目录

算法导论-第十讲 红黑树

1.红黑树的定义

2.红黑树的基本特征

3.红黑树的相关操作

4.完整程序

课后作业

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） 课时安排: 2学时

第十讲：红黑树 授课时间 :第十周周一第1、2节

教学内容（包括基本内容、重点、难点）：

基本内容： （1）红黑树的定义。

（2）插入操作

（3）删除操作

（4）可视化

教学重点、难点： 重点为红黑树的构建

教学媒体的选择： 本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计： 黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第十讲 红黑树

1. 红黑树的定义

　　红黑树（Red Black Tree），是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。

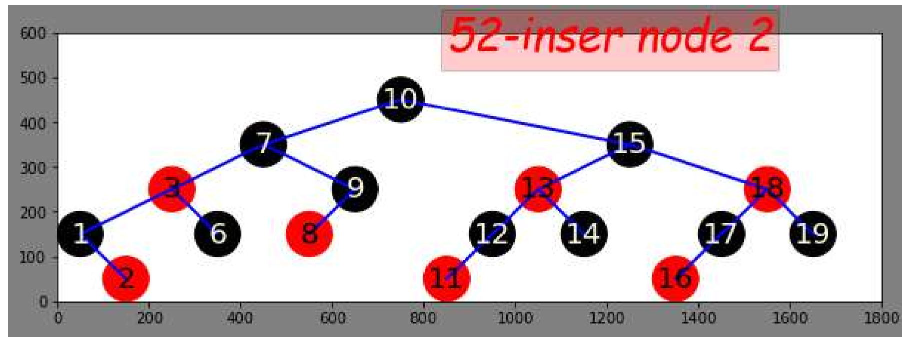
　　红黑树是在1972年由Rudolf Bayer发明的，当时被称为平衡二叉B树（symmetric binary B-trees）。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”。

　　红黑树是一种特化的AVL树（平衡二叉树），都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的： 它可以在 $O(\log n)$ 时间内做查找，插入和删除，这里的 n 是树中元素的数目。

2. 基本特征

　　红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:
　　（1）节点是红色或黑色；
　　（2）根节点黑色；
　　（3）所有叶子都是黑色。（叶子是NUIL节点）；
　　（4）每个红色节点的两个子节点都是黑色。（从每个叶子到根的所有路径上不能有两个连续的红色节点）；
　　（5）从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

　　这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。



红黑树中节点的Python程序如下：

In [4]:

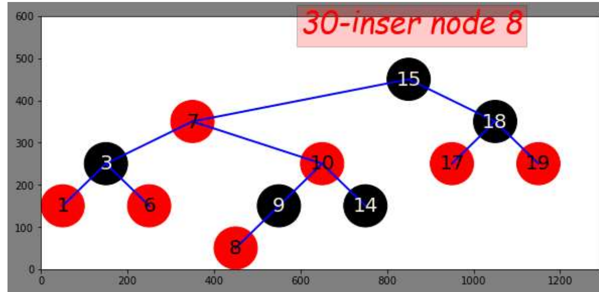
```
class RBNode:
    def __init__(self, val, color="R"):
        self.val = val
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

    def is_black_node(self):
        return self.color == "B"

    def set_black_node(self):
        self.color = "B"

    def set_red_node(self):
        self.color = "R"

    def print(self):
        if self.left:
            self.left.print()
        print(self.val)
        if self.right:
            self.right.print()
```

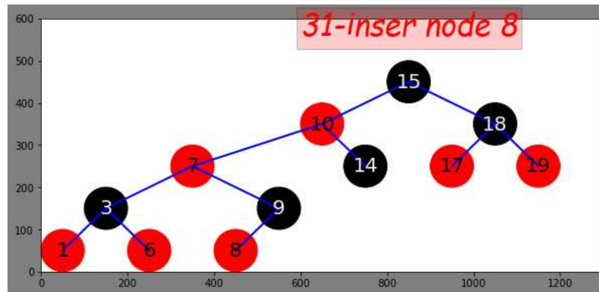
3. 红黑树的相关操作

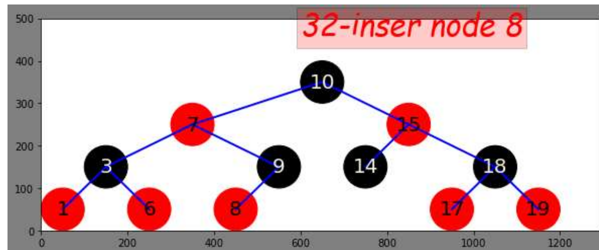
3.1 左旋右旋

当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。为了保持红黑树的性质，我们可以对相关节点做一系列的调整，通过对树进行旋转（例如左旋和右旋操作），即修改树中某些结点的颜色及指针结构，以达到对红黑树进行插入、删除结点等操作时，红黑树依然能保持它特有的性质。

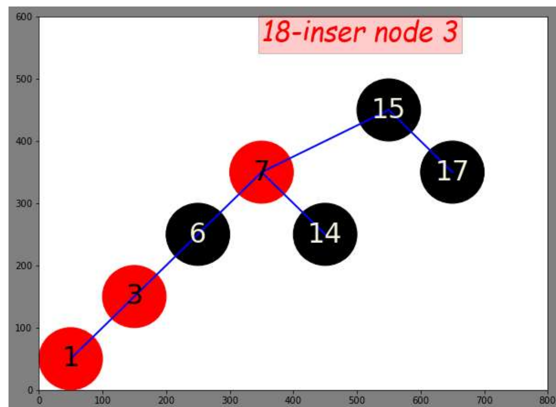
左旋的例子： 红黑树插入后，需要调整红黑树，调整方法就是先左旋（节点7），再右旋（节点15），来重新平衡

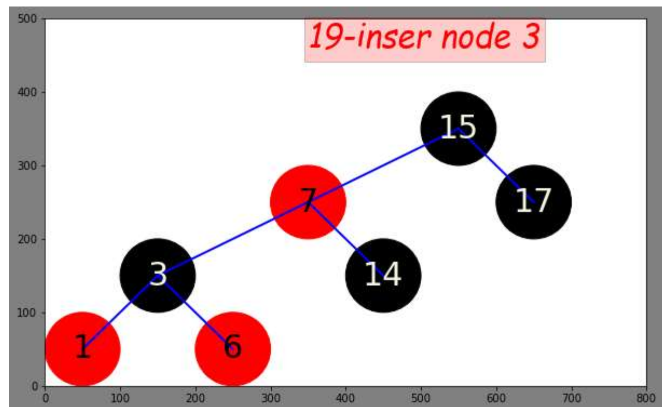






右旋的例子： 红黑树插入后，需要调整红黑树，调整方法就是右旋（节点6），来重新平衡





下面的Python代码给出了具体实现：

In [5]:

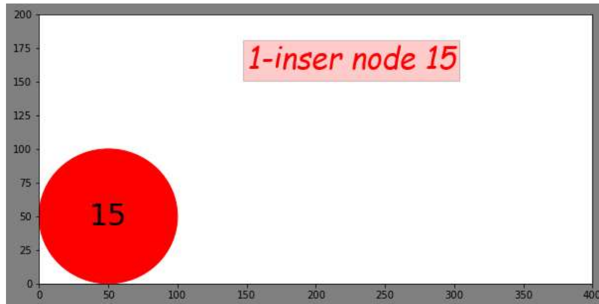
```
1  class RBTree:
2      '''
3      红黑树 五大特征
4      性质一：节点是红色或者是黑色；
5      性质二：根节点是黑色；
6      性质三：每个叶节点（NIL或空节点）是黑色；
7      性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节
8      性质五：从任一节点到其没个叶节点的所有路径都包含相同数目的黑色节点
9      '''
10     def __init__(self):
11         self.root = None
12
13     def left_rotate(self, node):
14         '''
15          * 左旋示意图：对节点x进行左旋
16          *     parent               parent
17          *    /                       /
18          *   node                   right
19          *  / \                     / \
20          * ln  right   ----->     node  ry
21          *    / \                 / \
22          *   ly ry               ln ly
23          * 左旋做了三件事：
24          * 1. 将right的左子节点ly赋给node的右子节点,并将node赋给right左子节点l
25          * 2. 将right的左子节点设为node，将node的父节点设为right
26          * 3. 将node的父节点parent(非空时)赋给right的父节点，同时更新parent的子
27         :param node: 要左旋的节点
28         :return:
29         '''
30         parent = node.parent
31         right = node.right
32
33         # 把右子子点的左子点节   赋给右节点 步骤1
34         node.right = right.left
35         if node.right:
36             node.right.parent = node
37
38         #把 node 变成基右子节点的左子节点 步骤2
39         right.left = node
40         node.parent = right
41
42         # 右子节点的你节点更并行为原来节点的父节点。 步骤3
43         right.parent = parent
44         if not parent:
45             self.root = right
46         else:
```

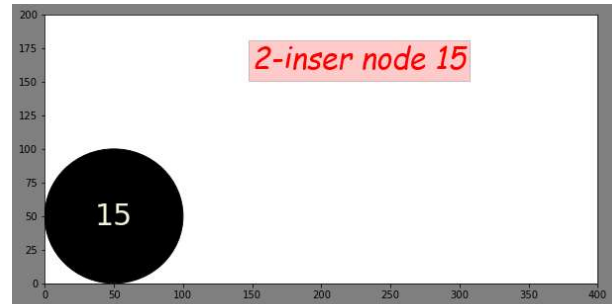
```
47                if parent.left == node:
48                    parent.left = right
49                else:
50                    parent.right = right
51            pass
52
53        def right_rotate(self, node):
54            print("right rotate", node.val)
55            '''
56             * 左旋示意图：对节点y进行右旋
57             *        parent           parent
58             *       /                   /
59             *      node                left
60             *     /    \               / \
61             *    left  ry   ----->   ln  node
62             *   / \                     / \
63             * ln  rn                   rn ry
64             * 右旋做了三件事：
65             * 1. 将left的右子节点rn赋给node的左子节点,并将node赋给rn右子节点的父节
66             * 2. 将left的右子节点设为node，将node的父节点设为left
67             * 3. 将node的父节点parent(非空时)赋给left的父节点，同时更新parent的子
68            :param node:
69            :return:
70            '''
71            parent = node.parent
72            left = node.left
73
74            # 处理步骤1
75            node.left = left.right
76            if node.left:
77                node.left.parent = node
78
79             # 处理步骤2
80            left.right = node
81            node.parent = left
82
83             # 处理步骤3
84            left.parent = parent
85            if not parent:
86                self.root = left
87            else:
88                if parent.left == node:
89                    parent.left = left
90                else:
91                    parent.right = left
92            pass
```

3.2 插入操作

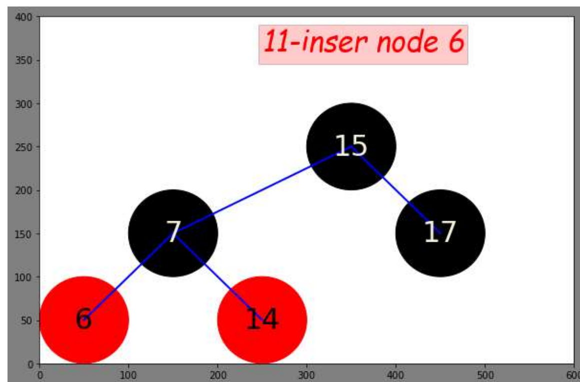
如果插入的是黑色节点时，则每次插入都会违返性质5, 都需要重新调整树。所以 插入时，每次都认为只插入红色节点。这样调整的次数就会减少很多。 但是还是有要调整的情况：

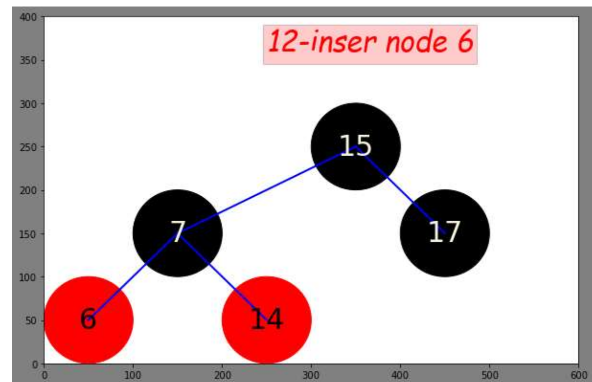
1.如果插入的是根节点，则直接把点变成黑色（性质2）， 示例中插入第一个节点15的情况，如下图所示：





2. 如果插入的节点的父节点是黑色节点，则不调整颜色。示例中插入一个节点6的情况，如下图所示：

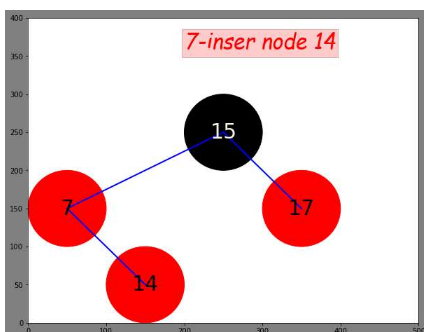


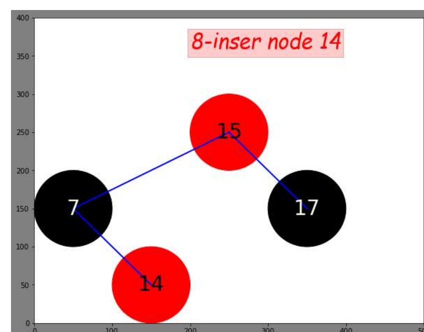


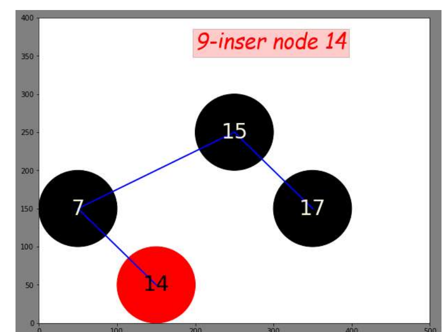
3.如果插入节点的父节点的红色节点（违反性质4），且父节点的兄弟节点为红色节点。

（1） 把父节点及其兄弟节点变成黑色，把祖父节点变成红色（使其不违反性质5）；

（2） 再检查祖父节点是否违反红黑树的性质（1或4）。 如下图所示：

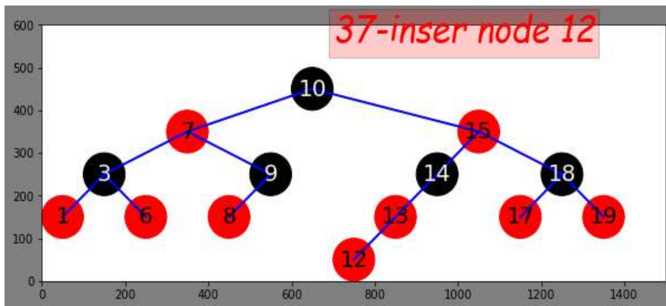


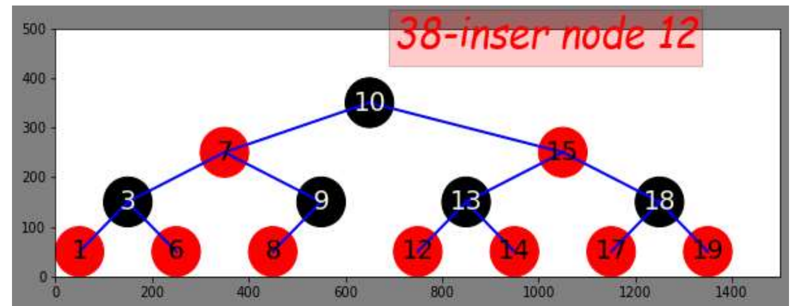




4. 如果插入节点的父节点的红色节点（违反性质四），且父节点的兄弟节点为黑色节点。 并且插入节点，父节点，及祖父节点同侧。 即node = node.parent.left && node.parent = node.parent.parent.left(同左则), 或 node = node.parent.right && node.parent = node.parent.parent.right（同右则）。

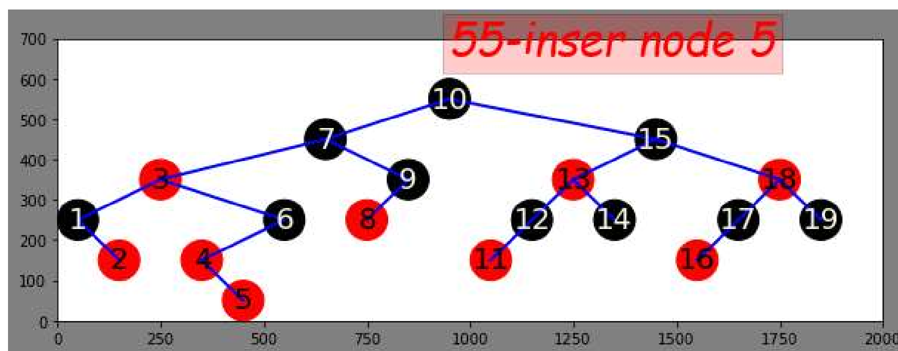
处理方法： 把父节点变成黑色节点，把祖父节点变成红色节点， 同时反向旋转祖父节点（同左则，右旋； 同右则左旋）。示例中，插入 节点12就是此种变形。如下图所示：

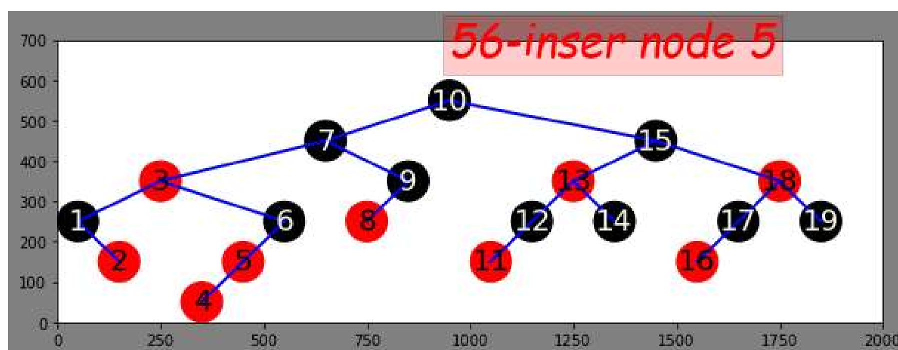


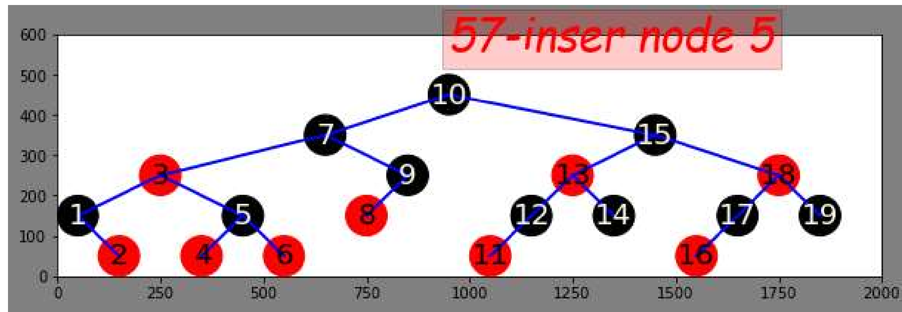


5. 如果插入节点的父节点的红色节点（违反性质四），且父节点的兄弟节点为黑色节点。 并且插入节点，父节点，及祖父节点不在同侧。

处理方法： 旋转父节点，使其变成同侧（第4种情况）， 再根据情况4来处理。示例中，插入 节点5就是此种变形。如下图所示：

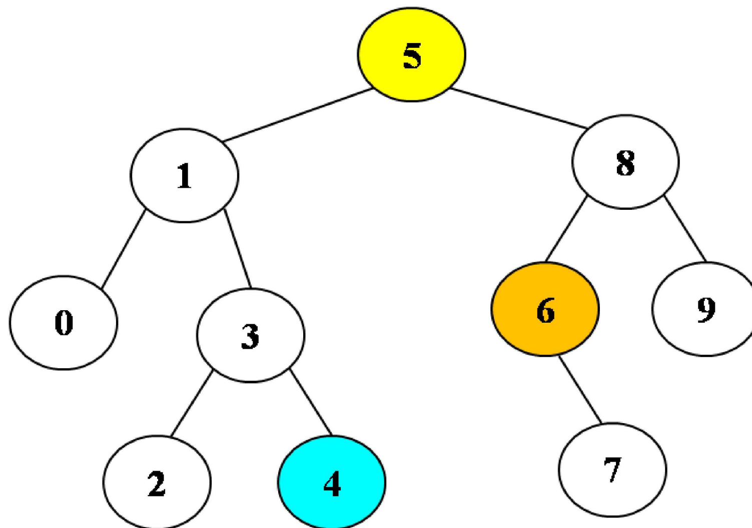






3.3 删除操作

两个节点的概念:
前驱节点: 节点的左子树中， 最大值的节点。
后继节点: 节点的右子树中，最小值的节点。



如上图所示：根结点5的前驱为4；根结点5的后继为6

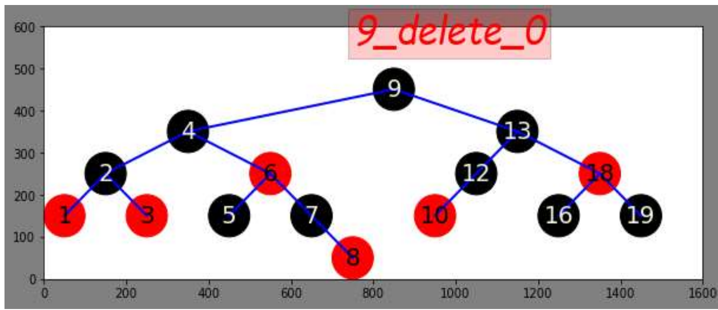
删除操作分成了三个步骤:

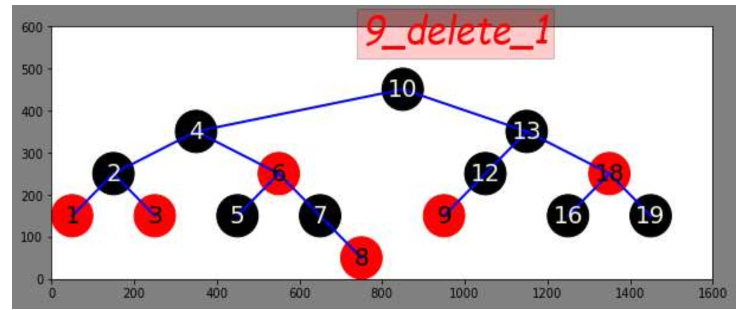
（1）获取要真正删除的叶子节点（把删除操作都归为删除叶子节点操作）。目的：把要删除的点，往叶子节点推。使删除操作变成删除叶子节点的操作。找到要删除节点的后继节点或前驱节点，如果存在，则替换掉当前节点。再以替换后的节点，找到后继或前驱节点。替换掉，直到无后继或前驱节点。

（2）对红黑树进行调整，使删除节点后的树，不违反红黑树的性质。

（3）真正的删除节点

获取删除节点9示例：9与10（后继节点）互换 变成了后面图的样子 将问题由删除根节点9，变成了删除叶子节点9.





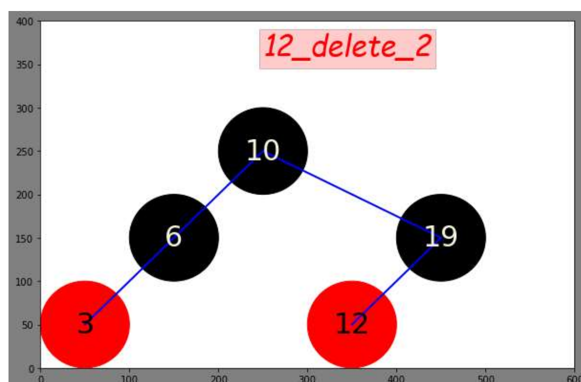
下面的Python代码给出了具体实现：

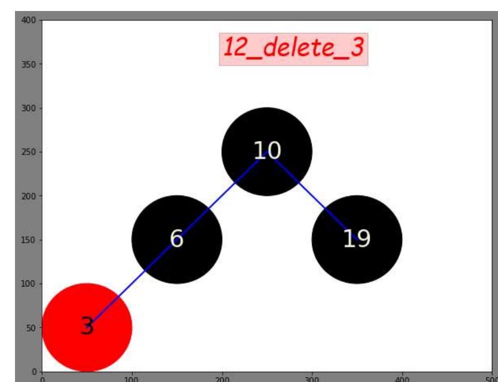
In [6]:

```
def pre_delete_node(self, node):
    '''
    删除前检查，返回最终要删除的点
    :param node:
    :return:
    '''
    post_node = self.get_post_node(node)
    if post_node:
        node.val, post_node.val = post_node.val, node.val
        return self.pre_delete_node(post_node)
    pre_node = self.get_pre_node(node)
    if pre_node:
        pre_node.val, node.val = node.val, pre_node.val
        return self.pre_delete_node(pre_node)
    #没有前驱节点，也没有后续节点
    return node
```

删除前调整红黑树:目的使删除节点后的树还是一棵红黑树

（1） 删除的节点是红色节点，可不需要调整直接删除，如下图所示：





（2） 删除的节点是根直接，直接删除

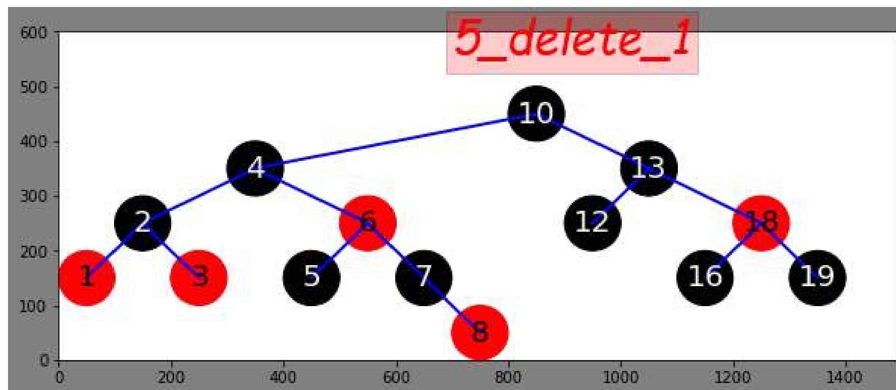
（3）如果是黑色节点，兄弟节点是红色节点， 旋转父节点： 把你节点变成黑色，兄弟节点变黑色。 重新平衡。

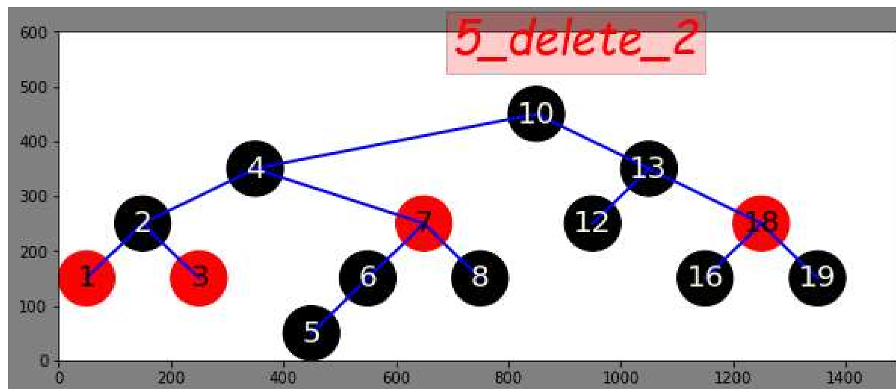
（4）删除黑色节点，兄弟结点也是黑色节点， 且兄弟节点的要么没有子节点，要么所有子节点都是黑色节点。

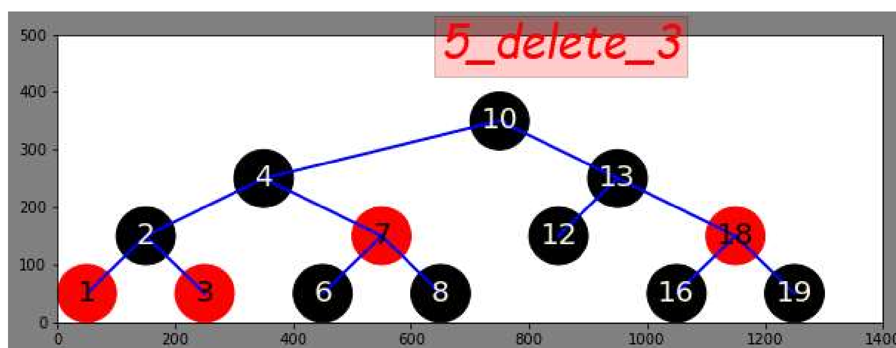
- 直接将兄弟节点变成红色节点
- 如果父节点是红色节点，直接把父节点变成黑色节点（调整结束）
- 如果父节点是黑色节点，再检查当前节点（递归检查）

（5）删除黑色节点，兄弟结点也是黑色节点， 兄弟节点的同则子节点（如果兄弟节点为左节点，则为史弟节点的左节点，右节点同理）为红色节点

- 变色： 兄弟同则子节点设置成黑色。 兄弟节点（黑色）和父节点（可能是红色，也可能是黑色）互换颜色
- 旋转父节点。







下面的Python代码给出了具体实现：

In [11]:

```
1   def check_delete_node(self, node):
2           '''
3           检查删除节点node
4           :param node:
5           :return:
6           '''
7           if self.root == node or node.is_red_node():
8               return
9
10          node_is_left = node.parent.left == node
11          brother = node.parent.right if node_is_left else node.parent.left
12          #brother 必不为空
13          if brother.is_red_node():
14              # 如果是黑色节点，兄弟节点是红色节点， 旋转父节点： 把你节点变成黑
15              if node_is_left:
16                  self.left_rotate(node.parent)
17              else:
18                  self.right_rotate(node.parent)
19              node.parent.set_red_node()
20              brother.set_black_node()
21              print("check node delete more ")
22              #再重新检查当前节点
23              self.check_delete_node(node)
24              return
25
26          all_none = not brother.left and not brother.right
27          all_black = brother.left and brother.right and brother.left.is_black_
28          if all_none or all_black:
29              brother.set_red_node()
30              if node.parent.is_red_node():
31                  node.parent.set_black_node()
32                  return
33              self.check_delete_node(node.parent)
34              return
35
36          #检查兄弟节点的同则子节点存丰并且是是红色节点
37          brother_same_right_red = node_is_left and brother.right and brother.r
38          brother_same_left_red = not node_is_left and brother.left and brother
39          if brother_same_right_red or brother_same_left_red:
40              if node.parent.is_red_node():
41                  brother.set_red_node()
42              else:
43                  brother.set_black_node()
44              node.parent.set_black_node()
45
46              if brother_same_right_red:
```

```
47                  brother.right.set_black_node()
48                  self.left_rotate(node.parent)
49              else:
50                  brother.left.set_black_node()
51                  self.right_rotate(node.parent)
52
53              return
54
55          # 检查兄弟节点的异则子节点存丰并且是是红色节点
56          brother_diff_right_red = not node_is_left and brother.right and broth
57          brother_diff_left_red = node_is_left and brother.left and brother.lef
58          if brother_diff_right_red or brother_diff_left_red:
59              brother.set_red_node()
60              if brother_diff_right_red:
61                  brother.right.set_black_node()
62                  self.left_rotate(brother)
63              else:
64                  brother.left.set_black_node()
65                  self.right_rotate(brother)
66
67              self.check_delete_node(node)
68              return
```

4. 红黑树的用途

1.红黑树在Linux非实时任务调度中的应用

Linux 的稳定内核版本在 2． 6． 24 之后，使用了新的调度程序 CFS，所有非实时可运行进程都以虚拟运行时间为 key 值挂在一棵红黑树上，以完成更公平高效地调度所有任务。CFS 弃用 active /expired 数组和动态计算优先级，不再跟踪任务的睡眠时间和区别是否交互任务，并且在调度中采用基于时间计算键值的红黑树来选取下一个任务，根据所有任务占用 CPU 时间的状态来确定调度任务优先级。

2.红黑树在Linux虚拟内存中的应用

32 位 Linux 内核虚拟地址空间划分 0 - 3G 为用户空间，3 - 4G 为内核空间，因此每个进程可以使用 4GB的虚拟空间。同时，Linux 定义了虚拟存储区域(VMA) 以便于更好表示进程所使用的虚拟空间，每个 VMA是某个进程的一段连续虚拟空间，其中的单元具有相同的特征，所有的虚拟区域按照地址排序由指针链接为一个链表。当发生缺页中断时搜索 VMA 到指定区域时，则需要频繁操作，因此选用了红黑树以减少查找时间。

2.红黑树在检测树的平衡性上的应用

红黑树是一种自平衡二叉搜索树，它的每个节点都被“着色”为红色或者黑色，这些节点的颜色被用来检测树的平衡性。红黑树作为嵌入式数据库中的索引机制，可以获得更好的性能，对于SQLite数据库，可以采用红黑树实现索引机制的优化。

5. 完整代码

In [ ]:

```
__author__ = 'jerome'
import functools
from tree_plt import show_rb_tree, save_rb_tree
import random

class RBNode:
    def __init__(self, val, color="R"):
        self.val = val
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

    def is_black_node(self):
        return self.color == "B"

    def is_red_node(self):
        return self.color == "R"

    def set_black_node(self):
        self.color = "B"

    def set_red_node(self):
        self.color = "R"

    def print(self):
        if self.left:
            self.left.print()
        print(self.val)
        if self.right:
            self.right.print()

def tree_log(func):
    @functools.wraps(func)
    def function(a, b):
        save_rb_tree(a.root, "{}-{}".format(a.index, a.action))
        a.index += 1
        func(a, b)
        save_rb_tree(a.root, "{}-{}".format(a.index, a.action))
        a.index += 1

    return function


class RBTree:
    '''
```

```
47      红黑树 五大特征
48      性质一：节点是红色或者是黑色；
49      性质二：根节点是黑色；
50      性质三：每个叶节点（NIL或空节点）是黑色；
51      性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节
52      性质五：从任一节点到其没个叶节点的所有路径都包含相同数目的黑色节点
53      '''
54      def __init__(self):
55          self.root = None
56          self.index = 1
57          self.action = ""
58  
59      def left_rotate(self, node):
60          '''
61           * 左旋示意图：对节点x进行左旋
62           *     parent               parent
63           *    /                       /
64           *   node                   right
65           *  / \                     / \
66           * ln  right   ----->     node  ry
67           *    / \                 / \
68           *   ly ry               ln ly
69           * 左旋做了三件事：
70           * 1. 将right的左子节点ly赋给node的右子节点,并将node赋给right左子节点l
71           * 2. 将right的左子节点设为node，将node的父节点设为right
72           * 3. 将node的父节点parent(非空时)赋给right的父节点，同时更新parent的子
73          :param node: 要左旋的节点
74          :return:
75          '''
76          parent = node.parent
77          right = node.right
78  
79          # 把右子子点的左子点节   赋给右节点 步骤1
80          node.right = right.left
81          if node.right:
82              node.right.parent = node
83  
84          #把 node 变成基右子节点的左子节点 步骤2
85          right.left = node
86          node.parent = right
87  
88          # 右子节点的你节点更并行为原来节点的父节点。 步骤3
89          right.parent = parent
90          if not parent:
91              self.root = right
92          else:
93              if parent.left == node:
94                  parent.left = right
```

```
95             else:
96                 parent.right = right
97         pass
98 
99     def right_rotate(self, node):
100         print("right rotate", node.val)
101         '''
102          * 左旋示意图：对节点y进行右旋
103          *         parent            parent
104          *        /                    /
105          *      node                 left
106          *     /    \                / \
107          *    left   ry    ----->   ln   node
108          *   / \                        / \
109          * ln   rn                    rn ry
110          * 右旋做了三件事：
111          * 1. 将left的右子节点rn赋给node的左子节点,并将node赋给rn右子节点的父节
112          * 2. 将left的右子节点设为node，将node的父节点设为left
113          * 3. 将node的父节点parent(非空时)赋给left的父节点，同时更新parent的子
114         :param node:
115         :return:
116         '''
117         parent = node.parent
118         left = node.left
119 
120         # 处理步骤1
121         node.left = left.right
122         if node.left:
123             node.left.parent = node
124 
125          # 处理步骤2
126         left.right = node
127         node.parent = left
128 
129          # 处理步骤3
130         left.parent = parent
131         if not parent:
132             self.root = left
133         else:
134             if parent.left == node:
135                 parent.left = left
136             else:
137                 parent.right = left
138         pass
139 
140     def insert_node(self, node):
141         '''
142         二叉树添加往红黑树中添加一个红色节点
```

```
143         :param node:
144         :return:
145         '''
146         if not self.root:
147             self.root = node
148             return
149 
150         cur = self.root
151         while cur:
152             if cur.val < node.val:
153                 if not cur.right:
154                     node.parent = cur
155                     cur.right = node
156                     break
157                 cur = cur.right
158                 continue
159 
160             if cur.val > node.val:
161                 if not cur.left:
162                     node.parent = cur
163                     cur.left = node
164                     break
165                 cur = cur.left
166         pass
167 
168     @tree_log
169     def check_node(self, node):
170         '''
171         检查节点及父节是否破坏了
172         性质二：根节点是黑色；
173         性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红
174         @@ 性质四可反向理解为， 节点和其父点必定不能够同时为红色节点
175         :param node:
176         :return:
177         '''
178         # 如果是父节点直接设置成黑色节点，退出
179         if self.root == node or self.root == node.parent:
180             self.root.set_black_node()
181             print("set black ", node.val)
182             return
183 
184         # 如果父节点是黑色节点，直接退出
185         if node.parent.is_black_node():
186             return
187 
188         # 如果父节点的兄弟节点也是红色节点,
189         grand = node.parent.parent
190         if not grand:
```

```
191                 self.check_node(node.parent)
192                 return
193             if grand.left and grand.left.is_red_node() and grand.right and grand.
194                 grand.left.set_black_node()
195                 grand.right.set_black_node()
196                 grand.set_red_node()
197                 self.check_node(grand)
198                 return
199 
200             # 如果父节点的兄弟节点也是黑色节点,
201             # node node.parent node.parent.parent 不同边
202             parent = node.parent
203             if parent.left == node and grand.right == node.parent:
204                 self.right_rotate(node.parent)
205                 self.check_node(parent)
206                 return
207             if parent.right == node and grand.left == node.parent:
208                 parent = node.parent
209                 self.left_rotate(node.parent)
210                 self.check_node(parent)
211                 return
212 
213             # node node.parent node.parent.parent 同边
214             parent.set_black_node()
215             grand.set_red_node()
216             if parent.left == node and grand.left == node.parent:
217                 self.right_rotate(grand)
218                 return
219             if parent.right == node and grand.right == node.parent:
220                 self.left_rotate(grand)
221                 return
222 
223         def add_node(self, node):
224             self.action = 'inser node {}'.format(node.val)
225             self.insert_node(node)
226             self.check_node(node)
227             pass
228 
229         def check_delete_node(self, node):
230             '''
231             检查删除节点node
232             :param node:
233             :return:
234             '''
235             if self.root == node or node.is_red_node():
236                 return
237 
238             node_is_left = node.parent.left == node
```

```
239            brother = node.parent.right if node_is_left else node.parent.left
240            #brother 必不为空
241            if brother.is_red_node():
242                # 如果是黑色节点，兄弟节点是红色节点， 旋转父节点： 把你节点变成黑
243                if node_is_left:
244                    self.left_rotate(node.parent)
245                else:
246                    self.right_rotate(node.parent)
247                node.parent.set_red_node()
248                brother.set_black_node()
249                print("check node delete more ")
250                #再重新检查当前节点
251                self.check_delete_node(node)
252                return
253
254            all_none = not brother.left and not brother.right
255            all_black = brother.left and brother.right and brother.left.is_black_
256            if all_none or all_black:
257                brother.set_red_node()
258                if node.parent.is_red_node():
259                    node.parent.set_black_node()
260                    return
261                self.check_delete_node(node.parent)
262                return
263
264            #检查兄弟节点的同则子节点存丰并且是是红色节点
265            brother_same_right_red = node_is_left and brother.right and brother.r
266            brother_same_left_red = not node_is_left and brother.left and brother
267            if brother_same_right_red or brother_same_left_red:
268
269                if node.parent.is_red_node():
270                    brother.set_red_node()
271                else:
272                    brother.set_black_node()
273                node.parent.set_black_node()
274
275                if brother_same_right_red:
276                    brother.right.set_black_node()
277                    self.left_rotate(node.parent)
278                else:
279                    brother.left.set_black_node()
280                    self.right_rotate(node.parent)
281
282                return
283
284            # 检查兄弟节点的异则子节点存丰并且是是红色节点
285            brother_diff_right_red = not node_is_left and brother.right and broth
286            brother_diff_left_red = node_is_left and brother.left and brother.lef
```

```
287         if brother_diff_right_red or brother_diff_left_red:
288             brother.set_red_node()
289             if brother_diff_right_red:
290                 brother.right.set_black_node()
291                 self.left_rotate(brother)
292             else:
293                 brother.left.set_black_node()
294                 self.right_rotate(brother)
295 
296             self.check_delete_node(node)
297             return
298 
299     def pre_delete_node(self, node):
300         '''
301         删除前检查，返回最终要删除的点
302         :param node:
303         :return:
304         '''
305         post_node = self.get_post_node(node)
306         if post_node:
307             node.val, post_node.val = post_node.val, node.val
308             return self.pre_delete_node(post_node)
309         pre_node = self.get_pre_node(node)
310         if pre_node:
311             pre_node.val, node.val = node.val, pre_node.val
312             return self.pre_delete_node(pre_node)
313         #没有前驱节点，也没有后续节点
314         return node
315 
316     def get_pre_node(self, node):
317         '''
318         获取 前驱 节点 ， 树中比node小的节点中最大的值
319         :param node:
320         :return:
321         '''
322         if not node.left:
323             return None
324         pre_node = node.left
325         while pre_node.right:
326             pre_node = pre_node.right
327         return pre_node
328 
329     def get_post_node(self, node):
330         '''
331         获取后续节点:
332         :param node:树中比node大的节点中最小的值
333         :return:
334         '''
```

```
335         if not node.right:
336             return None
337         post_node = node.right
338         while post_node.left:
339             post_node = post_node.left
340         return post_node
341 
342     def get_node(self, val):
343         '''
344         根据值查询节点信息
345         :param val:
346         :return:
347         '''
348         if not self.root:
349             return None
350         node = self.root
351         while node:
352             if node.val == val:
353                 break
354             if node.val > val:
355                 node = node.left
356                 continue
357             else:
358                 node = node.right
359         return node
360 
361     def delete_node(self, val):
362 
363         node = self.get_node(val)
364         if not node:
365             print("node error {}".format(val))
366             return
367         save_rb_tree(self.root, "{}_delete_0".format(val))
368         # 获取真正要删除的节点
369         node = self.pre_delete_node(node)
370         save_rb_tree(self.root, "{}_delete_1".format(val))
371         # node 节点必不为空，且子节点也都为空
372         self.check_delete_node(node)
373         save_rb_tree(self.root, "{}_delete_2".format(val))
374         #真正删除要删除的节点
375         self.real_delete_node(node)
376         save_rb_tree(self.root, "{}_delete_3".format(val))
377         pass
378 
379     def real_delete_node(self, node):
380         '''
381         真正删除节点函数
382         :param node:
```

```
383         :return:
384         '''
385         if self.root == node:
386             self.root = None
387             return
388         if node.parent.left == node:
389             node.parent.left = None
390             return
391         if node.parent.right == node:
392             node.parent.right = None
393         return
394 
395 if __name__ == '__main__':
396 
397     tree = RBTree()
398     data = list(range(1, 20))
399     random.shuffle(data)
400     print(data)
401     for i in data:
402         tree.add_node(RBNode(i))
403 
404     random.shuffle(data)
405     for i in data:
406         print("delete ", i)
407         tree.delete_node(i)
408 
409 
410     pass
```

引用及参考：

[1]《数据结构》李春葆著

[2] https://blog.csdn.net/net_wolf_007/article/details/79706498 (https://blog.csdn.net/net_wolf_007/article/details/79706498)

课后练习

1. 写出红黑树建树的Python代码或 C语言代码。
2. 对数组 $A=3,41,52,26,38,57,9,49$ 建立红黑树。

3. 对上述数组实现的红黑树实现添加{2，19，50，70}、删除{2，19，50，57}操作。

讨论、思考题、作业：

参考资料（含参考书、文献等）：算法笔记. 胡凡、曾磊，机械工业出版社，2016.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。