Red-black tree (C)

From LiteratePrograms
Jump to: navigation, search

Other implementations: **C** | Java

A red-black tree is a type of self-balancing binary search tree typically used to implement associative arrays. It has O(log *n*) worst-case time for each operation and is quite efficient in practice. Unfortunately, it's also quite complex to implement, requiring a number of subtle cases for both insertion and deletion.

This article walks through a C implementation of red-black trees, organized in a way to make correctness and completeness easier to understand.

Contents

[edit] Node structure and node relationships

A red-black tree is a type of binary search tree, so each node in the tree has a parent (except the root node) and at most two children. The tree as a whole will be identified by its root node. For ease of implementation, we will have each node retain a pointer to both its children as well as its parent node (NULL for the root). Keeping parent nodes costs space and is not strictly necessary, but makes it easy to follow the tree in any direction without maintaining an auxilary stack. Our red-black tree will implement an associative array, so we will store both the key and its associated value in a void*:

```
<<public type definitions>>=
enum rbtree_node_color { RED, BLACK };

typedef struct rbtree_node_t {
    void* key;
    void* value;
    struct rbtree_node_t* left;
    struct rbtree_node_t* right;
    struct rbtree_node_t* parent;
    enum rbtree_node_color color;
} *rbtree_node;

typedef struct rbtree_t {
    rbtree_node root;
} *rbtree;
```

Each node also stores its color, either red or black, using an enumeration. The role of the color bit will be explained in the properties. There is some internal fragmentation due to using an integer type to store a single bit, but we avoid optimizing this here for simplicity. In the source file we will typedef abbreviated names for our types:

```
<<private type definitions>>=
typedef rbtree_node node;
typedef enum rbtree_node_color color;
```

The parent of a node n is available by simply using n->parent. We're also interested in three other more complex relationships between nodes:

- The grandparent of a node, its parent's parent. We use assertions to make sure that we don't attempt to access the grandparent of a node that doesn't have one, such as the root node or its children:

```
<<private function prototypes>>=
static node grandparent(node n);
<<node relationships>>=
node grandparent(node n) {
    assert (n != NULL);
    assert (n->parent != NULL); /* Not the root node */
    assert (n->parent->parent != NULL); /* Not child of root */
    return n->parent->parent;
}
```

- The sibling of a node, defined as the other child of its parent. Note that the sibling *may* be NULL, if the parent has only one child.

```
<<private function prototypes>>=
static node sibling(node n);
<<node relationships>>=
node sibling(node n) {
    assert (n != NULL);
    assert (n->parent != NULL); /* Root node has no sibling */
    if (n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
}
```

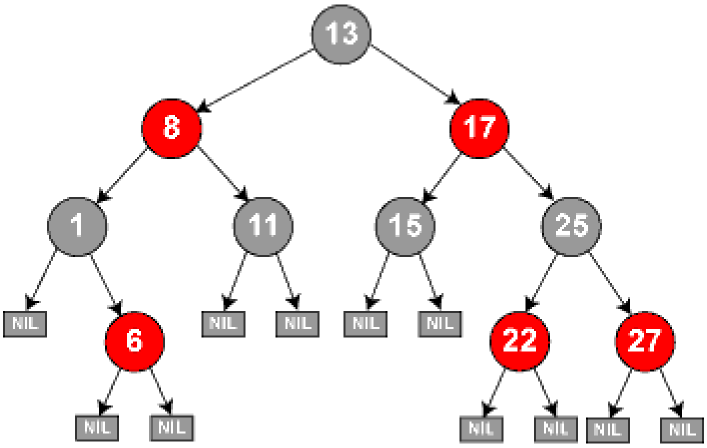
- The uncle of a node, defined as the sibling of its parent. The uncle may also be NULL, if the grandparent has only one child.

```
<<private function prototypes>>=
static node uncle(node n);
<<node relationships>>=
node uncle(node n) {
   assert (n != NULL);
   assert (n->parent != NULL); /* Root node has no uncle */
   assert (n->parent->parent != NULL); /* Children of root have no uncle */
   return sibling(n->parent);
}
```

Although in older C environments we might define these as macros for efficiency, we prefer here to rely on function inlining support in the compiler to keep the code simple. The use of assert() requires a header:

```
<<include files>>=
#include <assert.h>
```

[edit] Properties



We will at all times enforce the following five properties, which provide a theoretical guarantee that the tree remains balanced. We will have a helper function verify_properties() that asserts all five properties in a debug build, to help verify the correctness of our implementation and formally demonstrate their meaning. Note that many of these tests walk the tree, making them very expensive - for this reason we require the symbol VERIFY_RBTREE to be defined to turn them on.

As shown, the tree terminates in NIL leaves, which we represent using NULL (we set the child pointers of their parents to NULL). In an empty tree, the root pointer is NULL. This saves substantial space compared to explicit representation of leaves.

```
<<private function prototypes>>=
static void verify_properties(rbtree t);
<<verify properties functions>>=
void verify_properties(rbtree t) {
#ifdef VERIFY_RBTREE
   verify_property_1(t->root);
```

```
    verify_property_2(t->root);
    /* Property 3 is implicit */
    verify_property_4(t->root);
    verify_property_5(t->root);
#endif
}
```

1. Each node is either red or black:

```
<<private function prototypes>>=
static void verify_property_1(node root);
<<verify properties functions>>=
void verify_property_1(node n) {
    assert(node_color(n) == RED || node_color(n) == BLACK);
    if (n == NULL) return;
    verify_property_1(n->left);
    verify_property_1(n->right);
}
```

2. The root node is black.

```
<<private function prototypes>>=
static void verify_property_2(node root);
<<verify properties functions>>=
void verify_property_2(node root) {
    assert(node_color(root) == BLACK);
}
```

3. All leaves (shown as NIL in the above diagram) are black and contain no data. Since we represent these empty leaves using `NULL`, this property is implicitly assured by always treating `NULL` as black. To this end we create a `node_color()` helper function:

```
<<private function prototypes>>=
static color node_color(node n);
<<verify properties functions>>=
color node_color(node n) {
    return n == NULL ? BLACK : n->color;
}
```

4. Every red node has two children, and both are black (or equivalently, the parent of every red node is black).

```
<<private function prototypes>>=
static void verify_property_4(node root);
<<verify properties functions>>=
void verify_property_4(node n) {
    if (node_color(n) == RED) {
        assert (node_color(n->left)   == BLACK);
        assert (node_color(n->right)  == BLACK);
        assert (node_color(n->parent) == BLACK);
    }
    if (n == NULL) return;
    verify_property_4(n->left);
    verify_property_4(n->right);
}
```

5. All paths from any given node to its leaf nodes contain the same number of black nodes. This one is the trickiest to verify; we do it by traversing the tree, incrementing a black node count as we go. The first time we reach a leaf we save the count. When we subsequently reach other leaves, we compare the count to this saved count.

```
<<private function prototypes>>=
static void verify_property_5(node root);
static void verify_property_5_helper(node n, int black_count, int* black_count_path);
```

```
<<verify properties functions>>=
void verify_property_5(node root) {
    int black_count_path = -1;
    verify_property_5_helper(root, 0, &black_count_path);
}

void verify_property_5_helper(node n, int black_count, int* path_black_count) {
    if (node_color(n) == BLACK) {
        black_count++;
    }
    if (n == NULL) {
        if (*path_black_count == -1) {
            *path_black_count = black_count;
        } else {
            assert (black_count == *path_black_count);
        }
        return;
    }
    verify_property_5_helper(n->left,  black_count, path_black_count);
    verify_property_5_helper(n->right, black_count, path_black_count);
}
```

Properties 4 and 5 together guarantee that no path in the tree is more than about twice as long as any other path, which guarantees that it has O(log *n*) height.

[edit] Operations

[edit] Construction

An empty tree is represented by a tree with a NULL root. This object provides a starting point for other operations.

```
<<public function prototypes>>=
rbtree rbtree_create();
<<create operation>>=
rbtree rbtree_create() {
    rbtree t = malloc(sizeof(struct rbtree_t));
    t->root = NULL;
    verify_properties(t);
    return t;
}
```

We have a helper function to allocate and initialize a new node:

```
<<private function prototypes>>=
static node new_node(void* key, void* value, color node_color, node left, node right);
<<node constructor>>=
node new_node(void* key, void* value, color node_color, node left, node right) {
    node result = malloc(sizeof(struct rbtree_node_t));
    result->key = key;
    result->value = value;
    result->color = node_color;
    result->left = left;
    result->right = right;
    if (left  != NULL)  left->parent = result;
    if (right != NULL) right->parent = result;
    result->parent = NULL;
    return result;
}
```

The malloc() calls requires stdlib.h:

```
<<include files>>=
#include <stdlib.h>
```

[edit] Search

Read-only operations on a red-black tree, such as searching for a key and getting the corresponding value, require no modification from those used for binary search trees, because every red-black tree is a specialization of a simple binary search tree.

We begin by creating a helper function that gets a pointer to the node with a given key. If the key is not found, it returns NULL. This will be useful later for deletion:

```
<<public type definitions>>=
typedef int (*compare_func)(void* left, void* right);
<<private function prototypes>>=
static node lookup_node(rbtree t, void* key, compare_func compare);
<<lookup operation>>=
node lookup_node(rbtree t, void* key, compare_func compare) {
    node n = t->root;
    while (n != NULL) {
        int comp_result = compare(key, n->key);
        if (comp_result == 0) {
            return n;
        } else if (comp_result < 0) {
            n = n->left;
        } else {
            assert(comp_result > 0);
            n = n->right;
        }
    }
    return n;
}
```

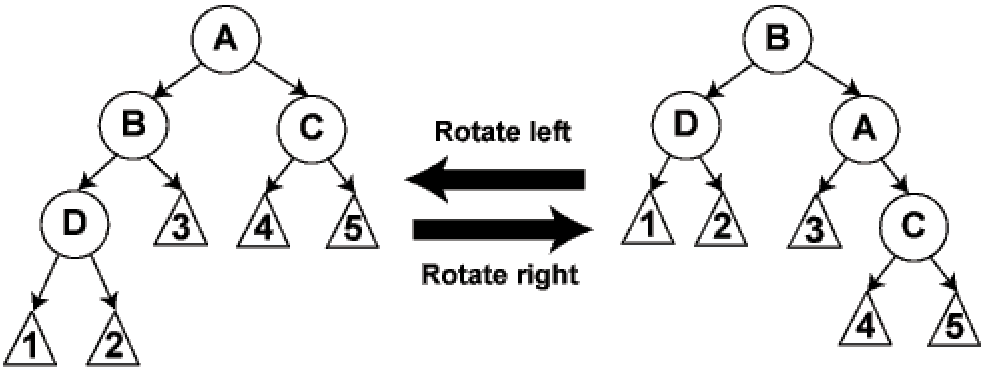
The client must pass in a comparison function to compare the data values, which have unknown type. Like the comparison function passed to the library function qsort(), it returns a negative value, zero, or positive value, depending if the left value is less than, equal to, or greater than the right value, respectively.

Now looking up a value is straightforward, by finding the node and extracting the data if lookup succeeded. We return NULL if the key was not found (implying that NULL cannot be used as a value unless all lookups are expected to succeed).

```
<<public function prototypes>>=
void* rbtree_lookup(rbtree t, void* key, compare_func compare);
<<lookup operation>>=
void* rbtree_lookup(rbtree t, void* key, compare_func compare) {
    node n = lookup_node(t, key, compare);
    return n == NULL ? NULL : n->value;
}
```

[edit] Rotations

Both insertion and deletion rely on a fundamental operation for reducing tree height called a *rotation*. A rotation locally changes the structure of the tree without changing the in-order order of the sequence of values that it stores.



We create two helper functions, one to perform a left rotation and one to perform a right rotation; each takes the highest node in the subtree as an argument:

```
<<private function prototypes>>=
static void rotate_left(rbtree t, node n);
static void rotate_right(rbtree t, node n);

<<rotation operations>>=
void rotate_left(rbtree t, node n) {
    node r = n->right;
    replace_node(t, n, r);
    n->right = r->left;
    if (r->left != NULL) {
        r->left->parent = n;
    }
    r->left = n;
    n->parent = r;
}

void rotate_right(rbtree t, node n) {
    node L = n->left;
    replace_node(t, n, L);
    n->left = L->right;
    if (L->right != NULL) {
        L->right->parent = n;
    }
    L->right = n;
    n->parent = L;
}
```

Here, replace_node() is a helper function that cuts a node away from its parent, substituting a new node (or NULL) in its place. It simplifies consistent updating of parent and child pointers. It needs the tree passed in because it may change which node is the root.

```
<<private function prototypes>>=
static void replace_node(rbtree t, node oldn, node newn);
<<replace node operation>>=
void replace_node(rbtree t, node oldn, node newn) {
    if (oldn->parent == NULL) {
        t->root = newn;
    } else {
        if (oldn == oldn->parent->left)
            oldn->parent->left = newn;
        else
            oldn->parent->right = newn;
    }
    if (newn != NULL) {
        newn->parent = oldn->parent;
    }
}
```

We'll find replace_node() useful again later on when discussing insertion and deletion.

[edit] Insertion

When inserting a new value, we first insert it into the tree as we would into an ordinary binary search tree. If the key already exists, we just replace the value (since we're implementing an associative array). Otherwise, we find the place in the tree where the new pair belongs, then attach a newly created red node containing the value:

```
<<public function prototypes>>=
void rbtree_insert(rbtree t, void* key, void* value, compare_func compare);
<<insertion operation>>=
void rbtree_insert(rbtree t, void* key, void* value, compare_func compare) {
    node inserted_node = new_node(key, value, RED, NULL, NULL);
    if (t->root == NULL) {
        t->root = inserted_node;
    } else {
        node n = t->root;
        while (1) {
            int comp_result = compare(key, n->key);
            if (comp_result == 0) {
                n->value = value;
                return;
            } else if (comp_result < 0) {
                if (n->left == NULL) {
                    n->left = inserted_node;
                    break;
                } else {
                    n = n->left;
                }
            } else {
                assert (comp_result > 0);
                if (n->right == NULL) {
                    n->right = inserted_node;
                    break;
                } else {
                    n = n->right;
                }
            }
        }
        inserted_node->parent = n;
    }
    insert_case1(t, inserted_node);
    verify_properties(t);
}
```

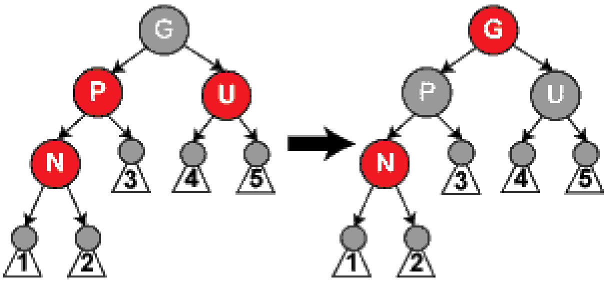
The problem is that the resulting tree may not satify our five red-black tree properties. The call to insert_case1() above begins the process of correcting the tree so that it satifies the properties once more.

Case 1: In this case, the new node is now the root node of the tree. Since the root node must be black, and changing its color adds the same number of black nodes to every path, we simply recolor it black. Because only the root node has no parent, we can assume henceforth that the node has a parent.

```
<<private function prototypes>>=
static void insert_case1(rbtree t, node n);
<<insertion operation>>=
void insert_case1(rbtree t, node n) {
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(t, n);
}
```

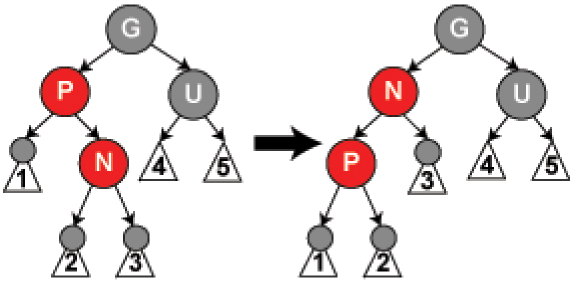
Case 2: In this case, the new node has a black parent. All the properties are still satisfied and we return.

```
<<private function prototypes>>=
static void insert_case2(rbtree t, node n);
<<insertion operation>>=
void insert_case2(rbtree t, node n) {
    if (node_color(n->parent) == BLACK)
        return; /* Tree is still valid */
    else
        insert_case3(t, n);
}
```



Case 3: In this case, the uncle node is red. We recolor the parent and uncle black and the grandparent red. However, the red grandparent node may now violate the red-black tree properties; we recursively invoke this procedure on it from case 1 to deal with this.

```
<<private function prototypes>>=
static void insert_case3(rbtree t, node n);
<<insertion operation>>=
void insert_case3(rbtree t, node n) {
    if (node_color(uncle(n)) == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(t, grandparent(n));
    } else {
        insert_case4(t, n);
    }
}
```

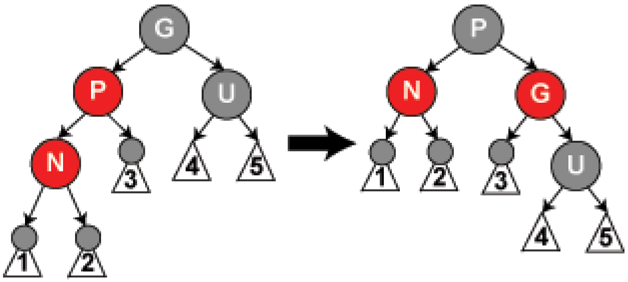


Case 4: In this case, we deal with two cases that are mirror images of one another:

- The new node is the right child of its parent and the parent is the left child of the grandparent. In this case we rotate left about the parent.
- The new node is the left child of its parent and the parent is the right child of the grandparent. In this case we rotate right about the parent.

Neither of these fixes the properties, but they put the tree in the correct form to apply case 5.

```
<<private function prototypes>>=
static void insert_case4(rbtree t, node n);
<<insertion operation>>=
void insert_case4(rbtree t, node n) {
   if (n == n->parent->right && n->parent == grandparent(n)->left) {
      rotate_left(t, n->parent);
      n = n->left;
   } else if (n == n->parent->left && n->parent == grandparent(n)->right) {
      rotate_right(t, n->parent);
      n = n->right;
   }
   insert_case5(t, n);
}
```



Case 5: In this final case, we deal with two cases that are mirror images of one another:

- The new node is the left child of its parent and the parent is the left child of the grandparent. In this case we rotate right about the grandparent.
- The new node is the right child of its parent and the parent is the right child of the grandparent. In this case we rotate left about the grandparent.

Now the properties are satisfied and all cases have been covered.

```
<<private function prototypes>>=
static void insert_case5(rbtree t, node n);
<<insertion operation>>=
void insert_case5(rbtree t, node n) {
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(t, grandparent(n));
    } else {
        assert (n == n->parent->right && n->parent == grandparent(n)->right);
        rotate_left(t, grandparent(n));
    }
}
```

Note that inserting is actually in-place, since all the calls above use tail recursion. Moreover, it performs at most two rotations, since the only recursive call occurs before making any rotations.

[edit] Removal

We begin by finding the node to be deleted with `lookup_node()` and deleting it precisely as we would in a binary search tree. There are two cases for removal, depending on whether the node to be deleted has at most one, or two non-leaf children. A node with at most one non-leaf child can simply be replaced with its non-leaf child. When deleting a node with two non-leaf children, we copy the value from the in-order predecessor (the maximum or rightmost element in the left subtree) into the node to be deleted, and then we then delete the predecessor node, which has only one non-leaf child. This same procedure also works in a red-black tree without affecting any properties.

Error creating thumbnail: /bin/bash: rsvg: command not
found

```
<<public function prototypes>>=
void rbtree_delete(rbtree t, void* key, compare_func compare);
<<deletion operation>>=
void rbtree_delete(rbtree t, void* key, compare_func compare) {
    node child;
    node n = lookup_node(t, key, compare);
    if (n == NULL) return;  /* Key not found, do nothing */
    if (n->left != NULL && n->right != NULL) {
        /* Copy key/value from predecessor and then delete it instead */
        node pred = maximum_node(n->left);
        n->key   = pred->key;
        n->value = pred->value;
        n = pred;
    }

    assert(n->left == NULL || n->right == NULL);
    child = n->right == NULL ? n->left  : n->right;
    if (node_color(n) == BLACK) {
        n->color = node_color(child);
        delete_case1(t, n);
    }
    replace_node(t, n, child);
    if (n->parent == NULL && child != NULL) // root should be black
        child->color = BLACK;
    free(n);

    verify_properties(t);
}
```

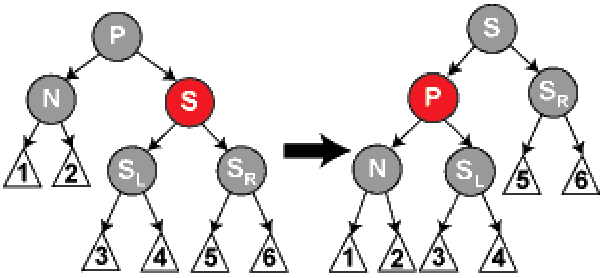
The maximum_node() helper function just walks right until it reaches the last non-leaf:

```
<<private function prototypes>>=
static node maximum_node(node root);
<<deletion operation>>=
static node maximum_node(node n) {
    assert (n != NULL);
    while (n->right != NULL) {
        n = n->right;
    }
    return n;
}
```

However, before deleting the node, we must ensure that doing so does not violate the red-black tree properties. If the node we delete is black, and we cannot change its child from red to black to compensate, then we would have one less black node on every path through the child node. We must adjust the tree around the node being deleted to compensate.

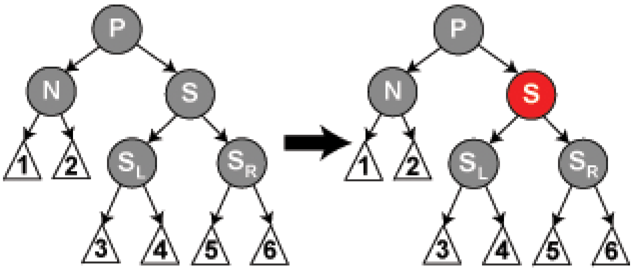
Case 1: In this case, N has become the root node. The deletion removed one black node from every path, so no properties are violated.

```
<<private function prototypes>>=
static void delete_case1(rbtree t, node n);
<<deletion operation>>=
void delete_case1(rbtree t, node n) {
    if (n->parent == NULL)
        return;
    else
        delete_case2(t, n);
}
```



Case 2: N has a red sibling. In this case we exchange the colors of the parent and sibling, then rotate about the parent so that the sibling becomes the parent of its former parent. This does not restore the tree properties, but reduces the problem to one of the remaining cases.

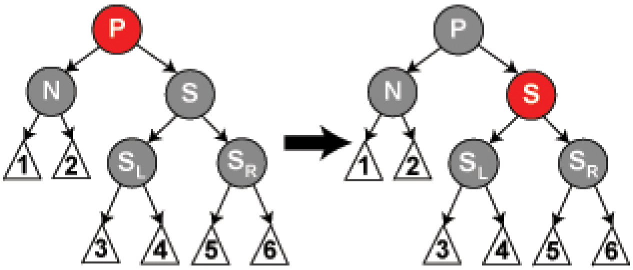
```
<<private function prototypes>>=
static void delete_case2(rbtree t, node n);
<<deletion operation>>=
void delete_case2(rbtree t, node n) {
   if (node_color(sibling(n)) == RED) {
      n->parent->color = RED;
      sibling(n)->color = BLACK;
      if (n == n->parent->left)
         rotate_left(t, n->parent);
      else
         rotate_right(t, n->parent);
   }
   delete_case3(t, n);
}
```



Case 3: In this case N's parent, sibling, and sibling's children are black. In this case we paint the sibling red. Now all paths passing through N's parent have one less black node than before the deletion, so we must recursively run this procedure from case 1 on N's parent.

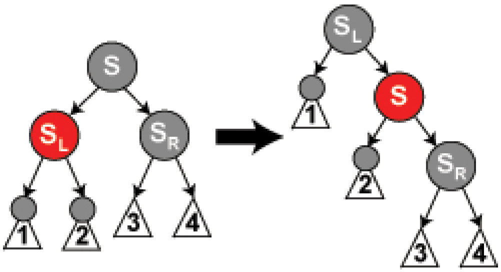
```
<<private function prototypes>>=
static void delete_case3(rbtree t, node n);
<<deletion operation>>=
void delete_case3(rbtree t, node n) {
   if (node_color(n->parent) == BLACK &&
      node_color(sibling(n)) == BLACK &&
      node_color(sibling(n)->left) == BLACK &&
      node_color(sibling(n)->right) == BLACK)
```

```
    {
        sibling(n)->color = RED;
        delete_case1(t, n->parent);
    }
    else
        delete_case4(t, n);
}
```



Case 4: N's sibling and sibling's children are black, but its parent is red. We exchange the colors of the sibling and parent; this restores the tree properties.

```
<<private function prototypes>>=
static void delete_case4(rbtree t, node n);
<<deletion operation>>=
void delete_case4(rbtree t, node n) {
    if (node_color(n->parent) == RED &&
        node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == BLACK &&
        node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        n->parent->color = BLACK;
    }
    else
        delete_case5(t, n);
}
```

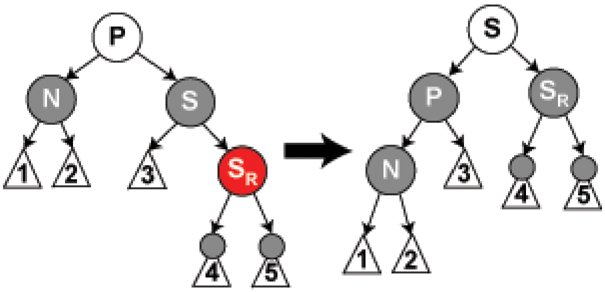


Case 5: There are two cases handled here which are mirror images of one another:

- N's sibling S is black, S's left child is red, S's right child is black, and N is the left child of its parent. We exchange the colors of S and its left sibling and rotate right at S.
- N's sibling S is black, S's right child is red, S's left child is black, and N is the right child of its parent. We exchange the colors of S and its right sibling and rotate left at S.

Both of these function to reduce us to the situation described in case 6.

```
<<private function prototypes>>=
static void delete_case5(rbtree t, node n);
<<deletion operation>>=
void delete_case5(rbtree t, node n) {
    if (n == n->parent->left &&
        node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == RED &&
        node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->left->color = BLACK;
        rotate_right(t, sibling(n));
    }
    else if (n == n->parent->right &&
             node_color(sibling(n)) == BLACK &&
             node_color(sibling(n)->right) == RED &&
             node_color(sibling(n)->left) == BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->right->color = BLACK;
        rotate_left(t, sibling(n));
    }
    delete_case6(t, n);
}
```



Case 6: There are two cases handled here which are mirror images of one another:

- N's sibling S is black, S's right child is red, and N is the left child of its parent. We exchange the colors of N's parent and sibling, make S's right child black, then rotate left at N's parent.
- N's sibling S is black, S's left child is red, and N is the right child of its parent. We exchange the colors of N's parent and sibling, make S's left child black, then rotate right at N's parent.

This accomplishes three things at once:

- We add a black node to all paths through N, either by adding a black S to those paths or by recoloring N's parent black.
- We remove a black node from all paths through S's red child, either by removing P from those paths or by recoloring S.

- We recolor S's red child black, adding a black node back to all paths through S's red child.

S's left child has become a child of N's parent during the rotation and so is unaffected.

```
<<private function prototypes>>=
static void delete_case6(rbtree t, node n);
<<deletion operation>>=
void delete_case6(rbtree t, node n) {
    sibling(n)->color = node_color(n->parent);
    n->parent->color = BLACK;
    if (n == n->parent->left) {
        assert (node_color(sibling(n)->right) == RED);
        sibling(n)->right->color = BLACK;
        rotate_left(t, n->parent);
    }
    else
    {
        assert (node_color(sibling(n)->left) == RED);
        sibling(n)->left->color = BLACK;
        rotate_right(t, n->parent);
    }
}
```

Again, the function calls all use tail recursion, so the algorithm is in-place. Additionally, no recursive calls will be made after a rotation, so no more than three rotations are made.

[edit] Files

As we went along, we have separated the public interface (to be used by the client) from the private (static) helper function prototypes. We place the type definitions and public interface in the header file and the helper function prototypes and function definitions in the source file:

```
<<rbtree.h>>=
#ifndef _RBTREE_H_

public type definitions

public function prototypes

#endif

<<rbtree.c>>=
#include "rbtree.h"
include files

private type definitions
```

private function prototypes

node relationships
verify properties functions
create operation
node constructor
lookup operation
rotation operations
replace node operation
insertion operation
deletion operation

[edit] Test driver

To ensure that all the cases of the complex insert and delete operations are exercised, we will perform a large number of operations on some simple integer data. All properties are verified after each operation, providing strong evidence of correctness.

```
<<testmain.c>>=
#include "rbtree.h"
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> /* rand() */

static int compare_int(void* left, void* right);
static void print_tree(rbtree t);
static void print_tree_helper(rbtree_node n, int indent);

int compare_int(void* leftp, void* rightp) {
    int left = (int)leftp;
    int right = (int)rightp;
    if (left < right)
        return -1;
    else if (left > right)
        return 1;
    else {
        assert (left == right);
        return 0;
    }
}

#define INDENT_STEP  4

void print_tree_helper(rbtree_node n, int indent);

void print_tree(rbtree t) {
    print_tree_helper(t->root, 0);
    puts("");
}
```

```
void print_tree_helper(rbtree_node n, int indent) {
    int i;
    if (n == NULL) {
        fputs("<empty tree>", stdout);
        return;
    }
    if (n->right != NULL) {
        print_tree_helper(n->right, indent + INDENT_STEP);
    }
    for(i=0; i<indent; i++)
        fputs(" ", stdout);
    if (n->color == BLACK)
        printf("%d\n", (int)n->key);
    else
        printf("<%d>\n", (int)n->key);
    if (n->left != NULL) {
        print_tree_helper(n->left, indent + INDENT_STEP);
    }
}

int main() {
    int i;
    rbtree t = rbtree_create();
    print_tree(t);

    for(i=0; i<5000; i++) {
        int x = rand() % 10000;
        int y = rand() % 10000;
#ifdef TRACE
        print_tree(t);
        printf("Inserting %d -> %d\n\n", x, y);
#endif
        rbtree_insert(t, (void*)x, (void*)y, compare_int);
        assert(rbtree_lookup(t, (void*)x, compare_int) == (void*)y);
    }
    for(i=0; i<60000; i++) {
        int x = rand() % 10000;
#ifdef TRACE
        print_tree(t);
        printf("Deleting key %d\n\n", x);
#endif
        rbtree_delete(t, (void*)x, compare_int);
    }
    return 0;
}
```

The TRACE preprocessor constant allows us to turn on code printing the tree after each operation. This allows for a sanity check that the tree looks as we expect, as well as providing a way to visualize the results of the operation. We print the right subtree before the left subtree so that the tree is displayed sideways.

[edit] Performance

With property verification and tracing turned off, this code performs quite efficiently. When compiled with gcc with the -O3 flag on a 2.4 GhZ CPU running Gentoo Linux, it was able to perform 500,000 consecutive insertions followed by 600,000 consecutive deletions in only 2.1 seconds. Its primary limitation is the same as that of the C library function `qsort()`, that it must perform frequent calls to the comparison function through a function pointer, incurring function call overhead. Red-black tree (C Plus Plus) overcomes this by using templates which, like `std::sort`, allow the comparison function to be inlined.

Download code

Retrieved from "http://en.literateprograms.org/index.php?title=Red-black_tree_(C)&oldid=19567"
Categories:

- Red-black tree
- Programming language:C
- Environment:Portable

Navigation menu

Personal tools

- Create account
- Log in

Namespaces

- Page
- Discussion

Variants

Views

- Read
- Edit
- View history
- Download code

Actions

Search

Go Search

Navigation

- Welcome
- Community portal
- Recent changes
- New articles
- Random article
- Help

Toolbox

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information

- This page was last modified on 12 November 2013, at 01:10.

- Privacy policy
- About LiteratePrograms
- Disclaimers