.c, 71
  - dl\_list.h, 77
- dl\_list\_insert\_node\_front
  - dl\_list.c, 71
  - dl\_list.h, 77
- dl\_list\_isempty
  - cds\_dl\_list.h, 45
  - dl\_list.c, 71
- dl\_list\_itr\_from\_index

- cds\_dl\_list.h, 45
  - dl\_list.c, 71
- dl\_list\_last
  - cds\_dl\_list.h, 46
  - dl\_list.c, 72
- dl\_list\_length
  - cds\_dl\_list.h, 46
  - dl\_list.c, 72
- dl\_list\_lock
  - cds\_dl\_list.h, 46
  - dl\_list.c, 72
- dl\_list\_new\_node
  - dl\_list.c, 72
  - dl\_list.h, 77
- dl\_list\_next
  - cds\_dl\_list.h, 46
  - dl\_list.c, 72
- dl\_list\_node\_t, 7
  - data, 7
  - next, 7
  - prev, 7
- dl\_list\_prepend
  - cds\_dl\_list.h, 46
  - dl\_list.c, 73
- dl\_list\_prev
  - cds\_dl\_list.h, 47
  - dl\_list.c, 73
- dl\_list\_remove\_at
  - dl\_list.c, 73
  - dl\_list.h, 78
- dl\_list\_remove\_node\_back
  - dl\_list.c, 73
  - dl\_list.h, 78
- dl\_list\_remove\_node\_front
  - dl\_list.c, 73
  - dl\_list.h, 78
- dl\_list\_remove\_node\_mid
  - dl\_list.c, 74
  - dl\_list.h, 78
- dl\_list\_t, 8
  - back, 8
  - cfunc, 8
  - free\_func, 9
  - front, 9
  - length, 9
  - mutex, 9
- dl\_list\_unlock
  - cds\_dl\_list.h, 47
  - dl\_list.c, 74
- free\_func
  - bs\_tree\_t, 6
  - dl\_list\_t, 9
  - sl\_list\_t, 11
- front
  - dl\_list\_t, 9
  - sl\_list\_t, 11
- general.c, 79
- cds\_compare\_double, 80
- cds\_compare\_float, 80
- cds\_compare\_int, 81
- cds\_compare\_long, 81
- cds\_compare\_longlong, 81
- cds\_compare\_string, 81
- cds\_compare\_uint, 82
- cds\_compare\_ulong, 82
- cds\_compare\_ulonglong, 82
- cds\_new\_double, 83
- cds\_new\_float, 83
- cds\_new\_int, 83
- cds\_new\_long, 83
- cds\_new\_longlong, 83
- cds\_new\_string, 84
- cds\_new\_uint, 84
- cds\_new\_ulong, 84
- cds\_new\_ulonglong, 84
- key
  - kvpair\_t, 9
- kvpair\_t, 9
  - key, 9
  - value, 9
- left
  - bs\_tree\_node\_t, 5
- length
  - bs\_tree\_t, 6
  - dl\_list\_t, 9
  - sl\_list\_t, 11
- mutex
  - bs\_tree\_t, 7
  - dl\_list\_t, 9
  - sl\_list\_t, 11
- next
  - dl\_list\_node\_t, 7
  - sl\_list\_node\_t, 10
- prev
  - dl\_list\_node\_t, 7
- queue.c, 85
  - queue\_free, 86
  - queue\_init, 86
  - queue\_isempty, 86
  - queue\_length, 86
  - queue\_lock, 87
  - queue\_pop, 87
  - queue\_pushback, 87
  - queue\_unlock, 87
- queue\_free
  - cds\_queue.h, 54
  - queue.c, 86
- queue\_init
  - cds\_queue.h, 54
  - queue.c, 86
- queue\_isempty

- cds\_queue.h, 54
  - queue.c, 86
- queue\_length
  - cds\_queue.h, 55
  - queue.c, 86
- queue\_lock
  - cds\_queue.h, 55
  - queue.c, 87
- queue\_pop
  - cds\_queue.h, 55
  - queue.c, 87
- queue\_pushback
  - cds\_queue.h, 55
  - queue.c, 87
- queue\_unlock
  - cds\_queue.h, 56
  - queue.c, 87
- right
  - bs\_tree\_node\_t, 5
- root
  - bs\_tree\_t, 7
- sl\_list.c, 87
  - sl\_list\_data, 89
  - sl\_list\_delete\_at, 89
  - sl\_list\_find, 90
  - sl\_list\_find\_index, 90
  - sl\_list\_find\_itr, 90
  - sl\_list\_first, 90
  - sl\_list\_free, 90
  - sl\_list\_free\_node, 91
  - sl\_list\_init, 91
  - sl\_list\_insert\_after, 91
  - sl\_list\_insert\_at, 91
  - sl\_list\_isempty, 92
  - sl\_list\_itr\_from\_index, 92
  - sl\_list\_length, 92
  - sl\_list\_lock, 92
  - sl\_list\_new\_node, 92
  - sl\_list\_next, 93
  - sl\_list\_prepend, 93
  - sl\_list\_remove\_at, 93
  - sl\_list\_unlock, 93
- sl\_list.h, 94
  - sl\_list\_find, 95
  - sl\_list\_free\_node, 95
  - sl\_list\_new\_node, 96
  - sl\_list\_remove\_at, 96
- sl\_list\_data
  - cds\_sl\_list.h, 58
  - sl\_list.c, 89
- sl\_list\_delete\_at
  - cds\_sl\_list.h, 58
  - sl\_list.c, 89
- sl\_list\_find
  - sl\_list.c, 90
  - sl\_list.h, 95
- sl\_list\_find\_index
  - cds\_sl\_list.h, 59
  - sl\_list.c, 90
- sl\_list\_find\_itr
  - cds\_sl\_list.h, 59
  - sl\_list.c, 90
- sl\_list\_first
  - cds\_sl\_list.h, 59
  - sl\_list.c, 90
- sl\_list\_free
  - cds\_sl\_list.h, 59
  - sl\_list.c, 90
- sl\_list\_free\_node
  - sl\_list.c, 91
  - sl\_list.h, 95
- sl\_list\_init
  - cds\_sl\_list.h, 60
  - sl\_list.c, 91
- sl\_list\_insert\_after
  - cds\_sl\_list.h, 60
  - sl\_list.c, 91
- sl\_list\_insert\_at
  - cds\_sl\_list.h, 60
  - sl\_list.c, 91
- sl\_list\_isempty
  - cds\_sl\_list.h, 60
  - sl\_list.c, 92
- sl\_list\_itr\_from\_index
  - cds\_sl\_list.h, 61
  - sl\_list.c, 92
- sl\_list\_length
  - cds\_sl\_list.h, 61
  - sl\_list.c, 92
- sl\_list\_lock
  - cds\_sl\_list.h, 61
  - sl\_list.c, 92
- sl\_list\_new\_node
  - sl\_list.c, 92
  - sl\_list.h, 96
- sl\_list\_next
  - cds\_sl\_list.h, 61
  - sl\_list.c, 93
- sl\_list\_node\_t, 10
  - data, 10
  - next, 10
- sl\_list\_prepend
  - cds\_sl\_list.h, 61
  - sl\_list.c, 93
- sl\_list\_remove\_at
  - sl\_list.c, 93
  - sl\_list.h, 96
- sl\_list\_t, 10
  - cfunc, 11
  - free\_func, 11
  - front, 11
  - length, 11
  - mutex, 11
- sl\_list\_unlock
  - cds\_sl\_list.h, 62

- sl\_list.c, [93](#)
- stack.c, [96](#)
  - stack\_free, [97](#)
  - stack\_init, [97](#)
  - stack\_isempty, [97](#)
  - stack\_length, [98](#)
  - stack\_lock, [98](#)
  - stack\_pop, [98](#)
  - stack\_push, [98](#)
  - stack\_unlock, [99](#)
- stack\_free
  - cds\_stack.h, [63](#)
  - stack.c, [97](#)
- stack\_init
  - cds\_stack.h, [63](#)
  - stack.c, [97](#)
- stack\_isempty
  - cds\_stack.h, [64](#)
  - stack.c, [97](#)
- stack\_length
  - cds\_stack.h, [64](#)
  - stack.c, [98](#)
- stack\_lock
  - cds\_stack.h, [64](#)
  - stack.c, [98](#)
- stack\_pop
  - cds\_stack.h, [64](#)
  - stack.c, [98](#)
- stack\_push
  - cds\_stack.h, [65](#)
  - stack.c, [98](#)
- stack\_unlock
  - cds\_stack.h, [65](#)
  - stack.c, [99](#)
- value
  - kvpair\_t, [9](#)