**前言：**  
    红黑树系列文章已经写到第5篇了。虽然第三篇文章：[**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)，用[**C语言**](http://lib.csdn.net/base/c)完整实现过红黑树，但个人感觉，代码还是不够清晰。特此，再奉献出一份c++的完整实现源码，以飨读者。

    此份c++实现源码，代码紧凑了许多，也清晰了不少，同时采取c++类实现的方式，代码也更容易维护以及重用。ok，有任何问题，欢迎指正。

**第一部分、红黑树的c++完整实现源码**

    本文包含红黑树c++实现的完整源码，所有的解释都含在注释中，所有的有关红黑树的原理及各种插入、删除操作的情况，都已在本人的红黑树系列的前4篇文章中，一一阐述。且在此红黑树系列第五篇文章中：[**红黑树从头至尾插入和删除结点的全程演示图**](http://blog.csdn.net/v_JULY_v/archive/2011/03/28/6284050.aspx)，把所有的插入、删除情况都一一展示尽了。  
    因此，有关红黑树的全部原理，请参考其它文章，重点可参考此文：[**红黑树算法的实现与剖析**](http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx)。因此，相关原理，本文不再赘述。

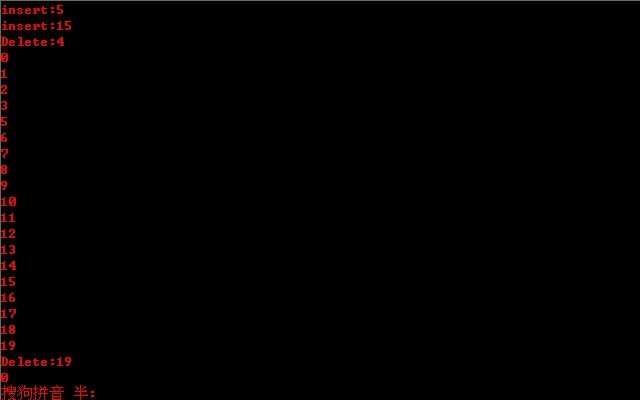
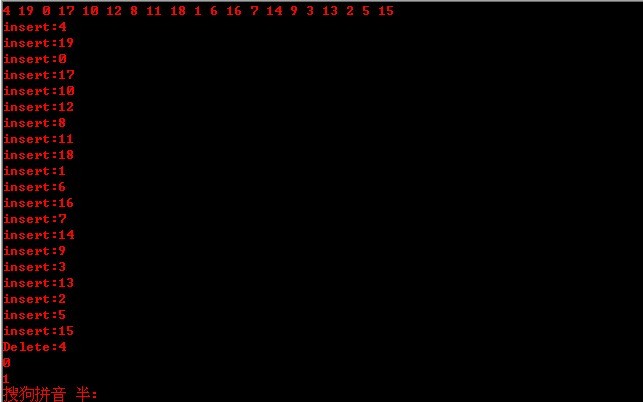
    ok，以下，即是红黑树c++实现的全部源码，先是RBTree.h，然后是RBTree.cpp。

**RBTree.h**

1. //file RBTree.h
2. //written by saturnman，20101008。
3. //updated by July，20110329。
4. /\*-----------------------------------------------
5. 版权声明：
6. July和saturnman对此份红黑树的c++实现代码享有全部的版权，
7. 谢绝转载，侵权必究。
8. ------------------------------------------------\*/
9. #ifndef \_RB\_TREE\_H\_
10. #define \_RB\_TREE\_H\_
11. #include<iostream>
12. #include<string>
13. #include<sstream>
14. #include<fstream>
15. **using** **namespace** std;
17. **template**<**class** KEY, **class** U>
18. **class** RB\_Tree
19. {
20. **private**:
21. RB\_Tree(**const** RB\_Tree& input){}
22. **const** RB\_Tree& operator=(**const** RB\_Tree& input){}
23. **private**:
24. **enum** COLOR{ RED, BLACK };
25. **class** RB\_Node
26. {
27. **public**:
28. RB\_Node()
29. {
30. //RB\_COLOR = BLACK;
31. right = NULL;
32. left = NULL;
33. parent = NULL;
34. }
35. COLOR RB\_COLOR;
36. RB\_Node\* right;
37. RB\_Node\* left;
38. RB\_Node\* parent;
39. KEY key;
40. U data;
41. };
42. **public**:
43. RB\_Tree()
44. {
45. **this**->m\_nullNode = **new** RB\_Node();
46. **this**->m\_root = m\_nullNode;
47. **this**->m\_nullNode->right = **this**->m\_root;
48. **this**->m\_nullNode->left = **this**->m\_root;
49. **this**->m\_nullNode->parent = **this**->m\_root;
50. **this**->m\_nullNode->RB\_COLOR = BLACK;
51. }
53. **bool** Empty()
54. {
55. **if** (**this**->m\_root == **this**->m\_nullNode)
56. {
57. **return** **true**;
58. }
59. **else**
60. {
61. **return** **false**;
62. }
63. }
65. //查找key
66. RB\_Node\* find(KEY key)
67. {
68. RB\_Node\* index = m\_root;
69. **while** (index != m\_nullNode)
70. {
71. **if** (key<index->key)
72. {
73. index = index->left;  //比当前的小，往左
74. }
75. **else** **if** (key>index->key)
76. {
77. index = index->right;  //比当前的大，往右
78. }
79. **else**
80. {
81. **break**;
82. }
83. }
84. **return** index;
85. }
87. //--------------------------插入结点总操作----------------------------------
88. //全部的工作，都在下述伪代码中：
89. /\*RB-INSERT(T, z)
90. 1  y ← nil[T]                 // y 始终指向 x 的父结点。
91. 2  x ← root[T]              // x 指向当前树的根结点，
92. 3  while x ≠ nil[T]
93. 4      do y ← x
94. 5         if key[z] < key[x]           //向左，向右..
95. 6            then x ← left[x]
96. 7            else x ← right[x]   //为了找到合适的插入点，x探路跟踪路径，直到x成为NIL 为止。
97. 8  p[z] ← y         //y置为 插入结点z 的父结点。
98. 9  if y = nil[T]
99. 10     then root[T] ← z
100. 11     else if key[z] < key[y]
101. 12             then left[y] ← z
102. 13             else right[y] ← z     //此 8-13行，置z 相关的指针。
103. 14  left[z] ← nil[T]
104. 15  right[z] ← nil[T]            //设为空，
105. 16  color[z] ← RED             //将新插入的结点z作为红色
106. 17  RB-INSERT-FIXUP(T, z)
107. \*/
108. //因为将z着为红色，可能会违反某一红黑性质，
109. //所以需要调用下面的RB-INSERT-FIXUP(T, z)来保持红黑性质。
110. **bool** Insert(KEY key, U data)
111. {
112. RB\_Node\* insert\_point = m\_nullNode;
113. RB\_Node\* index = m\_root;
114. **while** (index != m\_nullNode)
115. {
116. insert\_point = index;
117. **if** (key<index->key)
118. {
119. index = index->left;
120. }
121. **else** **if** (key>index->key)
122. {
123. index = index->right;
124. }
125. **else**
126. {
127. **return** **false**;
128. }
129. }
130. RB\_Node\* insert\_node = **new** RB\_Node();
131. insert\_node->key = key;
132. insert\_node->data = data;
133. insert\_node->RB\_COLOR = RED;
134. insert\_node->right = m\_nullNode;
135. insert\_node->left = m\_nullNode;
136. **if** (insert\_point == m\_nullNode) //如果插入的是一颗空树
137. {
138. m\_root = insert\_node;
139. m\_root->parent = m\_nullNode;
140. m\_nullNode->left = m\_root;
141. m\_nullNode->right = m\_root;
142. m\_nullNode->parent = m\_root;
143. }
144. **else**
145. {
146. **if** (key < insert\_point->key)
147. {
148. insert\_point->left = insert\_node;
149. }
150. **else**
151. {
152. insert\_point->right = insert\_node;
153. }
154. insert\_node->parent = insert\_point;
155. }
156. InsertFixUp(insert\_node);    //调用InsertFixUp修复红黑树性质。
157. }
159. //---------------------插入结点性质修复--------------------------------
160. //3种插入情况，都在下面的伪代码中(未涉及到所有全部的插入情况)。
161. /\*
162. RB-INSERT-FIXUP(T, z)
163. 1 while color[p[z]] = RED
164. 2     do if p[z] = left[p[p[z]]]
165. 3           then y ← right[p[p[z]]]
166. 4                if color[y] = RED
167. 5                   then color[p[z]] ← BLACK                    ? Case 1
168. 6                        color[y] ← BLACK                       ? Case 1
169. 7                        color[p[p[z]]] ← RED                   ? Case 1
170. 8                        z ← p[p[z]]                            ? Case 1
171. 9                   else if z = right[p[z]]
172. 10                           then z ← p[z]                       ? Case 2
173. 11                                LEFT-ROTATE(T, z)              ? Case 2
174. 12                           color[p[z]] ← BLACK                 ? Case 3
175. 13                           color[p[p[z]]] ← RED                ? Case 3
176. 14                           RIGHT-ROTATE(T, p[p[z]])            ? Case 3
177. 15           else (same as then clause with "right" and "left" exchanged)
178. 16 color[root[T]] ← BLACK
179. \*/
180. //然后的工作，就非常简单了，即把上述伪代码改写为下述的c++代码：
181. **void** InsertFixUp(RB\_Node\* node)
182. {
183. **while** (node->parent->RB\_COLOR == RED)
184. {
185. **if** (node->parent == node->parent->parent->left)   //
186. {
187. RB\_Node\* uncle = node->parent->parent->right;
188. **if** (uncle->RB\_COLOR == RED)   //插入情况1，z的叔叔y是红色的。
189. {
190. node->parent->RB\_COLOR = BLACK;
191. uncle->RB\_COLOR = BLACK;
192. node->parent->parent->RB\_COLOR = RED;
193. node = node->parent->parent;
194. }
195. **else** **if** (uncle->RB\_COLOR == BLACK)  //插入情况2：z的叔叔y是黑色的，。
196. {
197. **if** (node == node->parent->right) //且z是右孩子
198. {
199. node = node->parent;
200. RotateLeft(node);
201. }
202. //else                 //插入情况3：z的叔叔y是黑色的，但z是左孩子。
203. //{
204. node->parent->RB\_COLOR = BLACK;
205. node->parent->parent->RB\_COLOR = RED;
206. RotateRight(node->parent->parent);
207. //}
208. }
209. }
210. **else** //这部分是针对为插入情况1中，z的父亲现在作为祖父的右孩子了的情况，而写的。
211. //15 else (same as then clause with "right" and "left" exchanged)
212. {
213. RB\_Node\* uncle = node->parent->parent->left;
214. **if** (uncle->RB\_COLOR == RED)
215. {
216. node->parent->RB\_COLOR = BLACK;
217. uncle->RB\_COLOR = BLACK;
218. uncle->parent->RB\_COLOR = RED;
219. node = node->parent->parent;
220. }
221. **else** **if** (uncle->RB\_COLOR == BLACK)
222. {
223. **if** (node == node->parent->left)
224. {
225. node = node->parent;
226. RotateRight(node);     //与上述代码相比，左旋改为右旋
227. }
228. //else
229. //{
230. node->parent->RB\_COLOR = BLACK;
231. node->parent->parent->RB\_COLOR = RED;
232. RotateLeft(node->parent->parent);   //右旋改为左旋，即可。
233. //}
234. }
235. }
236. }
237. m\_root->RB\_COLOR = BLACK;
238. }
240. //左旋代码实现
241. **bool** RotateLeft(RB\_Node\* node)
242. {
243. **if** (node == m\_nullNode || node->right == m\_nullNode)
244. {
245. **return** **false**;//can't rotate
246. }
247. RB\_Node\* lower\_right = node->right;
248. lower\_right->parent = node->parent;
249. node->right = lower\_right->left;
250. **if** (lower\_right->left != m\_nullNode)
251. {
252. lower\_right->left->parent = node;
253. }
254. **if** (node->parent == m\_nullNode) //rotate node is root
255. {
256. m\_root = lower\_right;
257. m\_nullNode->left = m\_root;
258. m\_nullNode->right = m\_root;
259. //m\_nullNode->parent = m\_root;
260. }
261. **else**
262. {
263. **if** (node == node->parent->left)
264. {
265. node->parent->left = lower\_right;
266. }
267. **else**
268. {
269. node->parent->right = lower\_right;
270. }
271. }
272. node->parent = lower\_right;
273. lower\_right->left = node;
274. }
276. //右旋代码实现
277. **bool** RotateRight(RB\_Node\* node)
278. {
279. **if** (node == m\_nullNode || node->left == m\_nullNode)
280. {
281. **return** **false**;//can't rotate
282. }
283. RB\_Node\* lower\_left = node->left;
284. node->left = lower\_left->right;
285. lower\_left->parent = node->parent;
286. **if** (lower\_left->right != m\_nullNode)
287. {
288. lower\_left->right->parent = node;
289. }
290. **if** (node->parent == m\_nullNode) //node is root
291. {
292. m\_root = lower\_left;
293. m\_nullNode->left = m\_root;
294. m\_nullNode->right = m\_root;
295. //m\_nullNode->parent = m\_root;
296. }
297. **else**
298. {
299. **if** (node == node->parent->right)
300. {
301. node->parent->right = lower\_left;
302. }
303. **else**
304. {
305. node->parent->left = lower\_left;
306. }
307. }
308. node->parent = lower\_left;
309. lower\_left->right = node;
310. }
312. //--------------------------删除结点总操作----------------------------------
313. //伪代码，不再贴出，详情，请参考此红黑树系列第二篇文章：
314. //经典算法研究系列：五、红黑树算法的实现与剖析：
315. //http://blog.csdn.net/v\_JULY\_v/archive/2010/12/31/6109153.aspx。
316. **bool** Delete(KEY key)
317. {
318. RB\_Node\* delete\_point = find(key);
319. **if** (delete\_point == m\_nullNode)
320. {
321. **return** **false**;
322. }
323. **if** (delete\_point->left != m\_nullNode && delete\_point->right != m\_nullNode)
324. {
325. RB\_Node\* successor = InOrderSuccessor(delete\_point);
326. delete\_point->data = successor->data;
327. delete\_point->key = successor->key;
328. delete\_point = successor;
329. }
330. RB\_Node\* delete\_point\_child;
331. **if** (delete\_point->right != m\_nullNode)
332. {
333. delete\_point\_child = delete\_point->right;
334. }
335. **else** **if** (delete\_point->left != m\_nullNode)
336. {
337. delete\_point\_child = delete\_point->left;
338. }
339. **else**
340. {
341. delete\_point\_child = m\_nullNode;
342. }
343. delete\_point\_child->parent = delete\_point->parent;
344. **if** (delete\_point->parent == m\_nullNode)//delete root node
345. {
346. m\_root = delete\_point\_child;
347. m\_nullNode->parent = m\_root;
348. m\_nullNode->left = m\_root;
349. m\_nullNode->right = m\_root;
350. }
351. **else** **if** (delete\_point == delete\_point->parent->right)
352. {
353. delete\_point->parent->right = delete\_point\_child;
354. }
355. **else**
356. {
357. delete\_point->parent->left = delete\_point\_child;
358. }
359. **if** (delete\_point->RB\_COLOR == BLACK && !(delete\_point\_child == m\_nullNode && delete\_point\_child->parent == m\_nullNode))
360. {
361. DeleteFixUp(delete\_point\_child);
362. }
363. **delete** delete\_point;
364. **return** **true**;
365. }
367. //---------------------删除结点性质修复-----------------------------------
368. //所有的工作，都在下述23行伪代码中：
369. /\*
370. RB-DELETE-FIXUP(T, x)
371. 1 while x ≠ root[T] and color[x] = BLACK
372. 2     do if x = left[p[x]]
373. 3           then w ← right[p[x]]
374. 4                if color[w] = RED
375. 5                   then color[w] ← BLACK                        ?  Case 1
376. 6                        color[p[x]] ← RED                       ?  Case 1
377. 7                        LEFT-ROTATE(T, p[x])                    ?  Case 1
378. 8                        w ← right[p[x]]                         ?  Case 1
379. 9                if color[left[w]] = BLACK and color[right[w]] = BLACK
380. 10                   then color[w] ← RED                          ?  Case 2
381. 11                        x p[x]                                  ?  Case 2
382. 12                   else if color[right[w]] = BLACK
383. 13                           then color[left[w]] ← BLACK          ?  Case 3
384. 14                                color[w] ← RED                  ?  Case 3
385. 15                                RIGHT-ROTATE(T, w)              ?  Case 3
386. 16                                w ← right[p[x]]                 ?  Case 3
387. 17                         color[w] ← color[p[x]]                 ?  Case 4
388. 18                         color[p[x]] ← BLACK                    ?  Case 4
389. 19                         color[right[w]] ← BLACK                ?  Case 4
390. 20                         LEFT-ROTATE(T, p[x])                   ?  Case 4
391. 21                         x ← root[T]                            ?  Case 4
392. 22        else (same as then clause with "right" and "left" exchanged)
393. 23 color[x] ← BLACK
394. \*/
395. //接下来的工作，很简单，即把上述伪代码改写成c++代码即可。
396. **void** DeleteFixUp(RB\_Node\* node)
397. {
398. **while** (node != m\_root && node->RB\_COLOR == BLACK)
399. {
400. **if** (node == node->parent->left)
401. {
402. RB\_Node\* brother = node->parent->right;
403. **if** (brother->RB\_COLOR == RED)   //情况1：x的兄弟w是红色的。
404. {
405. brother->RB\_COLOR = BLACK;
406. node->parent->RB\_COLOR = RED;
407. RotateLeft(node->parent);
408. }
409. **else**     //情况2：x的兄弟w是黑色的，
410. {
411. **if** (brother->left->RB\_COLOR == BLACK && brother->right->RB\_COLOR == BLACK)
412. //且w的俩个孩子都是黑色的。
413. {
414. brother->RB\_COLOR = RED;
415. node = node->parent;
416. }
417. **else** **if** (brother->right->RB\_COLOR == BLACK)
418. //情况3：x的兄弟w是黑色的，w的右孩子是黑色（w的左孩子是红色）。
419. {
420. brother->RB\_COLOR = RED;
421. brother->left->RB\_COLOR = BLACK;
422. RotateRight(brother);
423. }
424. //else if(brother->right->RB\_COLOR == RED)
425. //情况4：x的兄弟w是黑色的，且w的右孩子时红色的。
426. //{
427. brother->RB\_COLOR = node->parent->RB\_COLOR;
428. node->parent->RB\_COLOR = BLACK;
429. brother->right->RB\_COLOR = BLACK;
430. RotateLeft(node->parent);
431. node = m\_root;
432. //}
433. }
434. }
435. **else**  //下述情况针对上面的情况1中，node作为右孩子而阐述的。
436. //22        else (same as then clause with "right" and "left" exchanged)
437. //同样，原理一致，只是遇到左旋改为右旋，遇到右旋改为左旋，即可。其它代码不变。
438. {
439. RB\_Node\* brother = node->parent->left;
440. **if** (brother->RB\_COLOR == RED)
441. {
442. brother->RB\_COLOR = BLACK;
443. node->parent->RB\_COLOR = RED;
444. RotateRight(node->parent);
445. }
446. **else**
447. {
448. **if** (brother->left->RB\_COLOR == BLACK && brother->right->RB\_COLOR == BLACK)
449. {
450. brother->RB\_COLOR = RED;
451. node = node->parent;
452. }
453. **else** **if** (brother->left->RB\_COLOR == BLACK)
454. {
455. brother->RB\_COLOR = RED;
456. brother->right->RB\_COLOR = BLACK;
457. RotateLeft(brother);
458. }
459. //else if(brother->left->RB\_COLOR==RED)
460. //{
461. brother->RB\_COLOR = node->parent->RB\_COLOR;
462. node->parent->RB\_COLOR = BLACK;
463. brother->left->RB\_COLOR = BLACK;
464. RotateRight(node->parent);
465. node = m\_root;
466. //}
467. }
468. }
469. }
470. m\_nullNode->parent = m\_root;   //最后将node置为根结点，
471. node->RB\_COLOR = BLACK;    //并改为黑色。
472. }
474. //
475. **inline** RB\_Node\* InOrderPredecessor(RB\_Node\* node)
476. {
477. **if** (node == m\_nullNode)       //null node has no predecessor
478. {
479. **return** m\_nullNode;
480. }
481. RB\_Node\* result = node->left;     //get node's left child
482. **while** (result != m\_nullNode)         //try to find node's left subtree's right most node
483. {
484. **if** (result->right != m\_nullNode)
485. {
486. result = result->right;
487. }
488. **else**
489. {
490. **break**;
491. }
492. }            //after while loop result==null or result's right child is null
493. **if** (result == m\_nullNode)
494. {
495. RB\_Node\* index = node->parent;
496. result = node;
497. **while** (index != m\_nullNode && result == index->left)
498. {
499. result = index;
500. index = index->parent;
501. }
502. result = index;         // first right parent or null
503. }
504. **return** result;
505. }
507. //
508. **inline** RB\_Node\* InOrderSuccessor(RB\_Node\* node)
509. {
510. **if** (node == m\_nullNode)       //null node has no successor
511. {
512. **return** m\_nullNode;
513. }
514. RB\_Node\* result = node->right;   //get node's right node
515. **while** (result != m\_nullNode)        //try to find node's right subtree's left most node
516. {
517. **if** (result->left != m\_nullNode)
518. {
519. result = result->left;
520. }
521. **else**
522. {
523. **break**;
524. }
525. }                              //after while loop result==null or result's left child is null
526. **if** (result == m\_nullNode)
527. {
528. RB\_Node\* index = node->parent;
529. result = node;
530. **while** (index != m\_nullNode && result == index->right)
531. {
532. result = index;
533. index = index->parent;
534. }
535. result = index;         //first parent's left or null
536. }
537. **return** result;
538. }
540. //debug
541. **void** InOrderTraverse()
542. {
543. InOrderTraverse(m\_root);
544. }
545. **void** CreateGraph(string filename)
546. {
547. //delete
548. }
549. **void** InOrderCreate(ofstream& file, RB\_Node\* node)
550. {
551. //delete
552. }
553. **void** InOrderTraverse(RB\_Node\* node)
554. {
555. **if** (node == m\_nullNode)
556. {
557. **return**;
558. }
559. **else**
560. {
561. InOrderTraverse(node->left);
562. cout << node->key << endl;
563. InOrderTraverse(node->right);
564. }
565. }
566. ~RB\_Tree()
567. {
568. clear(m\_root);
569. **delete** m\_nullNode;
570. }
571. **private**:
572. // utility function for destructor to destruct object;
573. **void** clear(RB\_Node\* node)
574. {
575. **if** (node == m\_nullNode)
576. {
577. **return**;
578. }
579. **else**
580. {
581. clear(node->left);
582. clear(node->right);
583. **delete** node;
584. }
585. }
586. **private**:
587. RB\_Node \*m\_nullNode;
588. RB\_Node \*m\_root;
589. };
590. #endif /\*\_RB\_TREE\_H\_\*/

**RBTree.cpp**

1. //file RBTree.cpp
2. //written by saturnman，20101008。
3. //updated by July，20110329。
5. //所有的头文件都已补齐，现在您可以直接复制此份源码上机验证了（版权所有，侵权必究）。
6. //July、updated，2011.05.06。
7. #include<iostream>
8. #include<algorithm>
9. #include<iterator>
10. #include<vector>
11. #include<sstream>
12. #include"RBTree.h"    //如果.h文件，和cpp文件放在一个文件里，此句去掉
13. **using** **namespace** std;
15. **int** main()
16. {
17. RB\_Tree<**int**,**int**> tree;
18. vector<**int**> v;
20. **for**(**int** i=0;i<20;++i)
21. {
22. v.push\_back(i);
23. }
24. random\_shuffle(v.begin(),v.end());
25. copy(v.begin(),v.end(),ostream\_iterator<**int**>(cout," "));
26. cout<<endl;
27. stringstream sstr;
28. **for**(i=0;i<v.size();++i)
29. {
30. tree.Insert(v[i],i);
31. cout<<"insert:"<<v[i]<<endl;   //添加结点
32. }
33. **for**(i=0;i<v.size();++i)
34. {
35. cout<<"Delete:"<<v[i]<<endl;
36. tree.Delete(v[i]);             //删除结点
37. tree.InOrderTraverse();
38. }
39. cout<<endl;
40. tree.InOrderTraverse();
41. **return** 0;
42. }

**运行效果图（先是一一插入各结点，然后再删除所有的结点）：**

**第二部分、程序有bug？**

**2.1、红黑树要求绝对平衡么？**

    据网友鑫反馈，上述c++源码虽说从上面的

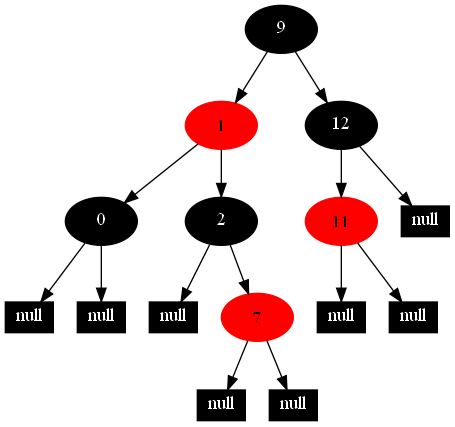
    1、首先在RBTree.h的最后里添加下述代码：

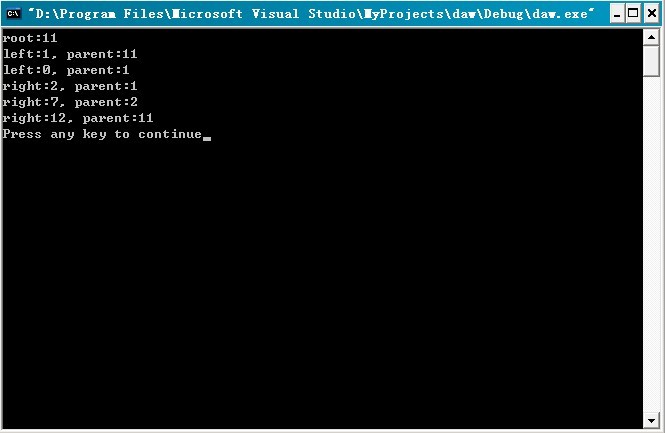
1. **public**:
2. **void** PrintTree()
3. {
4. \_printNode(m\_root);
5. }
6. **private**:
7. **void** \_printNode(RB\_Node \*node)
8. {
9. **if**(node == NULL || node == m\_nullNode) **return**;
11. **if**(node->parent == NULL || node->parent == m\_nullNode){
12. printf("root:%d/n", node->data);
13. }**else** **if**(node->parent->left == node){
14. printf("left:%d, parent:%d/n", node->data, node->parent->data);
15. }**else** **if**(node->parent->right == node){
16. printf("right:%d, parent:%d/n", node->data, node->parent->data);
17. }
19. \_printNode(node->left);
20. \_printNode(node->right);
21. }

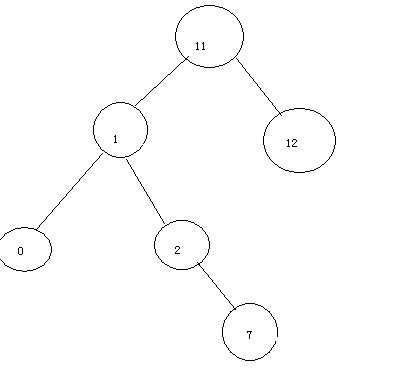
    2、改写RBTree.cpp文件，如下：

1. //file RBTree.cpp
2. //written by saturnman，20101008。
3. //updated by July，20110329。
5. //所有的头文件都已补齐，现在您可以直接复制此份源码上机验证了（版权所有，侵权必究）。
6. //July、updated，2011.05.06。
7. #include<iostream>
8. #include<algorithm>
9. #include<iterator>
10. #include<vector>
11. #include<sstream>
12. //#include"RBTree.h"    //如果.h文件，和cpp文件放在一个文件里，此句去掉
13. **using** **namespace** std;
15. **int** main()
16. {
17. RB\_Tree<**int**,**int**> tree;
19. tree.Insert(12, 12);
20. tree.Insert(1, 1);
21. tree.Insert(9, 9);
22. tree.Insert(2, 2);
23. tree.Insert(0, 0);
24. tree.Insert(11, 11);
25. tree.Insert(7, 7);

28. tree.Delete(9);
30. tree.PrintTree();
31. /\*vector<int> v;
33. for(int i=0;i<20;++i)
34. {
35. v.push\_back(i);
36. }
37. random\_shuffle(v.begin(),v.end());
38. copy(v.begin(),v.end(),ostream\_iterator<int>(cout," "));
39. cout<<endl;
40. stringstream sstr;
41. for(i=0;i<v.size();++i)
42. {
43. tree.Insert(v[i],i);
44. cout<<"insert:"<<v[i]<<endl;   //添加结点
45. }
46. for(i=0;i<v.size();++i)
47. {
48. cout<<"Delete:"<<v[i]<<endl;
49. tree.Delete(v[i]);             //删除结点
50. tree.InOrderTraverse();
51. }
52. cout<<endl;
53. tree.InOrderTraverse();\*/
54. **return** 0;
55. }

    后经测试，结果，的确有误，即依次插入以下节点，12，1，9，0，2，11，7后，红黑树变为如下：

然后删除根节点9，经过上述程序运行后，运行结果，如下：

即上述运行结果，所对应的红黑树的状态如下（此时，红黑树已经不再平衡，存在的问题确实已经很明显了）：

    是的，如你所见，上述程序删除根节点9之后，正确的红黑树的状态应该为7代替根节点9，7成为新的根节点，且节点7着为黑色，而上述结果则是完全错误，红黑树已经完全不平衡。至此，终于发现，此c++程序存在隐藏bug了。至于修正，则还得等一段时间。

**说明**

    但事实是，果真如此么？请看下文2.1节的修正。

**2.1、红黑树不要求严格平衡**

   修正：本程序没有任何问题。有一点非常之重要，之前就是因为未意识到而造成上述错觉，即：

   还是这句话，有任何问题，欢迎任何人提出或指正。

**第三部分、读者反馈**

**关于RB\_Tree插入删除操作的讨论**

July：

你好！关于RB\_Tree的完整实现代码，你已经在你的博客中写出了。但我认为，你的代码中有需要改正的地方。

***起 因***

       我这段时间正好在学习RB\_Tree，由于我忽略了RB\_Tree的性质(3)：每个叶子结点都是黑色的，导致我对RB\_Tree的操作纠结了好几天。在我还没意识到的时候，偶然间看到你的博客，想从中获得答案。然后就发现其中有值得商榷的地方。

***错 误***

       下图是你写的插入修正函数InsertFixUp的部分截图：

 你的文章地址：<http://blog.csdn.net/v_july_v/article/details/6285620>

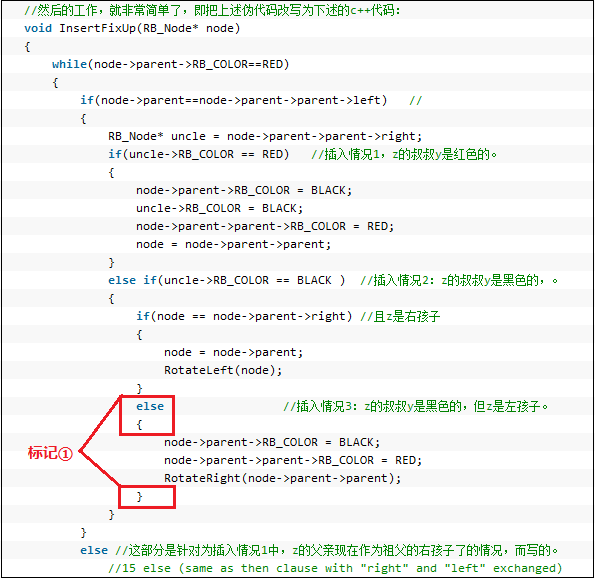


                                                         图  1

    正如《算法导论》所言，InsertFixUp 中每一次while循环都要面对3种情况：

case 1：z的叔叔y是红色的；

case 2：z的叔叔y是黑色的，且z是右孩子；

case 3：z的叔叔y是黑色的，且z是左孩子.

并且**case 2是落在case 3内的，所以这两种情况不是相互排斥的！**而在你的代码中，将case 2和case 3分别放在if和else中，导致它们相互独立。这是不对的。

***修 正***

       所以，在图1中“**标记①**”处的else是不能加的，应将其删除。

       遗憾的是，我认为你的RB\_Tree的删除修正操作DeleteFixUp也出现了类似的错误：对于DeleteFixUp所处理的4种情况也同样不是相互排斥的，而你用一组if…else if…else if…将case 2, 3, 4全部独立开来。

       以上便是鄙人的一点拙见，如果你认为有错误的地方，欢迎再讨论！

                                    杨  超

                                                               CSDN ID: crisischaos

                                                               2011.10.06

    考证：非常感谢杨兄来信指导。从算法导论一书原来的插入情况的修复伪代码来看：

1. //---------------------插入结点性质修复--------------------------------
2. //3种插入情况，都在下面的伪代码中(未涉及到所有全部的插入情况)。
3. /\*
4. RB-INSERT-FIXUP(T, z)
5. 1 while color[p[z]] = RED
6. 2     do if p[z] = left[p[p[z]]]
7. 3           then y ← right[p[p[z]]]
8. 4                if color[y] = RED
9. 5                   then color[p[z]] ← BLACK                    ? Case 1
10. 6                        color[y] ← BLACK                       ? Case 1
11. 7                        color[p[p[z]]] ← RED                   ? Case 1
12. 8                        z ← p[p[z]]                            ? Case 1
13. 9                   else if z = right[p[z]]
14. 10                           then z ← p[z]                       ? Case 2
15. 11                                LEFT-ROTATE(T, z)              ? Case 2
16. 12                           color[p[z]] ← BLACK                 ? Case 3
17. 13                           color[p[p[z]]] ← RED                ? Case 3
18. 14                           RIGHT-ROTATE(T, p[p[z]])            ? Case 3
19. 15           else (same as then clause with "right" and "left" exchanged)
20. 16 color[root[T]] ← BLACK
21. \*/
22. //然后的工作，就非常简单了，即把上述伪代码改写为下述的c++代码：  ....