AVL-tree-详解

阅读更多

1 前言

AVL-tree是一种平衡二叉树，搜索的平均复杂度是O(log n)。本篇博客将介绍AVL-tree的两种实现方式，其中第二种方式是参考<STL源码剖析>，较第一种方式更为简单高效，推荐第二种方式

2 定义

在计算机科学中，AVL树是最先发明的自平衡二叉查找树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树

节点的属性

1. **val**：节点的值
2. **left**：节点的左孩子
3. **right**：节点的右孩子
4. **parent**：节点的双亲
5. **height**：节点的高度

- 相比于普通的搜索二叉树，AVL树额外维护了一个高度属性，因为AVL树通过该高度属性来判断树的平衡性

平衡性质

1. 每个节点的左子树与右子树的高度最多不超过1
    - **节点的高度**：从给定节点到其最深叶节点所经过的边的数量

3 版本1

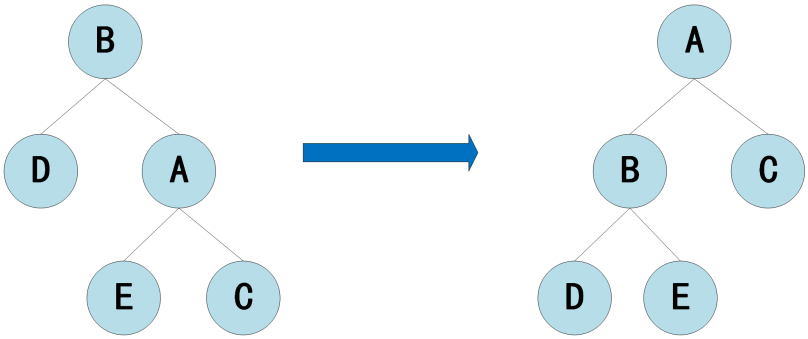
3.1 平衡性破坏分析

为了方便描述，定义几个符号

1. 对于一个节点X，调整(这里的调整特指旋转)之前其高度记为H_X
2. 仍然对于上述节点X，调整一次后高度记为H_{X+}

3.1.1 可旋性分析

首先分析左旋，见如下示意图



问题1：何时我们会进行左旋操作

只有当右子树的高度大于左子树的高度才会有左旋的需求。以上图为例，即$H_D=H_A-1$或$H_D=H_A-2$

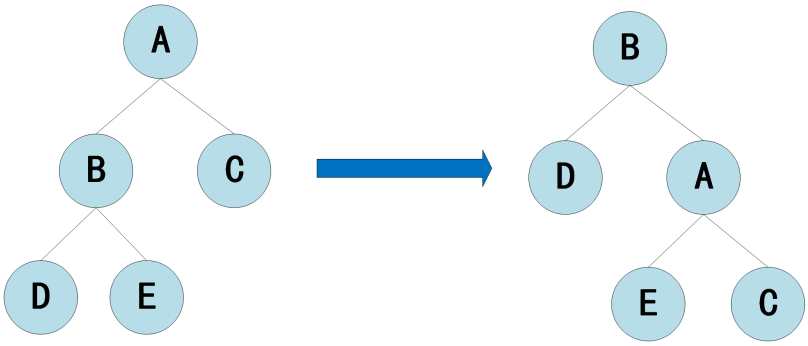
问题2：何时左旋后能够达到平衡状态

只有$H_C >= H_E$时，旋转后该子树的所有节点才满足AVL树的性质

下面对于问题2的结论进行分析

1. **旋转前，各节点高度如下**
    - $H_D=H_A-1$或$H_D=H_A-2$
    - $H_C=H_A-1$
    - $H_E=H_A-1$或$H_E=H_A-2$
2. **旋转后，各节点高度如下**
    - $H_{D+}=H_D=H_A-1$或H_A-2
    - $H_{E+}=H_E=H_A-1$或H_A-2
    - $H_{C+}=H_C=H_A-1$
    - 旋转后，B节点平衡，$H_B+=H_A$或$H_{B+}=H_A-1$
    - 旋转后，A节点平衡，$H_A+=H_A$或$H_{A+}=H_A+1$

然后分析右旋，见如下示意图



问题1：何时我们会进行右旋操作

只有当左子树的高度大于右子树的高度才会有右旋的需求。以上图为例，即$H_C=H_B-1$或$H_C=H_B-2$

问题2：何时右旋后能够达到平衡状态

只有$H_D >= H_E$时，旋转后该子树的所有节点才满足AVL树的性质

下面对于问题2的结论进行分析

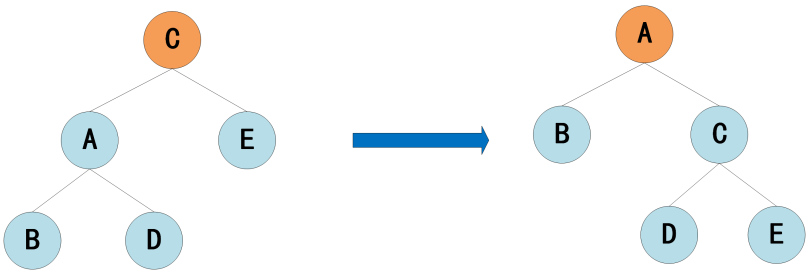
1. **旋转前，各节点高度如下**
    - $H_C=H_B-1$或$H_C=H_B-2$
    - $H_D=H_B-1$
    - $H_E=H_B-1$或$H_E=H_B-2$
2. **旋转后，各节点高度如下**
    - $H_{D+}=H_D=H_B-1$
    - $H_{E+}=H_E=H_B-1$或H_B-2
    - $H_{C+}=H_C=H_B-1$或H_B-2
    - 旋转后，A节点平衡，且$H_{A+}=H_B$或$H_{A+}=H_B-1$
    - 旋转后，B节点平衡，且$H_{B+}=H_B$或$H_{B+}=H_B+1$

注意：上述分析仅仅讨论了左旋或者右旋一棵子树时，旋转后该子树是否平衡，但是旋转后该子树的高度是可能变化的，也就是对于该子树的父节点而言，可能又会造成不平衡。因此在AVL性质维护函数中需要充分考虑到这一点

3.1.2 平衡性的维护

当平衡性质被破坏需要右旋来维护性质时

当C为平衡被破坏的节点，且C的左子树比右子树的高度大2，示意图如下



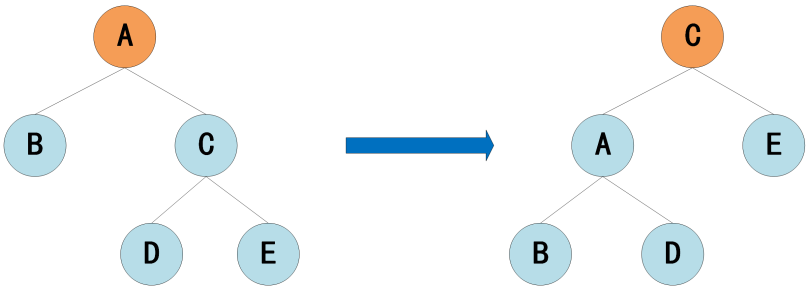
- 需要对C进行一次右旋
- 右旋的前提是$H_B >= H_D$
- 若不满足$H_B >= H_D$，则需要首先对A进行一次左旋，而左旋又存在前提
- 一直往下递归，直至满足旋转条件

其实，以上的过程存在一个可以优化的点：

假设对C节点的右旋条件不满足，即$H_B < H_D$，那么此时需要对A节点进行一次左旋。**但是A节点左旋后可能会导致A子树的高度发生变化(可能A节点的高度比旋转前少1)**。如果A子树的高度少1，那么对于节点C就成为一个平衡节点了，就不需要再进行右旋操作了。但是对于下面将要讲的伪代码中，并没有进行这样的优化。也就意味着不管A子树的高度是否发生变化，C的右旋仍然会执行，这样做并不会破坏平衡性(旋转前后都会处于平衡状态)，只是复杂度增高了

当平衡性质被破坏需要左旋来维护性质时

当A为平衡被破坏的节点，且A的右子树比左子树的高度大2，示意图如下



- 需要对A进行一次左旋
- 左旋的前提是$H_E >= H_D$
- 若不满足$H_E >= H_D$，则需要首先对C进行一次右旋，而右旋又存在前提
- 一直往下递归，直至满足旋转条件

这里存在一个相同的优化点，不再赘述

3.2 伪代码

3.2.1 基本操作

更新指定节点的高度，只有当该节点的左右孩子节点的高度都正确时，才能得到正确的结果

```
1 AVL-TREE-HEIGHT(T,x)
2 if x.left.height≥x.right.height    //左右节点均存在
3     x.height=x.left.height+1
4 else x.height=x.right.height+1
```

将一棵子树替换掉另一棵子树

```
1 AVL-TREE-TRANSPLANT(T,u,v)    //该函数与红黑树完全一致(都含有哨兵节点)
2 if u.p==T.nil
3     T.root=v
4 elseif u==u.p.left
5     u.p.left=v
```

```
else u.p.right=v
    v.p=u.p
```

3.2.2 左旋和右旋

左旋给定节点，更新旋转后节点的高度，并返回旋转后子树的根节点

```
AVL-TREE-LEFT-ROTATE(T,x)
y=x.right
x.right=y.left
if y.left≠T.nil
    y.left.p=x
y.p=x.p
if x.p== T.nil
    T.root=y
elseif x==x.p.left
    x.p.left=y
else x.p.right=y
y.left=x
x.p=y
AVL-TREE-HEIGHT(T,x)
AVL-TREE-HEIGHT(T,y)
//以上两行顺序不得交换
return y    //返回旋转后的子树根节点
```

右旋给定节点，更新旋转后节点的高度，并返回旋转后子树的根节点

```
AVL-TREE-RIGH-TROTATE(T,y)
x=y.left
y.left=x.right
if x.right≠T.nil
    x.right.p=y
x.p=y.p
if y.p==T.nil
    root=x
elseif y==y.p.left
    y.p.left=x
else y.p.right=x
x.right=y
y.p=x
AVL-TREE-HEIGHT(T,y)
AVL-TREE-HEIGHT(T,x)
//以上两行顺序不得交换
return x      //返回旋转后的子树根节点
```

3.2.3 性质维护

根据指定方向旋转给定节点，旋转后该子树满足平衡的性质，并且输入的节点必须满足如下性质

> X节点必定是以X为根节点的子树中**唯一**不满足平衡性的节点。意味着X节点的孩子子树的所有节点均满足平衡性。因此，必须从下往上找到第一个不平衡的节点来调用该方法

```
AVL-TREE-HOLD-ROTATE(T,x,orientation)
let stack1,stack2 be two stacks//不考虑实际用到的大小，直接用树的大小来分配堆栈空
stack1.push(x)
stack2.push(orientation)
cur=Nil
rotateRoot=Nil //对x尝试旋转后，返回最终旋转后的根节点
curOrientation=INVALID;
while(!stack1.Empty())
    cur=stack1.top()
    curOrientation=stack2.top()
    if curOrientation==LEFT //需要对cur尝试进行左旋
        if cur.right.right.height≥cur.right.left.height
            stack1.pop()
            stack2.pop()
            rotateRoot=AVL-TREE-LEFT-ROTATE(T,cur)
        else
            stack1.push(cur.right)//否则cur右孩子需要尝试进行右旋来调整
            stack2.push(RIGHT);
    elseif curOrientation ==RIGHT//需要对cur尝试进行右旋
        if cur.left.left.height≥cur.left.right.height
            stack1.pop()
            stack2.pop()
            rotateRoot=AVL-TREE-RIGHT-ROTATE(T,cur)
        else
            stack1.push(cur.left) //否则cur左孩子需要尝试进行左旋来调整
            stack2.push(LEFT)
return rotateRoot
```

3.2.4 插入

插入一个节点

```
AVL-TREE-TREE-INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil//循环结束时x指向空，y指向上一个x
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y//将这个叶节点作为z的父节点
```

```
10 if y==T.nil
11     T.root=z
12 elseif z.key<y.key
13     y.left=z
14 else y.right=z
15 z.left=T.nil
16 z.right=T.nil
17 AVL-TREE-FIXUP(T,z)
```

从指定节点向上遍历查找不平衡的节点，并维护平衡树的性质

```
1  AVL-TREE-FIXUP(T,y)
2  if y==T.nil//为了使删除函数也能调用该函数，因为删除函数传入的参数可能是哨兵
3      y=y.p
4  while(y≠T.nil) //沿着y节点向上遍历该条路径
5      AVL-TREE-HEIGHT(y)
6      if y.left.height==y.right.height+2 //左子树比右子树高2
7          y= AVL-TREE-HOLD-ROTATE(T,y,2)
8      elseif y.right.height=y.left.height+2
9          y= AVL-TREE-HOLD-ROTATE(T,y,1)
10     y=y.p
```

3.2.5 删除

删除一个节点

```
1  AVL-TREE-DELETE(T,z)
2  y=z   //x指向将要移动到y原本位置的节点，或者原本y节点的父节点
3  if z.left==T.nil
4      x=y.right
5      AVL-TREE-TRANSPLANT(T,z,z.right)
6  elseif z.right==T.nil
7      x=y.left
8      AVL-TREE-TRANSPLANT(T,z,z.left)
9   else y=AVL-TREE-MINIMUM(z.right) //找到z的后继，由于z存在左右孩子，故后继为右子
10     x=y.right
11     if y.p==z//如果y是z的右孩子，需要将x的parent指向y(使得x为哨兵节点也满足)
12         x.p=y
13     else AVL-TREE-TRANSPLANT (T,y,y.right)
14         y.right=z.right
15         y.right.p=y
16     AVL-TREE-TRANSPLANT (T,z,y)
17     y.left=z.left
18     y.left.p=y
19 AVL-TREE-FIXUP(T,x)
```

4 版本2

与版本1的实现不同，这个版本的分析将会更加简单，实现效率也会更高

4.1 几类不平衡情况

当插入一个节点后，某节点A为从插入节点往上找到的第一个平衡性被破坏的节点，**可以分为如下四种情况，又可分为两大类**

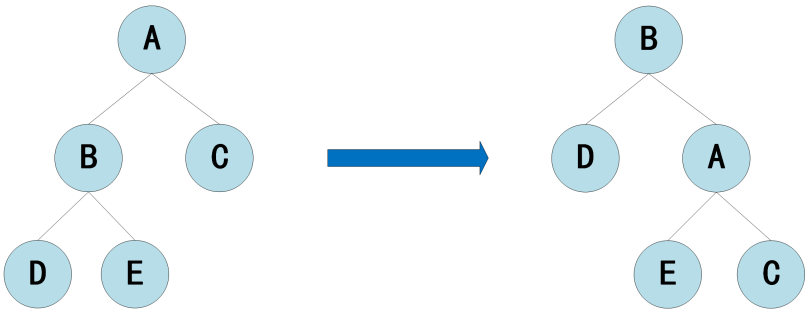
1. **第一类：外侧**
    - 插入点位于A的左子节点的左子树-左左
    - 插入点位于A的右子节点的右子树-右右
2. **第二类：内侧**
    - 插入点位于A的左子节点的右子树-左右
    - 插入点位于A的右子节点的左子树-右左

4.1.1 第一类不平衡(以左左为例)

为了方便描述，定义几个符号

1. 对于一个节点X，调整(这里的调整特指旋转)之前其高度记为H_X
2. 仍然对于上述节点X，调整一次后高度记为H_{X+}

插入点位于A的左子节点的左子树-左左，见如下示意图



调整前，各节点的高度如下

- $H_B=H_C+2$
- $H_A=H_C+2$(为什么A和B高度相同，因为B的高度已经更新过了，而A仍然是是插入新节点之前的高度，即尚未维护A节点的height字段)
- $H_D=H_B-1=H_C+1$
- H_E必定小于H_D(否则在新节点插入到节点D为根节点的子树之前，A节点就是不平衡的)，因此$H_E=H_C$

右旋调整后，各节点的高度如下

- $H_{D+}=H_D=H_C+1$
- $H_{E+}=H_E=H_C$

- $H_{C+}=H_C$
- 由于$H_{E+}=H_{C+}$，于是A节点平衡，且$H_{A+}=H_C+1$
- B节点也是平衡的，且$H_{B+}=H_C+2$

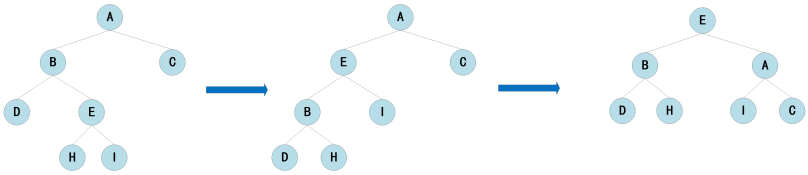
可以发现，调整前后子树根节点的高度都是H_C+2，因此该节点上层的节点的平衡性不会被破坏，于是通过一次右旋，不平衡性即被消除

4.1.2 第二类不平衡(以左右为例)

为了方便描述，定义几个符号

1. 对于一个节点X，调整(这里的调整特指旋转)之前其高度记为H_X
2. 仍然对于上述节点X，调整一次后高度记为H_{X+}
3. 仍然对于上述节点X，调整二次后高度记为H_{X++}

插入点位于A的左子节点的右子树-左右，见如下示意图



调整前，各节点的高度如下

- $H_B=H_C+2$
- $H_A=H_C+2$(为什么A和B高度相同，因为B的高度已经更新过了，而A仍然是是插入新节点之前的高度，即尚未维护A节点的height字段)
- $H_E=H_B-1=H_C+1$
- H_D必定小于H_E，因此$H_D=H_C$
- H_H与H_I至少有一个是H_C，另一个可以是H_C或H_C-1

对B节点进行一次左旋后，各节点高度如下

- $H_{D+}=H_D=H_C$
- $H_{H+}=H_H=H_C$ or H_C-1
- $H_{I+}=H_I=H_C$ or H_C-1
- $H_{C+}=H_C$
- $H_{A+}=H_A=H_C+2$
- $H_{B+}=H_C+1$
- 当$H_{I+}=H_C-1$时，节点E可能是不平衡的，但是没关系，这只是个中间状态，$H_E=H_C+2$

对A节点进行一次右旋，各节点高度如下

- $H_{D++}=H_{D+}=H_C$

- $H_{H++}=H_{H+}=H_C$ or H_C-1
- $H_{I++}=H_{I+}=H_C$ or H_C-1
- $H_{C++}=H_{C+}=H_C$
- 旋转后，B节点平衡，$H_{B++}=H_C+1$
- 旋转后，A节点平衡，$H_{A++}=H_C+1$
- 因此旋转后，E节点平衡，$H_{E++}=H_C+2$

可以发现，调整前后子树根节点的高度都是HC+2，因此该节点上层的节点的平衡性不会被破坏，于是通过一次左旋和一次右旋，不平衡性即被消除

4.2 伪代码

4.2.1 基本操作

更新指定节点的高度，只有当该节点的左右孩子节点的高度都正确时，才能得到正确的结果

```
AVL-TREE-HEIGHT(T,x)
if x.left.height≥x.right.height    //左右节点均存在
    x.height=x.left.height+1
else x.height=x.right.height+1
```

将一棵子树替换掉另一棵子树

```
AVL-TREE-TRANSPLANT(T,u,v)    //该函数与红黑树完全一致(都含有哨兵节点)
if u.p==T.nil
    T.root=v
elseif u==u.p.left
    u.p.left=v
else u.p.right=v
    v.p=u.p
```

4.2.2 左旋和右旋

左旋给定节点，更新旋转后节点的高度，并返回旋转后子树的根节点

```
 AVL-TREE-LEFT-ROTATE(T,x)
 y=x.right
 x.right=y.left
 if y.left≠T.nil
     y.left.p=x
 y.p=x.p
 if x.p== T.nil
     T.root=y
 elseif x==x.p.left
     x.p.left=y
```

```
else x.p.right=y
y.left=x
x.p=y
AVL-TREE-HEIGHT(T,x)
AVL-TREE-HEIGHT(T,y)
//以上两行顺序不得交换
return y    //返回旋转后的子树根节点
```

右旋给定节点，更新旋转后节点的高度，并返回旋转后子树的根节点

```
AVL-TREE-RIGHT-ROTATE(T,y)
x=y.left
y.left=x.right
if x.right≠T.nil
    x.right.p=y
x.p=y.p
if y.p==T.nil
    root=x
elseif y==y.p.left
    y.p.left=x
else y.p.right=x
x.right=y
y.p=x
AVL-TREE-HEIGHT(T,y)
AVL-TREE-HEIGHT(T,x)
//以上两行顺序不得交换
return x      //返回旋转后的子树根节点
```

4.2.3 插入

插入一个节点

```
AVL-TREE-INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil//循环结束时x指向空，y指向上一个x
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y//将这个叶节点作为z的父节点
if y==T.nil
    T.root=z
elseif z.key<y.key
    y.left=z
else y.right=z
z.left=T.nil
```

```
z.right=T.nil
AVL-TREE-BALANCE-FIX (T,z)
```

维护平衡性质，分别讨论第一类第二类的四种不平衡情况

```
AVL-TREE--BALANCE-FIX(T,z)
originHigh=z.h
AVL-TREE-HEIGHT(z)
r=z
if z.left.h==z.right.h+2
    if z.left.left.h>=z.left.right.h   //第一类，等号在插入过程中不可能取到，删
        r=AVL-TREE-RIGHT-ROTATE(z)
    elseif z.left.left.h<z.left.right.h     //第二类
        AVL-TREE-LEFT-ROTATE(z.left)
        r=AVL-TREE-RIGHT-ROTATE(z)
    //不可能出现左右子树高度相同的情况，但是DELETE-FIX中可能出现，注意
elseif z.right.h==z.left.h+2
    if z.right.right.h>=z.right.left.h   //第一类，等号在插入过程中不可能取到，
        r=AVL-TREE-LEFT-ROTATE(z)
    elseif z.right.right.h<z.right.left.h      //第二类
        AVL-TREE-RIGHT-ROTATE(z.right)
        r=AVL-TREE-LEFT-ROTATE(z)
    //不可能出现左右子树高度相同的情况，但是DELETE-FIX中可能出现，注意
if r.h!=originHigh and r!=root
    AVL-TREE--BALANCE-FIX(r.parent)
```

4.2.4 删除

删除给定关键字

```
AVL-TREE-DELETE(T,z)
y=z   //x指向将要移动到y原本位置的节点，或者原本y节点的父节点
p=y.parent   //p为被删除节点的父节点
if z.left==T.nil
    AVL-TREE-TRANSPLANT(T,z,z.right)
elseif z.right==T.nil
    AVL-TREE-TRANSPLANT(T,z,z.left)
else y=AVL-TREE-MINIMUM(z.right) //找到z的后继，由于z存在左右孩子，故后继为右子
    if y==z.right    //这个边界判断必须，因为p必须定位到被删除节点的父节点
        p=y
    else
        p=y.parent
    AVL-TREE-TRANSPLANT(y,y.right)
    y.right=z.right
    y.right.parent=y
    y.left=z.left
    y.left.parent=y
    AVL-TREE-TRANSPLANT (T,z,y)
```

```
19      y.height=z.height
20  if p!=nil
21      AVL-TREE--BALANCE-FIX(p)
```

5 Java源码

5.1 节点定义

```
1  public class AVLTreeNode {
2      /**
3       * 该节点的高度(从该节点到叶节点的最多边数)
4       */
5      int h;
6
7      /**
8       * 节点的值
9       */
10     int val;
11
12     /**
13      * 该节点的左孩子节点，右孩子节点，父节点
14      */
15     AVLTreeNode left, right, parent;
16
17     public AVLTreeNode(int val) {
18         h = 0;
19         this.val = val;
20         left = null;
21         right = null;
22         parent = null;
23     }
24 }
```

5.2 版本1

```
1   package org.liuyehcf.algorithm.datastructure.tree.avltree;
2
3   import java.util.*;
4
5   /**
6    * Created by Liuye on 2017/4/27.
7    */
8
9   public class AVLTree1 {
```

```
10      private enum RotateOrientation {
11          INVALID,
12          LEFT,
13          RIGHT
14      }
15
16      private AVLTreeNode root;
17
18      private AVLTreeNode nil;
19      private Map<AVLTreeNode, Integer> highMap;
20
21      public AVLTree1() {
22          nil = new AVLTreeNode(0);
23          nil.left = nil;
24          nil.right = nil;
25          nil.parent = nil;
26          root = nil;
27      }
28
29      public void insert(int val) {
30          AVLTreeNode x = root;
31          AVLTreeNode y = nil;
32          AVLTreeNode z = new AVLTreeNode(val);
33          while (x != nil) {
34              y = x;
35              if (val < x.val) {
36                  x = x.left;
37              } else {
38                  x = x.right;
39              }
40          }
41          z.parent = y;
42          if (y == nil) {
43              root = z;
44          } else if (z.val < y.val) {
45              y.left = z;
46          } else {
47              y.right = z;
48          }
49          z.left = nil;
50          z.right = nil;
51          fixUp(z);
52          if (!check())
53              throw new RuntimeException();
54      }
55
56      private void fixUp(AVLTreeNode y) {
```

```
57          if (y == nil) {
58              y = y.parent;
59          }
60          while (y != nil) {
61              updateHigh(y);
62              if (y.left.h == y.right.h + 2)
63                  y = holdRotate(y, RotateOrientation.RIGHT);
64              else if (y.right.h == y.left.h + 2)
65                  y = holdRotate(y, RotateOrientation.LEFT);
66              y = y.parent;
67          }
68      }
69
70      private AVLTreeNode holdRotate(AVLTreeNode x, RotateOrientation orienta
71          LinkedList<AVLTreeNode> stack1 = new LinkedList<AVLTreeNode>();
72          LinkedList<RotateOrientation> stack2 = new LinkedList<RotateOrienta
73          stack1.push(x);
74          stack2.push(orientation);
75          AVLTreeNode cur = nil;
76          AVLTreeNode rotateRoot = nil;
77          RotateOrientation curOrientation = RotateOrientation.INVALID;
78          while (!stack1.isEmpty()) {
79              cur = stack1.peek();
80              curOrientation = stack2.peek();
81              if (curOrientation == RotateOrientation.LEFT) {
82                  if (cur.right.right.h >= cur.right.left.h) {
83                      stack1.pop();
84                      stack2.pop();
85                      rotateRoot = leftRotate(cur);
86                  } else {
87                      stack1.push(cur.right);
88                      stack2.push(RotateOrientation.RIGHT);
89                  }
90              } else if (curOrientation == RotateOrientation.RIGHT) {
91                  if (cur.left.left.h >= cur.left.right.h) {
92                      stack1.pop();
93                      stack2.pop();
94                      rotateRoot = rightRotate(cur);
95                  } else {
96                      stack1.push(cur.left);
97                      stack2.push(RotateOrientation.LEFT);
98                  }
99              }
100         }
101         return rotateRoot;
102     }
103
```

```
104     private void updateHigh(AVLTreeNode z) {
105         z.h = Math.max(z.left.h, z.right.h) + 1;
106     }
107 
108     /**
109      * 左旋
110      *
111      * @param x
112      * @return 返回旋转后的根节点
113      */
114     private AVLTreeNode leftRotate(AVLTreeNode x) {
115         AVLTreeNode y = x.right;
116         x.right = y.left;
117         if (y.left != nil) {
118             y.left.parent = x;
119         }
120         y.parent = x.parent;
121         if (x.parent == nil) {
122             root = y;
123         } else if (x == x.parent.left) {
124             x.parent.left = y;
125         } else {
126             x.parent.right = y;
127         }
128         y.left = x;
129         x.parent = y;
130 
131         //更新高度
132         updateHigh(x);
133         updateHigh(y);
134         return y;
135     }
136 
137     /**
138      * 右旋
139      *
140      * @param y
141      * @return 返回旋转后的根节点
142      */
143     private AVLTreeNode rightRotate(AVLTreeNode y) {
144         AVLTreeNode x = y.left;
145         y.left = x.right;
146         if (x.right != nil) {
147             x.right.parent = y;
148         }
149         x.parent = y.parent;
150         if (y.parent == nil) {
```

```
151             root = x;
152         } else if (y == y.parent.left) {
153             y.parent.left = x;
154         } else {
155             y.parent.right = x;
156         }
157         x.right = y;
158         y.parent = x;
159 
160         //更新高度
161         updateHigh(y);
162         updateHigh(x);
163         return x;
164     }
165 
166     private boolean check() {
167         highMap = new HashMap<AVLTreeNode, Integer>();
168         return checkHigh(root) && checkBalance(root);
169     }
170 
171     private boolean checkHigh(AVLTreeNode root) {
172         if (root == nil) return true;
173         return checkHigh(root.left) && checkHigh(root.right) && root.h == h
174     }
175 
176     private int high(AVLTreeNode root) {
177         if (root == nil) {
178             return 0;
179         }
180         if (highMap.containsKey(root)) return highMap.get(root);
181         int leftHigh = high(root.left);
182         int rightHigh = high(root.right);
183         highMap.put(root, Math.max(leftHigh, rightHigh) + 1);
184         return highMap.get(root);
185     }
186 
187     private boolean checkBalance(AVLTreeNode root) {
188         if (root == nil) {
189             return true;
190         }
191         int leftHigh = root.left.h;
192         int rightHigh = root.right.h;
193         if (Math.abs(leftHigh - rightHigh) == 2) return false;
194         return checkBalance(root.left) && checkBalance(root.right);
195     }
196 
197     public boolean search(int val) {
```

```
198         return search(root, val) != nil;
199     }
200 
201     private AVLTreeNode search(AVLTreeNode x, int val) {
202         while (x != nil) {
203             if (x.val == val) return x;
204             else if (val < x.val) {
205                 x = x.left;
206             } else {
207                 x = x.right;
208             }
209         }
210         return nil;
211     }
212 
213     public void delete(int val) {
214         AVLTreeNode z = search(root, val);
215         if (z == nil) {
216             throw new RuntimeException();
217         }
218         AVLTreeNode y = z;
219         AVLTreeNode x = nil;
220         if (z.left == nil) {
221             x = y.right;
222             transplant(z, z.right);
223         } else if (z.right == nil) {
224             x = y.left;
225             transplant(z, z.left);
226         } else {
227             y = min(z.right);
228             x = y.right;
229             if (y.parent == z) {
230                 x.parent = y;
231             } else {
232                 transplant(y, y.right);
233                 y.right = z.right;
234                 y.right.parent = y;
235             }
236             transplant(z, y);
237 
238             y.left = z.left;
239             y.left.parent = y;
240 
241             //todo 这里不需要更新p的高度,因为p的子树的高度此时并不知道是否正确,因
242         }
243         fixUp(x);
244         if (!check())
```

```
245             throw new RuntimeException();
246         }
247 
248         private void transplant(AVLTreeNode u, AVLTreeNode v) {
249             v.parent = u.parent;
250             if (u.parent == nil) {
251                 root = v;
252             } else if (u == u.parent.left) {
253                 u.parent.left = v;
254             } else {
255                 u.parent.right = v;
256             }
257         }
258 
259         private AVLTreeNode min(AVLTreeNode x) {
260             while (x.left != nil) {
261                 x = x.left;
262             }
263             return x;
264         }
265 
266         public void inOrderTraverse() {
267             inOrderTraverse(root);
268             System.out.println();
269         }
270 
271         public void levelOrderTraversal() {
272             Queue<AVLTreeNode> queue = new LinkedList<AVLTreeNode>();
273             queue.offer(root);
274             while (!queue.isEmpty()) {
275                 int len = queue.size();
276                 for (int i = 0; i < len; i++) {
277                     AVLTreeNode peek = queue.poll();
278                     System.out.print("[" + peek.val + "," + peek.h + "], ");
279                     if (peek.left != nil) queue.offer(peek.left);
280                     if (peek.right != nil) queue.offer(peek.right);
281                 }
282             }
283             System.out.println();
284         }
285 
286         private void inOrderTraverse(AVLTreeNode root) {
287             if (root != nil) {
288                 inOrderTraverse(root.left);
289                 System.out.print(root.val + ", ");
290                 inOrderTraverse(root.right);
291             }
```

```
292     }
293
294     public static void main(String[] args) {
295         long start = System.currentTimeMillis();
296
297         Random random = new Random();
298
299         int TIMES = 10;
300
301         while (--TIMES > 0) {
302             System.out.println("剩余测试次数: " + TIMES);
303             AVLTree1 avlTree2 = new AVLTree1();
304
305             int N = 1000;
306             int M = N / 2;
307
308             Set<Integer> set = new HashSet<Integer>();
309             for (int i = 0; i < N; i++) {
310                 set.add(random.nextInt());
311             }
312
313             List<Integer> list = new ArrayList<Integer>(set);
314             Collections.shuffle(list, random);
315             //插入N个数据
316             for (int i : list) {
317                 avlTree2.insert(i);
318             }
319
320             //删除M个数据
321             Collections.shuffle(list, random);
322
323             for (int i = 0; i < M; i++) {
324                 set.remove(list.get(i));
325                 avlTree2.delete(list.get(i));
326             }
327
328             //再插入M个数据
329             for (int i = 0; i < M; i++) {
330                 int k = random.nextInt();
331                 set.add(k);
332                 avlTree2.insert(k);
333             }
334             list.clear();
335             list.addAll(set);
336             Collections.shuffle(list, random);
337
338             //再删除所有元素
```

```
            for (int i : list) {
                avlTree2.delete(i);
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("Run time: " + (end - start) / 1000 + "s");
    }
}
```

5.3 版本2

```
package org.liuyehcf.algorithm.datastructure.tree.avltree;

import java.util.*;

/**
 * Created by liuye on 2017/4/24 0024.
 */

public class AVLTree2 {
    private AVLTreeNode root;

    private AVLTreeNode nil;
    private Map<AVLTreeNode, Integer> highMap;

    public AVLTree2() {
        nil = new AVLTreeNode(0);
        nil.left = nil;
        nil.right = nil;
        nil.parent = nil;
        root = nil;
    }

    public void insert(int val) {
        AVLTreeNode x = root;
        AVLTreeNode y = nil;
        AVLTreeNode z = new AVLTreeNode(val);
        while (x != nil) {
            y = x;
            if (val < x.val) {
                x = x.left;
            } else {
                x = x.right;
            }
        }
        z.parent = y;
```

```
36          if (y == nil) {
37              root = z;
38          } else if (z.val < y.val) {
39              y.left = z;
40          } else {
41              y.right = z;
42          }
43          z.left = nil;
44          z.right = nil;
45          balanceFix(z);
46          if (!check())
47              throw new RuntimeException();
48      }
49
50      private void balanceFix(AVLTreeNode z) {
51          //当前节点的初始高度
52          int originHigh = z.h;
53
54          updateHigh(z);
55
56          //经过调整后的子树根节点(调整之前子树根节点为z)
57          AVLTreeNode r = z;
58
59          if (z.left.h == z.right.h + 2) {
60              //todo 这里的等号非常重要(插入过程时不可能取等号，删除过程可能取等号)
61              if (z.left.left.h >= z.left.right.h) {
62                  r = rightRotate(z);
63              } else if (z.left.left.h < z.left.right.h) {
64                  leftRotate(z.left);
65                  r = rightRotate(z);
66              }
67
68          } else if (z.right.h == z.left.h + 2) {
69              //todo 这里的等号非常重要(插入过程时不可能取等号，删除过程可能取等号)
70              if (z.right.right.h >= z.right.left.h) {
71                  r = leftRotate(z);
72              } else if (z.right.right.h < z.right.left.h) {
73                  rightRotate(z.right);
74                  r = leftRotate(z);
75              }
76          }
77
78          //递归其父节点
79          if (r.h != originHigh && r != root)
80              balanceFix(r.parent);
81      }
82
```

```
83      private void updateHigh(AVLTreeNode z) {
84          z.h = Math.max(z.left.h, z.right.h) + 1;
85      }
86
87      /**
88       * 左旋
89       *
90       * @param x
91       * @return 返回旋转后的根节点
92       */
93      private AVLTreeNode leftRotate(AVLTreeNode x) {
94          AVLTreeNode y = x.right;
95          x.right = y.left;
96          if (y.left != nil) {
97              y.left.parent = x;
98          }
99          y.parent = x.parent;
100         if (x.parent == nil) {
101             root = y;
102         } else if (x == x.parent.left) {
103             x.parent.left = y;
104         } else {
105             x.parent.right = y;
106         }
107         y.left = x;
108         x.parent = y;
109
110         //更新高度
111         updateHigh(x);
112         updateHigh(y);
113         return y;
114     }
115
116     /**
117      * 右旋
118      *
119      * @param y
120      * @return 返回旋转后的根节点
121      */
122     private AVLTreeNode rightRotate(AVLTreeNode y) {
123         AVLTreeNode x = y.left;
124         y.left = x.right;
125         if (x.right != nil) {
126             x.right.parent = y;
127         }
128         x.parent = y.parent;
129         if (y.parent == nil) {
```

```
130             root = x;
131         } else if (y == y.parent.left) {
132             y.parent.left = x;
133         } else {
134             y.parent.right = x;
135         }
136         x.right = y;
137         y.parent = x;
138 
139         //更新高度
140         updateHigh(y);
141         updateHigh(x);
142         return x;
143     }
144 
145     private boolean check() {
146         highMap = new HashMap<AVLTreeNode, Integer>();
147         return checkHigh(root) && checkBalance(root);
148     }
149 
150     private boolean checkHigh(AVLTreeNode root) {
151         if (root == nil) return true;
152         return checkHigh(root.left) && checkHigh(root.right) && root.h == h
153     }
154 
155     private int high(AVLTreeNode root) {
156         if (root == nil) {
157             return 0;
158         }
159         if (highMap.containsKey(root)) return highMap.get(root);
160         int leftHigh = high(root.left);
161         int rightHigh = high(root.right);
162         highMap.put(root, Math.max(leftHigh, rightHigh) + 1);
163         return highMap.get(root);
164     }
165 
166     private boolean checkBalance(AVLTreeNode root) {
167         if (root == nil) {
168             return true;
169         }
170         int leftHigh = root.left.h;
171         int rightHigh = root.right.h;
172         if (Math.abs(leftHigh - rightHigh) == 2) return false;
173         return checkBalance(root.left) && checkBalance(root.right);
174     }
175 
176     public boolean search(int val) {
```

```
177         return search(root, val) != nil;
178     }
179 
180     private AVLTreeNode search(AVLTreeNode x, int val) {
181         while (x != nil) {
182             if (x.val == val) return x;
183             else if (val < x.val) {
184                 x = x.left;
185             } else {
186                 x = x.right;
187             }
188         }
189         return nil;
190     }
191 
192     public void delete(int val) {
193         AVLTreeNode z = search(root, val);
194         if (z == nil) {
195             throw new RuntimeException();
196         }
197         //y代表真正被删除的节点
198         AVLTreeNode y = z;
199         //x为被删除节点的父节点，如果平衡被破坏，从该节点开始
200         AVLTreeNode p = y.parent;
201         if (z.left == nil) {
202             transplant(z, z.right);
203         } else if (z.right == nil) {
204             transplant(z, z.left);
205         } else {
206             y = min(z.right);
207             //todo 这里的分类讨论非常重要,否则将会定位到错误的父节点
208             if (y == z.right) {
209                 p = y;
210             } else {
211                 p = y.parent;
212             }
213             transplant(y, y.right);
214 
215             //todo 下面六句可以用z.val=y.val来代替,效果一样
216             y.right = z.right;
217             y.right.parent = y;
218 
219             y.left = z.left;
220             y.left.parent = y;
221 
222             transplant(z, y);
223             y.h = z.h;//todo 这里高度必须维护
```

```
224             //todo 这里不需要更新p的高度,因为p的子树的高度此时并不知道是否正确,因
225         }
226         if (p != nil)
227             balanceFix(p);
228         if (!check())
229             throw new RuntimeException();
230     }
231 
232     private void transplant(AVLTreeNode u, AVLTreeNode v) {
233         v.parent = u.parent;
234         if (u.parent == nil) {
235             root = v;
236         } else if (u == u.parent.left) {
237             u.parent.left = v;
238         } else {
239             u.parent.right = v;
240         }
241     }
242 
243     private AVLTreeNode min(AVLTreeNode x) {
244         while (x.left != nil) {
245             x = x.left;
246         }
247         return x;
248     }
249 
250     public void inOrderTraverse() {
251         inOrderTraverse(root);
252         System.out.println();
253     }
254 
255     public void levelOrderTraversal() {
256         Queue<AVLTreeNode> queue = new LinkedList<AVLTreeNode>();
257         queue.offer(root);
258         while (!queue.isEmpty()) {
259             int len = queue.size();
260             for (int i = 0; i < len; i++) {
261                 AVLTreeNode peek = queue.poll();
262                 System.out.print("[" + peek.val + "," + peek.h + "], ");
263                 if (peek.left != nil) queue.offer(peek.left);
264                 if (peek.right != nil) queue.offer(peek.right);
265             }
266         }
267         System.out.println();
268     }
269 
270     private void inOrderTraverse(AVLTreeNode root) {
```

```
271         if (root != nil) {
272             inOrderTraverse(root.left);
273             System.out.print(root.val + ", ");
274             inOrderTraverse(root.right);
275         }
276     }
277 
278     public static void main(String[] args) {
279         long start = System.currentTimeMillis();
280 
281         Random random = new Random();
282 
283         int TIMES = 10;
284 
285         while (--TIMES > 0) {
286             System.out.println("剩余测试次数: " + TIMES);
287             AVLTree2 avlTree = new AVLTree2();
288 
289             int N = 10000;
290             int M = N / 2;
291 
292             List<Integer> list = new ArrayList<Integer>();
293             for (int i = 0; i < N; i++) {
294                 list.add(random.nextInt());
295             }
296 
297             Collections.shuffle(list, random);
298             //插入N个数据
299             for (int i : list) {
300                 avlTree.insert(i);
301             }
302 
303             //删除M个数据
304             Collections.shuffle(list, random);
305 
306             for (int i = 0; i < M; i++) {
307                 int k = list.get(list.size() - 1);
308                 list.remove(list.size() - 1);
309                 avlTree.delete(k);
310             }
311 
312             //再插入M个数据
313             for (int i = 0; i < M; i++) {
314                 int k = random.nextInt();
315                 list.add(k);
316                 avlTree.insert(k);
317             }
```

```
318             Collections.shuffle(list, random);
319 
320             //再删除所有元素
321             for (int i : list) {
322                 avlTree.delete(i);
323             }
324         }
325         long end = System.currentTimeMillis();
326         System.out.println("Run time: " + (end - start) / 1000 + "s");
327     }
328 }
```