rbtree

a red-black balanced binary tree package
edition 1.0.8 for `rbtree` version 1.0.8
20 April 2007

Diab Jerius

Table of Contents

1 Copying

2 Usage

`rbtree` is a package of routines which implement a red-black balanced binary tree. The algorithms are for the most part taken from the book *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest.

Functions supported by this package include searches; tree traversals in either direction with actions performed at selectable traversals of each node; joining of trees; detachment of nodes; and finding data extrema.

2.1 Overview

This package provides two, parallel, interfaces to a tree. The first, preferred, method, is to return results from searches, etc. as node data rather than as references to nodes. This helps shield the user from the vagaries of the tree implementation, and, in most cases, is really what the user wants.

The second method deals directly with node handles. This is generally more efficient, but requires more care by the user. It also permits more specialized manipulations, such as moving nodes between trees without the need to create and destroy nodes. The routines which work with node handles generally have the string '`node`' in their names.

Trees are constructed with `rbtree_new` and are destroyed with `rbtree_delete`. `rbtree_new` is passed a comparison routine which determines how the tree will be ordered.

Nodes are created and inserted into trees with `rbtree_insert`. They can be removed from trees either by completely destroying them, via `rbtree_destroy` or `rbtree_destroy_node`, or by detaching them from the tree via `rbtree_detach_node`. The latter is useful if a node is to be moved from one tree to another. Detached nodes are themeselves destroyed with `rbtree_destroy_dnode`.

A tree is searched with `rbtree_search`, `rbtree_bnd_search`, `rbtree_search_node`, or `rbtree_bnd_search_node`. The first two return pointers to user data, the last two pointers to the node. The `rbtree_bnd_search` routines return the node or data upon success, or the nodes which bracket the data upon failure.

The minimum and maximum data can be also be determined: `rbtree_min`; `rbtree_max`; `rbtree_min_node`; and `rbtree_max_node`. The latter two return handles to the appropriate node.

Tree traversals are available in two flavors. The tree can be traversed and an action performed at the in-order traversals of nodes (`rbtree_traverse` or `rbtree_utraverse`) or at any combination of traversals (pre-order, in-order, or post-order) (`rbtree_walk` or `rbtree_uwalk`). In either case, the tree may be traversed in either ascending or descending order (in this package denoted as `LEFT_TO_RIGHT` or `RIGHT_TO_LEFT`).

Once a specific node has been identfied via `rbtree_min_node`, `rbtree_max_node` or `rbtree_search_node`, various manipulations are possible. It can be detached from the tree (`rbtree_detach_node`); you can find its predecessor or successor nodes (`rbtree_next_node`); its data can be retrieved and modified (`rbtree_node_get_data`; and `rbtree_node_put_data`); and you can delete it (`rbtree_destroy_node`);.

The number of nodes in a tree is available via `rbtree_count`.

Finally, two trees can be joined with `rbtree_join`.

2.2 Data Encapsulation

Each node contains a *data pointer* which associates a separate user-supplied data structure with the node. The pointer is stored at node creation, and is passed back after searches, etc. Nodes do not contain *any* user data. The user is responsible for deallocating any data when individual nodes are destroyed. When deleting entire trees via `rbtree_delete`, a user supplied routine will be invoked on each node's data pointer, if requested.

2.3 Node Comparison

There are two situations in which nodes will be compared, either to other nodes or to key data. During node insertion, the inserted node's data is compared against other nodes' data to determine the proper ordering of the nodes. This operation uses the comparison routine passed to `rbtree_new` when the tree is created. The second situation is during searches of the tree, where node data is compared to some user specified data. Since the comparison routine passed to `rbtree_new` assumes the same form for both pieces of data passed to it for comparison, using it would require that the user create a dummy user node data structure (the same as the one associated with each node), and fill it with the key data, which in most instances will probably be one field. `rbtree` provides the possibility of using another comparison routine (passed to the search or destroy routines), which may compare data with two dissimilar forms. For example, assume that the user node data has the following structure, and keys on the '`id`' value:

```
typedef struct
{
   int id;
   char *name;
} UserNodeData;
```

The node comparison routine, used to compare nodes during insertion, would look like this:

```
int node_node_compare(const void *dp1, const void *dp2)
{
  return ((UserNodeData *)dp1)->id - ((UserNodeData *)dp2)->id;
}
```

If you want to search the resultant tree, and not create a dummy `UserNodeData` structure, construct the search/destroy comparison routine as follows:

```
int key_node_compare(const void *dp1, const void *dp2)
{
  return *((int *) dp1) - ((UserNodeData *)dp2)->id;
}
```

and pass the search/destroy routine a pointer to an `int` set equal to the id for which you're searching.

Several predefined comparison routines are available, but are of limited use. See Section 3.1.20 [rbtree_node_cmp_s], page 18. See Section 3.1.21 [rbtree_node_cmp_v], page 19.

3 Library Routines

3.1 Public Routines

3.1.1 rbtree_bnd_search

Search a red-black binary tree keeping track of sibling nodes.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_bnd_search(
  RBTree rbtree,
  const void *data,
  void **prev,
  void **next,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`RBTree rbtree`
: the red-black tree to search

`const void *data`
: the data to search for

`void **prev`
: holding place for data that is found

`void **next`
: holding place for data that is found

`int (*cmp)(const void *,const void *)`
: The address of a comparison function. Set to `RBTREE_NULL_CMP` to use the tree's initial comparison function.

Description

This routine searches a binary tree for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the tree was initialized. Note that in the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '`-1`', '`0`', or '`1`' if, respectively, the first argument is less than, equal to, or greater than the second.

As the tree is being searched, `rbtree_bnd_search` keeps track of the preceding and succeeding in-order nodes. If the node is not found, the nodes which bracket the data are returned. The result of the search is returned via the parameters `prev`, and `next`. If a node

has data which matches the key data, both parameters are set to the node's data pointer. If a node is not found, and if there exists a previous in-order node, the parameter `prev` is set to that node's data pointer, else it is set to `NULL`. If there exists a succeeding in-order node, the parameter `next` is set to its data pointer, else it is set to `NULL`.

Returns

It returns the node's data pointer if the node can be found, `NULL` otherwise.

3.1.2 rbtree_bnd_search_node

Search a red-black binary tree keeping track of sibling nodes.

Synopsis

```
#include <rbtree/rbtree.h>

RBNode rbtree_bnd_search_node(
  RBTree rbtree,
  const void *data,
  RBNode *prev,
  RBNode *next,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`RBTree rbtree`
: binary tree to search

`const void *data`
: the data to search for

`RBNode *prev`
: set to the previous node

`RBNode *next`
: set to the next node

`int (*cmp)(const void *,const void *)`
: The address of a comparison function. Set to `RBTREE_NULL_CMP` to use the tree's initial comparison function.

Description

This routine searches a binary tree for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the tree was initialized. Note that in the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '`-1`', '`0`', or '`1`' if, respectively, the first argument is less than, equal to, or greater than the second.

As the tree is being searched, `rbtree_bnd_search_node` keeps track of the preceding in-order and succeeding nodes. If the node is not found, the nodes which bracket the data are returned. The result of the search is returned via the parameters `prev` and `next`. If a node has data which matches the key data, both parameters are set to point at the node. If a node is not found, and if there exists a previous in-order node, the parameter `prev` is set to point at the node, else it is set to `NULL`. If there exists a succeeding in-order node, the parameter `next` is set to point to it, else it is set to `NULL`.

Returns

It returns a handle to the found node, or `NULL` if not found.

3.1.3 rbtree_count

Determine the number of nodes in a red-black tree.

Synopsis

```
#include <rbtree/rbtree.h>

unsigned long rbtree_count(RBTree rbtree);
```

Parameters

`RBTree rbtree`
: the tree in question

Description

This routine returns the number of nodes in an `rbtree`. As each tree keeps a counter of how many nodes is in it, this is an inexpensive function to call.

3.1.4 rbtree_delete

Delete a red-black binary tree

Synopsis

```
#include <rbtree/rbtree.h>

void rbtree_delete(
  RBTree rbtree,
  void (*nfree)(void *),
  Visit visit,
  SiblingOrder sibling_order
);
```

Parameters

`RBTree rbtree`
: the handle of the rbtree to delete

`void (*nfree)(void *)`
: a routine which deallocates node user memory, may be `RBTREE_NULL_DELETE`

`Visit visit`
: the order of deletion, either `IN_ORDER`, or `POST_ORDER`, (`PRE_ORDER` is treated as `POST_ORDER`)

  Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

`SiblingOrder sibling_order`
: the direction that the tree is parsed

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine deletes a red-black binary tree, destroying all nodes in the specified order and direction. It optionally takes a user supplied function which, when passed a node's data pointer, deallocates user node data. It is strongly suggested that the deallocation function be specified.

3.1.5 rbtree_udelete

Delete a red-black binary tree.

Synopsis

```
#include <rbtree/rbtree.h>

void rbtree_udelete(
  RBTree rbtree,
  void (*nfree)(void *,void *),
  void *udata,
  Visit visit,
  SiblingOrder sibling_order
);
```

Parameters

`RBTree rbtree`
: the handle of the rbtree to delete

`void (*nfree)(void *,void *)`
: a routine which deallocates node user memory, may be `RBTREE_NULL_DELETE`

`void *udata`
: a pointer to data to be passed to the action routine

`Visit visit`
: the order of deletion. (`PRE_ORDER` is treated as `POST_ORDER`)

  Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

`SiblingOrder sibling_order`
: the direction that the tree is parsed

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine deletes a red-black binary tree, destroying all nodes in the specified order and direction. It optionally takes a user supplied function which, when passed a node's data pointer, deallocates possible user node data. It is strongly suggested that the deallocation function be specified. This routine differs from `rbtree_delete` in that it can pass along a pointer provided by the calling routine to the deallocation function routine, allowing arbitrary data to be available to it.

3.1.6 rbtree_destroy

Search for a node in a red-black binary tree and destroy it.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_destroy(
  RBTree rbtree,
  const void *data,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`RBTree rbtree`
: the red-black tree to search

`const void *data`
: the data to search for

`int (*cmp)(const void *,const void *)`
: The address of a comparison function. Set to `RBTREE_NULL_CMP` to use the tree's initial comparison function.

Description

`rbtree_destroy` searches a binary tree for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If

not, it uses that with which the tree was initialized. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '`-1`', '`0`', or '`1`' if, respectively, the first argument is less than, equal to, or greater than the second.

`rbtree_destroy` frees the memory required by the node. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.1.7 rbtree_destroy_dnode

Deallocate the memory associated with a detached node.

Synopsis

```
#include <rbtree/rbtree.h>

void rbtree_destroy_dnode(RBNode rbnode);
```

Parameters

`RBNode rbnode`
: the handle of the detached node to destroy

Description

This routine deallocates the memory associated with a detached red-black tree node. The user must have already destroyed the data associated with the node. The node handle must have been returned by `rbtree_detach_node`.

3.1.8 rbtree_destroy_node

Remove a node from a binary tree and destroy it.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_destroy_node(
  RBTree rbtree,
  RBNode rbnode
);
```

Parameters

`RBTree rbtree`
: the tree from which to remove the node

`RBNode rbnode`
: the node to remove

Description

This routine removes the passed node from the specified tree and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

Returns the node's data pointer if the node is not `NULL` or `NIL(tree)`, `NULL` otherwise.

3.1.9 rbtree_detach_node

Remove a node from an red-black tree without destroying it.

Synopsis

```
#include <rbtree/rbtree.h>

RBNode rbtree_detach_node(
  RBTree rbtree,
  RBNode rbnode
);
```

Parameters

`RBTree rbtree`
: the tree from which to remove the node

`RBNode rbnode`
: the node to detach

Description

`rbtree_detach_node` detaches the specified node from the specified tree, but doesn't delete it. This is useful for inserting the node into another tree. Because of the way nodes are deleted from trees, the node actually removed may not be the node passed to the routine. The user supplied data is correctly tracked.

Returns

Returns the node that was actually detached, or `NULL` if the node is `NULL` or `NIL(tree)`.

3.1.10 rbtree_insert

Create and insert a node into a red-black tree.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_insert(
  RBTree rbtree,
  void *data
);
```

Parameters

`RBTree rbtree`
: a handle to the tree into which to insert the node

`void *data`
: a pointer to the data

Description

`rbtree_insert` creates a node, sets its data pointer to the specified pointer, and inserts it into the specified tree.

Returns

It returns '`0`' if the insert was successful, '`1`' if it was unable to create the new node.

3.1.11 rbtree_insert_dnode

Insert a detached node into a tree.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_insert_dnode(
  RBTree rbtree,
  RBNode rbnode
);
```

Parameters

`RBTree rbtree`
: a handle to the tree into which to insert the node

`RBNode rbnode`
: the node to insert

Description

This routine inserts a detached node into a binary tree. It assumes that the tree's insert/delete comparison function can be applied to the data in the passed node. The node handle must have been obtained from `rbtree_detach_node`.

Returns

It returns '`1`' if the node is `NULL` or `NIL(tree)`, '`0`' otherwise.

3.1.12 rbtree_join

Join two red-black trees.

Synopsis

```
#include <rbtree/rbtree.h>

void rbtree_join(
  RBTree tree1,
  RBTree tree2
);
```

Parameters

`RBTree tree1`
: the tree to merge the second into

`RBTree tree2`
: the tree to merge into the first tree

Description

This routine joins two redblack trees, moving nodes from the second tree to the first. It doesn't delete the second tree, just empties it. This routine is more efficient than calling `rbtree_detach_node` and `rbtree_insert_dnode`.

3.1.13 rbtree_max

Determine the node with the maximum data.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_max(RBTree rbtree);
```

Parameters

`RBTree rbtree`
: the red-black tree to scan

Description

This routine searches the tree for the node with the maximum data, as determined by the comparison routine with which the tree was created.

Returns

It returns the node's data pointer, `NULL` if the tree has no data.

3.1.14 rbtree_max_node

Determine the node with the maximum data.

Synopsis

```
#include <rbtree/rbtree.h>
```

```
RBNode rbtree_max_node(RBTree rbtree);
```

Parameters

`RBTree rbtree`
: the tree to search

Description

This routine searches the tree for the node with the maximum data, as determined by the comparison routine with which the tree was created.

Returns

It returns a handle to the node if it exists, `NULL` if the tree is empty.

3.1.15 rbtree_min

Determine the node with the minimum data.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_min(RBTree rbtree);
```

Parameters

`RBTree rbtree`
: the red-black tree to scan

Description

This routine searches the tree for the node with the minimum data, as determined by the comparison routine with which the tree was created.

Returns

It returns the node's data pointer, `NULL` if the tree has no data.

3.1.16 rbtree_min_node

Determine the node with the minimum data.

Synopsis

```
#include <rbtree/rbtree.h>

RBNode rbtree_min_node(RBTree rbtree);
```

Parameters

`RBTree rbtree`
: the tree to search

Description

This routine searches the tree for the node with the minimum data, as determined by the comparison routine with which the tree was created.

Returns

It returns a handle to the node if it exists, `NULL` if the tree is empty.

3.1.17 rbtree_new

Create the root of a red-black tree.

Synopsis

```
#include <rbtree/rbtree.h>

RBTree rbtree_new(int (*cmp)(const void *,const void *));
```

Parameters

`int (*cmp)(const void *,const void *)`
: a node comparison routine used to sort the nodes

Description

This routine creates the data structures necessary to implement the root of a red-black tree. It requires a node comparison function which will be used to sort the nodes.

The comparison routine is usually only used for sorting, not searching, but may be if a search or destroy routine is called without a specific comparison routine. It should return '`-1`', '`0`', or '`1`' depending upon whether the first node is respectively less than, equal to, or greater than the second node.

Returns

It returns a handle to the new tree if successful, `NULL` if not.

3.1.18 rbtree_node_size

Return the size of an `RBNode`.

Synopsis

```
#include <rbtree/rbtree.h>

size_t rbtree_node_size(void);
```

Description

This routine returns the size of the internal structure used for each node. This does not include user supplied data. Please note that this is a function call, not a macro!

3.1.19 rbtree_next_node

Determine the next in-order node to a given node.

Synopsis

```
#include <rbtree/rbtree.h>

RBNode rbtree_next_node(
  RBTree rbtree,
  RBNode rbnode,
  SiblingOrder sib_order
);
```

Parameters

`RBTree rbtree`
: the tree to traverse

`RBNode rbnode`
: the node to find the neighbor of

`SiblingOrder sib_order`
: the direction of traversal

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

Determine the next in-order node (depending upon the user's desired traversal direction).

Returns

It returns a handle to the next node or `NULL` if there is none.

3.1.20 rbtree_node_cmp_s

comparison assuming the nodes' data pointer points at strings

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_node_cmp_s(
  const void *dp1,
  const void *dp2
);
```

Parameters

`const void *dp1`
: the first datum to compare

`const void *dp2`
: the second datum to compare

Description

comparison assuming the nodes' data pointer points at strings

3.1.21 rbtree_node_cmp_v

comparison based upon the arithmetic equality of the nodes' data pointers

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_node_cmp_v(
  const void *dp1,
  const void *dp2
);
```

Parameters

`const void *dp1`
: the first datum to compare

`const void *dp2`
: the second datum to compare

Description

comparison based upon the arithmetic equality of the nodes' data pointers

3.1.22 rbtree_node_get_data

Retrieve the data from a specified `rbtree` node.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_node_get_data(RBNode rbnode);
```

Parameters

`RBNode rbnode`
: the node from which to extract the data

Description

Retrieve the data from a specified `rbtree` node.

Returns

It returns the node's data pointer.

3.1.23 rbtree_node_put_data

Replace data pointer in a specified `rbtree` node.

Synopsis

```
#include <rbtree/rbtree.h>

void rbtree_node_put_data(
  RBNode rbnode,
  void *data
);
```

Parameters

`RBNode rbnode`
: the node into which to copy the data

`void *data`
: the data to copy

Description

This routine replaces the data pointer in the specified `rbtree` node with the passed pointer. It does not attempt to resort the tree, so the new data should not change the node's position in the tree (if it's not detached).

3.1.24 rbtree_search

Search a red-black tree for a node with equivalent data.

Synopsis

```
#include <rbtree/rbtree.h>

void *rbtree_search(
  RBTree rbtree,
  const void *data,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`RBTree rbtree`
: the red-black tree to search

`const void *data`
: the data to search for

`int (*cmp)(const void *,const void *)`
: The address of a comparison function. Set to `RBTREE_NULL_CMP` to use the tree's initial comparison function.

Description

`rbtree_search` searches the tree for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it

uses that with which the tree was initialized. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '`-1`', '`0`', or '`1`' if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.1.25 rbtree_search_node

Search a red-black tree for a node with equivalent data.

Synopsis

```
#include <rbtree/rbtree.h>

RBNode rbtree_search_node(
  RBTree rbtree,
  const void *data,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`RBTree rbtree`
: the tree to search

`const void *data`
: the data to match

`int (*cmp)(const void *,const void *)`
: The address of a comparison function. Set to `RBTREE_NULL_CMP` to use the tree's initial comparison function.

Description

This routine searches a red-black tree for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the tree was initialized. Note that in the former case the passed data need not have the same form as the data stored in the node. The comparison routine is called with the passed node's data as the first argument and the tree node's data as the second argument. It must return '`-1`', '`0`', or '`1`' if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns a handle to the matching node, or `NULL` if not found.

3.1.26 rbtree_traverse

Traverse a red-black binary tree in order, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_traverse(
  RBTree rbtree,
  int (*action)(void *nd),
  SiblingOrder sibling_order
);
```

Parameters

`RBTree rbtree`
: a handle to the rbtree to traverse

`int (*action)(void *nd)`
: the user supplied action routine applied to each node

`SiblingOrder sibling_order`
: the direction in which the tree is traversed

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine walks along an `rbtree`, calling a user supplied action function at the in-order traversals of each node. The tree traversal is aborted if the action routine returns non-zero. `rbtree_traverse` is substantially faster than calling `rbtree_walk` with an in-order visit, and uses fewer resources.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.1.27 rbtree_utraverse

Traverse a red-black binary tree in order, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_utraverse(
  RBTree rbtree,
  int (*action)(void *nd,void *udata),
  void *udata,
```

```
  SiblingOrder sibling_order
);
```

Parameters

`RBTree rbtree`
: a handle to the rbtree to traverse

`int (*action)(void *nd,void *udata)`
: the user supplied action routine applied to each node

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the direction in which the tree is traversed

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine walks along an `rbtree`, calling a user supplied action function at the in-order traversals of each node. The tree traversal is aborted if the action routine returns non-zero. `rbtree_utraverse` is substantially faster than calling `rbtree_uwalk` with an in-order visit, and uses fewer resources. This routine differs from `rbtree_traverse` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.1.28 rbtree_walk

Walk along a tree, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_walk(
  RBTree rbtree,
  int (*action)(void *nd,Visit visit,unsigned long level),
  SiblingOrder sibling_order,
  Visit visit
);
```

Parameters

`RBTree rbtree`
: a handle to the rbtree to traverse

`int (*action)(void *nd,Visit visit,unsigned long level)`
: the user supplied action routine applied to each node

`SiblingOrder sibling_order`
: The direction in which the tree is traversed.

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

`Visit visit`
: The node traversals at which to call the action routine. The logical or of possible `Visit` values.

  Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

Description

This routine traverses an `rbtree`, calling a user supplied action function at selectable traversals of each node (any combination of pre-order, in-order, or post-order). The function is informed of which node traversal and which tree level it is called from. The walk is aborted if the action routine returns non-zero.

The traversals at which the action routine is called at are specified by the logical or of `PRE_ORDER`, `IN_ORDER`, or `POST_ORDER`.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.1.29 rbtree_uwalk

Walk along a tree, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

int rbtree_uwalk(
  RBTree rbtree,
  int (*action)(void *nd,Visit visit,unsigned long level,void *udata),
  void *udata,
  SiblingOrder sibling_order,
  Visit visit
);
```

Parameters

`RBTree rbtree`
: a handle to the rbtree to traverse

`int (*action)(void *nd,Visit visit,unsigned long level,void *udata)`
: the user supplied action routine applied to each node

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the direction in which the tree is traversed

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

`Visit visit`
: The node traversals at which to call the action routine. The logical or of possible `Visit` values.

  Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

Description

This routine traverses an `rbtree`, calling a user supplied action function at selectable traversals of each node (any combination of pre-order, in-order, or post-order). The function is informed of which node traversal and which tree level it is called from. The walk is aborted if the action routine returns non-zero. This routine differs from `rbtree_walk` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

The traversals at which the action routine is called at are specified by the logical or of `PRE_ORDER`, `IN_ORDER`, or `POST_ORDER`.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.2 Private Routines

3.2.1 BndSearch

Search a (sub)tree for a node with comporable data, keeping track of siblings.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *BndSearch(
  Tree *tree,
  Node *x,
  Node **prev,
  Node **next,
```

```
  const void *data,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`Tree *tree`
: the tree to search

`Node *x`
: the node at which to start the search

`Node **prev`
: where to stick a pointer to the predecessor node

`Node **next`
: where to stick a pointer to the successor node

`const void *data`
: the data to search for

`int (*cmp)(const void *,const void *)`
: the address of a node comparison function

Description

This routine will search a tree for the node which has data comparable to that which is passed, using the passed comparison routine. If the node does not exist, it returns the predecessor and successor nodes. the start node need not be the root of the tree, allowing for sub-tree searches.

Returns

It returns a pointer to the found node upon sucess. If no node is found, it sets the parameters next and/or prev to point to the successor and predecessor nodes. If those nodes don't exist, the pointers are set to `NIL(tree)`.

3.2.2 Delete

Remove a node from a tree.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Delete(
  Tree *tree,
  Node *z
);
```

Parameters

`Tree *tree`
: the tree from which to delete the node

`Node *z`
: the node to delete

Description

`Delete` removes a node from a tree. The node removed may not be that requested to be removed, for algorithmic efficiency's sake. in this case, the data is swapped with the node that is removed, so everything works as expected. It does not deallocate the memory associated with the node.

Returns

It returns a pointer to the node that was actually removed.

3.2.3 Free_in_order

Traverse a tree, freeing nodes in-order.

Synopsis

```
#include <rbtree/rbtree.h>

static void Free_in_order(
  Tree *tree,
  Node *node,
  void (*nfree)(void *),
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to free

`Node *node`
: the starting node, must be `ROOT(tree)`

`void (*nfree)(void *)`
: a free routine for node data, may be `RBTREE_NULL_DELETE`

`SiblingOrder sibling_order`
: the order in which to traverse the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

Traverse the given tree, freeing (in-order) as we go. Recursive, so it uses more resources than `Free_post_order`.

3.2.4 uFree_in_order

Traverse a tree, freeing nodes in-order.

Synopsis

```
#include <rbtree/rbtree.h>

static void uFree_in_order(
  Tree *tree,
  Node *node,
  void (*nfree)(void *,void *),
  void *udata,
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to free

`Node *node`
: the starting node, must be `ROOT(tree)`

`void (*nfree)(void *,void *)`
: a free routine for node data, may be `RBTREE_NULL_DELETE`

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the order in which to traverse the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

Traverse the given tree, freeing (in-order) as we go. Recursive, so it uses more resources than `Free_post_order`. This routine differs from `Free_in_order` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

3.2.5 uFree_post_order

Free a tree, in post-order sequence.

Synopsis

```
#include <rbtree/rbtree.h>

static void uFree_post_order(
  Tree *tree,
  void (*nfree)(void *,void *),
  void *udata,
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to destroy

`void (*nfree)(void *,void *)`
: a routine to free the node data, may be `RBTREE_NULL_DELETE`

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the order in which to free the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine traverses the given tree, freeing (post-order) as we go. It's the fastest deletion routine. Use it when it doesn't matter in what order the data are deleted. This routine differs from `Free_post_order` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine. The node data is passed to a user-supplied free routine, if available.

3.2.6 Free_post_order

Free a tree, in post-order sequence.

Synopsis

```
#include <rbtree/rbtree.h>

static void Free_post_order(
  Tree *tree,
  void (*nfree)(void *),
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to destroy

`void (*nfree)(void *)`
: a routine to free the node data, may be `RBTREE_NULL_DELETE`

`SiblingOrder sibling_order`
: the order in which to free the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

This routine traverses the given tree, freeing (post-order) as we go. It's the fastest deletion routine. Use it when it doesn't matter in what order the data are deleted.

The node data is passed to a user-supplied free routine, if available.

3.2.7 Insert

Insert a node into a tree.

Synopsis

```
#include <rbtree/rbtree.h>

static void Insert(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree into which to insert the node

`Node *x`
: the node to insert

Description

This routine inserts the passed node into the passed tree, using the default comparison node routine.

3.2.8 JoinTrees

Join two trees.

Synopsis

```
#include <rbtree/rbtree.h>

static void JoinTrees(
  Tree *tree1,
  Tree *tree2
);
```

Parameters

`Tree *tree1`
: the tree to merge the second into

`Tree *tree2`
: the tree to merge into the first tree

Description

This routine joins two trees, detaching nodes from the second and inserting them in the first. As the second tree is completely gutted, the routine doesn't spend time keeping its redblack nature intact during the process.

3.2.9 LeftRotate

Perform a left rotation of a tree about a node.

Synopsis

```
#include <rbtree/rbtree.h>

static void LeftRotate(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree to rotate

`Node *x`
: the pivot node

Description

Perform a left rotation of a tree about a node.

3.2.10 Maximum

Return the right most node in the tree.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Maximum(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree to traverse

`Node *x`
: the origin node

Description

Return the right most node in the tree.

3.2.11 Minimum

Return the left most node in the tree.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Minimum(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree to traverse

`Node *x`
: the origin node

Description

Return the left most node in the tree.

3.2.12 Predecessor

Find the previous in-order node to a node.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Predecessor(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree to traverse

`Node *x`
: the origin node

Description

Find the previous in-order node to a node.

3.2.13 RightRotate

Perform a right rotation of a tree about a node.

Synopsis

```
#include <rbtree/rbtree.h>
```

```
static void RightRotate(
  Tree *tree,
  Node *y
);
```

Parameters

`Tree *tree`
: the tree to rotate

`Node *y`
: the pivot node

Description

Perform a right rotation of a tree about a node.

3.2.14 Search

Search a tree for a node with comparable data.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Search(
  Tree *tree,
  Node *x,
  const void *data,
  int (*cmp)(const void *,const void *)
);
```

Parameters

`Tree *tree`
: the tree to search

`Node *x`
: the node at which to start

`const void *data`
: the data to search for

`int (*cmp)(const void *,const void *)`
: the address of a node comparison routine

Description

This routine will search a tree for the node which has data comparable to that passed, using the passed comparison routine. The start node need not be the root of the tree, allowing for subtree searches.

Returns

It returns the node which matches, else `NIL(tree)`.

3.2.15 Successor

Find the next in-order node to a node.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *Successor(
  Tree *tree,
  Node *x
);
```

Parameters

`Tree *tree`
: the tree to traverse

`Node *x`
: the origin node

Description

Find the next in-order node to a node.

3.2.16 Traverse

Walk a tree, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

static int Traverse(
  Tree *tree,
  Node *node,
  int (*action)(void *),
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to walk

`Node *node`
: the node to start at, must be `ROOT(tree)`

`int (*action)(void *)`
: the action to perform at each node

`SiblingOrder sibling_order`
: the order in which to traverse the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

`Traverse` quickly walks a tree in a prescribed order, performing a user supplied action on each node, in-order. If the action function returns non-zero, the walk is aborted.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.2.17 uTraverse

Walk a tree, processing each node.

Synopsis

```
#include <rbtree/rbtree.h>

static int uTraverse(
  Tree *tree,
  Node *node,
  int (*action)(void *,void *udata),
  void *udata,
  SiblingOrder sibling_order
);
```

Parameters

`Tree *tree`
: the tree to walk

`Node *node`
: the node to start at, must be `ROOT(tree)`

`int (*action)(void *,void *udata)`
: the action to perform at each node

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the order in which to traverse the tree

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

Description

`uTraverse` quickly walks a tree in a prescribed order, performing a user supplied action on each node, in-order. If the action function returns non-zero, the walk is aborted. This routine differs from Traverse in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.2.18 Walk

Walk a tree, performing an action at specified traversals of each node.

Synopsis

```
#include <rbtree/rbtree.h>

static int Walk(
  Tree *tree,
  Node *node,
  int (*action)(void *data,Visit visit,unsigned long level),
  SiblingOrder sibling_order,
  Visit visit,
  unsigned long level
);
```

Parameters

`Tree *tree`
: the tree to walk

`Node *node`
: the node at which to start. must be `ROOT(tree)`

`int (*action)(void *data,Visit visit,unsigned long level)`
: the action routine

`SiblingOrder sibling_order`
: the order in which to walk

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

`Visit visit`
: the traversal(s) at which to call the action routine

  Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

`unsigned long level`
: the level of the passed node. '0' if the root node.

Description

`Walk` traverse a tree, calling a user supplied action routine at selectable traversals of each node (any combination of pre-order, in-order, or post-order). The action routine is informed

of which node traversal and which tree level it is called from. the walk is aborted if the action routine returns non-zero.

The traversals at which the action routine is called at are specified by the logical or of `PRE_ORDER`, `IN_ORDER`, or `POST_ORDER`.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.2.19 uWalk

Walk a tree, performing an action at specified traversals of each node.

Synopsis

```
#include <rbtree/rbtree.h>

static int uWalk(
  Tree *tree,
  Node *node,
  int (*action)(void *data,Visit visit,unsigned long level,void *udata),
  void *udata,
  SiblingOrder sibling_order,
  Visit visit,
  unsigned long level
);
```

Parameters

`Tree *tree`
: the tree to walk

`Node *node`
: the node at which to start. must be `ROOT(tree)`

`int (*action)(void *data,Visit visit,unsigned long level,void *udata)`
: the action routine

`void *udata`
: a pointer to data to be passed to the action routine

`SiblingOrder sibling_order`
: the order in which to walk

  Possible values for a `SiblingOrder` are as follows: `LEFT_TO_RIGHT`, `RIGHT_TO_LEFT`

`Visit visit`
: the traversal(s) at which to call the action routine

Possible values for a `Visit` are as follows: `PRE_ORDER`, `IN_ORDER`, `POST_ORDER`

`unsigned long level`
: the level of the passed node. '`0`' if the root node.

Description

`uWalk` traverse a tree, calling a user supplied action routine at selectable traversals of each node (any combination of pre-order, in-order, or post-order). The action routine is informed of which node traversal and which tree level it is called from. the walk is aborted if the action routine returns non-zero. This routine differs from `Walk` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

The traversals at which the action routine is called at are specified by the logical or of `PRE_ORDER`, `IN_ORDER`, or `POST_ORDER`.

Returns

If the action routines all return '`0`', it returns '`0`', else it returns the value returned by the last action routine called.

3.2.20 deleteNode

Delete a node.

Synopsis

```
#include <rbtree/rbtree.h>

static void deleteNode(Node *node);
```

Parameters

`Node *node`
: the node to delete

Description

This routine deallocates memory associated with a node. If asked to delete a `NULL` node, it just returns.

3.2.21 newNode

Allocate a new node and copy data into it.

Synopsis

```
#include <rbtree/rbtree.h>

static Node *newNode(
  Tree *tree,
  void *data
```

```
);
```

Parameters

`Tree *tree`
: the tree that will contain the node

`void *data`
: the data that the new node will contain

Description

Allocate a new node and copy data into it.