1. **final** Node<K,V> getNode(**int** hash, Object key) {
2. Node<K,V>[] tab; Node<K,V> first, e; **int** n; K k;
3. //定位键值对所在桶的位置
4. **if** ((tab = table) != **null** && (n = tab.length) > 0 &&
5. (first = tab[(n - 1) & hash]) != **null**) {
6. //选择该数组位置下链表的头节点
7. //如果节点hash与参数hash相等，节点key和参数key相等，则返回头节点
8. **if** (first.hash == hash && // always check first node
9. ((k = first.key) == key || (key != **null** && key.equals(k))))
10. **return** first;
11. //下一个节点是否为空
12. **if** ((e = first.next) != **null**) {
13. //节点实例是否属于TreeNode子类
14. //这里就是红黑树了
15. //如果 first 是 TreeNode 类型，则调用黑红树查找方法
16. **if** (first **instanceof** TreeNode)
17. **return** ((TreeNode<K,V>)first).getTreeNode(hash, key);
18. //对链表进行查找
19. **do** {
20. **if** (e.hash == hash &&
21. ((k = e.key) == key || (key != **null** && key.equals(k))))
22. **return** e;
23. } **while** ((e = e.next) != **null**);
24. }
25. }
26. **return** **null**;
27. }
29. //实现的红黑树的查找函数
30. **final** TreeNode<K,V> find(**int** h, Object k, Class<?> kc) {
31. TreeNode<K,V> p = **this**;
32. **do** {
33. **int** ph, dir; K pk;
34. TreeNode<K,V> pl = p.left, pr = p.right, q;
35. **if** ((ph = p.hash) > h)
36. //当前节点hash > 参数h
37. //当前节点赋值为左子节点
38. p = pl;
39. **else** **if** (ph < h)
40. //当前节点hash < 参数h
41. //当前节点赋值为右子节点
42. p = pr;
43. **else** **if** ((pk = p.key) == k || (k != **null** && k.equals(pk)))
44. //当前节点的key和参数k相等
45. //返回当前节点
46. **return** p;
47. **else** **if** (pl == **null**)
48. //如果当前节点左子节点为空
49. //将右子节点赋值给当前节点
50. p = pr;
51. **else** **if** (pr == **null**)
52. //如果当前节点右子节点为空
53. //将左子节点赋值给当前节点
54. p = pl;
55. **else** **if** ((kc != **null** ||
56. (kc = comparableClassFor(k)) != **null**) &&
57. (dir = compareComparables(kc, k, pk)) != 0)
58. //kc一般等于null
59. //kc = comparableClassFor(k) 不太清楚
60. p = (dir < 0) ? pl : pr;
61. **else** **if** ((q = pr.find(h, k, kc)) != **null**)
62. //递归调用右子节点，直到最后一个右子节点返回
63. //如果返回不是空
64. //返回该节点
65. **return** q;
66. **else**
67. //以上都不成立时，当前节点赋值为左子节点
68. p = pl;
69. } **while** (p != **null**);
70. **return** **null**;
71. }
73. //红黑树左旋操作
74. **static** <K,V> TreeNode<K,V> rotateLeft(TreeNode<K,V> root,
75. TreeNode<K,V> p) {
76. TreeNode<K,V> r, pp, rl;
77. **if** (p != **null** && (r = p.right) != **null**) {
78. //p节点不为空，p的右子节点不为空
79. **if** ((rl = p.right = r.left) != **null**)
80. //p的右子节点的左子节点(rl)不为空
81. //rl的父节点从p的右子节点变为p
82. rl.parent = p;
83. **if** ((pp = r.parent = p.parent) == **null**)
84. //p的父节点(pp)为空
85. //设置r为根节点，为黑色
86. (root = r).red = **false**;
87. **else** **if** (pp.left == p)
88. //pp不为空，且pp的左子节点为p
89. //pp的左子节点为r节点
90. pp.left = r;
91. **else**
92. //以上都不成立
93. //说明pp的右子节点是p
94. //所以左旋之后pp的右子节点是r
95. pp.right = r;
96. r.left = p;
97. p.parent = r;
98. }
99. **return** root;
100. }
102. //红黑树右旋操作
103. **static** <K,V> TreeNode<K,V> rotateRight(TreeNode<K,V> root,
104. TreeNode<K,V> p) {
105. TreeNode<K,V> l, pp, lr;
106. **if** (p != **null** && (l = p.left) != **null**) {
107. //p节点不为空，p的左子节点不为空
108. **if** ((lr = p.left = l.right) != **null**)
109. //p的左子节点(l)的右子节点(lr)不为空
110. //p的左子节点(l)的右子节点(lr)的父节点为p
111. lr.parent = p;
112. **if** ((pp = l.parent = p.parent) == **null**)
113. //p的父节点(pp)为空
114. //p的左子节点(l)为根节点
115. (root = l).red = **false**;
116. **else** **if** (pp.right == p)
117. //pp节点不为空，且右子节点为p
118. //pp的右子节点为l
119. pp.right = l;
120. **else**
121. //以上条件不成立
122. //说明pp的左子节点是p
123. //所以pp的左子节点为l
124. pp.left = l;
125. l.right = p;
126. p.parent = l;
127. }
128. **return** root;
129. }
131. //红黑树插入后的平衡操作
132. **static** <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,
133. TreeNode<K,V> x) {
134. x.red = **true**;
135. **for** (TreeNode<K,V> xp, xpp, xppl, xppr;;) {
136. **if** ((xp = x.parent) == **null**) {
137. //x的父节点(xp)为空
138. //x则为根节点，不是红色的
139. //返回x
140. x.red = **false**;
141. **return** x;
142. }
143. **else** **if** (!xp.red || (xpp = xp.parent) == **null**)
144. //x的父节点不是红色的，或者x的父节点的父节点为空
145. //说明这个xp是根节点
146. //返回root根节点
147. **return** root;
148. **if** (xp == (xppl = xpp.left)) {
149. //x的父节点的父节点(xpp)的左子节点(xppl)等于x的父节点(xp)
150. **if** ((xppr = xpp.right) != **null** && xppr.red) {
151. //xpp的右子节点(xppr)不为空并且xppr是红色的
152. xppr.red = **false**;
153. xp.red = **false**;
154. xpp.red = **true**;
155. x = xpp;
156. }
157. **else** {
158. //xpp的右子节点(xppr)为空或者xppr是黑色的
159. **if** (x == xp.right) {
160. //如果x的父节点的右子节点等于x节点
161. //进行左旋操作
162. root = rotateLeft(root, x = xp);
163. xpp = (xp = x.parent) == **null** ? **null** : xp.parent;
164. }
165. **if** (xp != **null**) {
166. //x的父节点不为空
167. //x的父节点为黑色
168. xp.red = **false**;
169. //x的父节点的父节点不为空
170. **if** (xpp != **null**) {
171. //x的父节点的父节点为红色
172. //根据x的父节点的父节点进行右旋操作
173. xpp.red = **true**;
174. root = rotateRight(root, xpp);
175. }
176. }
177. }
178. }
179. **else** {
180. **if** (xppl != **null** && xppl.red) {
181. //x的父节点的父节点的左子节点不为空并且为红色
182. xppl.red = **false**;
183. xp.red = **false**;
184. xpp.red = **true**;
185. x = xpp;
186. }
187. **else** {
188. **if** (x == xp.left) {
189. //x的父节点的左子节点等于x节点
190. root = rotateRight(root, x = xp);
191. xpp = (xp = x.parent) == **null** ? **null** : xp.parent;
192. }
193. **if** (xp != **null**) {
194. xp.red = **false**;
195. **if** (xpp != **null**) {
196. xpp.red = **true**;
197. root = rotateLeft(root, xpp);
198. }
199. }
200. }
201. }
202. }
203. }

