<https://blog.csdn.net/weiyuefei/article/details/52048014>

之前看了很多写红黑树的博客，但是感觉都讲的不太清楚！没说这样操作如何使他保持平衡的，于是疑惑重重，就看不下去了，一次不经意看到一个人说[维基百科的红黑树](http://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91)讲的好，我就随便点了一下一看——这下疯了~，怎么讲的这么好！可以说是把一个复杂的问题，讲得简单化！这太幸福了！ 于是我就慢慢学会了！强烈推荐维基的这个讲解，再也找不到比这还好的讲解了！不知道它上边其它的怎么样，反正这个很好！！既然学会了，走过来了，我也要留下脚印！

下面将是我对红黑树的总结，里面的性感的图片都是维基百科红黑树上的^\_^！我讨论的红黑树需建立在会平衡二叉树的基础上去学，即若不懂“旋转”操作，请看[平衡二叉树](http://www.cnblogs.com/fornever/archive/2011/11/15/2249492.html)的旋转操作。

**红黑树（RBT）的定义：它或者是一颗空树，或者是具有一下性质的二叉查找树：**

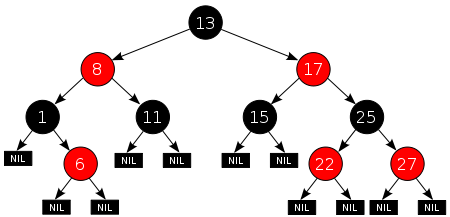
**1.节点非红即黑。**

**2.根节点是黑色。**

**3.所有NULL结点称为叶子节点，且认为颜色为黑**。

**4.所有红节点的子节点都为黑色。**

**5.从任一节点到其叶子节点的所有路径上都包含相同数目的黑节点。**



看完红黑树的定义是不是可晕？怎么这么多要求！！这怎么约束啊？我刚看到这5条约束，直接无语了，1-3、4还好说，第5点是怎么回事啊？怎么约束？整这么复杂的条件好干啥啊？我来简单说说呵：第3条，显然这里的叶子节点不是平常我们所说的叶子节点，如图标有NIL的为叶子节点，为什么不按常规出牌，因为按一般的叶子节点也行，但会使算法更复杂；第4条，即该树上决不允许存在两个连续的红节点；第5条，比如图中红8到1左边的叶子节点的路径包含2个黑节点，到6下的叶子节点的路径也包含2个黑节点。所有性质1-5合起来约束了该树的平衡性能--即该树上的最长路径不可能会大于2倍最短路径。为什么？因为第1条该树上的节点非红即黑，由于第4条该树上不允许存在两个连续的红节点，那么对于从一个节点到其叶子节点的一条最长的路径一定是红黑交错的，那么最短路径一定是纯黑色的节点；而又第5条从任一节点到其叶子节点的所有路径上都包含相同数目的黑节点，这么来说最长路径上的黑节点的数目和最短路径上的黑节点的数目相等！而又第2条根结点为黑、第3条叶子节点是黑，那么可知：最长路径<=2\*最短路径。一颗二叉树的平衡性能越好，那么它的效率越高！显然红黑树的平衡性能比AVL的略差些，但是经过大量试验证明，实际上红黑树的效率还是很不错了，仍能达到O(logN)，这个我不知道，我现在不可能做过大量试验，只是听人家这样说，O(∩\_∩)O哈哈~但你至少知道他的时间复杂度一定小于2O(logN)!

上边的性质看个10遍，看懂看透彻再看操作！

**插入操作**

由于性质的约束：插入点不能为黑节点，应插入红节点。因为你插入黑节点将破坏性质5，所以每次插入的点都是红结点，但是若他的父节点也为红，那岂不是破坏了性质4？对啊，所以要做一些“旋转”和一些节点的变色！另为叙述方便我们给要插入的节点标为N（红色），父节点为P，祖父节点为G，叔节点为U。下边将一一列出所有插入时遇到的情况：

**情形1**：该树为空树，直接插入根结点的位置，违反性质1，把节点颜色有红改为黑即可。

**情形2**：插入节点N的父节点P为黑色，不违反任何性质，无需做任何修改。

 情形1很简单，情形2中P为黑色，一切安然无事，但P为红就不一样了，下边是P为红的各种情况，也是真正要学的地方！

**情形3**：N为红，P为红，（祖节点一定存在，且为黑，下边同理）U也为红，这里不论P是G的左孩子，还是右孩子；不论N是P的左孩子，还是右孩子。

操作：如图把P、U改为黑色，G改为红色，未结束。

解析：N、P都为红，违反性质4；若把P改为黑，符合性质4，显然左边少了一个黑节点，违反性质5；所以我们把G，U都改为相反色，这样一来通过G的路径的黑节点数目没变，即符合4、5，但是G变红了，若G的父节点又是红的不就有违反了4，是这样，所以经过上边操作后未结束，需把G作为起始点，即把G看做一个插入的红节点继续向上检索----属于哪种情况，按那种情况操作~要么中间就结束，要么知道根结点（此时根结点变红，一根结点向上检索，那木有了，那就把他变为黑色吧）。

**情形4**：N为红，P为红，U为黑，P为G的左孩子，N为P的左孩子（或者P为G的右孩子，N为P的左孩子；反正就是同向的）。



操作：如图P、G变色，P、G变换即左左单旋（或者右右单旋），结束。

解析：要知道经过P、G变换（旋转），变换后P的位置就是当年G的位置，所以红P变为黑，而黑G变为红都是为了不违反性质5，而维持到达叶节点所包含的黑节点的数目不变！还可以理解为：也就是相当于（只是相当于，并不是实事，只是为了更好理解；）把红N头上的红节点移到对面黑U的头上；这样即符合了性质4也不违反性质5，这样就结束了。

**情形5**：N为红，P为红，U为黑，P为G的左孩子，N为P的右孩子（或者P为G的右孩子，N为P的左孩子；反正两方向相反）。



操作：需要进行两次变换（旋转），图中只显示了一次变换-----首先P、N变换，颜色不变；然后就变成了情形4的情况，按照情况4操作，即结束。

解析：由于P、N都为红，经变换，不违反性质5；然后就变成4的情形，此时G与G现在的左孩子变色，并变换，结束。

**删除操作**

我们知道删除需先找到“替代点”来替代删除点而被删除，也就是删除的是替代点，而替代点N的至少有一个子节点为NULL，那么，若N为红色，则两个子节点一定都为NULL（必须地），那么直接把N删了，不违反任何性质，ok，结束了；若N为黑色，另一个节点M不为NULL，则另一个节点M一定是红色的，且M的子节点都为NULL（按性质来的，不明白，自己分析一下）那么把N删掉，M占到N的位置，并改为黑色，不违反任何性质，ok，结束了；若N为黑色，另一个节点也为NULL，则把N删掉，该位置置为NULL，显然这个黑节点被删除了，破坏了性质5，那么要以N节点为起始点检索看看属于那种情况，并作相应的操作，另还需说明N为黑点（也许是NULL，也许不是，都一样），P为父节点，S为兄弟节点（这个我真想给兄弟节点叫B（brother）多好啊，不过人家图就是S我也不能改，在重画图，太浪费时间了！S也行呵呵，就当是sister也行，哈哈）分为以下5中情况：

**情形1**：S为红色（那么父节点P一定是黑，子节点一定是黑），N是P的左孩子（或者N是P的右孩子）。



操作：P、S变色，并交换----相当于AVL中的右右中旋转即以P为中心S向左旋（或者是AVL中的左左中的旋转），未结束。

解析：我们知道P的左边少了一个黑节点，这样操作相当于在N头上又加了一个红节点----不违反任何性质，但是到通过N的路径仍少了一个黑节点，需要再把对N进行一次检索，并作相应的操作才可以平衡（暂且不管往下看）。

**情形2**：P、S及S的孩子们都为黑。



操作：S改为红色，未结束。  
解析：S变为红色后经过S节点的路径的黑节点数目也减少了1，那个从P出发到其叶子节点到所有路径所包含的黑节点数目（记为num）相等了。但是这个num比之前少了1，因为左右子树中的黑节点数目都减少了！一般地，P是他父节点G的一个孩子，那么由G到其叶子节点的黑节点数目就不相等了，所以说没有结束，需把P当做新的起始点开始向上检索。

**情形3**：P为红（S一定为黑），S的孩子们都为黑。



操作：P该为黑，S改为红，结束。

解析：这种情况最简单了，既然N这边少了一个黑节点，那么S这边就拿出了一个黑节点来共享一下，这样一来，S这边没少一个黑节点，而N这边便多了一个黑节点，这样就恢复了平衡，多么美好的事情哈！

**情形4**：P任意色，S为黑，N是P的左孩子，S的右孩子SR为红，S的左孩子任意（或者是N是P的右孩子，S的左孩子为红，S的右孩子任意）。



操作：SR（SL）改为黑，P改为黑，S改为P的颜色，P、S变换--这里相对应于AVL中的右右中的旋转（或者是AVL中的左左旋转），结束。  
解析：P、S旋转有变色，等于给N这边加了一个黑节点，P位置（是位置而不是P）的颜色不变，S这边少了一个黑节点；SR有红变黑，S这边又增加了一个黑节点；这样一来又恢复了平衡，结束。

**情形5**：P任意色，S为黑，N是P的左孩子，S的左孩子SL为红，S的右孩子SR为黑（或者N是P的有孩子，S的右孩子为红，S的左孩子为黑）。



操作：SL（或SR）改为黑，S改为红，SL（SR）、S变换；此时就回到了情形4，SL（SR）变成了黑S，S变成了红SR（SL），做情形4的操作即可，这两次变换，其实就是对应AVL的右左的两次旋转（或者是AVL的左右的两次旋转）。  
解析：这种情况如果你按情形4的操作的话，由于SR本来就是黑色，你无法弥补由于P、S的变换（旋转）给S这边造成的损失！所以我没先对S、SL进行变换之后就变为情形4的情况了，何乐而不为呢？

好了，这五种情况都讨论完了，我想强调的是：注意哪些分方向的情况，每个分方向的情形就两种情况，不要搞迷了！下边我写的代码，不用关心是什么方向，我主要是用一个指针数组即child[2]，0代表左，1代表右，进行两个节点的变换（旋转）的时候只需向conversion(&T,direction);传入父节点指针的地址及子节点在父节点的方位（0或1）；有兴趣可以看代码.

欢迎大家留言指正哦^\_^

下边贴上我的C代码：  
简介：主要是用递归实现插入、删除，回溯时检索并恢复平衡。

1. #include <stdio.h>
2. #include <stdlib.h>
4. #define RED 0
5. #define BACK 1
7. typedef int Elemtype;
9. *//定义一个红黑树的结点*
10. typedef struct Red\_Back\_Tree
11. {
12. Elemtype e;
13. int color;
14. struct Red\_Back\_Tree \* child[2];
15. }\* RBT;
17. *// 两个节点变换函数*
18. void conversion(RBT \*T,int direction);
20. *// 删除一个节点的所用函数*
21. int DeleteRBT(RBT \*T,Elemtype e); *// 删除主(接口)函数*
22. int find\_replace\_point(RBT gogal,RBT \*l); *// 寻找替代点*
23. int keep\_balance\_for\_delete(RBT \*T,int direction); *// 删除的平衡操作*
24. int do\_with\_start\_point(RBT gogal,RBT \*T,int direction); *// 处理第一个起始点*
26. *// 插入一个节点的所用函数*
27. int InsertRBT(RBT \*T,Elemtype e); *// 插入接口函数*
28. int \_InsertRBT(RBT \*T,Elemtype e); *// 插入主函数*
29. int keep\_balance\_for\_insert(RBT \*T,int firdirection,Elemtype e);*// 插入的平衡操作*
30. RBT create\_one\_node(Elemtype e); *// 新建一个节点*


34. void conversion(RBT \*T,int direction)
35. {
36. RBT f=(\*T),s=f->child[direction],ss=s->child[!direction];
38. f->child[direction]=ss;
39. s->child[!direction]=f;
40. \*T=s;
41. }
43. *//★★★★★★★★★★★★★★★★★删除操作★★★★★★★★★★★★★★★★★★★★★★★★★★★*
45. int do\_with\_start\_point(RBT gogal,RBT \*T,int direction)
46. {
47. gogal->e=(\*T)->e;
48. if(BACK==((\*T)->color))
49. {
50. if(NULL!=(\*T)->child[direction])
51. {
52. (\*T)->e=(\*T)->child[direction]->e;
53. free((\*T)->child[direction]);
54. (\*T)->child[direction]=NULL;
55. return 1;
56. }
57. else
58. {
59. free((\*T));
60. \*T=NULL;
61. return 0;
62. }
63. }
64. else
65. {
66. free((\*T));
67. (\*T)=NULL;
68. return 1;
69. }
70. }
72. int keep\_balance\_for\_delete(RBT \*T,int direction)
73. {
74. RBT p=(\*T),b=p->child[!direction];
76. if(RED==b->color)
77. {
78. p->color=RED;
79. b->color=BACK;
80. *// conversion(&p,!direction);//很恐怖的一个写法，偶然中发现：这里传的地址是假的!不是T!!*
81. *// 考我怎么这么傻逼!!如果不是及时发现，到调试时将是无限恐怖*
82. *// 将是一个巨大的隐藏的BUG!!!将会带来巨大的麻烦!!!*
83. conversion(T,!direction);
84. return keep\_balance\_for\_delete(&((\*T)->child[direction]),direction);
85. }
86. else if(BACK==p->color && BACK==b->color &&
87. (NULL==b->child[0] || BACK==b->child[0]->color) &&
88. (NULL==b->child[1] || BACK==b->child[1]->color)) *//这里感觉不美，就一次为NULL却每次要*
89. { *//判断是否为NULL，不美……*
90. b->color=RED;
91. return 0;
92. }
93. else if(RED==p->color &&
94. (NULL==b->child[0] || BACK==b->child[0]->color) &&
95. (NULL==b->child[1] || BACK==b->child[1]->color))
96. {
97. p->color=BACK;
98. b->color=RED;
99. return 1;
100. }
101. *// 第一次调试*
102. *// 调试原因：由于删除0点未按预料的操作应该是情况④，却按⑤操作*
103. *// 错误的地方：RED==b->child[!direction] ! 丢了->color 这个错误我上边错了几次，不过编译器报错改了过来*
104. *// 这次的编译器不报错，看代码也看不错来，最后追究到这里，一一对照才发现!!!*
105. *// else if(BACK==b->color && (NULL!=b->child[!direction] && RED==b->child[!direction]))*
106. else if(BACK==b->color && (NULL!=b->child[!direction] && RED==b->child[!direction]->color))
107. {
108. b->color=p->color;
109. p->color=BACK;
110. b->child[!direction]->color=BACK;
111. conversion(T,!direction);
112. return 1;
113. }
114. else
115. {
116. b->child[direction]->color=p->color;
117. p->color=BACK;
118. conversion(&(p->child[!direction]),direction);*//这里的p写的才算不错!即p也(\*T)都行，一样!*
119. conversion(T,!direction);
120. return 1;
121. }
123. }
125. int find\_replace\_point(RBT gogal,RBT \*l)
126. {
127. if(NULL!=(\*l)->child[0])
128. {
129. if(find\_replace\_point(gogal,&(\*l)->child[0])) return 1;
130. return keep\_balance\_for\_delete(l,0);
131. *//...*
132. }
133. *// 第二次调试---其实没F5,F10,F11，根据结果猜测，到这里看看还真是的!*
134. *// 调试原因：删除0好了，删除1又错了---2不见了，1还在*
135. *// 错误的地方：就在这里，找到替代点，却没有“替代”，这等于把替代点删了...*
136. *// 这里很明显，gogal这个删除点指针根本就没用...我当时忘了吧!!修改如下!*
137. *// else //替代点为起始点*
138. *// {*
139. *// return do\_with\_start\_point(l,1);*
140. *// }*
141. else
142. {
143. return do\_with\_start\_point(gogal,l,1);
144. }
145. }
147. int DeleteRBT(RBT \*T,Elemtype e)
148. {
149. if(!(\*T)) return -1;
150. else if(e>(\*T)->e)
151. {
152. if(DeleteRBT(&((\*T)->child[1]),e)) return 1;
153. return keep\_balance\_for\_delete(T,1);
154. *//...*
155. }
156. else if(e<(\*T)->e)
157. {
158. if(DeleteRBT(&((\*T)->child[0]),e)) return 1;
159. return keep\_balance\_for\_delete(T,0);
160. *//...*
161. }
162. else
163. {
164. if(NULL!=(\*T)->child[1]) *//真正的删除点不是起始点，需找替代点*
165. {
166. if(find\_replace\_point((\*T),&((\*T)->child[1]))) return 1;
167. return keep\_balance\_for\_delete(T,1);
168. *//...*
169. }
170. else *//真正的删除点就是起始点*
171. {
172. return do\_with\_start\_point((\*T),T,0);
173. }
174. }
175. }
176. *//★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★*

179. *//★★★★★★★★★★★★★★★★★★★插入操作★★★★★★★★★★★★★★★★★★★★★★★★★*
181. RBT create\_one\_node(Elemtype e)
182. {
183. RBT p=(RBT)malloc(sizeof(struct Red\_Back\_Tree));
184. p->e=e; p->color=RED;
185. p->child[0]=p->child[1]=NULL;
186. return p;
187. }
189. int keep\_balance\_for\_insert(RBT \*T,int firdirection,Elemtype e)
190. {
191. RBT p=(\*T)->child[firdirection],u=(\*T)->child[!firdirection];
192. int secdirection=( (e>p->e) ? 1 : 0 ); *// 查处第二个方向*
194. if(NULL!=u && RED==u->color) */\*\*\*\*\*③叔节点为红色\*\*\*\*\*/*
195. {
196. p->color=BACK;
197. u->color=BACK;
198. (\*T)->color=RED;
199. return 1; *//继续...*
200. }
201. else */\*\*\*\*\*④叔节点为黑色\*\*\*\*\*/*
202. {
203. if(firdirection!=secdirection) conversion(&((\*T)->child[firdirection]),secdirection);
204. (\*T)->color=RED; (\*T)->child[firdirection]->color=BACK;
205. conversion(T,firdirection);
206. return 0;
207. }
208. }
210. int \_InsertRBT(RBT \*T,Elemtype e)
211. {
212. int info=0;
213. if(NULL==(\*T)) */\*\*\*\*\*①插入到根节点\*\*\*\*\*/* *//这里只是包含这种情况*
214. {
215. \*T=create\_one\_node(e);
216. (\*T)->color=RED;
217. info=1;
218. }
219. else if(e>((\*T)->e))
220. {
221. info=\_InsertRBT(&(\*T)->child[1],e);
222. if(info<1) return info;
223. else if(info==1) */\*\*\*\*\*②父节点为黑\*\*\*\*\*\*/*
224. {
225. if(BACK==((\*T)->color)) info--;
226. else info++;
227. }
228. else
229. {
230. info=keep\_balance\_for\_insert(T,1,e);
231. }
233. }
234. else if(e<((\*T)->e))
235. {
236. info=\_InsertRBT(&((\*T)->child[0]),e);
237. if(info<1) return info;
238. else if(info==1)
239. {
240. if(BACK==((\*T)->color)) info--;
241. else info++;
242. }
243. else
244. {
245. info=keep\_balance\_for\_insert(T,0,e);
246. }
248. }
249. else return info=-1;
251. return info;
252. }
254. int InsertRBT(RBT \*T,Elemtype e) *//插入节点函数返回值： -1->改点已存在 0->成功插入*
255. {
256. int info=0; *// info: -1->已存在 0->结束 1->回溯到父节点 2->回溯到祖节点*
258. *//2011年11月30日9:13:47 昨天晚上最后又想来这里这个if可以不要即可，也就是把它也放到\_InsertRBT*
259. *//内处理，在InsertRBT中有个判断即可!即改成下边的写法!*
260. *// if(NULL==(\*T)) /\*\*\*\*\*①插入到根节点\*\*\*\*\*/*
261. *// {*
262. *// \*T=create\_one\_node(e);*
263. *// (\*T)->color=BACK;*
264. *// }*
265. *// else*
266. *// {*
267. *// info=\_InsertRBT(T,e); // 经过再三思考，这里info的返回值只可能为：-1 0 1*
268. *// if(info>0) (\*T)->color=BACK,info=0; // 查看根节点是否为红*
269. *// }*
271. info=\_InsertRBT(T,e);
272. if(info==1) (\*T)->color=BACK,info=0;
273. *// 为了防止根结点变为红，它其实是处理了两种情况的后遗症*
274. *// 分别是：③情况回溯上来，根节点变红 ①情况插入点即为根节点，为红*
275. *// 这里没有直接把根结点变黑，主要是为了与\_InsertRBT保持一致的写法，其实都行!*
276. return info;
277. }
278. *//★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★*

281. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*JUST FOR TEST\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//*
282. RBT queue[1000];
283. void print(RBT cur)
284. {
285. int front=0,rear=0;
286. int count=1,temp=0;
288. if(NULL==cur)
289. {
290. printf("NULL\n");
291. return ;
292. }
294. queue[rear]=cur;
295. while(front<=rear)
296. {
297. cur=queue[front++]; count--;
298. if(NULL!=cur->child[0]) queue[++rear]=cur->child[0],temp++;
299. if(NULL!=cur->child[1]) queue[++rear]=cur->child[1],temp++;
301. printf("%d color->",cur->e);
302. if(BACK==cur->color) printf("BACK |");
303. else printf("RED |");
305. if(0==count)
306. {
307. count=temp;
308. temp=0;
309. printf("\n");
310. }
311. }
312. }
313. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//*
315. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DEAR MAIN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//*
316. int main()
317. {
318. RBT T=NULL;
319. int i,nodenum=100;
321. print(T);
322. printf("\n");
324. printf("\n插入操作\n");
325. for(i=0;i<nodenum;i++)
326. {
327. InsertRBT(&T,i);
328. printf("插入%d\n",i);
329. print(T);
330. printf("\n");
331. }
333. *// print(T);*
334. printf("\n删除操作:\n");
336. for(i=0;i<nodenum;i++)
337. {
338. DeleteRBT(&T,i);
339. printf("删除%d\n",i);
340. print(T);
341. printf("\n");
342. }
344. return 0;
345. }