c-rbtree

C-Util Community

Oct 26, 2022

LIBRARY DOCUMENTATION

The **c-rbtree** project provides a Red-Black-Tree API, that is fully implemented in ISO-C11 and has no external dependencies. Furthermore, tree traversal, memory allocations, and key comparisons are completely controlled by the API user. The implementation only provides the RB-Tree specific rebalancing and coloring.

CHAPTER

ONE

API

The `c-rbtree.h` header exposes the full API of the c-rbtree library. It provides access to the Red-Black Tree structure as well as helper functions to perform standard tree operations.

A tree is represented by the *CRBTree* structure. It contains a *single* field, which is a pointer to the root node. If NULL, the tree is empty. If non-NULL, there is at least a single element in the tree.

Each node of the tree is represented by the *CRBNode* structure. It has three fields. The `left` and `right` members can be accessed by the API user directly to traverse the tree. The third member is a combination of the parent pointer and a set of flags. API users are required to embed the *CRBNode* object into their own objects and then use `offsetof()` (i.e., `container_of()` and friends) to turn *CRBNode* pointers into pointers to their own enclosing type:

1.1 Tree Structure

The tree structure of c-rbtree is directly exposed in its API. Callers are allowed to access the node and tree structures directly to traverse the tree. Tree modifications, however, should be performed via the functions provided by the library.

struct **CRBNode**

Node of a Red-Black Tree

Each node in an RB-Tree must embed a *CRBNode* object. This object contains pointers to its left and right child, which can be freely accessed by the API user at any time. They are NULL, if the node does not have a left/right child.

The `__parent_and_flags` field must never be accessed directly. It encodes the pointer to the parent node, and the color of the node. Use the accessor functions instead.

There is no reason to initialize a *CRBNode* object before linking it. However, if you need a boolean state that tells you whether the node is linked or not, you should initialize the node via *c_rbnode_init()* or *C_RBNODE_INIT()*.

union **[anonymous]**

Anonymous union for alignment guarantees

unsigned long **__parent_and_flags**

Internal state encoding the parent pointer and state

CRBNode ***left**

Left child, or NULL

CRBNode ***right**

Right child, or NULL

C_RBNODE_INIT(_var)

Initialize RBNode Object

Parameters

- **_var** – Backpointer to the variable

Set the contents of the specified node to its unlinked, unused state, ready to be linked into a tree.

Returns
Evaluates to the initializer for *_var*.

struct **CRBTree**

Red-Black Tree Top-Level Structure

Each Red-Black Tree is rooted in an CRBTree object. This object contains a pointer to the root node of the tree. The API user is free to access the `root` member at any time, and use it to traverse the tree.

To initialize an RB-Tree, set it to NULL / all zero.

union **[anonymous]**

Anonymous union for alignment guarantees

CRBNode ***root**

Pointer to the root node, or NULL

C_RBTREE_INIT

Initialize RBTree Object

Set the contents of the specified tree to its pristine, empty state.

Returns
Evaluates to the initializer for a *CRBTree* object.

void **c_rbnode_init**(*CRBNode* *n)

Mark a node as unlinked

Parameters

- **n** – Node to operate on

This marks the node `n` as unlinked. The node will be set to a valid state that can never happen if the node is linked in a tree. Furthermore, this state is fully known to the implementation, and as such handled gracefully in all cases.

You are *NOT* required to call this on your node. *c_rbtree_add()* ca handle uninitialized nodes just fine. However, calling this allows to use *c_rbnode_is_linked()* to check for the state of a node. Furthermore, iterators and accessors can be called on initialized (yet unlinked) nodes.

Use the *C_RBNODE_INIT* macro if you want to initialize static variables.

c_rbnode_entry(_what, _t, _m)

Get parent container of tree node

Parameters

- **_what** – Tree node, or NULL
- **_t** – Type of parent container
- **_m** – Member name of tree node in `_t`

If the tree node `_what` is embedded into a surrounding structure, this will turn the tree node pointer `_what` into a pointer to the parent container (using `offsetof(3)`, or sometimes called `container_of(3)`).

If `_what` is NULL, this will also return NULL.

Returns
Pointer to parent container, or NULL.

CRBNode ***c_rbnode_parent**(*CRBNode* *n)

Return parent pointer

Parameters

- **n** – Node to access

This returns a pointer to the parent of the given node `n`. If `n` does not have a parent, NULL is returned. If `n` is not linked, `n` itself is returned.

You should not call this on unlinked or uninitialized nodes! If you do, you better know its semantics.

Returns
Pointer to parent.

_Bool **c_rbnode_is_linked**(*CRBNode* *n)

Check whether a node is linked

Parameters

- **n** – Node to check, or NULL

This checks whether the passed node is linked. If you pass NULL, or if the node is not linked into a tree, this will return false. Otherwise, this returns true.

Note that you must have either linked the node or initialized it, before calling this function. Never call this function on uninitialized nodes. Furthermore, removing a node via *c_rbnode_unlink_stale()* does *NOT* mark the node as unlinked. You have to call *c_rbnode_init()* yourself after removal, or use *c_rbnode_unlink()*.

Returns
true if the node is linked, false if not.

void **c_rbnode_unlink**(*CRBNode* *n)

Safely remove node from tree and reinitialize it

Parameters

- **n** – Node to remove, or NULL

This is almost the same as *c_rbnode_unlink_stale()*, but extends it slightly, to be more convenient to use in many cases:

- If `n` is unlinked or NULL, this is a no-op.
- `n` is reinitialized after being removed.

void **c_rbtree_init**(*CRBTree* *t)

Initialize a new RB-Tree

Parameters

- **t** – Tree to operate on

This initializes a new, empty RB-Tree. An RB-Tree must be initialized before any other functions are called on it. Alternatively, you can zero its memory or assign *C_RBTREE_INIT*.

_Bool **c_rbtree_is_empty**(*CRBTree* *t)

Check whether an RB-tree is empty

Parameters

- **t** – Tree to operate on

This checks whether the passed RB-Tree is empty.

Returns
True if tree is empty, false otherwise.

1.2 Search

While the API supports direct traversal via the open-coded structures, it can be cumbersome to use at times. If you, instead, provide a callback to compare entries in the tree, you can use the following helpers to search the tree for specific entries, or slots to insert new entries.

type **CRBCompareFunc**

Function type to compare a node to a key

If you use the tree-traversal helpers (which are optional), you need to provide this callback so they can compare nodes in a tree to the key you look for.

The tree is provided as optional context `t` to this callback. The key you look for is provided as `k`, the current node that should be compared to is provided as `n`. This function should work like `strcmp()`, that is, return <0 if `key` orders before `n`, 0 if both compare equal, and >0 if it orders after `n`.

CRBNode ***c_rbtree_find_node**(*CRBTree* *t, *CRBCompareFunc* f, const void *k)

Find node

Parameters

- **t** – Tree to search through
- **f** – Comparison function
- **k** – Key to search for

This searches through `t` for a node that compares equal to `k`. The function `f` must be provided by the caller, which is used to compare nodes to `k`. See the documentation of *CRBCompareFunc* for details.

If there are multiple entries that compare equal to `k`, this will return a pseudo-randomly picked node. If you need stable lookup functions for trees where duplicate entries are allowed, you better code your own lookup.

Returns
Pointer to matching node, or NULL.

c_rbtree_find_entry(_t, _f, _k, _s, _m)

Find entry

Parameters

- **_t** – Tree to search through
- **_f** – Comparison function
- **_k** – Key to search for
- **_s** – Type of the structure that embeds the nodes
- **_m** – Name of the node-member in type @_t

This is very similar to *c_rbtree_find_node()*, but instead of returning a pointer to the *CRBNode*, it returns a pointer to the surrounding object. This object must embed the *CRBNode* object. The type of the surrounding object must be given as `_s`, and the name of the embedded *CRBNode* member as `_m`.

See *c_rbtree_find_node()* and *c_rbnode_entry()* for more details.

Returns
Pointer to found entry, NULL if not found.

CRBNode ****c_rbtree_find_slot**(*CRBTree* *t, *CRBCompareFunc* f, const void *k, *CRBNode* **p)

Find slot to insert new node

Parameters

- **t** – Tree to search through
- **f** – Comparison function
- **k** – Key to search for
- **p** – Output storage for parent pointer

This searches through `t` just like *c_rbtree_find_node()* does. However, instead of returning a pointer to a node that compares equal to `k`, this searches for a slot to insert a node with key `k`. A pointer to the slot is returned, and a pointer to the parent of the slot is stored in `p`. Both can be passed directly to *c_rbtree_add()*, together with your node to insert.

If there already is a node in the tree, that compares equal to `k`, this will return NULL and store the conflicting node in `p`. In all other cases, this will return a pointer (non-NULL) to the empty slot to insert the node at. `p` will point to the parent node of that slot.

If you want trees that allow duplicate nodes, you better code your own insertion function.

Returns
Pointer to slot to insert node, or NULL on conflicts.

1.3 Iterators

The `c_rbtree_for_each*()` macros provide simple for-loop wrappers to iterate an RB-Tree. They come in a set of flavours:

entry
This combines *c_rbnode_entry()* with the loop iterator, so the iterator always has the type of the surrounding object, rather than *CRBNode*.

safe
The loop iterator always keeps track of the next element to visit. This means, you can safely modify the current element, while retaining loop-integrity. You still must not touch any other entry of the tree. Otherwise, the loop-iterator will be corrupted. Also remember to only modify the tree in a way compatible with your iterator-order. That is, if you use in-order iteration (default), you can unlink your current object, including re-balancing the tree. However, if you use post-order, you must not trigger a tree rebalance operation, since it is not an invariant of post-order iteration.

postorder
Rather than the default in-order iteration, this iterates the tree in post-order.

unlink
This unlinks the current element from the tree before the loop code is run. Note that the tree is not rebalanced. That is, you must never break out of the loop. If you do so, the tree is corrupted.

1.4 Traversal

If you prefer open-coding the tree traversal over the built-in iterators, c-rbtree provides a set of helpers to find starting position and end nodes for different kind of tree traversals.

CRBNode ***c_rbnode_leftmost**(*CRBNode* *n)

Return leftmost child

Parameters

- **n** – Current node, or NULL

This returns the leftmost child of `n`. If `n` is NULL, this will return NULL. In all other cases, this function returns a valid pointer. That is, if `n` does not have any left children, this returns `n`.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to leftmost child, or NULL.

CRBNode ***c_rbnode_rightmost**(*CRBNode* *n)

Return rightmost child

Parameters

- **n** – Current node, or NULL

This returns the rightmost child of `n`. If `n` is NULL, this will return NULL. In all other cases, this function returns a valid pointer. That is, if `n` does not have any right children, this returns `n`.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to rightmost child, or NULL.

CRBNode ***c_rbnode_leftdeepest**(*CRBNode* *n)

Return left-deepest child

Parameters

- **n** – Current node, or NULL

This returns the left-deepest child of `n`. If `n` is NULL, this will return NULL. In all other cases, this function returns a valid pointer. That is, if `n` does not have any children, this returns `n`.

The left-deepest child is defined as the deepest child without any left (grand-...)siblings.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to left-deepest child, or NULL.

CRBNode ***c_rbnode_rightdeepest**(*CRBNode* *n)

Return right-deepest child

Parameters

- **n** – Current node, or NULL

This returns the right-deepest child of `n`. If `n` is NULL, this will return NULL. In all other cases, this function returns a valid pointer. That is, if `n` does not have any children, this returns `n`.

The right-deepest child is defined as the deepest child without any right (grand-...)siblings.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to right-deepest child, or NULL.

CRBNode *c_rbnode_next(*CRBNode* *n)

Return next node

Parameters

- **n** – Current node, or NULL

An RB-Tree always defines a linear order of its elements. This function returns the logically next node to n. If n is NULL, the last node or unlinked, this returns NULL.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to next node, or NULL.

CRBNode *c_rbnode_prev(*CRBNode* *n)

Return previous node

Parameters

- **n** – Current node, or NULL

An RB-Tree always defines a linear order of its elements. This function returns the logically previous node to n. If n is NULL, the first node or unlinked, this returns NULL.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to previous node, or NULL.

CRBNode *c_rbnode_next_postorder(*CRBNode* *n)

Return next node in post-order

Parameters

- **n** – Current node, or NULL

This returns the next node to n, based on a left-to-right post-order traversal. If n is NULL, the root node, or unlinked, this returns NULL.

This implements a left-to-right post-order traversal: First visit the left child of a node, then the right, and lastly the node itself. Children are traversed recursively.

This function can be used to implement a left-to-right post-order traversal:

```
for (n = c_rbtree_first_postorder(t); n; n = c_rbnode_next_postorder(n))
        visit(n);
```

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to next node, or NULL.

CRBNode *c_rbnode_prev_postorder(*CRBNode* *n)

Return previous node in post-order

Parameters

- **n** – Current node, or NULL

This returns the previous node to `n`, based on a left-to-right post-order traversal. That is, it is the inverse operation to *c_rbnode_next_postorder()*. If `n` is NULL, the left-deepest node, or unlinked, this returns NULL.

This function returns the logical previous node in a directed post-order traversal. That is, it effectively does a pre-order traversal (since a reverse post-order traversal is a pre-order traversal). This function does NOT do a right-to-left post-order traversal! In other words, the following invariant is guaranteed, if `c_rbnode_next_postorder(n)` is non-NULL:

```
n == c_rbnode_prev_postorder(c_rbnode_next_postorder(n))
```

This function can be used to implement a right-to-left pre-order traversal, using the fact that a reverse post-order traversal is also a valid pre-order traversal:

```
for (n = c_rbtree_last_postorder(t); n; n = c_rbnode_prev_postorder(n))
        visit(n);
```

This would effectively perform a right-to-left pre-order traversal: first visit a parent, then its right child, then its left child. Both children are traversed recursively.

Worst case runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to previous node in post-order, or NULL.

CRBNode ***c_rbtree_first**(*CRBTree* *t)

Return first node

Parameters

- **t** – Tree to operate on

An RB-Tree always defines a linear order of its elements. This function returns the logically first node in `t`. If `t` is empty, NULL is returned.

Fixed runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to first node, or NULL.

CRBNode ***c_rbtree_last**(*CRBTree* *t)

Return last node

Parameters

- **t** – Tree to operate on

An RB-Tree always defines a linear order of its elements. This function returns the logically last node in `t`. If `t` is empty, NULL is returned.

Fixed runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to last node, or NULL.

CRBNode ***c_rbtree_first_postorder**(*CRBTree* *t)

Return first node in post-order

Parameters

- **t** – Tree to operate on

This returns the first node of a left-to-right post-order traversal. That is, it returns the left-deepest leaf. If the tree is empty, this returns NULL.

This can also be interpreted as the last node of a right-to-left pre-order traversal.

Fixed runtime (n: number of elements in tree): O(log(n))

Returns
Pointer to first node in post-order, or NULL.

CRBNode ***c_rbtree_last_postorder**(*CRBTree* *t)

Return last node in post-order

Parameters

- **t** – Tree to operate on

This returns the last node of a left-to-right post-order traversal. That is, it always returns the root node, or NULL if the tree is empty.

This can also be interpreted as the first node of a right-to-left pre-order traversal.

Fixed runtime (n: number of elements in tree): O(1)

Returns
Pointer to last node in post-order, or NULL.

1.5 Tree Modification

Insertion into and removal from an RB-Tree require rebalancing to make sure the tree stays balanced. The following functions ensure the tree integrity is kept.

void **c_rbtree_move**(*CRBTree* *to, *CRBTree* *from)

Move tree

Parameters

- **to** – Destination tree
- **from** – Source tree

This imports the entire tree from `from` into `to`. `to` must be empty! `from` will be empty afterwards.

Note that this operates in O(1) time. Only the root-entry is updated to point to the new tree-root.

void **c_rbnode_link**(*CRBNode* *p, *CRBNode* **l, *CRBNode* *n)

Link node into tree

Parameters

- **p** – Parent node to link under
- **l** – Left/right slot of `p` to link at
- **n** – Node to add

This links `n` into an tree underneath another node. The caller must provide the exact spot where to link the node. That is, the caller must traverse the tree based on their search order. Once they hit a leaf where to insert the node, call this function to link it and rebalance the tree.

For this to work, the caller must provide a pointer to the parent node. If the tree might be empty, you must resort to *c_rbtree_add()*.

In most cases you are better off using *c_rbtree_add()*. See there for details how tree-insertion works.

void **c_rbtree_add**(*CRBTree* *t, *CRBNode* *p, *CRBNode* **l, *CRBNode* *n)

Add node to tree

Parameters

- **t** – Tree to operate one
- **p** – Parent node to link under, or NULL
- **l** – Left/right slot of @p (or root) to link at
- **n** – Node to add

This links @n into the tree given as `t`. The caller must provide the exact spot where to link the node. That is, the caller must traverse the tree based on their search order. Once they hit a leaf where to insert the node, call this function to link it and rebalance the tree.

A typical insertion would look like this (`t` is your tree, `n` is your node):

```
CRBNode **i, *p;

i = &t->root;
p = NULL;
while (*i) {
        p = *i;
        if (compare(n, *i) < 0)
                i = &(*i)->left;
        else
                i = &(*i)->right;
}

c_rbtree_add(t, p, i, n);
```

Once the node is linked into the tree, a simple lookup on the same tree can be coded like this:

```
CRBNode *i;

i = t->root;
while (i) {
        int v = compare(n, i);
        if (v < 0)
                i = (*i)->left;
        else if (v > 0)
                i = (*i)->right;
        else
                break;
}
```

When you add nodes to a tree, the memory contents of the node do not matter. That is, there is no need to initialize the node via *c_rbnode_init()*. However, if you relink nodes multiple times during their lifetime, it is usually very convenient to use *c_rbnode_init()* and *c_rbnode_unlink()* (rather than *c_rbnode_unlink_stale()*). In those cases, you should validate that a node is unlinked before you call *c_rbtree_add()*.

void **c_rbnode_unlink_stale**(*CRBNode* *n)

Remove node from tree

Parameters

- **n** – Node to remove

This removes the given node from its tree. Once unlinked, the tree is rebalanced.

This does *NOT* reset `n` to being unlinked. If you need this, use `c_rbtree_unlink()`.

INDEX

C