NAME

rb3ptr − Generic Red-black tree implementation and macros for typesafe access

SYNOPSIS

The rb3ptr API is split into multiple sections.

DATATYPES contains essential structures and definitions.

BASIC FUNCTIONS has operations on the generic Red-black tree implementation that provide an ordered container API.

NAVIGATION FUNCTIONS contains additional functionality for navigation in a binary search tree.

MACROS explains the type-specific wrappers for better type-safety and more convenient access with fixed comparison functions.

BSD MACROS lists functionality to emulate parts of the BSD <*sys/tree.h*> API.

DATATYPES

struct rb3_tree;

struct rb3_head;

typedef int (*rb3_cmp)(struct rb3_head ****head*, void ****data***);**

BASIC FUNCTIONS

void rb3_reset_tree(struct rb3_tree ****tree*);**

int rb3_isempty(struct rb3_tree ****tree*);**

struct rb3_head *rb3_get_min(struct rb3_tree ****tree*);**

struct rb3_head *rb3_get_max(struct rb3_tree ****tree*);**

struct rb3_head *rb3_get_prev(struct rb3_head ****head*);**

struct rb3_head *rb3_get_next(struct rb3_head ****head*);**

struct rb3_head *rb3_get_minmax(struct rb3_tree ****tree*, int** ***dir***);**

struct rb3_head *rb3_get_prevnext(struct rb3_head ****head*, int** ***dir***);**

struct rb3_head *rb3_find(struct rb3_tree ****tree*, struct rb3_cmp** ***cmp***, void *****data***);**

struct rb3_head *rb3_find_parent(struct rb3_tree ****tree*, struct rb3_cmp** ***cmp***, void *****data***, struct rb3_head ****parent_out***, int *****dir_out***);**

void rb3_link_and_rebalance(struct rb3_head ****head*, struct rb3_head *** ***parent***, int** ***dir***);**

void rb3_unlink_and_rebalance(struct rb3_head ****head*);**

void rb3_replace(struct rb3_head ****head*, struct rb3_head *** ***newhead***);**

struct rb3_head *rb3_insert(struct rb3_tree ****tree*, struct rb3_head *** ***head***, struct rb3_cmp** ***cmp***, void *****data***);**

struct rb3_head *rb3_delete(struct rb3_tree ****tree*, struct rb3_cmp** ***cmp***, void *****data***);**

NAVIGATION FUNCTIONS

Note: Valid values for *dir* are **RB3_LEFT** and **RB3_RIGHT** (0 and 1).

struct rb3_head *rb3_get_root(struct rb3_tree ****tree*);**

struct rb3_head *rb3_get_base(struct rb3_tree *_tree_);

int rb3_is_base(struct rb3_head *_head_);

int rb3_is_node_linked(struct rb3_head *_head_);

int rb3_get_parent_dir(struct rb3_head *_head_);

int rb3_has_child(struct rb3_head *_head_, int _dir_);

struct rb3_head *rb3_get_parent(struct rb3_head *_head_);

struct rb3_head *rb3_get_child(struct rb3_head *_head_, int _dir_);

struct rb3_head *rb3_get_prev_ancestor(struct rb3_head *_head_);

struct rb3_head *rb3_get_next_ancestor(struct rb3_head *_head_);

struct rb3_head *rb3_get_prev_descendant(struct rb3_head *_head_);

struct rb3_head *rb3_get_next_descendant(struct rb3_head *_head_);

struct rb3_head *rb3_get_prevnext_ancestor(struct rb3_head *_head_, int _dir_);

struct rb3_head *rb3_get_prevnext_descendant(struct rb3_head *_head_, int _dir_);

MACROS

RB3_GEN_IMPL()

RB3_GEN_IMPL_STATIC()

RB3_GEN_INLINE(*NAME*, *NODETYPE*, *GET_HEAD*, *GET_NODE***)**

RB3_GEN_NODECMP(*NAME*, *NODETYPE*, *GET_HEAD*, *GET_NODE*, *NODECMP***)**

RB3_FOREACH(*NAME*, *TREE*, *NODE***);**

RB3_FOREACH_REVERSE(*NAME*, *TREE*, *NODE***);**

RB3_FOREACH_DIR(*NAME*, *TREE*, *NODE***);**

RB3_FOREACH_SAFE(*NAME*, *TREE*, *NODE*, *TMPNODE***);**

RB3_FOREACH_REVERSE_SAFE(*NAME*, *TREE*, *NODE*, *TMPNODE***);**

RB3_FOREACH_DIR_SAFE(*NAME*, *TREE*, *NODE*, *TMPNODE***);**

BSD MACROS

These MACROS expose a BSD <*sys/tree.h*> compatible interface. Unfortunately, **RB_LEFT**() and **RB_RIGHT**() cannot be supported due to missing information in the signature.

RB_PROTOTYPE()

RB_PROTOTYPE_STATIC()

RB_GENERATE()

RB_GENERATE_STATIC()

RB_INIT(*tree***)**

RB_INSERT(*NAME*, *tree*, *elm***)**

RB_FIND(*NAME*, *tree*, *elm***)**

RB_REMOVE(*NAME*, *tree*, *elm***)**

RB_MIN(*NAME*, *tree***)**

RB_MAX(*NAME*, *tree***)**

RB_PREV(*NAME***,** *tree***,** *elm***)**

RB_NEXT(*NAME***,** *tree***,** *elm***)**

DOCUMENTATION

This section contains explanations for the structures and prototypes listed above.

DATATYPES

struct rb3_tree is the basic tree type. It holds the root link for one red-black tree in a running program.

struct rb3_head is the linking information for a node in the tree. Data that should be linked in a tree must contain such a structure. The tree implementation does not care about the actual data, but simply maintains the links between the link structures.

rb3_cmp is the function type of comparisons to direct tree searches. At each visited node, the function is called with the node and a user-provided data as arguments. It should return an integer less than, equal to, or greater than 0, indicating whether the node in the tree compares less than, equal to, or greater than the user-provided data. This function is always user-provided. Typically it will use **offsetof**(3) or the linux **container_of**() macro to get at the actual data in which the **struct rb3_head** node is embedded.

BASIC FUNCTIONS

rb3_reset_tree() initializes a **struct rb3_tree** for subsequent use. Note that zeroing the structure (e.g., with **memset()** or static initialization) will **not** do the work. There are no resources allocated, so there is no matching "destructor" routine.

rb3_isempty() tests if a tree does not contain any nodes. This of course is true after initialization.

rb3_get_min() and **rb3_get_max**() return the leftmost / rightmost element linked in a tree. If the tree is empty, NULL is returned.

rb3_get_prev() and **rb3_get_next**() return the previous / next node linked in the same tree (with respect to in-order traversal). If no such node exists, NULL is returned.

rb3_get_minmax() and **rb3_get_prevnext**() can be used instead of **rb3_get_min**(), **rb3_get_max**(), **rb3_get_prev**(), and **rb3_next**(). They take the direction as runtime parameter (**RB3_LEFT** or **RB3_RIGHT**).

rb3_find() finds a node in a tree. If no node comparing equal (i.e., the comparison function returns 0 given the visited node and the user-provided data) is found in the tree, NULL is returned.

rb3_find_parent() is similar to **rb3_find**(), but when the search is unsuccessful, the appropriate insertion point for a node matching the search is returned in the out-arguments. **rb3_link_and_rebalance**() can then be used to add the node. (**rb3_insert**() combines these two operations in a single function call).

rb3_link_and_rebalance() can be used to link a given node into a tree given an insertion point (parent node and its child direction). The appropriate insertion point can be found using **rb3_find_parent**().

rb3_unlink_and_rebalance() can be used to unlink a given node from a tree without any search. The node must be known to be linked in a tree.

rb3_replace() unlinks a node and puts another one in its place. This operation is constant-time; no

rebalancing is required.

rb3_insert() can be used to insert a new node into a tree at a suitable insertion point. It takes a tree, the new node to insert, and a **rb3_cmp** function implementing the node ordering to direct the search. If a node comparing equal (i.e., the comparison function returns 0 given the visited node and the user-provided node) is found in the tree, that node is returned. Otherwise, the to-be-inserted node is linked into the tree and NULL is returned.

rb3_delete() does a node search in a tree given a comparison function and data. If a matching node is found, it is unlinked from the tree and a pointer to it is returned. Otherwise, NULL is returned. Note that the node is not cleared (zeroed), so if you want .BR rb3_is_node_linked () to work after the function returns, you should clear the node manually.

NAVIGATION FUNCTIONS

rb3_get_root() returns the root node in the tree, or NULL if the tree is empty.

rb3_get_base() returns the base head of the tree, which always exists. If the tree is nonempty, the root node is linked as left child of the base node. This is an implementation detail and need not be relied upon in most situations.

rb3_is_base() tests whether a link structure is the base node in a tree. This only can distinguish the base node of a tree that was initialized with **rb3_reset_tree**(), from non-base nodes that are cleared (zeroed) or properly linked in a tree.

rb3_is_node_linked() tests whether the given non-base node is linked in a (any) tree. This can only distinguish nodes that are properly linked in a tree from unlinked (zeroed) nodes.

rb3_get_parent_dir() returns RB3_LEFT or RB3_RIGHT depending on whether the given link node is the left or right child of its parent. This is a single bitwise operation on the link structure, so is more efficient than testing both childs of the parent's link structure.

rb3_has_child() tests whether the given link has a child in the given direction.

rb3_get_parent() returns the parent link structure of the given node. If the given node is the root node, the base head is returned. If this is not what you want, test if the return value has itself a parent. (The base head is the only head that has no parent).

rb3_get_child() returns the left or right child of the given node, depending on the given direction value (**RB3_LEFT** or **RB3_RIGHT**)

rb3_get_prev_ancestor() returns the nearest left ancestor of the given head link structure. If none exists, NULL is returned.

rb3_get_next_ancestor() returns the nearest right ancestor of the given head link structure. If none exists, NULL is returned.

rb3_get_prev_descendant() returns the nearest left descendant of the given head link structure. If none exists, NULL is returned.

rb3_get_next_descendant() returns the nearest right descendant of the given head link structure. If none exists, NULL is returned.

rb3_get_prevnext_ancestor() returns the nearest left or right ancestor (depending on the given direction) of the given head link structure. If none exists, NULL is returned.

rb3_get_prevnext_descendant() returns the nearest left or right descendant (depending on the given direction) of the given head link structure. If none exists, NULL is returned.

MACROS

RB3_GEN_IMPL() evaluates to a complete implementation of the rb3ptr API with *extern* linkage. Use this only if you can't use a separately compiled rb3ptr libray. Macros are hard to debug.

RB3_GEN_IMPL_STATIC() evaluates to a complete implementation of the rb3ptr API with *static* linkage. Use this only if no other file in the same projects need rb3ptr's functionality.

RB3_GEN_INLINE() evaluates to an implementation of the non-comparison-related functionality of rb3ptr wrapped for a specific datatype. *NAME* should be a prefix for these functions, such as for example *footree*. *NODETYPE* should be the node type managed by this set of generated functions, such as for example *struct* foo (see the example below). *GET_HEAD and GET_NODE* should be macros or functions for the generated implementation's use to retrieve the embedded link structure from a node, or vice versa.

RB3_GEN_NODECMP() TODO

RB3_FOREACH() is a for-loop iteration macro. *NAME* should be the prefix used in *RB3_GEN_INLINE*(). *TREE* should be a tree of the generated type (**struct NAME**). *NODE* should by a value of type *NODETYPE* *. It is used as iteration variable.

RB3_FOREACH_REVERSE() **RB3_FOREACH_DIR**() **RB3_FOREACH_SAFE**() **RB3_FOREACH_REVERSE_SAFE**() **RB3_FOREACH_DIR_SAFE**() TODO

BSD MACROS

For documentation of the BSD macros please refer to **tree**(3)

EXAMPLE

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <rb3ptr.h>

/*
 * Include the generic implementation. Alternatively, you can link with a
 * seperately compiled generic implementation
 */
RB3_GEN_IMPL_STATIC();

/*
 * Define a node datatype and a compare operation
 */
struct foo {
	/* the node type must include a struct rb3_head. */
	struct rb3_head head;
	int val;
};
```

```
int foo_compare(struct foo *a, struct foo *b)
{
	return (a−>val > b−>val) − (a−>val < b−>val);
}

struct rb3_head *get_head(struct foo *foo)
{
	return &foo−>head;
}

struct foo *get_foo(struct rb3_head *head)
{
	return (struct foo *)((char *) head − offsetof(struct foo, head));
}

RB3_GEN_TREE_DEFINITION(footree);
RB3_GEN_INLINE_PROTO_STATIC(footree, struct foo, get_head, get_foo);
RB3_GEN_NODECMP_PROTO_STATIC(footree, /* no suffix for these compare functions */, struct foo, get_head, get_f
RB3_GEN_NODECMP_STATIC(footree, /* no suffix for these compare functions */, struct foo, get_head, get_foo, foo_c

void testoperations(void)
{
	struct footree tree;
	struct foo *iter;
	struct foo foo[42];
	size_t i;

	footree_reset_tree(&tree);
	for (i = 0; i < 42; i++)
		foo[i].val = rand();
	for (i = 0; i < 42; i++)
		footree_insert(&tree, &foo[i]);
	for (iter = footree_get_min(&tree); iter != NULL; iter = footree_get_next(iter))
		printf("iter %d0, iter−>val);
	for (i = 0; i < 42; i++)
		footree_delete(&tree, &foo[i]);
}
```