**红黑树**

**一.红黑树定义**

红黑树(Red-Black Tree，简称R-B Tree)，它一种特殊的二叉查找树。

红黑树是特殊的二叉查找树，意味着它满足二叉查找树的特征：任意一个节点所包含的键值，大于等于左孩子的键值，小于等于右孩子的键值。

除了具备该特性之外，红黑树还包括许多额外的信息。

红黑树的每个节点上都有存储位表示节点的颜色，颜色是红(Red)或黑(Black)。

**红黑树的特性:**

(1) 每个节点或者是黑色，或者是红色。

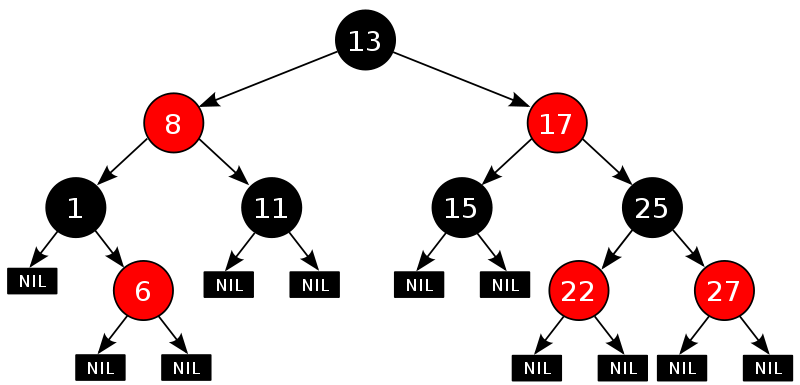
(2) 根节点是黑色

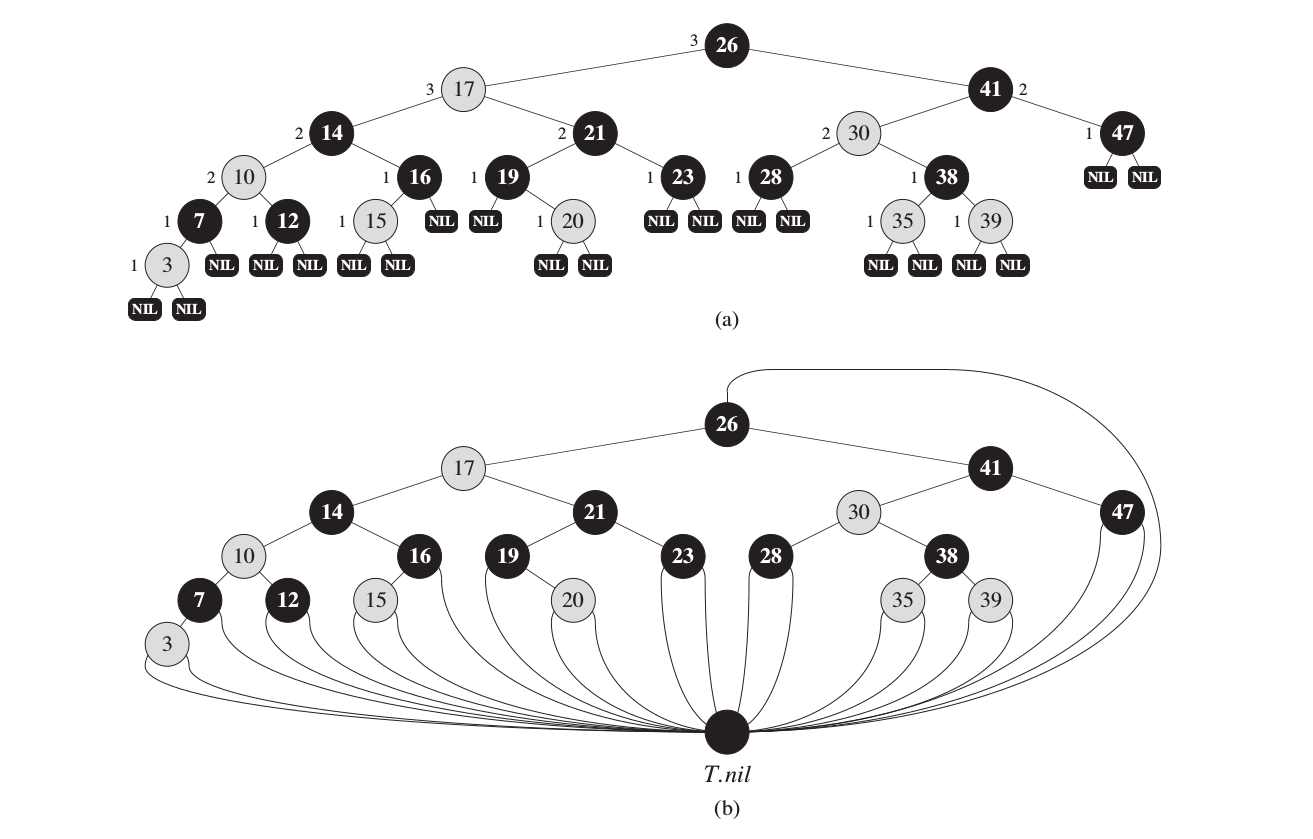
(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]

(4) 如果一个节点是红色的，则它的子节点必须是黑色的。

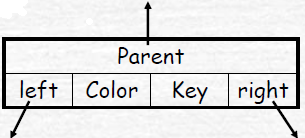
(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

关于它的特性，需要注意的是：  
第一，特性(3)中的叶子节点，是只为空(NIL或null)的节点。  
第二，特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对接近平衡的二叉树。





树结点的结构



**2红黑树性质**

结点的黑高bh(x):从某个结点出发（不含该节点），到达一个叶节点的任意一条简单路径上的黑色节点个数

**引理：一颗有n个内部节点的红黑树的高度至多为2lg(n+1)**

1. 先证明以任意节点x为根的子树中至少包含2^(bh(x))-1个内部节点

归纳法证明：

（1）如果x的高度为0，那么以x为根节点的子树包含2^(bh(x))-1 = 2^0-1=0个

（2）一个高度为正值的且有两个子节点的内部节点x。每个子节点有黑高bh(x)或者bh(x)-1。由归纳假设可以得出，每个子节点至少有2^(bh(x)-1)-1个内部节点

(3)因此，以x为根的子树至少包含(2^(bh(x)-1)-1)+ (2^(bh(x)-1)-1)+1 = 2^bh(x)-1

2. 设h是红黑树的高度,根据性质4，从根节点到叶节点的任何一条简单路径至少有一半的节点为黑色。因此，根的黑高至少为h/2,于是

n≥2^(h/2)-1

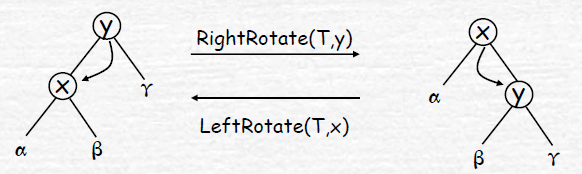
整理得到：**h ≤ 2log2(n+1)**

结论：

红黑树搜索、插入、删除的时间复杂度为O(logn)

**3 红黑树的旋转**

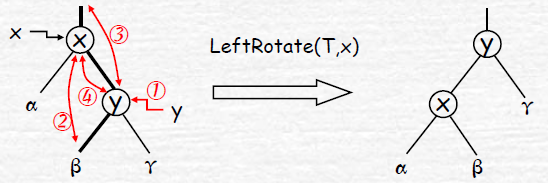
左、右旋转的图示



注：旋转过程中二叉搜索树(BST)性质不变：

α ≤x ≤ β ≤y ≤ γ

**左旋实现的步骤**



步骤解释：需要变动的是3根粗链

① y←right[x] //记录指向y节点的指针

② right[x]←left[y], p[left[y]]←x //β连到x右

③ p[y]←p[x], p[x]的左或右指针指向y //y连到p[x]

④ Left[y]←x, p[x]←y //x连到y左

左旋算法实现

|  |
| --- |
| LeftRotate(T, x)  {  y ← right[x]; //假定right[x] ≠ nil[T] right[x] ← left[y]; p[left[y]] ← x;  p[y] ← p[x];  if p[x]=nil[T] then //x是根  root[T] ← y; //修改树指针  else if x=left[p[x]] then // 如果x是左节点  left[p[x]] ← y;  else // 右节点  right[p[x]] ← y;  left[y] ← x; p[x] ← y; // 令x成为y 的左子树  }  T(n)=O(1) |

**4 、红黑树的插入**

step 1：将z节点按BST树规则插入红黑树中，z是叶子节点；

step 2：将z涂红；

step 3：调整使其满足红黑树的性质；

之所以插入涂成红色：

1.假若涂黑色，那么将会违背性质5, 性质5调整起来比较麻烦。 涂成红色，会违背性质2,性质4. 性质2可以忽略，性质4比较容易解决。

2.因为红黑树中黑节点至少是红节点的两倍，因此插入节点的父节点为黑色的概率较大，而此时并不需要作任何调整，因此效率较高。

|  |
| --- |
| RBInsert(T, z){  y ← nil[T]; //y用于记录：当前扫描节点的双亲节点  x ← root[T]; //从根开始扫描  while x ≠ nil[T] do //查找插入位置  {  y ← x;  if key[z] ＜ key[x] then //z插入x的左边  x ← left[x];  else  x ← right[x]; //z插入x的右边  }  p[z] ← y; //y是z的双亲  if y = nil[T] then //z插入空树  root[T] ← z; //z是根  else  if key[z] ＜ key[y] then  left[y] ← z; //z是y的左子插入  else  right[y] ← z; //z是y的右子插入  left[z] ← right[z] ← nil[T];  color[z] ← red;  RBInsertFixup(T, z);  }  时间：T(n)=O(logn) |

调整分析

idea：通过旋转和改变颜色，自下而上调整（z进行上溯），使树满足红黑树；

z插入后违反情况：

∵z作为红点，其两个孩子为黑（nil[T]）

∴满足性质1，3，5

可能违反性质2：z是根

可能违反性质4：p[z]是红

调整步骤：

(1)若z为根，将其涂黑；

(2)若z为非根，则p[z]存在

①若p[z]为黑，无需调整

②若p[z]为红，违反性质4，则需调整

∵ p[z]为红，它不为根

∴ p[p[z]]存在且为黑

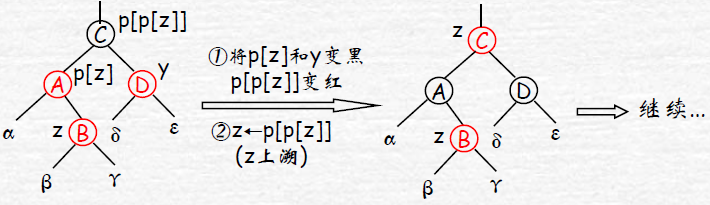
分6种情况进行调整：

其中

case1~3为z的双亲p[z]是其祖父p[p[z]]的左孩子，

case4~6为z的双亲p[z]是其祖父p[p[z]]的右孩子。

Case 1：z的叔叔y是红色



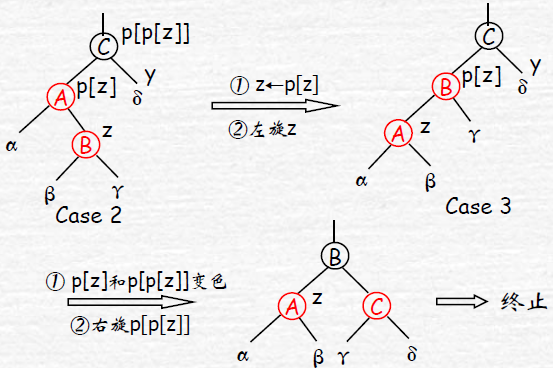
注：(1)变换后，新的z(上溯后)可能违反性质4，故调整最多至根；

(2)若红色传播到根，将根涂黑，则树的黑高增1；//临界处理

(3)z是p[z]的左、右孩子均一样处理；

Case 2：当z的叔叔y是黑色，且z是双亲p[z]的右孩子

Case 3：当z的叔叔y是黑色，且z是双亲p[z]的左孩子



插入调整算法

|  |
| --- |
| RBInsertFixup(T, z)  {  while ( color[p[z]]=red ) do //若p[z]为黑，无需调整，不进入此循环  {  if p[z]=left[p[p[z]]] then //case 1,2,3  {  y ← right[p[p[z]]]; //y是z的叔叔  if color[y]=red then //case 1  {  color[y]=black; color[p[z]]=black;  color[p[p[z]]]=red; z ← p[p[z]];  }  else //case 2 or case 3 y为黑  else //case 2 or case 3 y为黑  {  if z=right[p[z]] then //case 2  {  z ← p[z]; //上溯至双亲  leftRotate(T, z);  } //以下为case 3  color[p[z]]=black; color[p[p[z]]]=red;  RightRotate(T, p[p[z]]); //p[z]为黑，退出循环  } //case 1’s endif  } //case 2 or 3’ s  else //case 4,5,6’s 与上面对称  { … … }  } //endwhile  color[root[t]] ← black; |

插入算法的时间复杂性

调整算法的时间：O(logn)

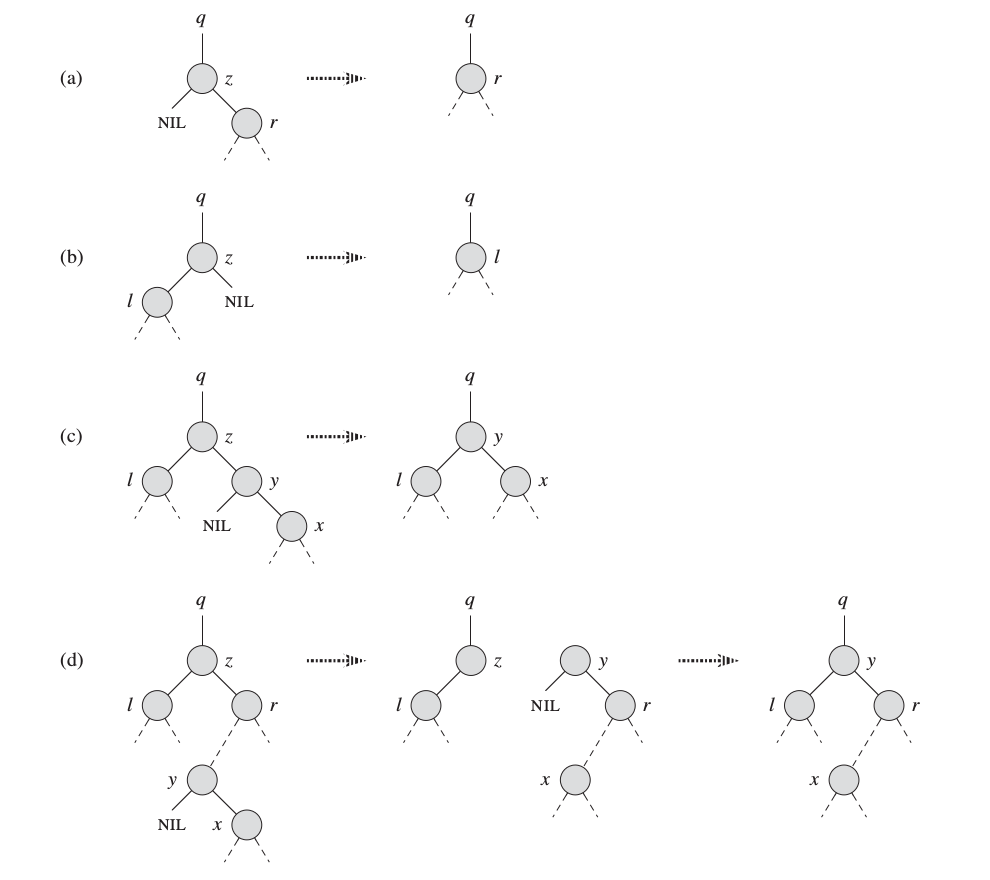
整个插入算法的时间：O(logn)

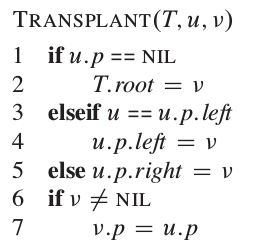
5、红黑树的删除

找到真正的删除点：当被删除结点n存在左右孩子时，真正的删除点应该是n的中序遍在前驱(或后继)

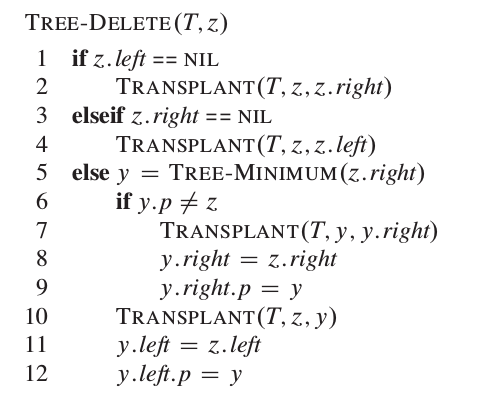
删除节点操作与BST是一样的，不过需要根据被删除节点颜色做一个调整

一般BST树删除节点操作

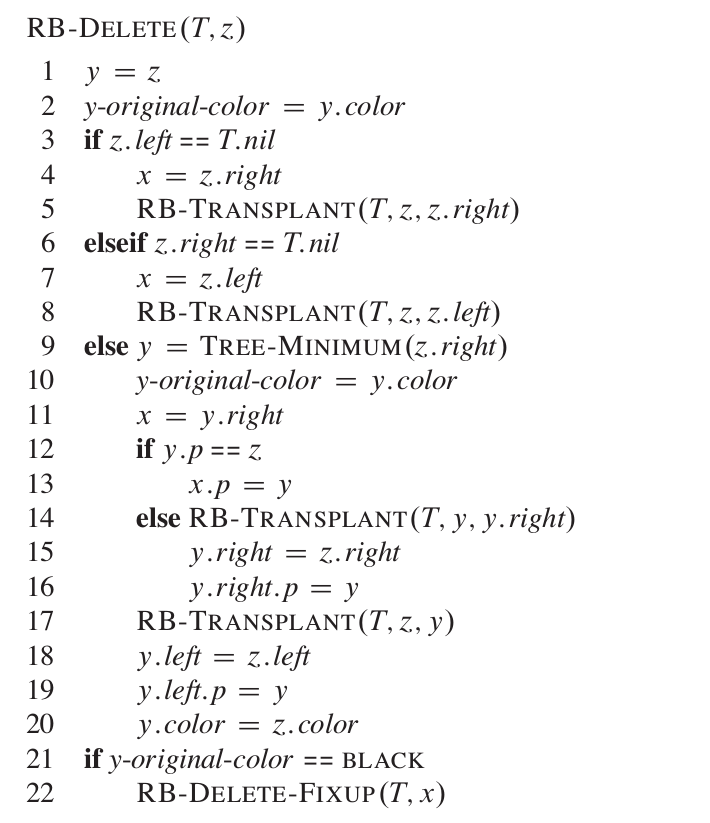




TRANSPLANT()：用一棵以v为根的子树来替换一棵以u为根的子树



RBT删除节点的分析讨论



讨论

x：或是y的唯一孩子；或是哨兵nil[T]

可以想象将y的黑色涂到x上，于是

- 若x为红，只要将其涂黑，调整即可终止；

- 若x为黑，将y的黑色涂上之后，x是一个双黑节点，违反性质1。

处理步骤如下

step 1：若x是根，直接移去多余一层黑色(树黑高减1)，终止；

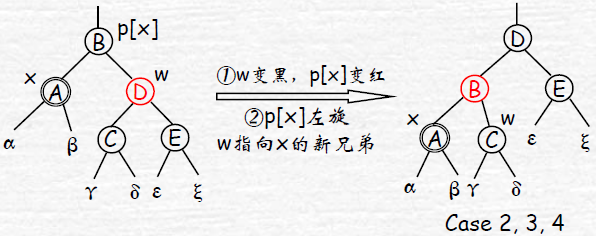
step 2：若x原为红，将y的黑色涂到x上，终止；

step 3：若x非根节点，且为黑色，则x为双黑。通过变色、旋转使多余黑色向上传播，直到某个红色节点或传到根；

**调整分8种情况**

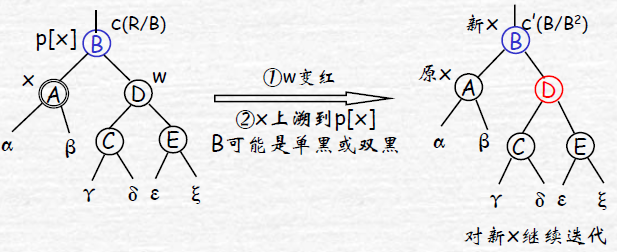
case 1~4为x是p[x]的左子；case 5~8为x是p[x]的右子

case 1：x的兄弟w是红色



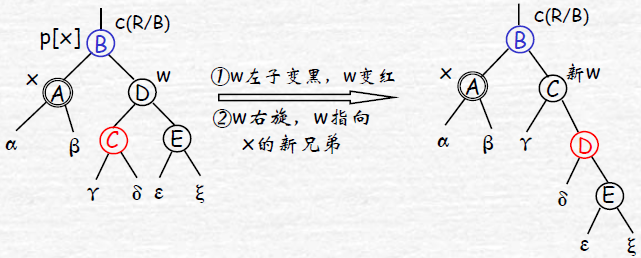
目标：将case1转为case2,3,4处理

case 2：x的黑兄弟w的两个孩子均为黑



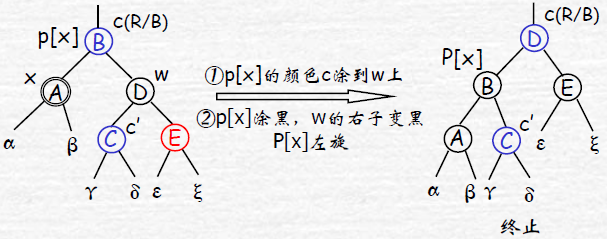
目标： x上移到B，通过A和D的黑色上移

case 3：x的黑兄弟w的右子为黑且左子为红



目标：将case3转为case4

case 4：x的黑兄弟w的右子为红(左子为黑或红)



目标：终结处理。x的黑色上移给B，B的原色下移给D，D将黑色下移给C和E，通过旋转解决矛盾点C

|  |
| --- |
| RB-DELETE-FIXUP(T, x) 恢复与保持红黑性质的工作  while x ≠ root[T] and color[x] = BLACK  do if x = left[p[x]]  then w ← right[p[x]]  if color[w] = RED  then color[w] ← BLACK ▹ Case 1  color[p[x]] ← RED ▹ Case 1  LEFT-ROTATE(T, p[x]) ▹ Case 1  w ← right[p[x]] ▹ Case 1  if color[left[w]] = BLACK and color[right[w]] = BLACK  then color[w] ← RED ▹ Case 2  x ← p[x] ▹ Case 2  else if color[right[w]] = BLACK  then color[left[w]] ← BLACK ▹ Case 3  color[w] ← RED ▹ Case 3  RIGHT-ROTATE(T, w) ▹ Case 3  w ← right[p[x]] ▹ Case 3  color[w] ← color[p[x]] ▹ Case 4  color[p[x]] ← BLACK ▹ Case 4  color[right[w]] ← BLACK ▹ Case 4  LEFT-ROTATE(T, p[x]) ▹ Case 4  x ← root[T] ▹ Case 4  else (same as then clause with "right" and "left" exchanged)  color[x] ← BLACK |

红黑树插入、删除算法总结

其平均和最差检索时间复杂度O(log2n)

为了恢复红黑树的平衡，需要自底向根进行调整

红黑树的结点组成

数据、颜色、左指针、右指针、父指针

如不使用父指针，也可以使用堆栈，

保存从根结点到新插入或删除结点的路径上遇到的每个结点，方便回溯

5 、红黑树的应用

典型的用途是：

C++ STL中的关联式容器：集合set、多重集合multiset、映射map、多重映射multimap等均采用了红黑树的变体

set<int> s; map<int, string>s

Java中的容器Set，Map的构造器TreeSet，TreeMap也使用了红黑树

在Linux内核中，用于组织虚拟区间的数据结构也是红黑树