一、伸展树

本文介绍了二叉查找树的一种改进数据结构–伸展树（Splay Tree）。它的主要特点是不会保证树一直是平衡的，但各种操作的平摊时间复杂度是O(log n)，因而，从平摊复杂度上看，二叉查找树也是一种平衡二叉树。另外，相比于其他树状数据结构（如红黑树，AVL树等），伸展树的空间要求与编程复杂度要小得多。

伸展树的出发点是这样的：考虑到局部性原理（刚被访问的内容下次可能仍会被访问，查找次数多的内容可能下一次会被访问），为了使整个查找时间更小，被查频率高的那些节点应当经常处于靠近树根的位置。这样，很容易得想到以下这个方案：每次查找节点之后对树进行重构，把被查找的节点搬移到树根，这种自调整形式的二叉查找树就是伸展树。每次对伸展树进行操作后，它均会通过旋转的方法把被访问节点旋转到树根的位置。

为了将当前被访问节点旋转到树根，我们通常将节点自底向上旋转，直至该节点成为树根为止。“旋转”的巧妙之处就是在不打乱数列中数据大小关系（指中序遍历结果是全序的）情况下，所有基本操作的平摊复杂度仍为O（log n）。

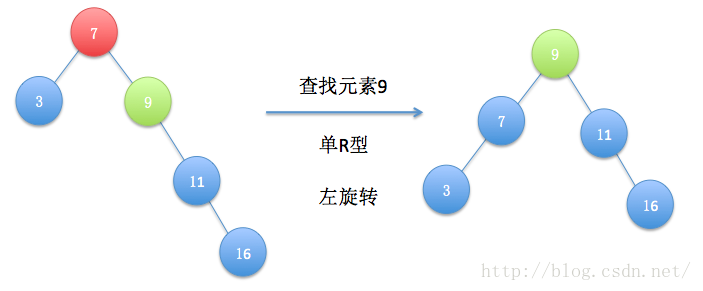
在AVL树中我我们知道有4种旋转方式（实际是两种）RR型,LL型,RL型,LR型，如果不清楚AVL树旋转方式的可以参考[平衡二叉树（AVL）图解与实现](http://blog.csdn.net/u014634338/article/details/42465089" \t "_blank)，个人觉还是说得很明白了，这里我们还是已图解的方式来讲解伸展树的操作，对伸展树的旋转操作中我沿用了AVL树中的部分名称，当然这个不是很严谨，主要是明白其中的原理，至于怎么称呼这个都是个人习惯而已。

首先给出伸展树的结构定义：

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. **typedef** **struct** SplayNode \*Tree;
2. **typedef** **int** ElementType;
3. **struct** SplayNode
4. {
5. Tree parent; //该结点的父节点，方便操作
6. ElementType val; //结点值
7. Tree lchild;
8. Tree rchild;
9. SplayNode(**int** val=0) //默认构造函数
10. {
11. parent=NULL;
12. lchild=rchild=NULL;
13. **this**->val=val;
14. }
15. };

二、伸展树的旋转操作

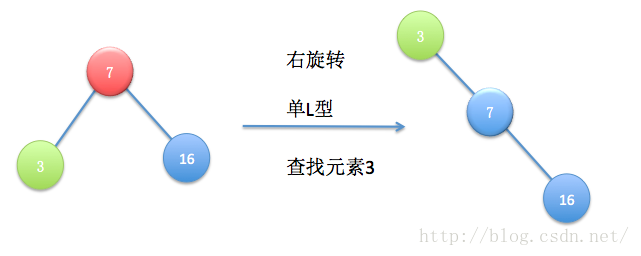


什么叫单R型呢，在上图中，我们查找的元素是9,其父节点是7，并且7是根结点，查找结点是其父节点的右孩子，而且把9变成根结点只需一次左旋转即可（即将9提升一层），这样的情况我们叫单R型，经过一次左旋转后结点9替代了原来的根结点7，变成新的根结点（注意这里因为图简单，9最终变成了根结点，在树复杂的情况，一般不会一次就变成了根结点，但肯定会变成原子树的根，这也就是程序中说的当前子树中的新根）。为了后面更加轻松，这里把单左旋代码贴出，可以对比图示和代码分析分析，便于理解

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //单左旋操作
2. //参数:根，旋转结点(旋转中心)
3. //返回:当前子树中的新根
4. Tree left\_single\_rotate(Tree &root,Tree node)
5. {
6. **if** (node==NULL)
7. **return** NULL;
8. Tree parent=node->parent; //其父结点
9. Tree grandparent=parent->parent; //其祖父结点
10. parent->rchild=node->lchild; //设置其父节点的右孩子
11. **if** (node->lchild) //如果有左孩子则更新node结点左孩子的父节点信息
12. node->lchild->parent=parent;
13. node->lchild=parent; //更新node结点的左孩子信息
14. parent->parent=node; //更新原父节点的信息
15. node->parent=grandparent;
17. **if** (grandparent) //更新祖父孩子结点的信息
18. {
20. **if** (grandparent->lchild==parent)
21. grandparent->lchild=node;
22. **else**
23. grandparent->rchild=node;
24. }
25. **else** //不存在祖父节点，则原父节点为根，那么旋转后node为根
26. root=node;
27. **return** node;
28. }

(2)单L型

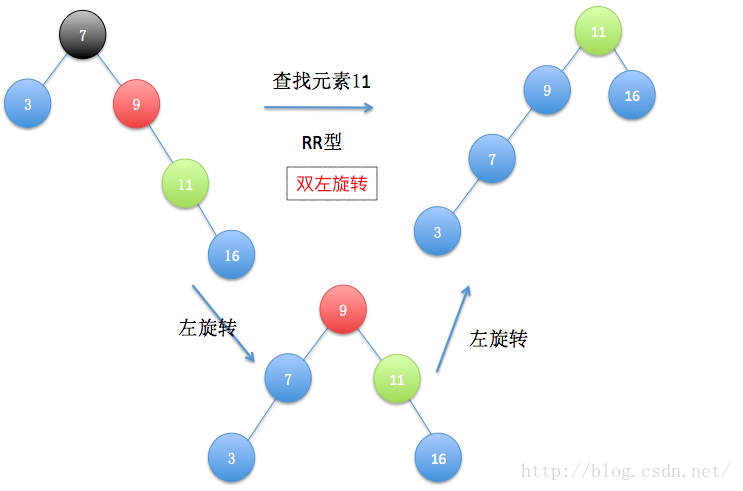


单L型和单R型是对称的，也就是说查找结点3是其父节点的左子树，并且其父节点是根结点，这样一次右旋转后3就是根结点了。

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //单右旋操作
2. //参数:根，旋转结点(旋转中心)
3. //返回:当前子树中的新根
4. Tree right\_single\_rotate(Tree &root,Tree node)
5. {
6. **if** (node==NULL)
7. **return** NULL;
8. Tree parent,grandparent;
9. parent=node->parent;
10. grandparent=parent->parent;
11. parent->lchild=node->rchild;
12. **if** (node->rchild)
13. node->rchild->parent=parent;
14. node->rchild=parent;
15. parent->parent=node;
16. node->parent=grandparent;
17. **if** (grandparent)
18. {
19. **if** (grandparent->lchild==parent)
20. grandparent->lchild=node;
21. **else**
22. grandparent->rchild=node;
23. }
24. **else**
25. root=node;
26. **return** node;
28. }

(3)RR型

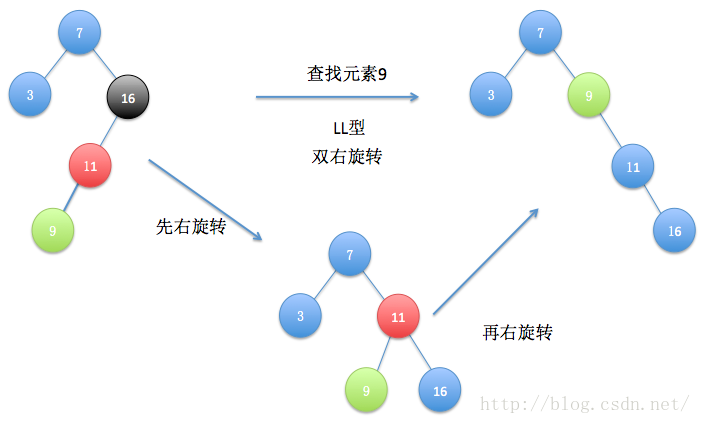


所谓RR型，简单点说就是两次R型，两次左旋转，这种情况是查找结点有父节点，同时也有祖父结点，并且三则在同右侧，这种就是RR型，针对这种情况，先把查找结点的父节点旋转一次，即提升一层，然后再以查找结点再次旋转，这样查找结点就到了根结点了，都是左旋转，只是旋转对象不一样罢了。

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //两次单左旋操作
2. //参数：根，最后将变成子树根结点的结点
3. **void** left\_double\_rotate(Tree &root,Tree node)
4. {
5. left\_single\_rotate(root,node->parent);
6. left\_single\_rotate(root,node);
7. }

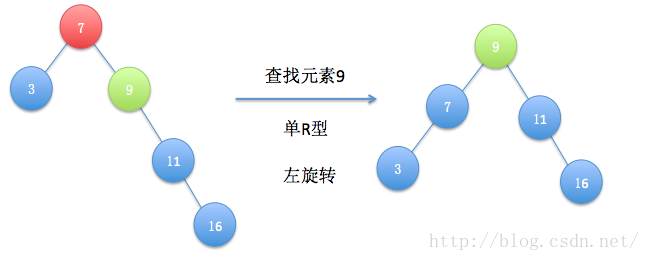
(4)LL型



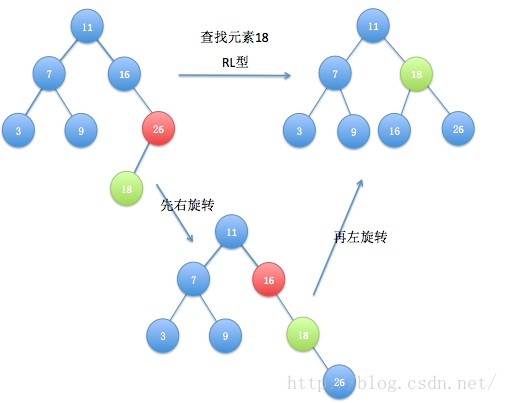
**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //两次单右旋操作
2. //参数：根，最后将变成子树根结点的结点
3. **void** right\_double\_rotate(Tree &root,Tree node)
4. {
5. right\_single\_rotate(root,node->parent); //先提升其父节点
6. right\_single\_rotate(root,node);         //最后提升自己
7. }

LL型和RR型是对称的，经过一次双右旋结果如上图，但是这样就结束了吗？回想一下，伸展树的旋转操作目的是干什么，不是为了把查找结点推送至树根么，是的，但是现在这种情况结点9还不是树根，但是这种情况不是我们前面讲过的单R型吗？所以再来次左旋就可以了，也就是下面这个样子：



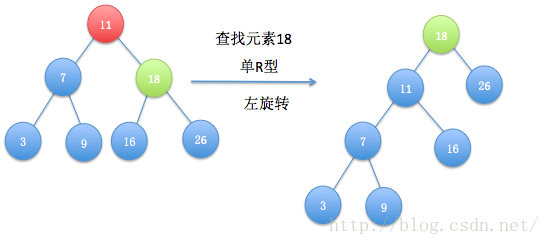
(5)RL型



**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

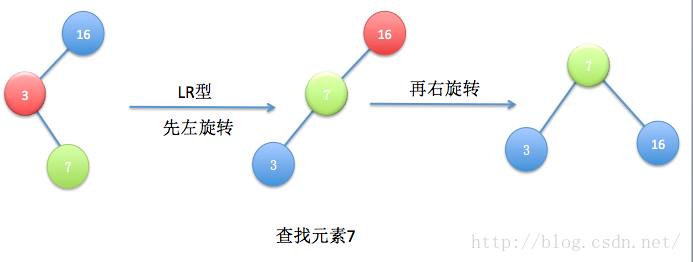
1. //双旋操作（RL型），于AVL树类似
2. //参数：根，最后将变成子树根结点的结点
3. **void** RL\_rotate(Tree&root,Tree node)
4. {
5. right\_single\_rotate(root,node); //先右后左
6. left\_single\_rotate(root,node);
7. }

这个和AVL树中的RL是一样的，旋转完成后，还需要一步左旋：



OK，到位了。

(6)LR型



**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //双旋操作（LR型），于AVL树类似
2. //参数：根，最后将变成子树根结点的结点
3. **void** LR\_rotate(Tree &root,Tree node)
4. {
5. left\_single\_rotate(root,node); //先左
6. right\_single\_rotate(root,node);//后右
7. }

OK，到这里伸展树的几种情况就介绍完了，怎么这么多旋转方式，其实大可不必这样，这个只是我学习的时候自己总结的，和网上的也打不相同，主要是自己理解了它的旋转方式后，就好了，至于命名这些都影响不大，我这里把它们分为这几种的方式，主要是为了封装成函数，方便我的调用，这样逻辑更清楚一下，自己懂了以后就可以根据自己理解来组织代码了。

三、伸展树的操作

伸展树的操作和AVL树一样无非就是查，插，删，下面我们分别来介绍它们。

(1)先看看查找函数search：

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //查找函数，带调整功能
2. //参数:根结点，需要查找的val
3. //返回:true or false
4. **bool** search(Tree &root,ElementType val)
5. {
7. Tree parent=NULL;
8. Tree \*temp=NULL;
9. temp=search\_val(root,val, parent);
10. **if** (\*temp && \*temp!=root)
11. {
12. SplayTree(root,\*temp);
13. **return** **true**;
14. }
15. **return** **false**;
16. }

查找函数中里面有另一个具体的查找函数，我们先不管它，先梳理逻辑，首先我们通过内部的查找函数，查找值为val的结点，找到后返回结点给temp，如果查找成功，并且当前结点不是根结点，那么我们将进行树的调整，将结点temp推到树根，否则直接退出，这就是search的功能，简单明了。

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //具体的查找函数
2. //参数:根，需要查找的val,父节点指针
3. //成功:返回其结点
4. //失败：返回其引用,方便后面的插入操作
5. Tree \*search\_val(Tree &root,ElementType val,Tree &parent)
6. {
7. **if** (root==NULL)
8. **return** &root;
9. **if** (root->val>val)
10. **return** search\_val(root->lchild,val,parent=root);
11. **else** **if**(root->val<val)
12. **return** search\_val(root->rchild,val,parent=root);
13. **return** &root;
14. }

这里我们有必要介绍一下内部的查找函数，因为这是一个通用的接口，后面都会用到它，这个查找函数，如果查找成功则返回结点的引用，否则返回它该插入地方的引用，也就是其最后的parent的某个孩子，parent是查找成功或失败结点的父节点，也是引用类型。OK，这就是我们的查找函数，这里没有强化它的查找功能，只是方便我们后面的插入和删除工作。

(2)插入

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //插入函数
2. //参数：根，需要插入的val
3. //返回:true or false
4. **bool** insert(Tree &root,ElementType val)
5. {
6. Tree \*temp=NULL;
7. Tree parent=NULL;
8. //先查找，如果成功则无需插入，否则返回该结点的引用。
9. temp=search\_val(root,val,parent);
11. **if** (\*temp==NULL) //需要插入数据
12. {
13. Tree node=**new** SplayNode(val);
14. \*temp=node; //因为是引用型，所以这里直接赋值，简化了很多了。
15. node->parent=parent; //设置父节点。
16. **return** **true**;
17. }
18. **return** **false**;
19. }

可以看到这个插入函数也是很短的，注意观察，里面有我们熟悉的东西，没错就是前面所讲的内部查找函数，这里对插入结点，我们先进行查找，如果查找成功就不进行插入，否则返回该插入地址的引用，这样我们直接让\*temp=node，便完成了插入工作，简化了很多工作，然后设置父节点信息，插入成功。

(3)伸展

当我们查找一个val后，我们需要对树进行伸展，下面就是我们的伸展函数

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //Splay调整操作
2. **void** SplayTree(Tree &root,Tree node)
3. {
4. **while** (root->lchild!=node && root->rchild!=node && root!=node) //当前结点不是根，或者不是其根的左右孩子，则根据情况进行旋转操作
5. up(root, node);
6. **if** (root->lchild==node) //当前结点为根的左孩子，只需进行一次单右旋
7. root=right\_single\_rotate(root, node);
8. **else** **if**(root->rchild==node) //当前结点为根的右孩子，只需进行一次单左旋
9. root=left\_single\_rotate(root, node);
10. }

可以看到，里面有个up函数，在这个函数外，还有单独的if判断结构，这两个if就是判断特殊情况的，也就是我们只需进行一个单旋便可以晋级为根结点的情况，这个很简单，结合一下图就可以看出来了。OK，看看我们的up函数

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //根据情况，选择不同的旋转方式
2. **void** up(Tree &root,Tree node)
3. {
4. Tree parent,grandparent;
5. **int** i,j;
6. parent=node->parent;
7. grandparent=parent->parent;
8. i=grandparent->lchild==parent ? -1:1;
9. j=parent->lchild==node ?-1:1;
10. **if** (i==-1 && j==-1) //AVL树中的LL型
11. right\_double\_rotate(root, node);
12. **else** **if**(i==-1 && j==1) //AVL树中的LR型
13. LR\_rotate(root, node);
14. **else** **if**(i==1 && j==-1) //AVL树中的RL型
15. RL\_rotate(root, node);
16. **else**                    //AVL树中的RR型
17. left\_double\_rotate(root, node);
18. }

up顾名思义就是往上，也就是把查找结点往上推送，在这个函数里面我们判断了旋转类型，是LL型，还是RR型，还是LR型，亦或是RL型，然后再调用我们前面展示过的旋转函数。只需旋转函数最好结合图然后再看代码，这样很容易理解，不要只看代码。

到这里我们的查找和插入，以及伸展过程我们都展示了，这里很重要一个函数就是查找函数，还有就是几种旋转方式。

(4)删除

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //删除操作
2. **void** remove(Tree &root,ElementType val)
3. {
4. Tree parent=NULL;
5. Tree \*temp;
6. Tree \*replace;
7. Tree replace2;
8. temp=search\_val(root,val, parent); //先进行查找操作
9. **if**(\*temp) //如果查找到了
10. {
11. **if** (\*temp!=root) //判断是否是根结点，不是根结点，则需要调整至根结点
12. SplayTree(root, \*temp);
14. //调至根结点或者本来就是根结点后进行删除，先查看是否有替代元素
15. **if** (root->rchild)
16. {
17. //有替代元素
18. replace=Find\_Min(root->rchild); //找到替换元素
19. root->val=(\*replace)->val;  //替换
20. **if** ((\*replace)->lchild==NULL) //左子树为空
21. {
22. replace2=\*replace;
23. \*replace=(\*replace)->rchild; //重接其右孩子
24. **delete** replace2;
26. }
27. **else** **if**((\*replace)->rchild==NULL) //右子树为空
28. {
29. replace2=\*replace;
30. \*replace=(\*replace)->lchild; //重接其左孩子
31. **delete** replace2;
32. }
33. }
34. **else**
35. {
36. //无替代元素，则根直接移向左子树，不管左子树是否为空都可以处理
37. replace2=root;
38. root=root->lchild;
39. **delete** replace2;
40. }
41. }
42. }

在删除函数中，我们首先进行了查找，查找失败就退出，查找成功后，我们便把它推送到根结点，然后再用我们BST删除方式，找替代元素，这样化繁为简，只是这里统一采用引用方式，要简单很多。

下面是这是我们的熟悉的找替代元素的函数:

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. //操作当前子树的最小结点
2. //返回:其最小结点的引用
3. Tree \*Find\_Min(Tree &root)
4. {
5. **if** (root->lchild)
6. **return** Find\_Min(root->lchild);
7. **return** &root;
8. }

OK，到这里我们的伸展树就介绍完了，在这里可以看到我们伸展树里面的函数和AVL树里面的函数差别很大，在AVL树里面，我们即采用了引用(部分),同时又可以通过返回值来设置，再加上手生，写得有点杂乱，这里的伸展树，我就统一采用引用方式，能不返回值就返回值，这样可以简化很多操作，加之伸展树本来就比AVL树简单，不同判断平衡因子，因此写起来就更加简单了。

下面就是我们总的代码：

**[cpp]** [view plain](https://blog.csdn.net/u014634338/article/details/49586689) [copy](https://blog.csdn.net/u014634338/article/details/49586689)

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <iostream>
4. **using** **namespace** std;
5. **typedef** **struct** SplayNode \*Tree;
6. **typedef** **int** ElementType;
7. **struct** SplayNode
8. {
9. Tree parent; //该结点的父节点，方便操作
10. ElementType val; //结点值
11. Tree lchild;
12. Tree rchild;
13. SplayNode(**int** val=0) //默认构造函数
14. {
15. parent=NULL;
16. lchild=rchild=NULL;
17. **this**->val=val;
18. }
19. };
21. **bool** search(Tree &,ElementType);
22. Tree \*search\_val(Tree&,ElementType,Tree&);
23. **bool** insert(Tree &,ElementType);
24. Tree left\_single\_rotate(Tree&,Tree);
25. Tree right\_single\_rotate(Tree &,Tree );
26. **void** LR\_rotate(Tree&,Tree );
27. **void** RL\_rotate(Tree&,Tree );
28. **void** right\_double\_rotate(Tree&,Tree );
29. **void** left\_double\_rotate(Tree&,Tree );
30. **void** SplayTree(Tree &,Tree);
31. **void** up(Tree &,Tree );
32. Tree \*Find\_Min(Tree &);
33. **void** remove(Tree &,ElementType);
35. //查找函数，带调整功能
36. //参数:根结点，需要查找的val
37. //返回:true or false
38. **bool** search(Tree &root,ElementType val)
39. {
41. Tree parent=NULL;
42. Tree \*temp=NULL;
43. temp=search\_val(root,val, parent);
44. **if** (\*temp && \*temp!=root)
45. {
46. SplayTree(root,\*temp);
47. **return** **true**;
48. }
49. **return** **false**;
50. }
52. //具体的查找函数
53. //参数:根，需要查找的val,父节点指针
54. //成功:返回其结点
55. //失败：返回其引用,方便后面的插入操作
56. Tree \*search\_val(Tree &root,ElementType val,Tree &parent)
57. {
58. **if** (root==NULL)
59. **return** &root;
60. **if** (root->val>val)
61. **return** search\_val(root->lchild,val,parent=root);
62. **else** **if**(root->val<val)
63. **return** search\_val(root->rchild,val,parent=root);
64. **return** &root;
65. }
67. //插入函数
68. //参数：根，需要插入的val
69. //返回:true or false
70. **bool** insert(Tree &root,ElementType val)
71. {
72. Tree \*temp=NULL;
73. Tree parent=NULL;
74. //先查找，如果成功则无需插入，否则返回该结点的引用。
75. temp=search\_val(root,val,parent);
77. **if** (\*temp==NULL) //需要插入数据
78. {
79. Tree node=**new** SplayNode(val);
80. \*temp=node; //因为是引用型，所以这里直接赋值，简化了很多了。
81. node->parent=parent; //设置父节点。
82. **return** **true**;
83. }
84. **return** **false**;
85. }
86. //单左旋操作
87. //参数:根，旋转结点(旋转中心)
88. //返回:当前子树中的新根
89. Tree left\_single\_rotate(Tree &root,Tree node)
90. {
91. **if** (node==NULL)
92. **return** NULL;
93. Tree parent=node->parent; //其父结点
94. Tree grandparent=parent->parent; //其祖父结点
95. parent->rchild=node->lchild; //设置其父节点的右孩子
96. **if** (node->lchild) //如果有左孩子则更新node结点左孩子的父节点信息
97. node->lchild->parent=parent;
98. node->lchild=parent; //更新node结点的左孩子信息
99. parent->parent=node; //更新原父节点的信息
100. node->parent=grandparent;
102. **if** (grandparent) //更新祖父孩子结点的信息
103. {
105. **if** (grandparent->lchild==parent)
106. grandparent->lchild=node;
107. **else**
108. grandparent->rchild=node;
109. }
110. **else** //不存在祖父节点，则原父节点为根，那么旋转后node为根
111. root=node;
112. **return** node;
113. }
114. //单右旋操作
115. //参数:根，旋转结点(旋转中心)
116. //返回:当前子树中的新根
117. Tree right\_single\_rotate(Tree &root,Tree node)
118. {
119. **if** (node==NULL)
120. **return** NULL;
121. Tree parent,grandparent;
122. parent=node->parent;
123. grandparent=parent->parent;
124. parent->lchild=node->rchild;
125. **if** (node->rchild)
126. node->rchild->parent=parent;
127. node->rchild=parent;
128. parent->parent=node;
129. node->parent=grandparent;
130. **if** (grandparent)
131. {
132. **if** (grandparent->lchild==parent)
133. grandparent->lchild=node;
134. **else**
135. grandparent->rchild=node;
136. }
137. **else**
138. root=node;
139. **return** node;
141. }
142. //双旋操作（LR型），于AVL树类似
143. //参数：根，最后将变成子树根结点的结点
144. **void** LR\_rotate(Tree &root,Tree node)
145. {
146. left\_single\_rotate(root,node); //先左
147. right\_single\_rotate(root,node);//后右
148. }
149. //双旋操作（RL型），于AVL树类似
150. //参数：根，最后将变成子树根结点的结点
151. **void** RL\_rotate(Tree&root,Tree node)
152. {
153. right\_single\_rotate(root,node); //先右后左
154. left\_single\_rotate(root,node);
155. }
157. //两次单右旋操作
158. //参数：根，最后将变成子树根结点的结点
159. **void** right\_double\_rotate(Tree &root,Tree node)
160. {
161. right\_single\_rotate(root,node->parent); //先提升其父节点
162. right\_single\_rotate(root,node);         //最后提升自己
163. }
164. //两次单左旋操作
165. //参数：根，最后将变成子树根结点的结点
166. **void** left\_double\_rotate(Tree &root,Tree node)
167. {
168. left\_single\_rotate(root,node->parent);
169. left\_single\_rotate(root,node);
170. }
171. //Splay调整操作
172. **void** SplayTree(Tree &root,Tree node)
173. {
174. **while** (root->lchild!=node && root->rchild!=node && root!=node) //当前结点不是根，或者不是其根的左右孩子，则根据情况进行旋转操作
175. up(root, node);
176. **if** (root->lchild==node) //当前结点为根的左孩子，只需进行一次单右旋
177. root=right\_single\_rotate(root, node);
178. **else** **if**(root->rchild==node) //当前结点为根的右孩子，只需进行一次单左旋
179. root=left\_single\_rotate(root, node);
180. }
182. //根据情况，选择不同的旋转方式
183. **void** up(Tree &root,Tree node)
184. {
185. Tree parent,grandparent;
186. **int** i,j;
187. parent=node->parent;
188. grandparent=parent->parent;
189. i=grandparent->lchild==parent ? -1:1;
190. j=parent->lchild==node ?-1:1;
191. **if** (i==-1 && j==-1) //AVL树中的LL型
192. right\_double\_rotate(root, node);
193. **else** **if**(i==-1 && j==1) //AVL树中的LR型
194. LR\_rotate(root, node);
195. **else** **if**(i==1 && j==-1) //AVL树中的RL型
196. RL\_rotate(root, node);
197. **else**                    //AVL树中的RR型
198. left\_double\_rotate(root, node);
199. }
201. //操作当前子树的最小结点
202. //返回:其最小结点的引用
203. Tree \*Find\_Min(Tree &root)
204. {
205. **if** (root->lchild)
206. **return** Find\_Min(root->lchild);
207. **return** &root;
208. }
210. //删除操作
211. **void** remove(Tree &root,ElementType val)
212. {
213. Tree parent=NULL;
214. Tree \*temp;
215. Tree \*replace;
216. Tree replace2;
217. temp=search\_val(root,val, parent); //先进行查找操作
218. **if**(\*temp) //如果查找到了
219. {
220. **if** (\*temp!=root) //判断是否是根结点，不是根结点，则需要调整至根结点
221. SplayTree(root, \*temp);
223. //调制根结点或者本来就是根结点后进行删除，先查看是否有替代元素
224. **if** (root->rchild)
225. {
226. //有替代元素
227. replace=Find\_Min(root->rchild); //找到替换元素
228. root->val=(\*replace)->val;  //替换
229. **if** ((\*replace)->lchild==NULL) //左子树为空
230. {
231. replace2=\*replace;
232. \*replace=(\*replace)->rchild; //重接其右孩子
233. **delete** replace2;
235. }
236. **else** **if**((\*replace)->rchild==NULL) //右子树为空
237. {
238. replace2=\*replace;
239. \*replace=(\*replace)->lchild; //重接其左孩子
240. **delete** replace2;
241. }
242. }
243. **else**
244. {
245. //无替代元素，则根直接移向左子树，不管左子树是否为空都可以处理
246. replace2=root;
247. root=root->lchild;
248. **delete** replace2;
249. }
250. }
251. }
253. //前序
254. **void** PreOrder(Tree root)
255. {
256. **if** (root==NULL)
257. **return**;
258. printf("%d ",root->val);
259. PreOrder(root->lchild);
260. PreOrder(root->rchild);
261. }
262. //中序
263. **void** InOrder(Tree root)
264. {
265. **if** (root==NULL)
266. **return**;
267. InOrder(root->lchild);
268. printf("%d ",root->val);
269. InOrder(root->rchild);
270. }
271. **int** main()
272. {
273. Tree root=NULL;
274. insert(root, 11);
275. insert(root, 7);
276. insert(root, 18);
277. insert(root, 3);
278. insert(root, 9);
279. insert(root, 16);
280. insert(root, 26);
281. insert(root, 14);
282. insert(root, 15);
284. search(root,14);
285. printf("查找14:\n");
286. printf("前序:");
287. PreOrder(root);
288. printf("\n");
289. printf("中序:");
290. InOrder(root);
291. printf("\n");
293. //    remove(root,16);
294. //    remove(root,26);
295. //    remove(root,11);
296. remove(root,16);
297. printf("删除16:\n");
298. printf("前序:");
299. PreOrder(root);
300. printf("\n");
301. printf("中序:");
302. InOrder(root);
303. printf("\n");
304. **return** 0;
305. }

程序我反复测试过，目前还没有发现问题，当然也是还存在bug，不管是代码还是思考过程，如果有问题，还希望各位不吝指教。感谢。

main函数中某测试数据的图：

