CSC

1.0.0

Generated by Doxygen 1.8.6

Wed Jan 2 2019 14:39:35

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

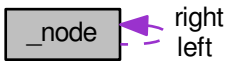
Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 _node Struct Reference

Collaboration diagram for _node:



Data Fields

- void * **data**
- struct _node * **left**
- struct _node * **right**

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

- /home/tamer/csc/src/cbst.c

3.2 cbitset Struct Reference

Data Fields

- bitset_type * data
- size_t nbits
- size_t size

3.2.1 Field Documentation

3.2.1.1 bitset_type* cbitset::data

The internal data of the bitset.

3.2.1.2 size_t cbitset::nbits

The number of bits the bitset can hold.

3.2.1.3 size_t cbitset::size

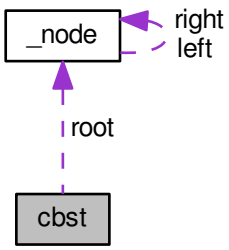
The number of elements stored in `cbitset::data`.

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

- /home/tamer/csc/src/cbitset.c

3.3 cbst Struct Reference

Collaboration diagram for cbst:



Data Fields

- _node * **root**
- size_t **size**

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

- /home/tamer/csc/src/cbst.c

3.4 cvector Struct Reference

Data Fields

- void ** data
- size_t size
- size_t capacity

3.4.1 Field Documentation

3.4.1.1 size_t cvector::capacity

The number of elements the vector is capable of storing before needing to resize.

3.4.1.2 void** cvector::data

The internal data store of the vector.

3.4.1.3 size_t cvector::size

The number of elements currently in the vector.

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

- /home/tamer/csc/src/cvector.c

Chapter 4

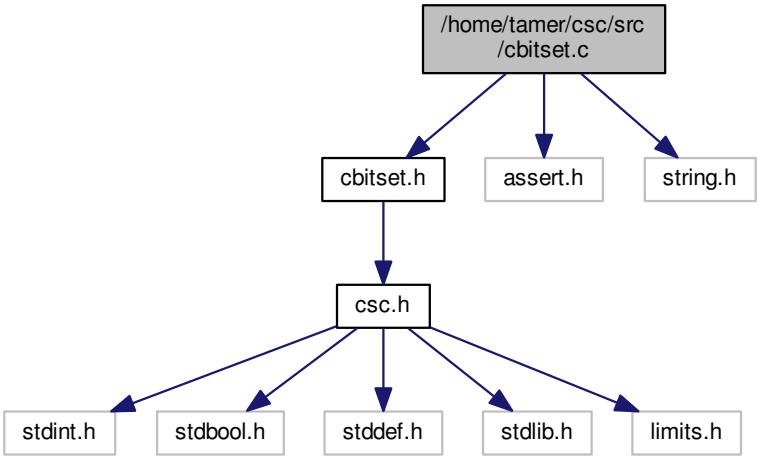
File Documentation

4.1 /home/tamer/csc/src/cbitset.c File Reference

implementation of the cbitset data structure.

```
#include "cbitset.h"
#include <assert.h>
#include <string.h>
```

Include dependency graph for cbitset.c:



Data Structures

- struct cbitset

Macros

- #define **CSC_BITSIZE** ((sizeof(bitset_type)) * (8))

Typedefs

- typedef uint_fast64_t **bitset_type**

Functions

- cbitset * csc_cbitset_create (size_t nbits)
  creates a cbitset.
- void csc_cbitset_destroy (cbitset *b)
  destroys a cbitset.
- size_t csc_cbitset_size (const cbitset *b)
  return the number of bits the bitset is capable of holding.
- CSCError csc_cbitset_set (cbitset *b, size_t bit)
  sets the 0-indexed bit supplied in the bitset.
- CSCError csc_cbitset_clear (cbitset *b, size_t bit)
  clears the 0-indexed bit supplied in the bitset.
- CSCError csc_cbitset_flip (cbitset *b, size_t bit)
  flips the 0-indexed bit supplied in the bitset.
- bool csc_cbitset_at (const cbitset *b, size_t bit, CSCError *e)
  retrieves the state of the bit at the specified 0-indexed position.
- void csc_cbitset_set_all (cbitset *b)
  sets all the bits in the bitset.
- void csc_cbitset_clear_all (cbitset *b)
  clears all the bits in the bitset.

4.1.1 Detailed Description

implementation of the cbitset data structure.

Author

Tamer Aly

Date

27 Dec 2018

See Also

cbitset.h

4.1.2 Function Documentation

4.1.2.1 bool csc_cbitset_at (const cbitset * *b*, size_t *bit*, CSCError * *e*)

retrieves the state of the bit at the specified 0-indexed position.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to check.
e	**optional** parameter to retrieve any errors. Can be `NULL`.

Returns

On success, `e` is `CSCError::E_NOERR`. If `bit` is out of range, `e` is `CSCError::E_OUTOFRANGE`. If the bit is set, `true` is returned and `false` otherwise.

4.1.2.2 CSCError csc_cbitset_clear (cbitset * *b*, size_t *bit*)

clears the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to clear.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.1.2.3 void csc_cbitset_clear_all (cbitset * *b*)

clears all the bits in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(n)`

Parameters

b	the bitset.

4.1.2.4 cbitset* csc_cbitset_create (size_t *nbits*)

creates a cbitset.

This function creates a cbitset capable of holding `nbits` of data. Note that `nbits` must be greater than 0. The bitset is initialized with all of the bits cleared.

Parameters

nbits	the number of bits the bitset should manage.

Returns

a pointer to a cbitset if successful. On failure or if `nbits` is 0, `NULL` is returned.

See Also

csc_cbitset_destroy

4.1.2.5 void csc_cbitset_destroy (cbitset * b)

destroys a cbitset.

This function destroys a cbitset by cleaning up any resources it holds. After a call to this function `b` should no longer be used.

`b` is expected to be **non-null**.

Parameters

b	the bitset.

See Also

csc_cbitset_create

4.1.2.6 CSCError csc_cbitset_flip (cbitset * b, size_t bit)

flips the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to flip.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.1.2.7 CSCError csc_cbitset_set (cbitset * b, size_t bit)

sets the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to set.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.1.2.8 void csc_cbitset_set_all (cbitset * b)

sets all the bits in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(n)`

Parameters

b	the bitset.

4.1.2.9 size_t csc_cbitset_size (const cbitset * b)

return the number of bits the bitset is capable of holding.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.

Returns

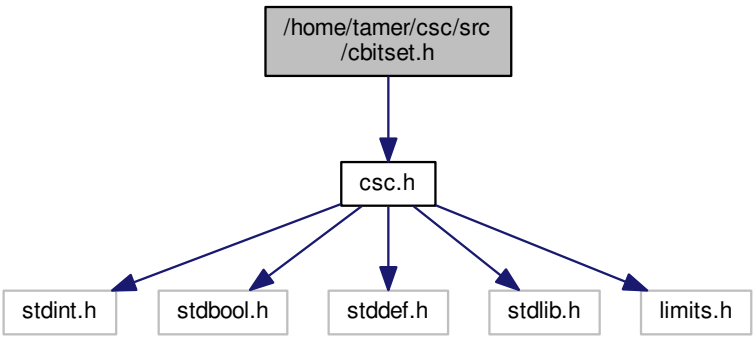
the number of bits the bitset is capable of holding.

4.2 /home/tamer/csc/src/cbitset.h File Reference

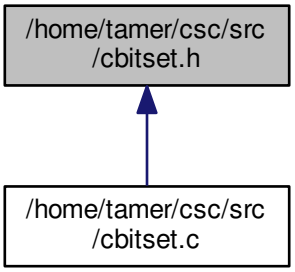
defines the interface to the cbitset data structure.

```
#include "csc.h"
```

Include dependency graph for cbitset.h:



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



Typedefs

- typedef struct cbitset cbitset
  the cbitset data structure.

Functions

- cbitset * csc_cbitset_create (size_t nbits)
  creates a cbitset.
- void csc_cbitset_destroy (cbitset *b)
  destroys a cbitset.
- size_t csc_cbitset_size (const cbitset *b)
  return the number of bits the bitset is capable of holding.
- CSCError csc_cbitset_set (cbitset *b, size_t bit)
  sets the 0-indexed bit supplied in the bitset.
- CSCError csc_cbitset_clear (cbitset *b, size_t bit)
  clears the 0-indexed bit supplied in the bitset.
- CSCError csc_cbitset_flip (cbitset *b, size_t bit)
  flips the 0-indexed bit supplied in the bitset.
- bool csc_cbitset_at (const cbitset *b, size_t bit, CSCError *e)
  retrieves the state of the bit at the specified 0-indexed position.
- void csc_cbitset_set_all (cbitset *b)
  sets all the bits in the bitset.
- void csc_cbitset_clear_all (cbitset *b)
  clears all the bits in the bitset.

4.2.1 Detailed Description

defines the interface to the cbitset data structure.

Author

Tamer Aly

Date

27 Dec 2018 Here is some code to get you started with basic usage of a cbitset:

```
// create a bitset capable of holding 10 bits
cbitset* b = csc_cbitset_create(10);

// set bit 2nd bit
CSCError e = csc_cbitset_set(b, 1);
if (e != E_NOERR) {
     // handle the error
}

// check if the 3rd bit is set
if (csc_cbitset_at(b, 2, &e)) {
     // the bit is set
} else {
     // the bit isn't set
}

// flip the 4th bit
e = csc_cbitset_flip(b, 3);
if (e != E_NOERR) {
     // handle the error
}

// access the 11th bit (an error)
if (csc_cbitset_at(b, 10, &e)) {
     // can't happen
} else {
     if (e != E_NOERR) {
         // out of range
     } else {
         // can't happen
     }
}


// clean up
csc_cbitset_destroy(b);
```

See Also

cbitset.c

4.2.2 Typedef Documentation

4.2.2.1 typedef struct cbitset cbitset

the cbitset data structure.

A bitset is a data structure that allows a user to manipulate state that can be represented in a single bit. Normally, this is used to store the state of several boolean conditions in a space efficient manner. You can think of a bitset as a space-optimized version of a vector of `bool` types.

4.2.3 Function Documentation

4.2.3.1 bool csc_cbitset_at (const cbitset * b, size_t bit, CSCError * e)

retrieves the state of the bit at the specified 0-indexed position.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to check.
e	**optional** parameter to retrieve any errors. Can be `NULL`.

Returns

On success, `e` is `CSCError::E_NOERR`. If `bit` is out of range, `e` is `CSCError::E_OUTOFRANGE`. If the bit is set, `true` is returned and `false` otherwise.

4.2.3.2 CSCError csc_cbitset_clear (cbitset * *b*, size_t *bit*)

clears the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to clear.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.2.3.3 void csc_cbitset_clear_all (cbitset * *b*)

clears all the bits in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(n)`

Parameters

b	the bitset.

4.2.3.4 cbitset* csc_cbitset_create (size_t *nbits*)

creates a cbitset.

This function creates a cbitset capable of holding `nbits` of data. Note that `nbits` must be greater than 0. The bitset is initialized with all of the bits cleared.

Parameters

nbits	the number of bits the bitset should manage.

Returns

a pointer to a cbitset if successful. On failure or if `nbits` is 0, `NULL` is returned.

See Also

csc_cbitset_destroy

4.2.3.5 void csc_cbitset_destroy (cbitset * b)

destroys a cbitset.

This function destroys a cbitset by cleaning up any resources it holds. After a call to this function `b` should no longer be used.

`b` is expected to be **non-null**.

Parameters

b	the bitset.

See Also

csc_cbitset_create

4.2.3.6 CSCError csc_cbitset_flip (cbitset * b, size_t bit)

flips the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to flip.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.2.3.7 CSCError csc_cbitset_set (cbitset * b, size_t bit)

sets the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.
bit	the 0-indexed bit to set.

Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

4.2.3.8 void csc_cbitset_set_all (cbitset * b)

sets all the bits in the bitset.

`b` is expected to be **non-null**.

Time Complexity: `O(n)`

Parameters

b	the bitset.

4.2.3.9 size_t csc_cbitset_size (const cbitset * b)

return the number of bits the bitset is capable of holding.

`b` is expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the bitset.

Returns

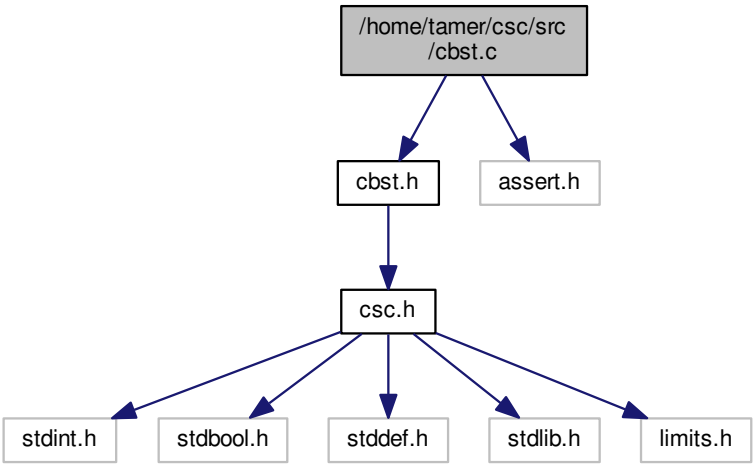
the number of bits the bitset is capable of holding.

4.3 /home/tamer/csc/src/cbst.c File Reference

This file defines the implements the cbst data structure and interface functions.

```
#include "cbst.h"
#include <assert.h>
```

Include dependency graph for cbst.c:



Data Structures

- struct _node
- struct cbst

Typedefs

- typedef struct _node **_node**

Functions

- cbst * csc_cbst_create ()

  cbst "constructor" function
- void csc_cbst_destroy (cbst *b)

  cbst "destructor" function
- CSCError csc_cbst_add (cbst *b, void *elem, csc_compare cmp)

  adds an element into the BST.
- void * csc_cbst_rm (cbst *b, const void *elem, csc_compare cmp)

  removes an element from the BST.
- void * csc_cbst_find (const cbst *b, const void *elem, csc_compare cmp)

  finds the element in the specified BST.
- size_t csc_cbst_size (const cbst *b)

  returns the size of the BST.
- bool csc_cbst_empty (const cbst *b)

  checks if the BST is empty.
- void csc_cbst_foreach (cbst *b, csc_foreach fn, void *context)

  *applies the callback function to each element of the BST in an **in-order** traversal.The user may pass in additional context using the `context` param or pass in `NULL` if not required.*

4.3.1 Detailed Description

This file defines the implements the cbst data structure and interface functions.

Author

Tamer Aly

Date

27 Dec 2018

4.3.2 Function Documentation

4.3.2.1 CSCError csc_cbst_add (cbst * *b*, void * *elem*, csc_compare *cmp*)

adds an element into the BST.

This function adds `elem` into the supplied BST. Note that adding the element into the BST does **not** make the BST own the element. The user is still responsible for cleaning up that memory.Moreover, duplicate elements are not allowed.

Both `elem` and `b` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to add.

Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`. If a duplicate element is attempted to be added, `CSCError::E_INVALIDOPERATION`.

4.3.2.2 cbst* csc_cbst_create ( )

cbst "constructor" function

This function is used to create a `cbst`. If the function is successful, the function returns a pointer to a `cbst` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed cbst.

See Also

csc_cbst_destroy

4.3.2.3 void csc_cbst_destroy (cbst * *b*)

cbst "destructor" function

This function is used to clean up resources used by a `cbst` created via the csc_cbst_create function. This function must be called whenever a cbst is no longer used.

See Also

csc_cbst_create

4.3.2.4 bool csc_cbst_empty (const cbst * *b*)

checks if the BST is empty.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the BST.

Returns

`true` if the BST is empty. In other words, true if `csc_cbst_size(v)` == 0. Otherwise, `false`.

4.3.2.5 void* csc_cbst_find (const cbst * *b,* const void * *elem,* csc_compare *cmp*)

finds the element in the specified BST.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to find.
cmp	the comparison function to use. See csc_compare for more details.

Returns

the element or `NULL` if the element couldn't be found.

4.3.2.6 void csc_cbst_foreach (cbst * *b*, csc_foreach *fn*, void * *context*)

applies the callback function to each element of the BST in an **in-order** traversal.The user may pass in additional context using the `context` param or pass in `NULL` if not required.

Time Complexity: `O(n)`

Parameters

b	the BST.
fn	the callback function to apply to each element.
context	user-defined data that will be applied to the callback. Can be `NULL` if unused.

4.3.2.7 void* csc_cbst_rm (cbst * *b*, const void * *elem*, csc_compare *cmp*)

removes an element from the BST.

This function removes `elem` from the supplied BST if it exists. In order to remove the element, the function must search for the element in the BST using the supplied `cmp` function. The removed element is returned.

All three parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

See Also

csc_compare
csc_cbst_find

4.3.2.8 size_t csc_cbst_size (const cbst * *b*)

returns the size of the BST.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the BST.

Returns

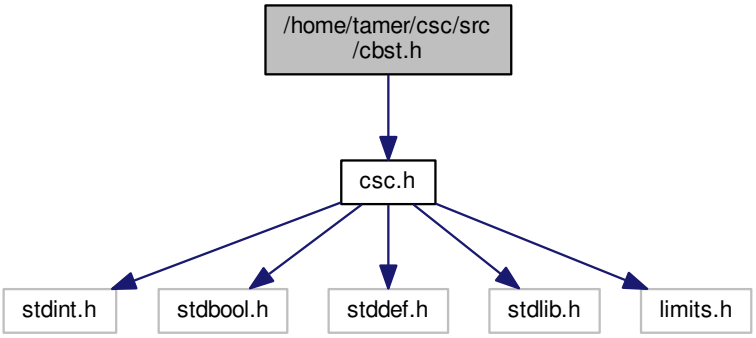
the size of the BST.

4.4 /home/tamer/csc/src/cbst.h File Reference

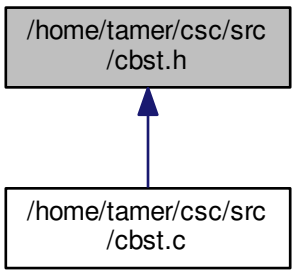
This file defines the interface to the cbst data structure.

```
#include "csc.h"
```

Include dependency graph for cbst.h:



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



Typedefs

- typedef struct cbst cbst

  implementation of a binary search tree.

Functions

- cbst * csc_cbst_create ()

  cbst "constructor" function
- void csc_cbst_destroy (cbst *b)

  cbst "destructor" function
- CSCError csc_cbst_add (cbst *b, void *elem, csc_compare cmp)

  adds an element into the BST.
- void * csc_cbst_rm (cbst *b, const void *elem, csc_compare cmp)

  removes an element from the BST.
- void * csc_cbst_find (const cbst *b, const void *elem, csc_compare cmp)

  finds the element in the specified BST.
- size_t csc_cbst_size (const cbst *b)

  returns the size of the BST.
- bool csc_cbst_empty (const cbst *b)

  checks if the BST is empty.
- void csc_cbst_foreach (cbst *b, csc_foreach fn, void *context)

  *applies the callback function to each element of the BST in an **in-order** traversal.The user may pass in additional context using the `context` param or pass in `NULL` if not required.*

4.4.1 Detailed Description

This file defines the interface to the cbst data structure.

Author

Tamer Aly

Date

27 Dec 2018 cbst implements a standard binary search tree (BST). In this implementation, attempting to add duplicate or `NULL` keys is not allowed. See #csc_bst_add for more details.

Here is a brief code sample to get you started with using cbst:

```
// a callback function to call on each element of the tree
void print_and_free_elem(void* elem, void* context);

//
// somewhere in main
//

// create a tree
cbst* tree = csc_cbst_create();
if (tree == NULL) {
    // couldn't create the tree
}

// add some elements
int elems[] = {5, 3, 7, 2, 4, 6, 8};
for (int i = 0; i < sizeof(elems) / sizeof(int); ++i) {
     int* x = malloc(sizeof(*x));
     if (x == NULL) {
         // couldn't allocate memory
     }
     *x = elems[i];
     CSCError e = csc_cbst_add(tree, x, csc_cmp_int);
     if (e != E_NOERR) {
         // handle the error
     }
}

// remove an element
int* e = (int*) csc_cbst_rm(b, &elems[0], csc_cmp_int);
if (e == NULL) {
```

```
    // the element didn't exist
} else {
    // e is the element which was removed.
    // here is an opportunity to do any resource cleanup or operations on e.
    free(e);
}

// find an element
int* elem = (int*) csc_cbst_find(b, &elems[2], csc_cmp_int);
if (elem == NULL) {
    // element wasn't found.
}

// get the number of elements in the tree
size_t n_elems = csc_cbst_size(b);

// iterate over all of the elements in the tree in an in-order traversal
csc_cbst_foreach(b, print_elem, NULL);

// clean up
csc_cbst_destroy(b);

// implement the callback
void print_and_free_elem(void* elem, void* context)
{
    CSC_UNUSED(context);
    int* e = (int*) elem;
    printf("%d\n", *e);
    free(e);
}
```

4.4.2 Typedef Documentation

4.4.2.1 typedef struct cbst cbst

implementation of a binary search tree.

See Also

csc_cbst_create

4.4.3 Function Documentation

4.4.3.1 CSCError csc_cbst_add (cbst * b, void * elem, csc_compare cmp)

adds an element into the BST.

This function adds `elem` into the supplied BST. Note that adding the element into the BST does **not** make the BST own the element. The user is still responsible for cleaning up that memory.Moreover, duplicate elements are not allowed.

Both `elem` and `b` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to add.

Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`. If a duplicate element is attempted to be added, `CSCError::E_INVALIDOPERATION`.

4.4.3.2 cbst* csc_cbst_create ( )

cbst "constructor" function

This function is used to create a `cbst`. If the function is successful, the function returns a pointer to a `cbst` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed cbst.

See Also

csc_cbst_destroy

4.4.3.3 void csc_cbst_destroy (cbst * *b*)

cbst "destructor" function

This function is used to clean up resources used by a `cbst` created via the csc_cbst_create function. This function must be called whenever a cbst is no longer used.

See Also

csc_cbst_create

4.4.3.4 bool csc_cbst_empty (const cbst * *b*)

checks if the BST is empty.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the BST.

Returns

`true` if the BST is empty. In other words, true if `csc_cbst_size(v)` == 0. Otherwise, `false`.

4.4.3.5 void* csc_cbst_find (const cbst * *b*, const void * *elem*, csc_compare *cmp*)

finds the element in the specified BST.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to find.
cmp	the comparison function to use. See csc_compare for more details.

Returns

the element or `NULL` if the element couldn't be found.

4.4.3.6 void csc_cbst_foreach (cbst * *b*, csc_foreach *fn*, void * *context*)

applies the callback function to each element of the BST in an **in-order** traversal.The user may pass in additional context using the `context` param or pass in `NULL` if not required.

Time Complexity: `O(n)`

Parameters

b	the BST.
fn	the callback function to apply to each element.
context	user-defined data that will be applied to the callback. Can be `NULL` if unused.

4.4.3.7 void* csc_cbst_rm (cbst * *b*, const void * *elem*, csc_compare *cmp*)

removes an element from the BST.

This function removes `elem` from the supplied BST if it exists. In order to remove the element, the function must search for the element in the BST using the supplied `cmp` function. The removed element is returned.

All three parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(h)` worst case, `O(log(n))` average case where `h` is the height of the tree and `n` is the number of elements the tree holds.

Parameters

b	the BST.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

See Also

csc_compare
csc_cbst_find

4.4.3.8 size_t csc_cbst_size (const cbst * *b*)

returns the size of the BST.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

b	the BST.

Returns

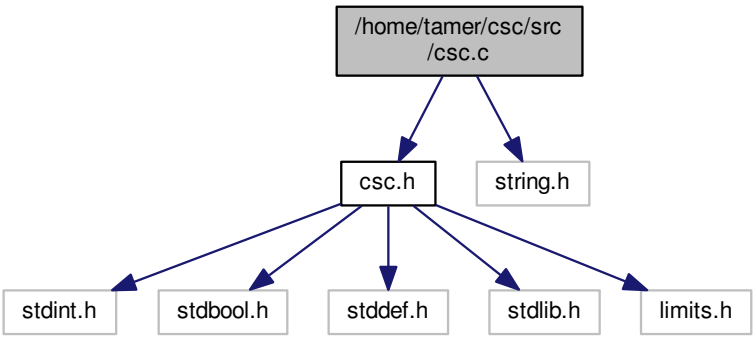
the size of the BST.

4.5 /home/tamer/csc/src/csc.c File Reference

the implementation file for csc.h.

```
#include "csc.h"
#include <string.h>
```

Include dependency graph for csc.c:



Macros

- #define CSC_DEFINE_BUILTIN_CMP(type)

  implements a builtin type comparison function

Functions

- void csc_swap (void **a, void **b)

  generic swap function to swap two `void*`
- void csc_error_str (CSCError e, char *buf, size_t len)

  returns a library-defined error string depending on the error.

4.5.1 Detailed Description

the implementation file for csc.h.

Author

Tamer Aly

Date

27 Dec 2018

See Also

csc.h

4.5.2 Macro Definition Documentation

4.5.2.1 #define CSC_DEFINE_BUILTIN_CMP(*type*)

Value:

```
int csc_cmp_##type(const void* a, const void* b) \
{\
    type vA = *(type*)a;\
    type vB = *(type*)b;\
    if (vA == vB) { return 0; }\
    else if (vA < vB) { return -1; }\
    else { return 1; }\
}
```

implements a builtin type comparison function

When defined with a type, this macro will implement the function signature that the `CSC_DECLARE_BUILTIN_CMP` defines. Note that the function signature **must** be declared first using `CSC_DECLARE_BUILTIN_CMP` in the header file.

See Also

csc.h

4.5.3 Function Documentation

4.5.3.1 void csc_error_str (CSCError *e*, char * *buf*, size_t *len*)

returns a library-defined error string depending on the error.

This is a convenience function that populates `buf` of length `len` with a library-defined error message that depends on the value of `e`. It is recommended that `len` is *at least* CSC_MAX_ERROR_MSG_LEN.

This function should ideally be used after a library call returning a CSCError for a simple diagnostic error handling mechanism. For example:

```
CSCError e = csc_some_func(args...);
if (e != E_NOERR) { // uh oh. An error.
     char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
     csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
     puts(buf);
}
```

Parameters

e	the error.
buf	the buffer to fill. Must be **non-null**.
len	the length of the buffer. Recommended to be >= CSC_MAX_ERROR_MSG_LEN.

See Also

CSC_MAX_ERROR_MSG_LEN

4.5.3.2 void csc_swap (void ** a, void ** b)

generic swap function to swap two `void*`

This is a generic swap function to swap the provided elements.

Parameters

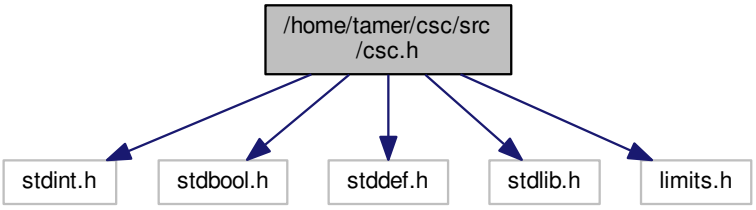
a	A is the first elem.
b	B is the second elem.

4.6 /home/tamer/csc/src/csc.h File Reference

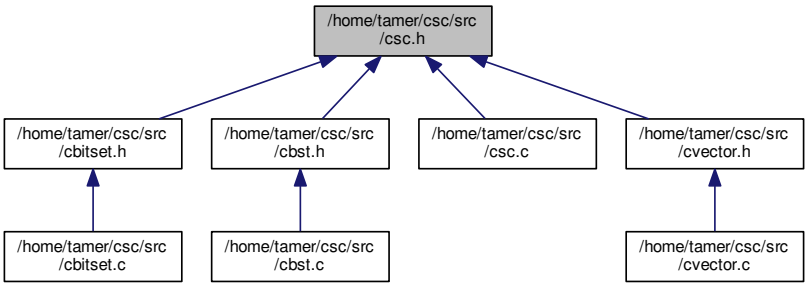
the main include file for the csc library.

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <limits.h>
```

Include dependency graph for csc.h:



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



Macros

- #define CSC_UNUSED(x) (void)x

  macro that silences compiler warnings about unused function parameters.
- #define CSC_64

  This macro is only defined if compiling on a 64 bit architecture.
- #define CSC_32

  This macro is only defined if compiling on a 32 bit architecture.

- #define CSC_DECLARE_BUILTIN_CMP(type) int csc_cmp_##type(const void* a, const void* b)

  convenience macro defining comparison functions for built in types.
- #define CSC_MAX_ERROR_MSG_LEN 128

  the maximum message length a CSCError is guaranteed to generate.

Typedefs

- typedef enum CSCError CSCError

  the list of errors that can be returned by the library.
- typedef int(* csc_compare)(const void *a, const void *b)

  comparison function callback for comparing two elements
- typedef void(* csc_foreach)(void *elem, void *context)

  callback function for iterating the elements of a container.

Enumerations

- enum CSCError {
  E_NOERR = 0, E_OUTOFMEM, E_OUTOFRANGE, E_INVALIDOPERATION,
  E_ERR_N }

  the list of errors that can be returned by the library.

Functions

- void csc_swap (void **a, void **b)

  generic swap function to swap two `void*`
- void csc_error_str (CSCError e, char *buf, size_t len)

  returns a library-defined error string depending on the error.
- **CSC_DECLARE_BUILTIN_CMP** (int)

4.6.1 Detailed Description

the main include file for the csc library.

Author

Tamer Aly

Date

27 Dec 2018 This is the main include file that must be included alongside any other source and header combination for a particular data structure in the library. This file defines several helper functions that are used throughout the library.

4.6.2 Macro Definition Documentation

4.6.2.1 #define CSC_DECLARE_BUILTIN_CMP(*type*) int csc_cmp_##type(const void* a, const void* b)

convenience macro defining comparison functions for built in types.

This macro defines a comparison function for built-in C types.

For example, defining CSC_DECLARE_BUILTIN_CMP(int) would create the signature:

```
int csc_cmp_int(const void* a, const void* b);
```

Note that this macro only creates the signature of the function. See csc.c for how to implement the signature.

See Also

csc.c

4.6.2.2 #define CSC_MAX_ERROR_MSG_LEN 128

the maximum message length a CSCError is guaranteed to generate.

See Also

csc_error_str

4.6.2.3 #define CSC_UNUSED(*x*) (void)x

macro that silences compiler warnings about unused function parameters.

This macro is used to silence compiler warnings about unused function parameters. Mostly, this is for unused context parameters in generic callback functions used internally by the library.

4.6.3 Typedef Documentation

4.6.3.1 typedef int(∗ csc_compare)(const void ∗a, const void ∗b)

comparison function callback for comparing two elements

This is a comparison function callback for comparing two elements that is used for routine functions like sorting or searching a generic container. When creating a custom comparison function for your type, the following protocol **must** be adhered to: a return value < 0 means `a` is less than `b`. a return value of > 0 means `a` is greater than `b`. a return value of 0 means `a` is equal to `b`.

As a convenience, the library provides comparison functions for all the C built in types.

Parameters

a	the first element
b	the second element

See Also

CSC_DECLARE_BUILTIN_CMP

4.6.3.2 typedef void(∗ csc_foreach)(void ∗elem, void ∗context)

callback function for iterating the elements of a container.

This callback function defines an operation that will be applied to each element of a container.

Parameters

elem	the element to process
context	user-defined data that can be passed into the function. Can be `NULL` if unused.

4.6.3.3 typedef enum CSCError CSCError

the list of errors that can be returned by the library.

This enumeration defines all of the errors that can be returned by certain library calls. These errors can provide more diagnostic information than a simple true/false return. Whenever this error type is returned, a type of `CSCError::E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCError::E_ERR_N`, indicates an error condition.

It is recommended that you check for this error code whenever possible:

```
CSCError e = csc_function(args...);
if (e != E_NOERR) {
     // handle the error by printing a simple diagnostic message.
     char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
     csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
     puts(buf);
}
```

See Also

csc_error_str

4.6.4 Enumeration Type Documentation

4.6.4.1 enum CSCError

the list of errors that can be returned by the library.

This enumeration defines all of the errors that can be returned by certain library calls. These errors can provide more diagnostic information than a simple true/false return. Whenever this error type is returned, a type of `CSCError::E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCError::E_ERR_N`, indicates an error condition.

It is recommended that you check for this error code whenever possible:

```
CSCError e = csc_function(args...);
if (e != E_NOERR) {
     // handle the error by printing a simple diagnostic message.
     char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
     csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
     puts(buf);
}
```

See Also

csc_error_str

Enumerator

E_NOERR This indicates no errors occurred in the operation.

E_OUTOFMEM This error indicates the operation failed since memory could not be allocated.

E_OUTOFRANGE This error indicates that the operation failed due to accessing an out of range element. i.e. array index of -1.

E_INVALIDOPERATION This error indicates that the operation failed because an invalid operation was attempted.

E_ERR_N This is never returned by any function calls and can be ignored.

4.6.5 Function Documentation

4.6.5.1 void csc_error_str (CSCError *e*, char * *buf*, size_t *len*)

returns a library-defined error string depending on the error.

This is a convenience function that populates `buf` of length `len` with a library-defined error message that depends on the value of `e`. It is recommended that `len` is *at least* CSC_MAX_ERROR_MSG_LEN.

This function should ideally be used after a library call returning a CSCError for a simple diagnostic error handling mechanism. For example:

```
CSCError e = csc_some_func(args...);
if (e != E_NOERR) { // uh oh. An error.
     char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
     csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
     puts(buf);
}
```

Parameters

e	the error.
buf	the buffer to fill. Must be **non-null**.
len	the length of the buffer. Recommended to be >= CSC_MAX_ERROR_MSG_LEN.

See Also

CSC_MAX_ERROR_MSG_LEN

4.6.5.2 void csc_swap (void ** *a*, void ** *b*)

generic swap function to swap two `void*`

This is a generic swap function to swap the provided elements.

Parameters

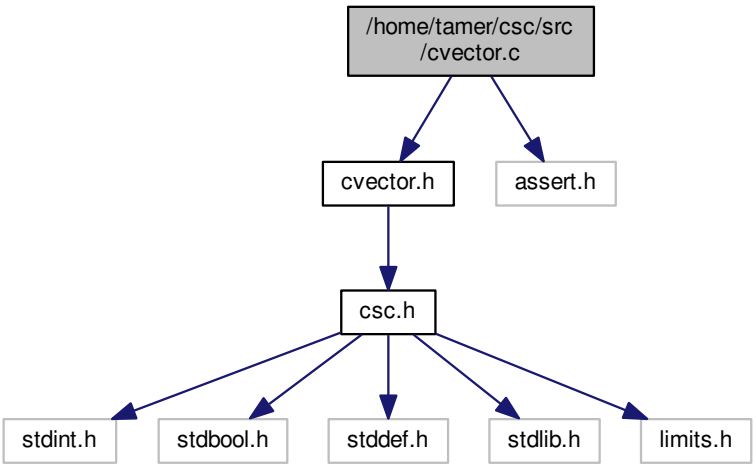
a	A is the first elem.
b	B is the second elem.

4.7 /home/tamer/csc/src/cvector.c File Reference

contains the implementation of the cvector structure.

```
#include "cvector.h"
#include <assert.h>
```

Include dependency graph for cvector.c:



Data Structures

- struct cvector

Functions

- cvector * csc_cvector_create ()
  cvector "constructor" function
- size_t csc_cvector_size (const cvector *v)
  returns the size of the vector.
- size_t csc_cvector_capacity (const cvector *v)
  returns the capacity of the vector.
- void csc_cvector_destroy (cvector *v)
  cvector "destructor" function
- void csc_cvector_foreach (cvector *v, csc_foreach fn, void *context)
  applies the callback function to each element of the vector.
- CSCError csc_cvector_add (cvector *v, void *elem)
  adds an element into the vector.
- void * csc_cvector_at (const cvector *v, size_t idx)
  returns the element at the specified index.
- void * csc_cvector_rm (cvector *v, const void *elem, csc_compare cmp)
  removes an element from the vector.
- void * csc_cvector_rm_at (cvector *v, size_t idx)
  removes the element at the specified 0-indexed index from the vector.
- void * csc_cvector_find (const cvector *v, const void *elem, csc_compare cmp)
  finds the element in the specified vector.
- bool csc_cvector_empty (const cvector *v)
  checks if the vector is empty.
- CSCError csc_cvector_reserve (cvector *v, size_t num_elems)

reserves memory for the specified number of elements in the vector.

- CSCError csc_cvector_shrink_to_fit (cvector *v)

  shrinks the capacity to match the size of the vector.

4.7.1 Detailed Description

contains the implementation of the cvector structure.

Author

Tamer Aly

Date

27 Dec 2018

4.7.2 Function Documentation

4.7.2.1 CSCError csc_cvector_add (cvector * *v*, void * *elem*)

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that adding the element into the vector does **not** make the vector own the element. The user is still responsible for cleaning up that memory.

Both `elem` and `v` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

Time Complexity: `O(1)` best case, `O(n)` worst case, `O(1)` amortized.

Parameters

v	the vector.
elem	the element to add.

Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`.

4.7.2.2 void* csc_cvector_at (const cvector * *v*, size_t *idx*)

returns the element at the specified index.

This function performs a range check to ensure that `idx` is less than the size of the vector.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.
idx	the index.

Returns

the element at that index in the vector or `NULL` if the index is out of range.

4.7.2.3 size_t csc_cvector_capacity (const cvector * v)

returns the capacity of the vector.

This function returns the number of elements the vector can hold before it needs to be resized.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

the capacity of the vector.

4.7.2.4 cvector* csc_cvector_create ( )

cvector "constructor" function

This function is used to create a `cvector`. If the function is successful, the function returns a pointer to a `cvector` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed cvector.

See Also

csc_cvector_destroy

4.7.2.5 void csc_cvector_destroy (cvector * v)

cvector "destructor" function

This function is used to clean up resources used by a `cvector` created via the csc_cvector_create function. This function must be called whenever a cvector is no longer used.

See Also

csc_cvector_create

4.7.2.6 bool csc_cvector_empty (const cvector * v)

checks if the vector is empty.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

`true` if the vector is empty. In other words, true if `csc_cvector_size(v)` == 0. Otherwise, `false`.

4.7.2.7 void* csc_cvector_find (const cvector * *v*, const void * *elem*, csc_compare *cmp*)

finds the element in the specified vector.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, O(n) average and worst case.

Parameters

v	the vector.
elem	the element to find.
cmp	the comparison function to use. See csc_compare for more details.

Returns

the element or `NULL` if the element couldn't be found.

4.7.2.8 void csc_cvector_foreach (cvector * *v*, csc_foreach *fn*, void * *context*)

applies the callback function to each element of the vector.

This callback function defines an operation that will be applied to each element of the `cvector`. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

Time Complexity: `O(n)`

Parameters

v	the vector.
fn	the callback function to apply to each element.
context	user-defined data that will be applied to the callback. Can be `NULL` if unused.

See Also

csc_foreach

4.7.2.9 CSCError csc_cvector_reserve (cvector * *v*, size_t *num_elems*)

reserves memory for the specified number of elements in the vector.

This functions reserves enough memory in the vector such that it is able to hold *at least* `num_elems` without needing to expand. If the number of elements that will be contained in the vector is known or can be estimated, you may be able to improve the performance of your application by allocating the memory for the elements up front using this function. As always with performance, your milage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is $<$ `csc_cvector_size(v)`, that is an error.

All parameters are expected to be **non-null**.

Time Complexity: OS-specific.

Parameters

v	the vector.

num_elems	the number of elements to allocate memory for.

Returns

On success `CSCError::E_NOERR`. If the requested size is less than the current size, `CSCError::E_INVALIDOPERATION`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.7.2.10 void* csc_cvector_rm (cvector * *v*, const void * *elem*, csc_compare *cmp*)

removes an element from the vector.

This function removes `elem` from the supplied vector if it exists. In order to remove the element, the function must search for the element in the vector using the supplied `cmp` function.

All three parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(n)` average and worst case.

Parameters

v	the vector.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

See Also

csc_compare
csc_cvector_find

4.7.2.11 void* csc_cvector_rm_at (cvector * *v*, size_t *idx*)

removes the element at the specified 0-indexed index from the vector.

This function the element at index `idx` from `v`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.
idx	the index.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

4.7.2.12 CSCError csc_cvector_shrink_to_fit (cvector * *v*)

shrinks the capacity to match the size of the vector.

After a call to this function, the following will be true:

```
csc_cvector_size(v) == csc_cvector_capacity(v);
```

This function may be useful in low-memory settings where the vector's capacity greatly exceeds the size and the extra memory won't be required.

All parameters are expected to be **non-null**.

Time Complexity: OS-specific.

Parameters

v	the vector.

Returns

On success `CSCError::E_NOERR`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.7.2.13 size_t csc_cvector_size (const cvector * v)

returns the size of the vector.

This function returns the number of elements currently in the vector.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

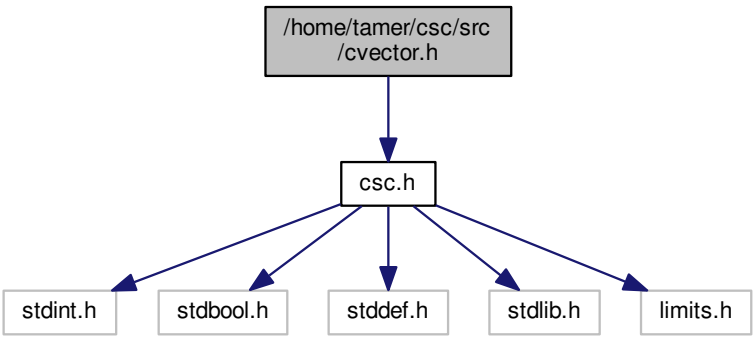
the size of the vector.

4.8 /home/tamer/csc/src/cvector.h File Reference

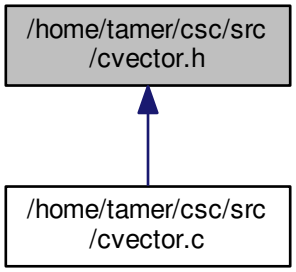
contains interface of the cvector structure.

```
#include "csc.h"
```

Include dependency graph for cvector.h:



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



Typedefs

- typedef struct cvector cvector

  implementation of a generic dynamic array.

Functions

- cvector ∗ csc_cvector_create ()

  cvector "constructor" function
- void csc_cvector_destroy (cvector ∗v)

  cvector "destructor" function
- CSCError csc_cvector_add (cvector ∗v, void ∗elem)

  adds an element into the vector.
- void ∗ csc_cvector_rm (cvector ∗v, const void ∗elem, csc_compare cmp)

  removes an element from the vector.
- void ∗ csc_cvector_rm_at (cvector ∗v, size_t idx)

  removes the element at the specified 0-indexed index from the vector.
- void ∗ csc_cvector_find (const cvector ∗v, const void ∗elem, csc_compare cmp)

  finds the element in the specified vector.
- size_t csc_cvector_size (const cvector ∗v)

  returns the size of the vector.
- size_t csc_cvector_capacity (const cvector ∗v)

  returns the capacity of the vector.
- void csc_cvector_foreach (cvector ∗v, csc_foreach fn, void ∗context)

  applies the callback function to each element of the vector.
- void ∗ csc_cvector_at (const cvector ∗v, size_t idx)

  returns the element at the specified index.
- bool csc_cvector_empty (const cvector ∗v)

  checks if the vector is empty.
- CSCError csc_cvector_reserve (cvector ∗v, size_t num_elems)

  reserves memory for the specified number of elements in the vector.
- CSCError csc_cvector_shrink_to_fit (cvector ∗v)

  shrinks the capacity to match the size of the vector.

4.8.1 Detailed Description

contains interface of the cvector structure.

Author

Tamer Aly

Date

27 Dec 2018 Here is example code to get you started on using the `cvector`:

```
// for-each callback function signature
void print_elem(void* elem, void* context);

//
// somewhere in main....
//

// create a vector
cvector* v = csc_cvector_create();
if (v == NULL) {
     // couldn't create the vector.
}

// add some elements into the vector.
for (int i = 0; i < 10; i++) {
     int* x = malloc(sizeof(*x));
     if (x == NULL) {
         // couldn't allocate memory.
     }
     *x = i;

     CSCError e = csc_cvector_add(v, x);
     if (e != E_NOERR) {
         // couldn't add the element.
     }
}

// get the size
size_t size = csc_cvector_size(v);

// print the elements "manually"
for (size_t i = 0; i < size; i++) {
    int* x = (int*) csc_cvector_at(v, i);
    printf("%d\n", *x);
}

// remove the 2nd element
CSCError e = csc_cvector_rm_at(v, 1);
if (e != E_NOERR) {
    // couldn't remove the element
}

// print the elements "functionally"
csc_cvector_foreach(v, print_elem, NULL);

// clean up resources
for (size_t i = 0; i < csc_cvector_size(v); ++i) {
     void* x = csc_cvector_at(v, i);
     free(x);
}
csc_cvector_destroy(v);

//
// somewhere outside of main...
//

// implement the callback
void print_elem(void* elem, void* context)
{
     CSC_UNUSED(context); // no need for context
     printf("%d\n", *(int*)elem);
}
```

4.8.2 Typedef Documentation

4.8.2.1 typedef struct cvector cvector

implementation of a generic dynamic array.

cvector implements a dynamic array that mimics `std::vector` from C++.

See Also

csc_cvector_create

4.8.3 Function Documentation

4.8.3.1 CSCError csc_cvector_add (cvector * *v*, void * *elem*)

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that adding the element into the vector does **not** make the vector own the element. The user is still responsible for cleaning up that memory.

Both `elem` and `v` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

Time Complexity: `O(1)` best case, `O(n)` worst case, `O(1)` amortized.

Parameters

v	the vector.
elem	the element to add.

Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`.

4.8.3.2 void* csc_cvector_at (const cvector * *v*, size_t *idx*)

returns the element at the specified index.

This function performs a range check to ensure that `idx` is less than the size of the vector.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.
idx	the index.

Returns

the element at that index in the vector or `NULL` if the index is out of range.

4.8.3.3 size_t csc_cvector_capacity (const cvector * *v*)

returns the capacity of the vector.

This function returns the number of elements the vector can hold before it needs to be resized.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

the capacity of the vector.

4.8.3.4 cvector* csc_cvector_create ( )

cvector "constructor" function

This function is used to create a `cvector`. If the function is successful, the function returns a pointer to a `cvector` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed cvector.

See Also

csc_cvector_destroy

4.8.3.5 void csc_cvector_destroy (cvector * *v*)

cvector "destructor" function

This function is used to clean up resources used by a `cvector` created via the csc_cvector_create function. This function must be called whenever a cvector is no longer used.

See Also

csc_cvector_create

4.8.3.6 bool csc_cvector_empty (const cvector * *v*)

checks if the vector is empty.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

`true` if the vector is empty. In other words, true if `csc_cvector_size(v)` == 0. Otherwise, `false`.

4.8.3.7 void* csc_cvector_find (const cvector * *v*, const void * *elem*, csc_compare *cmp*)

finds the element in the specified vector.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, O(n) average and worst case.

Parameters

v	the vector.
elem	the element to find.
cmp	the comparison function to use. See csc_compare for more details.

Returns

the element or `NULL` if the element couldn't be found.

4.8.3.8 void csc_cvector_foreach (cvector * *v*, csc_foreach *fn*, void * *context*)

applies the callback function to each element of the vector.

This callback function defines an operation that will be applied to each element of the `cvector`. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

Time Complexity: `O(n)`

Parameters

v	the vector.
fn	the callback function to apply to each element.
context	user-defined data that will be applied to the callback. Can be `NULL` if unused.

See Also

csc_foreach

4.8.3.9 CSCError csc_cvector_reserve (cvector * *v*, size_t *num_elems*)

reserves memory for the specified number of elements in the vector.

This functions reserves enough memory in the vector such that it is able to hold *at least* `num_elems` without needing to expand. If the number of elements that will be contained in the vector is known or can be estimated, you may be able to improve the performance of your application by allocating the memory for the elements up front using this function. As always with performance, your milage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is $<$ `csc_cvector_size(v)`, that is an error.

All parameters are expected to be **non-null**.

Time Complexity: OS-specific.

Parameters

v	the vector.
num_elems	the number of elements to allocate memory for.

Returns

On success `CSCError::E_NOERR`. If the requested size is less than the current size, `CSCError::E_INVALIDOPERATION`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.8.3.10 void* csc_cvector_rm (cvector * *v*, const void * *elem*, csc_compare *cmp*)

removes an element from the vector.

This function removes `elem` from the supplied vector if it exists. In order to remove the element, the function must search for the element in the vector using the supplied `cmp` function.

All three parameters are expected to be **non-null**.

Time Complexity: `O(1)` best case, `O(n)` average and worst case.

Parameters

v	the vector.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

See Also

csc_compare
csc_cvector_find

4.8.3.11 void* csc_cvector_rm_at (cvector * *v*, size_t *idx*)

removes the element at the specified 0-indexed index from the vector.

This function the element at index `idx` from `v`.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.
idx	the index.

Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

4.8.3.12 CSCError csc_cvector_shrink_to_fit (cvector * *v*)

shrinks the capacity to match the size of the vector.

After a call to this function, the following will be true:

```
csc_cvector_size(v) == csc_cvector_capacity(v);
```

This function may be useful in low-memory settings where the vector's capacity greatly exceeds the size and the extra memory won't be required.

All parameters are expected to be **non-null**.

Time Complexity: OS-specific.

Parameters

v	the vector.

Returns

On success `CSCError::E_NOERR`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.8.3.13 size_t csc_cvector_size (const cvector * v)

returns the size of the vector.

This function returns the number of elements currently in the vector.

All parameters are expected to be **non-null**.

Time Complexity: `O(1)`

Parameters

v	the vector.

Returns

the size of the vector.

Index