**前言：**    红黑树作为一种经典而高级的[**数据结构**](http://lib.csdn.net/base/datastructure)，相信，已经被不少人实现过，但不是因为程序不够完善而无法运行，就是因为程序完全没有注释，初学者根本就看不懂。  
    此份红黑树的c源码最初从[**Linux**](http://lib.csdn.net/base/linux)-lib-rbtree.c而来，后经一网友那谁（<http://www.cppblog.com/converse/>）用c写了出来。在此，向原作者表示敬意。

    考虑到原来的程序没有注释，所以我特把这份源码放到编译器里，一行一行的完善，一行一行的给它添加注释，至此，红黑树c带注释的源码，就摆在了您眼前，有不妥、不正之处，还望不吝指正。  
------------

红黑树的六篇文章：

[**1、教你透彻了解红黑树**](http://blog.csdn.net/v_JULY_v/archive/2010/12/29/6105630.aspx)

[**2、红黑树算法的实现与剖析**](http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx)

[**3、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)

[**4、一步一图一代码，R-B Tree**](http://blog.csdn.net/v_JULY_v/archive/2011/01/09/6124989.aspx)

[**5、红黑树插入和删除结点的全程演示**](http://blog.csdn.net/v_JULY_v/archive/2011/03/28/6284050.aspx)

[**6、红黑树的c++完整实现源码**](http://blog.csdn.net/v_JULY_v/archive/2011/03/29/6285620.aspx)

-------------------------

ok，咱们开始吧。  
    相信，经过我前俩篇博文对红黑树的介绍，你应该对红黑树有了透彻的理解了（没看过的朋友，可事先查上面的倆篇文章，或与此文的源码剖析对应着看）。

    本套源码剖析把重点放在红黑树的3种插入情况，与红黑树的4种删除情况。其余的能从略则尽量简略。

目录：  
一、左旋代码分析  
二、右旋  
三、红黑树查找结点  
四、红黑树的插入  
五、红黑树的3种插入情况  
六、红黑树的删除  
七、红黑树的4种删除情况  
八、[**测试**](http://lib.csdn.net/base/softwaretest)用例

好的，咱们还是先从树的左旋、右旋代码，开始（大部分分析，直接给注释）：

1. //一、左旋代码分析
2. /\*-----------------------------------------------------------
3. |   node           right
4. |   / /    ==>     / /
5. |   a  right     node  y
6. |       / /       / /
7. |       b  y     a   b    //左旋
8. -----------------------------------------------------------\*/
9. **static** rb\_node\_t\* rb\_rotate\_left(rb\_node\_t\* node, rb\_node\_t\* root)
10. {
11. rb\_node\_t\* right = node->right;    //指定指针指向 right<--node->right
13. **if** ((node->right = right->left))
14. {
15. right->left->parent = node;  //好比上面的注释图，node成为b的父母
16. }
17. right->left = node;   //node成为right的左孩子
19. **if** ((right->parent = node->parent))
20. {
21. **if** (node == node->parent->right)
22. {
23. node->parent->right = right;
24. }
25. **else**
26. {
27. node->parent->left = right;
28. }
29. }
30. **else**
31. {
32. root = right;
33. }
34. node->parent = right;  //right成为node的父母
36. **return** root;
37. }

40. //二、右旋
41. /\*-----------------------------------------------------------
42. |       node            left
43. |       / /             / /
44. |    left  y   ==>    a    node
45. |   / /                    / /
46. |  a   b                  b   y  //右旋与左旋差不多，分析略过
47. -----------------------------------------------------------\*/
48. **static** rb\_node\_t\* rb\_rotate\_right(rb\_node\_t\* node, rb\_node\_t\* root)
49. {
50. rb\_node\_t\* left = node->left;
52. **if** ((node->left = left->right))
53. {
54. left->right->parent = node;
55. }
56. left->right = node;
58. **if** ((left->parent = node->parent))
59. {
60. **if** (node == node->parent->right)
61. {
62. node->parent->right = left;
63. }
64. **else**
65. {
66. node->parent->left = left;
67. }
68. }
69. **else**
70. {
71. root = left;
72. }
73. node->parent = left;
75. **return** root;
76. }

79. //三、红黑树查找结点
80. //----------------------------------------------------
81. //rb\_search\_auxiliary：查找
82. //rb\_node\_t\* rb\_search：返回找到的结点
83. //----------------------------------------------------
84. **static** rb\_node\_t\* rb\_search\_auxiliary(key\_t key, rb\_node\_t\* root, rb\_node\_t\*\* save)
85. {
86. rb\_node\_t \*node = root, \*parent = NULL;
87. **int** ret;
89. **while** (node)
90. {
91. parent = node;
92. ret = node->key - key;
93. **if** (0 < ret)
94. {
95. node = node->left;
96. }
97. **else** **if** (0 > ret)
98. {
99. node = node->right;
100. }
101. **else**
102. {
103. **return** node;
104. }
105. }
107. **if** (save)
108. {
109. \*save = parent;
110. }
112. **return** NULL;
113. }
115. //返回上述rb\_search\_auxiliary查找结果
116. rb\_node\_t\* rb\_search(key\_t key, rb\_node\_t\* root)
117. {
118. **return** rb\_search\_auxiliary(key, root, NULL);
119. }

122. //四、红黑树的插入
123. //---------------------------------------------------------
124. //红黑树的插入结点
125. rb\_node\_t\* rb\_insert(key\_t key, data\_t data, rb\_node\_t\* root)
126. {
127. rb\_node\_t \*parent = NULL, \*node;
129. parent = NULL;
130. **if** ((node = rb\_search\_auxiliary(key, root, &parent)))  //调用rb\_search\_auxiliary找到插入结
132. 点的地方
133. {
134. **return** root;
135. }
137. node = rb\_new\_node(key, data);  //分配结点
138. node->parent = parent;
139. node->left = node->right = NULL;
140. node->color = RED;
142. **if** (parent)
143. {
144. **if** (parent->key > key)
145. {
146. parent->left = node;
147. }
148. **else**
149. {
150. parent->right = node;
151. }
152. }
153. **else**
154. {
155. root = node;
156. }
158. **return** rb\_insert\_rebalance(node, root);   //插入结点后，调用rb\_insert\_rebalance修复红黑树
160. 的性质
161. }

164. //五、红黑树的3种插入情况
165. //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。
166. //--------------------------------------------------------------
167. //红黑树修复插入的3种情况
168. //为了在下面的注释中表示方便，也为了让下述代码与我的倆篇文章相对应，
169. //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。
170. //--------------------------------------------------------------
171. **static** rb\_node\_t\* rb\_insert\_rebalance(rb\_node\_t \*node, rb\_node\_t \*root)
172. {
173. rb\_node\_t \*parent, \*gparent, \*uncle, \*tmp;  //父母p[z]、祖父p[p[z]]、叔叔y、临时结点\*tmp
175. **while** ((parent = node->parent) && parent->color == RED)
176. {     //parent 为node的父母，且当父母的颜色为红时
177. gparent = parent->parent;   //gparent为祖父
179. **if** (parent == gparent->left)  //当祖父的左孩子即为父母时。
180. //其实上述几行语句，无非就是理顺孩子、父母、祖父的关系。:D。
181. {
182. uncle = gparent->right;  //定义叔叔的概念，叔叔y就是父母的右孩子。
184. **if** (uncle && uncle->color == RED) //情况1：z的叔叔y是红色的
185. {
186. uncle->color = BLACK;   //将叔叔结点y着为黑色
187. 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。
188. gparent->color = RED;
189. node = gparent;     //将祖父当做新增结点z，指针z上移俩层，且着为红色。
190. //上述情况1中，只考虑了z作为父母的右孩子的情况。
191. }
192. **else**                     //情况2：z的叔叔y是黑色的，
193. {
194. **if** (parent->right == node)  //且z为右孩子
195. {
196. root = rb\_rotate\_left(parent, root); //左旋[结点z，与父母结点]
197. tmp = parent;
198. parent = node;
199. node = tmp;     //parent与node 互换角色
200. }
201. //情况3：z的叔叔y是黑色的，此时z成为了左孩子。
202. //注意，1：情况3是由上述情况2变化而来的。
203. //......2：z的叔叔总是黑色的，否则就是情况1了。
204. parent->color = BLACK;   //z的父母p[z]着为黑色
205. gparent->color = RED;    //原祖父结点着为红色
206. root = rb\_rotate\_right(gparent, root); //右旋[结点z，与祖父结点]
207. }
208. }
210. **else**
211. {
212. // if (parent == gparent->right) 当祖父的右孩子即为父母时。（解释请看本文评论下第23楼，同时，感谢SupremeHover指正！）
213. 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]
214. **if** (uncle && uncle->color == RED)  //情况1：z的叔叔y是红色的
215. {
216. uncle->color = BLACK;
217. parent->color = BLACK;
218. gparent->color = RED;
219. node = gparent;           //同上。
220. }
221. **else**                               //情况2：z的叔叔y是黑色的，
222. {
223. **if** (parent->left == node)  //且z为左孩子
224. {
225. root = rb\_rotate\_right(parent, root);  //以结点parent、root右旋
226. tmp = parent;
227. parent = node;
228. node = tmp;       //parent与node 互换角色
229. }
230. //经过情况2的变化，成为了情况3.
231. parent->color = BLACK;
232. gparent->color = RED;
233. root = rb\_rotate\_left(gparent, root);   //以结点gparent和root左旋
234. }
235. }
236. }
238. root->color = BLACK; //根结点，不论怎样，都得置为黑色。
239. **return** root;      //返回根结点。
240. }

243. //六、红黑树的删除
244. //------------------------------------------------------------
245. //红黑树的删除结点
246. rb\_node\_t\* rb\_erase(key\_t key, rb\_node\_t \*root)
247. {
248. rb\_node\_t \*child, \*parent, \*old, \*left, \*node;
249. color\_t color;
251. **if** (!(node = rb\_search\_auxiliary(key, root, NULL)))  //调用rb\_search\_auxiliary查找要删除的
253. 结点
254. {
255. printf("key %d is not exist!/n");
256. **return** root;
257. }
259. old = node;
261. **if** (node->left && node->right)
262. {
263. node = node->right;
264. **while** ((left = node->left) != NULL)
265. {
266. node = left;
267. }
268. child = node->right;
269. parent = node->parent;
270. color = node->color;
272. **if** (child)
273. {
274. child->parent = parent;
275. }
276. **if** (parent)
277. {
278. **if** (parent->left == node)
279. {
280. parent->left = child;
281. }
282. **else**
283. {
284. parent->right = child;
285. }
286. }
287. **else**
288. {
289. root = child;
290. }
292. **if** (node->parent == old)
293. {
294. parent = node;
295. }
297. node->parent = old->parent;
298. node->color = old->color;
299. node->right = old->right;
300. node->left = old->left;
302. **if** (old->parent)
303. {
304. **if** (old->parent->left == old)
305. {
306. old->parent->left = node;
307. }
308. **else**
309. {
310. old->parent->right = node;
311. }
312. }
313. **else**
314. {
315. root = node;
316. }
318. old->left->parent = node;
319. **if** (old->right)
320. {
321. old->right->parent = node;
322. }
323. }
324. **else**
325. {
326. **if** (!node->left)
327. {
328. child = node->right;
329. }
330. **else** **if** (!node->right)
331. {
332. child = node->left;
333. }
334. parent = node->parent;
335. color = node->color;
337. **if** (child)
338. {
339. child->parent = parent;
340. }
341. **if** (parent)
342. {
343. **if** (parent->left == node)
344. {
345. parent->left = child;
346. }
347. **else**
348. {
349. parent->right = child;
350. }
351. }
352. **else**
353. {
354. root = child;
355. }
356. }
358. free(old);
360. **if** (color == BLACK)
361. {
362. root = rb\_erase\_rebalance(child, parent, root); //调用rb\_erase\_rebalance来恢复红黑树性
364. 质
365. }
367. **return** root;
368. }

371. //七、红黑树的4种删除情况
372. //----------------------------------------------------------------
373. //红黑树修复删除的4种情况
374. //为了表示下述注释的方便，也为了让下述代码与我的倆篇文章相对应，
375. //x表示要删除的结点，\*other、w表示兄弟结点，
376. //----------------------------------------------------------------
377. **static** rb\_node\_t\* rb\_erase\_rebalance(rb\_node\_t \*node, rb\_node\_t \*parent, rb\_node\_t \*root)
378. {
379. rb\_node\_t \*other, \*o\_left, \*o\_right;   //x的兄弟\*other，兄弟左孩子\*o\_left,\*o\_right
381. **while** ((!node || node->color == BLACK) && node != root)
382. {
383. **if** (parent->left == node)
384. {
385. other = parent->right;
386. **if** (other->color == RED)   //情况1：x的兄弟w是红色的
387. {
388. other->color = BLACK;
389. parent->color = RED;   //上俩行，改变颜色，w->黑、p[x]->红。
390. root = rb\_rotate\_left(parent, root);  //再对p[x]做一次左旋
391. other = parent->right;  //x的新兄弟new w 是旋转之前w的某个孩子。其实就是左旋后
393. 的效果。
394. }
395. **if** ((!other->left || other->left->color == BLACK) &&
396. (!other->right || other->right->color == BLACK))
397. //情况2：x的兄弟w是黑色，且w的俩个孩子也
399. 都是黑色的
401. {                         //由于w和w的俩个孩子都是黑色的，则在x和w上得去掉一黑色，
402. other->color = RED;   //于是，兄弟w变为红色。
403. node = parent;    //p[x]为新结点x
404. parent = node->parent;  //x<-p[x]
405. }
406. **else**                       //情况3：x的兄弟w是黑色的，
407. {                          //且，w的左孩子是红色，右孩子为黑色。
408. **if** (!other->right || other->right->color == BLACK)
409. {
410. **if** ((o\_left = other->left))   //w和其左孩子left[w]，颜色交换。
411. {
412. o\_left->color = BLACK;    //w的左孩子变为由黑->红色
413. }
414. other->color = RED;           //w由黑->红
415. root = rb\_rotate\_right(other, root);  //再对w进行右旋，从而红黑性质恢复。
416. other = parent->right;        //变化后的，父结点的右孩子，作为新的兄弟结点
418. w。
419. }
420. //情况4：x的兄弟w是黑色的
422. other->color = parent->color;  //把兄弟节点染成当前节点父节点的颜色。
423. parent->color = BLACK;  //把当前节点父节点染成黑色
424. **if** (other->right)      //且w的右孩子是红
425. {
426. other->right->color = BLACK;  //兄弟节点w右孩子染成黑色
427. }
428. root = rb\_rotate\_left(parent, root);  //并再做一次左旋
429. node = root;   //并把x置为根。
430. **break**;
431. }
432. }
433. //下述情况与上述情况，原理一致。分析略。
434. **else**
435. {
436. other = parent->left;
437. **if** (other->color == RED)
438. {
439. other->color = BLACK;
440. parent->color = RED;
441. root = rb\_rotate\_right(parent, root);
442. other = parent->left;
443. }
444. **if** ((!other->left || other->left->color == BLACK) &&
445. (!other->right || other->right->color == BLACK))
446. {
447. other->color = RED;
448. node = parent;
449. parent = node->parent;
450. }
451. **else**
452. {
453. **if** (!other->left || other->left->color == BLACK)
454. {
455. **if** ((o\_right = other->right))
456. {
457. o\_right->color = BLACK;
458. }
459. other->color = RED;
460. root = rb\_rotate\_left(other, root);
461. other = parent->left;
462. }
463. other->color = parent->color;
464. parent->color = BLACK;
465. **if** (other->left)
466. {
467. other->left->color = BLACK;
468. }
469. root = rb\_rotate\_right(parent, root);
470. node = root;
471. **break**;
472. }
473. }
474. }
476. **if** (node)
477. {
478. node->color = BLACK;  //最后将node[上述步骤置为了根结点]，改为黑色。
479. }
480. **return** root;  //返回root
481. }

484. //八、测试用例
485. //主函数
486. **int** main()
487. {
488. **int** i, count = 100;
489. key\_t key;
490. rb\_node\_t\* root = NULL, \*node = NULL;
492. srand(time(NULL));
493. **for** (i = 1; i < count; ++i)
494. {
495. key = rand() % count;
496. **if** ((root = rb\_insert(key, i, root)))
497. {
498. printf("[i = %d] insert key %d success!/n", i, key);
499. }
500. **else**
501. {
502. printf("[i = %d] insert key %d error!/n", i, key);
503. exit(-1);
504. }
506. **if** ((node = rb\_search(key, root)))
507. {
508. printf("[i = %d] search key %d success!/n", i, key);
509. }
510. **else**
511. {
512. printf("[i = %d] search key %d error!/n", i, key);
513. exit(-1);
514. }
515. **if** (!(i % 10))
516. {
517. **if** ((root = rb\_erase(key, root)))
518. {
519. printf("[i = %d] erase key %d success/n", i, key);
520. }
521. **else**
522. {
523. printf("[i = %d] erase key %d error/n", i, key);
524. }
525. }
526. }
528. **return** 0;
529. }

ok，完。

后记：

一、欢迎任何人就此份源码，以及我的前述倆篇文章，进行讨论、提议。  
但任何人，引用此份源码剖析，必须得注明作者本人July以及出处。  
红黑树系列，已经写了三篇文章，相信，教你透彻了解红黑树的目的，应该达到了。