NAME

SPLAY_PROTOTYPE, **SPLAY_GENERATE**, **SPLAY_ENTRY**, **SPLAY_HEAD**, **SPLAY_INITIALIZER**, **SPLAY_ROOT**, **SPLAY_EMPTY**, **SPLAY_NEXT**, **SPLAY_MIN**, **SPLAY_MAX**, **SPLAY_FIND**, **SPLAY_LEFT**, **SPLAY_RIGHT**, **SPLAY_FOREACH**, **SPLAY_INIT**, **SPLAY_INSERT**, **SPLAY_REMOVE**, **RB_PROTOTYPE**, **RB_PROTOTYPE_STATIC**, **RB_PROTOTYPE_INSERT**, **RB_PROTOTYPE_INSERT_COLOR**, **RB_PROTOTYPE_REMOVE**, **RB_PROTOTYPE_REMOVE_COLOR**, **RB_PROTOTYPE_FIND**, **RB_PROTOTYPE_NFIND**, **RB_PROTOTYPE_NEXT**, **RB_PROTOTYPE_PREV**, **RB_PROTOTYPE_MINMAX**, **RB_PROTOTYPE_REINSERT**, **RB_GENERATE**, **RB_GENERATE_STATIC**, **RB_GENERATE_INSERT**, **RB_GENERATE_INSERT_COLOR**, **RB_GENERATE_REMOVE**, **RB_GENERATE_REMOVE_COLOR**, **RB_GENERATE_FIND**, **RB_GENERATE_NFIND**, **RB_GENERATE_NEXT**, **RB_GENERATE_PREV**, **RB_GENERATE_MINMAX**, **RB_GENERATE_REINSERT**, **RB_ENTRY**, **RB_HEAD**, **RB_INITIALIZER**, **RB_ROOT**, **RB_EMPTY**, **RB_NEXT**, **RB_PREV**, **RB_MIN**, **RB_MAX**, **RB_FIND**, **RB_NFIND**, **RB_LEFT**, **RB_RIGHT**, **RB_PARENT**, **RB_FOREACH**, **RB_FOREACH_FROM**, **RB_FOREACH_SAFE**, **RB_FOREACH_REVERSE**, **RB_FOREACH_REVERSE_FROM**, **RB_FOREACH_REVERSE_SAFE**, **RB_INIT**, **RB_INSERT**, **RB_INSERT_NEXT**, **RB_INSERT_PREV**, **RB_REMOVE**, **RB_REINSERT**, **RB_AUGMENT RB_AUGMENT_CHECK, RB_UPDATE_AUGMENT** - implementations of splay and rank-balanced (wavl) trees

SYNOPSIS

#include <sys/tree.h>

SPLAY_PROTOTYPE(*NAME*, *TYPE*, *FIELD*, *CMP*);

SPLAY_GENERATE(*NAME*, *TYPE*, *FIELD*, *CMP*);

SPLAY_ENTRY(*TYPE*);

SPLAY_HEAD(*HEADNAME*, *TYPE*);

*struct TYPE **
SPLAY_INITIALIZER(*SPLAY_HEAD *head*);

SPLAY_ROOT(*SPLAY_HEAD *head*);

bool
SPLAY_EMPTY(*SPLAY_HEAD *head*);

*struct TYPE **

SPLAY_NEXT(*NAME, SPLAY_HEAD *head, struct TYPE *elm*);

*struct TYPE **
SPLAY_MIN(*NAME, SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_MAX(*NAME, SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_FIND(*NAME, SPLAY_HEAD *head, struct TYPE *elm*);

*struct TYPE **
SPLAY_LEFT(*struct TYPE *elm, SPLAY_ENTRY NAME*);

*struct TYPE **
SPLAY_RIGHT(*struct TYPE *elm, SPLAY_ENTRY NAME*);

SPLAY_FOREACH(*VARNAME, NAME, SPLAY_HEAD *head*);

void
SPLAY_INIT(*SPLAY_HEAD *head*);

*struct TYPE **
SPLAY_INSERT(*NAME, SPLAY_HEAD *head, struct TYPE *elm*);

*struct TYPE **
SPLAY_REMOVE(*NAME, SPLAY_HEAD *head, struct TYPE *elm*);

RB_PROTOTYPE(*NAME, TYPE, FIELD, CMP*);

RB_PROTOTYPE_STATIC(*NAME, TYPE, FIELD, CMP*);

RB_PROTOTYPE_INSERT(*NAME, TYPE, ATTR*);

RB_PROTOTYPE_INSERT_COLOR(*NAME, TYPE, ATTR*);

RB_PROTOTYPE_REMOVE(*NAME, TYPE, ATTR*);

RB_PROTOTYPE_REMOVE_COLOR(*NAME, TYPE, ATTR*);

RB_PROTOTYPE_FIND(*NAME*, *TYPE*, *ATTR*);

RB_PROTOTYPE_NFIND(*NAME*, *TYPE*, *ATTR*);

RB_PROTOTYPE_NEXT(*NAME*, *TYPE*, *ATTR*);

RB_PROTOTYPE_PREV(*NAME*, *TYPE*, *ATTR*);

RB_PROTOTYPE_MINMAX(*NAME*, *TYPE*, *ATTR*);

RB_PROTOTYPE_REINSERT(*NAME*, *TYPE*, *ATTR*);

RB_GENERATE(*NAME*, *TYPE*, *FIELD*, *CMP*);

RB_GENERATE_STATIC(*NAME*, *TYPE*, *FIELD*, *CMP*);

RB_GENERATE_INSERT(*NAME*, *TYPE*, *FIELD*, *CMP*, *ATTR*);

RB_GENERATE_INSERT_COLOR(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_REMOVE(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_REMOVE_COLOR(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_FIND(*NAME*, *TYPE*, *FIELD*, *CMP*, *ATTR*);

RB_GENERATE_NFIND(*NAME*, *TYPE*, *FIELD*, *CMP*, *ATTR*);

RB_GENERATE_NEXT(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_PREV(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_MINMAX(*NAME*, *TYPE*, *FIELD*, *ATTR*);

RB_GENERATE_REINSERT(*NAME*, *TYPE*, *FIELD*, *CMP*, *ATTR*);

RB_ENTRY(*TYPE*);

RB_HEAD(*HEADNAME*, *TYPE*);

RB_INITIALIZER(*RB_HEAD *head*);

*struct TYPE **
RB_ROOT(*RB_HEAD *head*);

bool
RB_EMPTY(*RB_HEAD *head*);

*struct TYPE **
RB_NEXT(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_PREV(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_MIN(*NAME, RB_HEAD *head*);

*struct TYPE **
RB_MAX(*NAME, RB_HEAD *head*);

*struct TYPE **
RB_FIND(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_NFIND(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_LEFT(*struct TYPE *elm, RB_ENTRY NAME*);

*struct TYPE **
RB_RIGHT(*struct TYPE *elm, RB_ENTRY NAME*);

*struct TYPE **
RB_PARENT(*struct TYPE *elm, RB_ENTRY NAME*);

RB_FOREACH(*VARNAME, NAME, RB_HEAD *head*);

RB_FOREACH_FROM(*VARNAME, NAME, POS_VARNAME*);

RB_FOREACH_SAFE(*VARNAME, NAME, RB_HEAD *head, TEMP_VARNAME*);

RB_FOREACH_REVERSE(*VARNAME, NAME, RB_HEAD *head*);

RB_FOREACH_REVERSE_FROM(*VARNAME, NAME, POS_VARNAME*);

RB_FOREACH_REVERSE_SAFE(*VARNAME, NAME, RB_HEAD *head, TEMP_VARNAME*);

void
RB_INIT(*RB_HEAD *head*);

*struct TYPE **
RB_INSERT(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_INSERT_NEXT(*NAME, RB_HEAD *head, struct TYPE *elm, struct TYPE *next*);

*struct TYPE **
RB_INSERT_PREV(*NAME, RB_HEAD *head, struct TYPE *elm, struct TYPE *prev*);

*struct TYPE **
RB_REMOVE(*NAME, RB_HEAD *head, struct TYPE *elm*);

*struct TYPE **
RB_REINSERT(*NAME, RB_HEAD *head, struct TYPE *elm*);

void
RB_AUGMENT(*NAME, struct TYPE *elm*);

bool
RB_AUGMENT_CHECK(*NAME, struct TYPE *elm*);

void
RB_UPDATE_AUGMENT(*NAME, struct TYPE *elm*);

DESCRIPTION

These macros define data structures for different types of trees: splay trees and rank-balanced (wavl) trees.

In the macro definitions, *TYPE* is the name tag of a user defined structure that must contain a field of type *SPLAY_ENTRY*, or *RB_ENTRY*, named *ENTRYNAME*. The argument *HEADNAME* is the name tag of a user defined structure that must be declared using the macros **SPLAY_HEAD**(), or

RB_HEAD(). The argument *NAME* has to be a unique name prefix for every tree that is defined.

The function prototypes are declared with **SPLAY_PROTOTYPE**(), **RB_PROTOTYPE**(), or **RB_PROTOTYPE_STATIC**(). The function bodies are generated with **SPLAY_GENERATE**(), **RB_GENERATE**(), or **RB_GENERATE_STATIC**(). See the examples below for further explanation of how these macros are used.

SPLAY TREES

A splay tree is a self-organizing data structure. Every operation on the tree causes a splay to happen. The splay moves the requested node to the root of the tree and partly rebalances it.

This has the benefit that request locality causes faster lookups as the requested nodes move to the top of the tree. On the other hand, every lookup causes memory writes.

The Balance Theorem bounds the total access time for *m* operations and *n* inserts on an initially empty tree as **O**(*(m + n)lg n*). The amortized cost for a sequence of *m* accesses to a splay tree is **O**(*lg n*).

A splay tree is headed by a structure defined by the **SPLAY_HEAD**() macro. A structure is declared as follows:

SPLAY_HEAD(*HEADNAME*, *TYPE*) *head*;

where *HEADNAME* is the name of the structure to be defined, and struct *TYPE* is the type of the elements to be inserted into the tree.

The **SPLAY_ENTRY**() macro declares a structure that allows elements to be connected in the tree.

In order to use the functions that manipulate the tree structure, their prototypes need to be declared with the **SPLAY_PROTOTYPE**() macro, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by **SPLAY_ENTRY**().

The function bodies are generated with the **SPLAY_GENERATE**() macro. It takes the same arguments as the **SPLAY_PROTOTYPE**() macro, but should be used only once.

Finally, the *CMP* argument is the name of a function used to compare tree nodes with each other. The function takes two arguments of type *struct TYPE* *. If the first argument is smaller than the second, the function returns a value smaller than zero. If they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The **SPLAY_INIT**() macro initializes the tree referenced by *head*.

The splay tree can also be initialized statically by using the **SPLAY_INITIALIZER**() macro like this:

SPLAY_HEAD(*HEADNAME, TYPE*) *head* = **SPLAY_INITIALIZER**(*&head*);

The **SPLAY_INSERT**() macro inserts the new element *elm* into the tree.

The **SPLAY_REMOVE**() macro removes the element *elm* from the tree pointed by *head*.

The **SPLAY_FIND**() macro can be used to find a particular element in the tree.

```
struct TYPE find, *res;
find.key = 30;
res = SPLAY_FIND(NAME, head, &find);
```

The **SPLAY_ROOT**(), **SPLAY_MIN**(), **SPLAY_MAX**(), and **SPLAY_NEXT**() macros can be used to traverse the tree:

```
for (np = SPLAY_MIN(NAME, &head); np != NULL; np = SPLAY_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the **SPLAY_FOREACH**() macro:

SPLAY_FOREACH(*np, NAME, head*)

The **SPLAY_EMPTY**() macro should be used to check whether a splay tree is empty.

RANK-BALANCED TREES

Rank-balanced (RB) trees are a framework for defining height-balanced binary search trees, including AVL and red-black trees. Each tree node has an associated rank. Balance conditions are expressed by conditions on the differences in rank between any node and its children. Rank differences are stored in each tree node.

The balance conditions implemented by the RB macros lead to weak AVL (wavl) trees, which combine the best aspects of AVL and red-black trees. Wavl trees rebalance after an insertion in the same way AVL trees do, with the same worst-case time as red-black trees offer, and with better balance in the resulting tree. Wavl trees rebalance after a removal in a way that requires less restructuring, in the worst case, than either AVL or red-black trees do. Removals can lead to a tree almost as unbalanced as a red-black tree; insertions lead to a tree becoming as balanced as an AVL tree.

A rank-balanced tree is headed by a structure defined by the **RB_HEAD**() macro. A structure is declared as follows:

RB_HEAD(*HEADNAME*, *TYPE*) *head*;

where *HEADNAME* is the name of the structure to be defined, and struct *TYPE* is the type of the elements to be inserted into the tree.

The **RB_ENTRY**() macro declares a structure that allows elements to be connected in the tree.

In order to use the functions that manipulate the tree structure, their prototypes need to be declared with the **RB_PROTOTYPE**() or **RB_PROTOTYPE_STATIC**() macro, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by **RB_ENTRY**(). Individual prototypes can be declared with **RB_PROTOTYPE_INSERT**(), **RB_PROTOTYPE_INSERT_COLOR**(), **RB_PROTOTYPE_REMOVE**(), **RB_PROTOTYPE_REMOVE_COLOR**(), **RB_PROTOTYPE_FIND**(), **RB_PROTOTYPE_NFIND**(), **RB_PROTOTYPE_NEXT**(), **RB_PROTOTYPE_PREV**(), **RB_PROTOTYPE_MINMAX**(), and **RB_PROTOTYPE_REINSERT**() in case not all functions are required. The individual prototype macros expect *NAME*, *TYPE*, and *ATTR* arguments. The *ATTR* argument must be empty for global functions or *static* for static functions.

The function bodies are generated with the **RB_GENERATE**() or **RB_GENERATE_STATIC**() macro. These macros take the same arguments as the **RB_PROTOTYPE**() and **RB_PROTOTYPE_STATIC**() macros, but should be used only once. As an alternative individual function bodies are generated with the **RB_GENERATE_INSERT**(), **RB_GENERATE_INSERT_COLOR**(), **RB_GENERATE_REMOVE**(), **RB_GENERATE_REMOVE_COLOR**(), **RB_GENERATE_FIND**(), **RB_GENERATE_NFIND**(), **RB_GENERATE_NEXT**(), **RB_GENERATE_PREV**(), **RB_GENERATE_MINMAX**(), and **RB_GENERATE_REINSERT**() macros.

Finally, the *CMP* argument is the name of a function used to compare tree nodes with each other. The function takes two arguments of type *struct TYPE* *. If the first argument is smaller than the second, the function returns a value smaller than zero. If they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The **RB_INIT**() macro initializes the tree referenced by *head*.

The rank-balanced tree can also be initialized statically by using the **RB_INITIALIZER**() macro like this:

RB_HEAD(*HEADNAME*, *TYPE*) *head* = **RB_INITIALIZER**(*&head*);

The **RB_INSERT**() macro inserts the new element *elm* into the tree.

The **RB_INSERT_NEXT**() macro inserts the new element *elm* into the tree immediately after a given element.

The **RB_INSERT_PREV**() macro inserts the new element *elm* into the tree immediately before a given element.

The **RB_REMOVE**() macro removes the element *elm* from the tree pointed by *head*.

The **RB_FIND**() and **RB_NFIND**() macros can be used to find a particular element in the tree.

The **RB_FIND**() macro returns the element in the tree equal to the provided key, or NULL if there is no such element.

The **RB_NFIND**() macro returns the least element greater than or equal to the provided key, or NULL if there is no such element.

```
struct TYPE find, *res, *resn;
find.key = 30;
res = RB_FIND(NAME, head, &find);
resn = RB_NFIND(NAME, head, &find);
```

The **RB_ROOT**(), **RB_MIN**(), **RB_MAX**(), **RB_NEXT**(), and **RB_PREV**() macros can be used to traverse the tree:

```
for (np = RB_MIN(NAME, &head); np != NULL; np = RB_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the **RB_FOREACH**() or **RB_FOREACH_REVERSE**() macro:

RB_FOREACH(*np*, *NAME*, *head*)

The macros **RB_FOREACH_SAFE**() and **RB_FOREACH_REVERSE_SAFE**() traverse the tree referenced by head in a forward or reverse direction respectively, assigning each element in turn to np. However, unlike their unsafe counterparts, they permit both the removal of np as well as freeing it from within the loop safely without interfering with the traversal.

Both **RB_FOREACH_FROM**() and **RB_FOREACH_REVERSE_FROM**() may be used to continue an interrupted traversal in a forward or reverse direction respectively. The head pointer is not required. The pointer to the node from where to resume the traversal should be passed as their last argument, and

will be overwritten to provide safe traversal.

The **RB_EMPTY**() macro should be used to check whether a rank-balanced tree is empty.

The **RB_REINSERT**() macro updates the position of the element *elm* in the tree. This must be called if a member of a **tree** is modified in a way that affects comparison, such as by modifying a node's key. This is a lower overhead alternative to removing the element and reinserting it again.

The **RB_AUGMENT**() macro updates augmentation data of the element *elm* in the tree. By default, it has no effect. It is not meant to be invoked by the RB user. If **RB_AUGMENT**() is defined by the RB user, then when an element is inserted or removed from the tree, it is invoked for every element in the tree that is the root of an altered subtree, working from the bottom of the tree up to the top. It is typically used to maintain some associative accumulation of tree elements, such as sums, minima, maxima, and the like.

The **RB_AUGMENT_CHECK**() macro updates augmentation data of the element *elm* in the tree. By default, it does nothing and returns false. If **RB_AUGMENT_CHECK**() is defined, then when an element is inserted or removed from the tree, it is invoked for every element in the tree that is the root of an altered subtree, working from the bottom of the tree up toward the top, until it returns false to indicate that it did not change the element and so working further up the tree would change nothing. It is typically used to maintain some associative accumulation of tree elements, such as sums, minima, maxima, and the like.

The **RB_UPDATE_AUGMENT**() macro updates augmentation data of the element *elm* and its ancestors in the tree. If **RB_AUGMENT**() is defined by the RB user, then when an element in the tree is changed, without changing the order of items in the tree, invoking this function on that element restores consistency of the augmentation state of the tree as if the element had been removed and inserted again.

EXAMPLES

The following example demonstrates how to declare a rank-balanced tree holding integers. Values are inserted into it and the contents of the tree are printed in order. To maintain the sum of the values in the tree, each element maintains the sum of its value and the sums from its left and right subtrees. Lastly, the internal structure of the tree is printed.

```
#include <sys/tree.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>

struct node {
```

```
        RB_ENTRY(node) entry;
        int i, sum;
};

int
intcmp(struct node *e1, struct node *e2)
{
        return (e1->i < e2->i ? -1 : e1->i > e2->i);
}

int
sumaug(struct node *e)
{
        e->sum = e->i;
        if (RB_LEFT(e, entry) != NULL)
                e->sum += RB_LEFT(e, entry)->sum;
        if (RB_RIGHT(e, entry) != NULL)
                e->sum += RB_RIGHT(e, entry)->sum;
}
#define RB_AUGMENT(entry) sumaug(entry)

RB_HEAD(inttree, node) head = RB_INITIALIZER(&head);
RB_GENERATE(inttree, node, entry, intcmp)

int testdata[] = {
        20, 16, 17, 13, 3, 6, 1, 8, 2, 4, 10, 19, 5, 9, 12, 15, 18,
        7, 11, 14
};

void
print_tree(struct node *n)
{
        struct node *left, *right;

        if (n == NULL) {
                printf("nil");
                return;
        }
        left = RB_LEFT(n, entry);
        right = RB_RIGHT(n, entry);
```

```
		if (left == NULL && right == NULL)
			printf("%d", n->i);
		else {
			printf("%d(", n->i);
			print_tree(left);
			printf(",");
			print_tree(right);
			printf(")");
		}
	}

	int
	main(void)
	{
		int i;
		struct node *n;

		for (i = 0; i < sizeof(testdata) / sizeof(testdata[0]); i++) {
			if ((n = malloc(sizeof(struct node))) == NULL)
				err(1, NULL);
			n->i = testdata[i];
			RB_INSERT(inttree, &head, n);
		}

		RB_FOREACH(n, inttree, &head) {
			printf("%d\n", n->i);
		}
		print_tree(RB_ROOT(&head));
		printf("Sum of values = %d0, RB_ROOT(&head)->sum);
		printf("\n");
		return (0);
	}
```

NOTES

Trying to free a tree in the following way is a common error:

```
	SPLAY_FOREACH(var, NAME, head) {
		SPLAY_REMOVE(NAME, head, var);
		free(var);
	}
```

```
free(head);
```

Since *var* is freed, the **FOREACH**() macro refers to a pointer that may have been reallocated already. Proper code needs a second variable.

```
for (var = SPLAY_MIN(NAME, head); var != NULL; var = nxt) {
        nxt = SPLAY_NEXT(NAME, head, var);
        SPLAY_REMOVE(NAME, head, var);
        free(var);
}
```

Both **RB_INSERT**() and **SPLAY_INSERT**() return NULL if the element was inserted in the tree successfully, otherwise they return a pointer to the element with the colliding key.

Accordingly, **RB_REMOVE**() and **SPLAY_REMOVE**() return the pointer to the removed element otherwise they return NULL to indicate an error.

SEE ALSO

arb(3), queue(3)

Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan, "Rank-Balanced Trees", *ACM Transactions on Algorithms*, 4, 11, http://sidsen.azurewebsites.net/papers/rb-trees-talg.pdf, June 2015.

HISTORY

The tree macros first appeared in FreeBSD 4.6.

AUTHORS

The author of the tree macros is Niels Provos.