红黑树的 c 实现源码与剖析

原作者：那谁
源码剖析作者：July
==========
July 说明：
由于原来的程序没有任何一行注释，我把它深入剖析，并一行一行的添加了注释，
详情请参见此文：
教你彻底实现红黑树：红黑树的 c 源码实现与剖析
http://blog.csdn.net/v_JULY_v/archive/2011/01/03/6114226.aspx
关于红黑树系列的教程，还可看下以下俩篇文章：
教你透彻了解红黑树：
http://blog.csdn.net/v_JULY_v/archive/2010/12/29/6105630.aspx
红黑树算法的层层剖析与逐步实现
http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx

Ok，君自看。

//以下是最初的源程序。
//如果你看不太懂，那么就证明了我所做的源码剖析工作有意义了。:D 。
//详情，参见 My Blog[谷歌或百度搜"结构之法"]
//http://blog.csdn.net/v_JULY_v

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int  key_t;
typedef int  data_t;

typedef enum color_t
{
    RED = 0,
    BLACK = 1
}color_t;

typedef struct rb_node_t
{
    struct rb_node_t *left, *right, *parent;
    key_t key;
    data_t data;
    color_t color;
```

```
}rb_node_t;

/* forward declaration */
rb_node_t* rb_insert(key_t key, data_t data, rb_node_t* root);
rb_node_t* rb_search(key_t key, rb_node_t* root);
rb_node_t* rb_erase(key_t key, rb_node_t* root);

int main()
{
    int i, count = 900000;
    key_t key;
    rb_node_t* root = NULL, *node = NULL;

     srand(time(NULL));
    for (i = 1; i < count; ++i)
    {
        key = rand() % count;
        if ((root = rb_insert(key, i, root)))
        {
            printf("[i = %d] insert key %d success!\n", i, key);
        }
        else
        {
            printf("[i = %d] insert key %d error!\n", i, key);
            exit(-1);
        }

        if ((node = rb_search(key, root)))
        {
            printf("[i = %d] search key %d success!\n", i, key);
        }
        else
        {
            printf("[i = %d] search key %d error!\n", i, key);
            exit(-1);
        }
        if (!(i % 10))
        {
            if ((root = rb_erase(key, root)))
            {
                printf("[i = %d] erase key %d success\n", i, key);
            }
            else
            {
```

```
                printf("[i = %d] erase key %d error\n", i, key);
            }
        }
    }

    return 0;
}

static rb_node_t* rb_new_node(key_t key, data_t data)
{
    rb_node_t *node = (rb_node_t*)malloc(sizeof(struct rb_node_t));

    if (!node)
    {
        printf("malloc error!\n");
        exit(-1);
    }
    node->key = key, node->data = data;

    return node;
}

/*----------------------------------------------------------
|   node                right
|   / \      ==>        / \
|   a   right         node   y
|         / \          / \
|         b   y        a   b      //原程序给的这丁点注释错了，我已修正。
----------------------------------------------------------*/
static rb_node_t* rb_rotate_left(rb_node_t* node, rb_node_t* root)
{
    rb_node_t* right = node->right;

    if ((node->right = right->left))
    {
        right->left->parent = node;
    }
    right->left = node;

    if ((right->parent = node->parent))
    {
        if (node == node->parent->right)
        {
            node->parent->right = right;
```

```
            }
            else
            {
                node->parent->left = right;
            }
        }
        else
        {
            root = right;
        }
        node->parent = right;

        return root;
}


/*----------------------------------------------------
|            node                  left
|            / \                  / \
|       left   y     ==>       a       node
|      / \                             / \
|    a     b                           b     y    //右旋与左旋差不多，分析略过
----------------------------------------------------*/
static rb_node_t* rb_rotate_right(rb_node_t* node, rb_node_t* root)
{
        rb_node_t* left = node->left;

        if ((node->left = left->right))
        {
            left->right->parent = node;
        }
        left->right = node;

        if ((left->parent = node->parent))
        {
            if (node == node->parent->right)
            {
                node->parent->right = left;
            }
            else
            {
                node->parent->left = left;
            }
        }
```

```
        else
        {
                root = left;
        }
        node->parent = left;

        return root;
}

static rb_node_t* rb_insert_rebalance(rb_node_t *node, rb_node_t *root)
{
        rb_node_t *parent, *gparent, *uncle, *tmp;

        while ((parent = node->parent) && parent->color == RED)
        {
                gparent = parent->parent;

                if (parent == gparent->left)
                {
                        uncle = gparent->right;
                        if (uncle && uncle->color == RED)
                        {
                                uncle->color = BLACK;
                                parent->color = BLACK;
                                gparent->color = RED;
                                node = gparent;
                        }
                        else
                        {
                                if (parent->right == node)
                                {
                                        root = rb_rotate_left(parent, root);
                                        tmp = parent;
                                        parent = node;
                                        node = tmp;
                                }

                                parent->color = BLACK;
                                gparent->color = RED;
                                root = rb_rotate_right(gparent, root);
                        }
                }
                 else
                 {
```

```
                uncle = gparent->left;
                if (uncle && uncle->color == RED)
                {
                        uncle->color = BLACK;
                        parent->color = BLACK;
                        gparent->color = RED;
                        node = gparent;
                }
                else
                {
                        if (parent->left == node)
                        {
                                root = rb_rotate_right(parent, root);
                                tmp = parent;
                                parent = node;
                                node = tmp;
                        }

                        parent->color = BLACK;
                        gparent->color = RED;
                        root = rb_rotate_left(gparent, root);
                }
        }
    }

    root->color = BLACK;

    return root;
}

static rb_node_t* rb_erase_rebalance(rb_node_t *node, rb_node_t *parent, rb_node_t *root)
{
    rb_node_t *other, *o_left, *o_right;

    while ((!node || node->color == BLACK) && node != root)
    {
        if (parent->left == node)
        {
                other = parent->right;
                if (other->color == RED)
                {
                        other->color = BLACK;
                        parent->color = RED;
                        root = rb_rotate_left(parent, root);
```

```
            other = parent->right;
        }
        if ((!other->left || other->left->color == BLACK) &&
            (!other->right || other->right->color == BLACK))
        {
            other->color = RED;
            node = parent;
            parent = node->parent;
        }
        else
        {
            if (!other->right || other->right->color == BLACK)
            {
                if ((o_left = other->left))
                {
                    o_left->color = BLACK;
                }
                other->color = RED;
                root = rb_rotate_right(other, root);
                other = parent->right;
            }
            other->color = parent->color;
            parent->color = BLACK;
            if (other->right)
            {
                other->right->color = BLACK;
            }
            root = rb_rotate_left(parent, root);
            node = root;
            break;
        }
    }
    else
    {
        other = parent->left;
        if (other->color == RED)
        {
            other->color = BLACK;
            parent->color = RED;
            root = rb_rotate_right(parent, root);
            other = parent->left;
        }
        if ((!other->left || other->left->color == BLACK) &&
            (!other->right || other->right->color == BLACK))
```

```
                {
                    other->color = RED;
                    node = parent;
                    parent = node->parent;
                }
                else
                {
                    if (!other->left || other->left->color == BLACK)
                    {
                        if ((o_right = other->right))
                        {
                            o_right->color = BLACK;
                        }
                        other->color = RED;
                        root = rb_rotate_left(other, root);
                        other = parent->left;
                    }
                    other->color = parent->color;
                    parent->color = BLACK;
                    if (other->left)
                    {
                        other->left->color = BLACK;
                    }
                    root = rb_rotate_right(parent, root);
                    node = root;
                    break;
                }
            }
        }

        if (node)
        {
            node->color = BLACK;
        }

        return root;
}

static rb_node_t* rb_search_auxiliary(key_t key, rb_node_t* root, rb_node_t** save)
{
        rb_node_t *node = root, *parent = NULL;
        int ret;

        while (node)
```

```
        {
                parent = node;
                ret = node->key - key;
                if (0 < ret)
                {
                        node = node->left;
                }
                else if (0 > ret)
                {
                        node = node->right;
                }
                else
                {
                        return node;
                }
        }

        if (save)
        {
                *save = parent;
        }

        return NULL;
}

rb_node_t* rb_insert(key_t key, data_t data, rb_node_t* root)
{
        rb_node_t *parent = NULL, *node;

        parent = NULL;
        if ((node = rb_search_auxiliary(key, root, &parent)))
        {
                return root;
        }

        node = rb_new_node(key, data);
        node->parent = parent;
          node->left = node->right = NULL;
        node->color = RED;

        if (parent)
        {
                if (parent->key > key)
                {
```

```
                parent->left = node;
            }
            else
            {
                parent->right = node;
            }
        }
        else
        {
            root = node;
        }

        return rb_insert_rebalance(node, root);
}

rb_node_t* rb_search(key_t key, rb_node_t* root)
{
        return rb_search_auxiliary(key, root, NULL);
}

rb_node_t* rb_erase(key_t key, rb_node_t *root)
{
        rb_node_t *child, *parent, *old, *left, *node;
        color_t color;

        if (!(node = rb_search_auxiliary(key, root, NULL)))
        {
            printf("key %d is not exist!\n");
            return root;
        }

        old = node;

        if (node->left && node->right)
        {
            node = node->right;
            while ((left = node->left) != NULL)
            {
                node = left;
            }
            child = node->right;
            parent = node->parent;
            color = node->color;
```

```
if (child)
{
	child->parent = parent;
}
if (parent)
{
	if (parent->left == node)
	{
		parent->left = child;
	}
	else
	{
		parent->right = child;
	}
}
else
{
	root = child;
}

if (node->parent == old)
{
	parent = node;
}

node->parent = old->parent;
node->color = old->color;
node->right = old->right;
node->left = old->left;

if (old->parent)
{
	if (old->parent->left == old)
	{
		old->parent->left = node;
	}
	else
	{
		old->parent->right = node;
	}
}
 else
{
	root = node;
```

```
        }

        old->left->parent = node;
        if (old->right)
        {
                old->right->parent = node;
        }
}
else
{
        if (!node->left)
        {
                child = node->right;
        }
        else if (!node->right)
        {
                child = node->left;
        }
        parent = node->parent;
        color = node->color;

        if (child)
        {
                child->parent = parent;
        }
        if (parent)
        {
                if (parent->left == node)
                {
                        parent->left = child;
                }
                else
                {
                        parent->right = child;
                }
        }
        else
        {
                root = child;
        }
}

free(old);
```

```
    if (color == BLACK)
    {
        root = rb_erase_rebalance(child, parent, root);
    }

    return root;
}
```

程序完。

==========

运行结果：

```
[i = 1] insert key 52 success!
[i = 1] search key 52 success!
[i = 2] insert key 80 success!
[i = 2] search key 80 success!
[i = 3] insert key 2 success!
[i = 3] search key 2 success!
[i = 4] insert key 81 success!
[i = 4] search key 81 success!
[i = 5] insert key 96 success!
[i = 5] search key 96 success!
[i = 6] insert key 16 success!
[i = 6] search key 16 success!
[i = 7] insert key 92 success!
[i = 7] search key 92 success!
[i = 8] insert key 20 success!
[i = 8] search key 20 success!
[i = 9] insert key 63 success!
[i = 9] search key 63 success!
[i = 10] insert key 16 success!
[i = 10] search key 16 success!
[i = 10] erase key 16 success
[i = 11] insert key 99 success!
[i = 11] search key 99 success!
[i = 12] insert key 36 success!
[i = 12] search key 36 success!
[i = 13] insert key 36 success!
[i = 13] search key 36 success!
[i = 14] insert key 33 success!
[i = 14] search key 33 success!
[i = 15] insert key 26 success!
[i = 15] search key 26 success!
[i = 16] insert key 84 success!
[i = 16] search key 84 success!
[i = 17] insert key 89 success!
```

[i = 17] search key 89 success!
[i = 18] insert key 13 success!
[i = 18] search key 13 success!
[i = 19] insert key 63 success!
[i = 19] search key 63 success!
[i = 20] insert key 20 success!
[i = 20] search key 20 success!
[i = 20] erase key 20 success
[i = 21] insert key 9 success!
[i = 21] search key 9 success!
[i = 22] insert key 27 success!
[i = 22] search key 27 success!
[i = 23] insert key 57 success!
[i = 23] search key 57 success!
[i = 24] insert key 51 success!
[i = 24] search key 51 success!
[i = 25] insert key 1 success!
[i = 25] search key 1 success!
[i = 26] insert key 60 success!
[i = 26] search key 60 success!
[i = 27] insert key 45 success!
[i = 27] search key 45 success!
[i = 28] insert key 2 success!
[i = 28] search key 2 success!
[i = 29] insert key 20 success!
[i = 29] search key 20 success!
[i = 30] insert key 54 success!
[i = 30] search key 54 success!
[i = 30] erase key 54 success
[i = 31] insert key 41 success!
[i = 31] search key 41 success!
[i = 32] insert key 10 success!
[i = 32] search key 10 success!
[i = 33] insert key 74 success!
[i = 33] search key 74 success!
[i = 34] insert key 68 success!
[i = 34] search key 68 success!
[i = 35] insert key 9 success!
[i = 35] search key 9 success!
[i = 36] insert key 12 success!
[i = 36] search key 12 success!
[i = 37] insert key 7 success!
[i = 37] search key 7 success!
[i = 38] insert key 38 success!

```
[i = 38] search key 38 success!
[i = 39] insert key 83 success!
[i = 39] search key 83 success!
[i = 40] insert key 42 success!
[i = 40] search key 42 success!
[i = 40] erase key 42 success
[i = 41] insert key 7 success!
[i = 41] search key 7 success!
[i = 42] insert key 98 success!
[i = 42] search key 98 success!
[i = 43] insert key 18 success!
[i = 43] search key 18 success!
[i = 44] insert key 37 success!
[i = 44] search key 37 success!
[i = 45] insert key 72 success!
[i = 45] search key 72 success!
[i = 46] insert key 91 success!
[i = 46] search key 91 success!
[i = 47] insert key 89 success!
[i = 47] search key 89 success!
[i = 48] insert key 97 success!
[i = 48] search key 97 success!
.....
Press any key to continue
```

为了给你个对比，我摘一段我给的源码剖析[注释完美]

五、红黑树的 3 种插入情况

接下来，咱们重点分析针对红黑树插入的 3 种情况，而进行的修复工作。

```
//--------------------------------------------------------
//红黑树修复插入的 3 种情况
//为了表示方面下面的注释中，也为了让下述代码与我的俩篇文章相对应，
//用 z 表示当前结点，p[z]表示父母、p[p[z]]表示祖父、y 表示叔叔。
//--------------------------------------------------------
static rb_node_t* rb_insert_rebalance(rb_node_t *node, rb_node_t *root)
{
    rb_node_t *parent, *gparent, *uncle, *tmp;   //父母 p[z]、祖父 p[p[z]]、叔叔 y、临时结点
*tmp

    while ((parent = node->parent) && parent->color == RED)
    {       //parent 为 node 的父母，且当父母的颜色为红时
        gparent = parent->parent;    //gparent 为祖父
```

```
        if (parent == gparent->left)    //当祖父的左孩子即为父母时。
                                        //其实上述几行语句，无非就是理顺孩子、父母、祖父的关系。:D。
        {
            uncle = gparent->right;    //定义叔叔的概念，叔叔 y 就是父母的右孩子。

            if (uncle && uncle->color == RED) //情况 1：z 的叔叔 y 是红色的
            {
                uncle->color = BLACK;      //将叔叔结点 y 着为黑色
                parent->color = BLACK;    //z 的父母 p[z]也着为黑色。解决 z，p[z]都是红色的问题。
                gparent->color = RED;
                node = gparent;         //将祖父当做新增结点 z，指针 z 上移俩层，且着为红色。
    //上述情况 1 中，只考虑了 z 作为父母的右孩子的情况。
            }
            else                                //情况 2：z 的叔叔 y 是黑色的，
            {
                if (parent->right == node)    //且 z 为右孩子
                {
                    root = rb_rotate_left(parent, root); //左旋[结点 z，与父母结点]
                    tmp = parent;
                    parent = node;
                    node = tmp;         //parent 与 node  互换角色
                }
                            //情况 3：z 的叔叔 y 是黑色的，此时 z 成为了左孩子。
                                //注意，1：情况 3 是由上述情况 2 变化而来的。
                                //.....2：z 的叔叔总是黑色的，否则就是情况 1 了。
                parent->color = BLACK;      //z 的父母 p[z]着为黑色
                gparent->color = RED;       //原祖父结点着为红色
                root = rb_rotate_right(gparent, root); //右旋[结点 z，与祖父结点]
            }
        }

        else
        {
    //这部分是特别为情况 1 中，z 作为左孩子情况，而写的。
            uncle = gparent->left;    //祖父的左孩子作为叔叔结点。[原理还是与上部分一样的]
            if (uncle && uncle->color == RED)    //情况 1：z 的叔叔 y 是红色的
            {
                uncle->color = BLACK;
                parent->color = BLACK;
                gparent->color = RED;
```

```
                node = gparent;                 //同上。
            }
            else                                        //情况 2：z 的叔叔 y 是黑色的，
            {
                if (parent->left == node)    //且 z 为左孩子
                {
                    root = rb_rotate_right(parent, root);   //以结点 parent、root 右旋
                    tmp = parent;
                    parent = node;
                    node = tmp;             //parent 与 node  互换角色
                }
                  //经过情况 2 的变化，成为了情况 3.
                parent->color = BLACK;
                gparent->color = RED;
                root = rb_rotate_left(gparent, root);     //以结点 gparent 和 root 左旋
            }
        }
    }

    root->color = BLACK; //根结点，不论怎样，都得置为黑色。
    return root;          //返回根结点。
}
//:D。这下，你应该发现我所添加的注释的价值了。
```

本文来自CSDN博客，转载请标明出处：
http://blog.csdn.net/v_JULY_v/archive/2011/01/03/6114226.aspx

一切的详情请参见此文：
教你彻底实现红黑树：红黑树的 c 源码实现与剖析
http://blog.csdn.net/v_JULY_v/archive/2011/01/03/6114226.aspx
关于红黑树系列的教程，还可看下以下俩篇文章：
教你透彻了解红黑树：
http://blog.csdn.net/v_JULY_v/archive/2010/12/29/6105630.aspx
红黑树算法的层层剖析与逐步实现
http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx

我博客里还有有关微软等公司数据结构+算法面试 100 题的资料，
详情，参见 My Blog：
http://blog.csdn.net/v_JULY_v

-----July、二零一一年一月三日。