AVL树学习

1.什么是AVL树

AVL树是带有平衡条件的二叉查找树，是其每个结点的左子树和右子树的高度差最多差1的二叉查找树。如图1，左边的树是AVL树，但右边的不是。

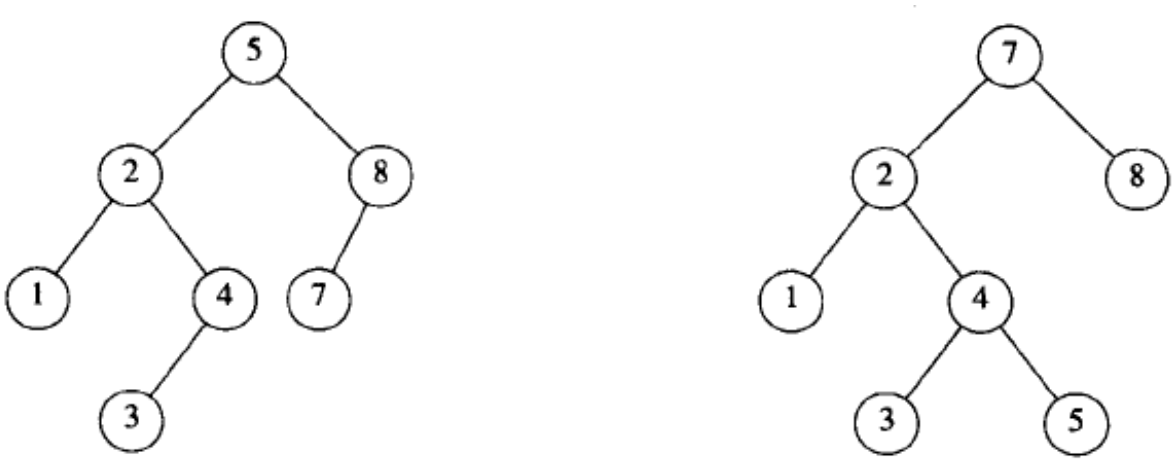

图1 左边的是AVL树，右边的不是

2.AVL树的基本操作

当向一棵AVL树中插入一个结点时，可能破坏AVL树的特性（例如，将6插入图1的AVL树中将会破坏项为8的结点平衡条件）。因此，我们需要对树进行简单的修正来恢复AVL树的平衡，这种修正操作叫做旋转（rotation）。在插入以后，只有那些从插入点到根节点的路径上的结点的平衡可能被破坏，因为只有这些结点的子树可能发生变化，在这条路径上可以发现一个结点，它的新平衡破坏了AVL条件，可以证明，恢复这个结点的平衡就可以保证整个树的AVL性质。

我们把必须平衡的结点叫做a。则AVl树的不平衡可能是由以下四种情形造成的：

（1）对a的左儿子的左子树进行一次插入

（2）对a的左儿子的右子树进行一次插入

（3）对a的右儿子的左子树进行一次插入

（4）对a的右儿子的右子树进行一次插入

对于（1）和（4），可以通过单旋转完成AVL树的修正，对于（2）和（3），可以通过双旋转来完成AVL树的修正。以下详细介绍单旋转和双旋转。

1.1单旋转

图2显示了向右单旋转如何调整情形（1），其中k2表示插入后不平衡结点a。

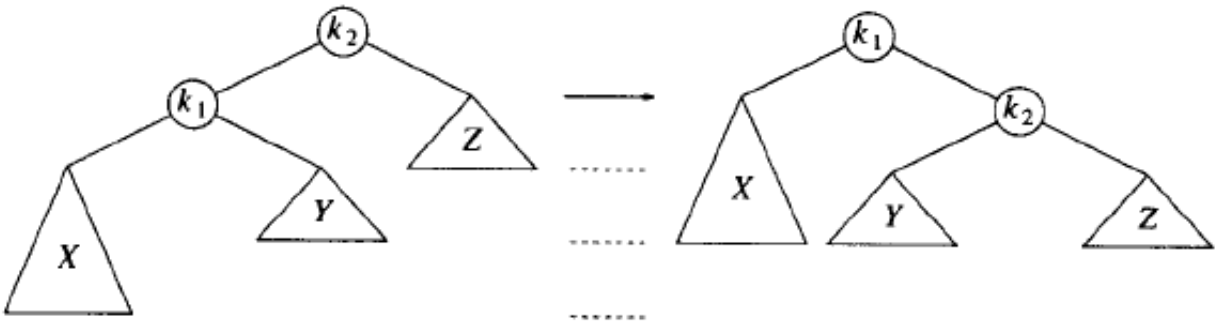

图2 向右单旋转修正情形（1）

图3显示了在将6插入左边原始的AVL树后结点8便不再平衡，于是，我们在7和8之间做一次单旋转，结果得到右边的树。

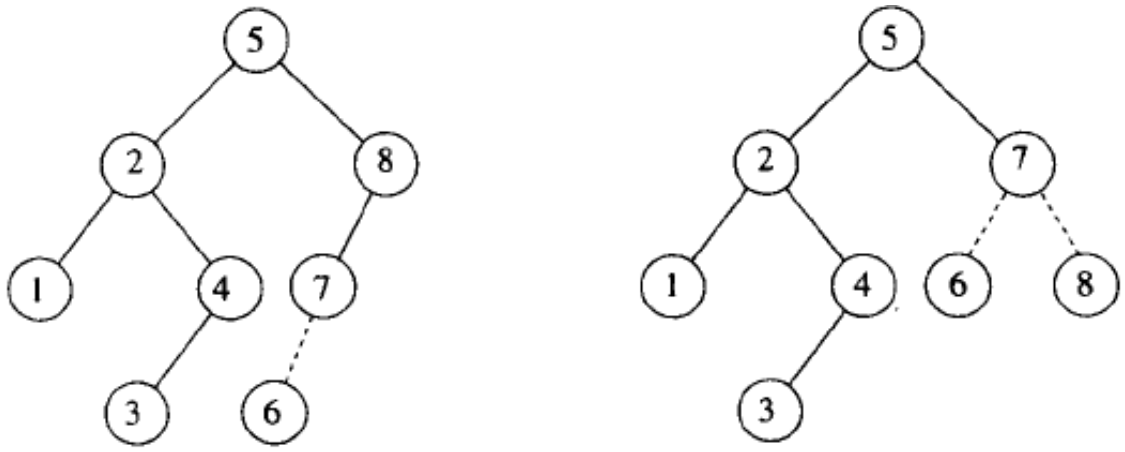

图3 插入6破坏了AVL性质，而后经过单旋转又将AVL性质恢复

图4说明了向左单旋转如何调整情形（4），与情形（1）对称。

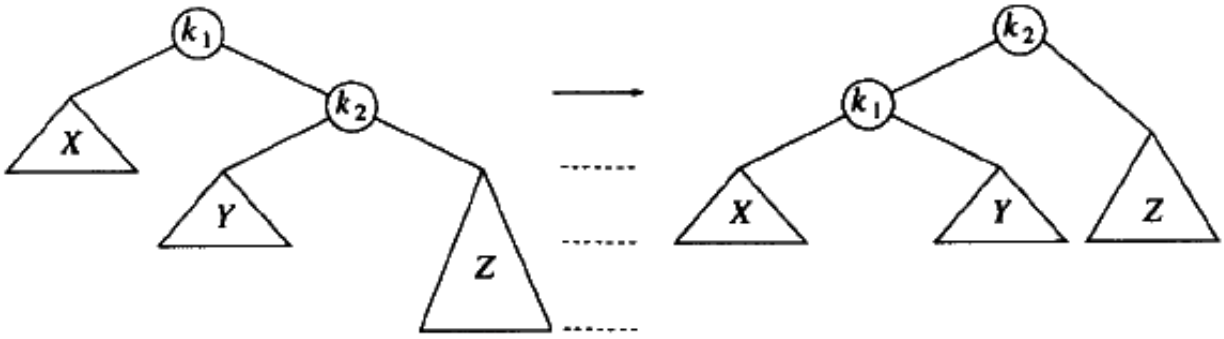

图4 向左单旋转修正情形（4）

图5中，左边的图是在原来的AVL树中加入结点7，结点5不再平衡，破坏了AVL特性。右边的图是经过单旋转修正之后的树。

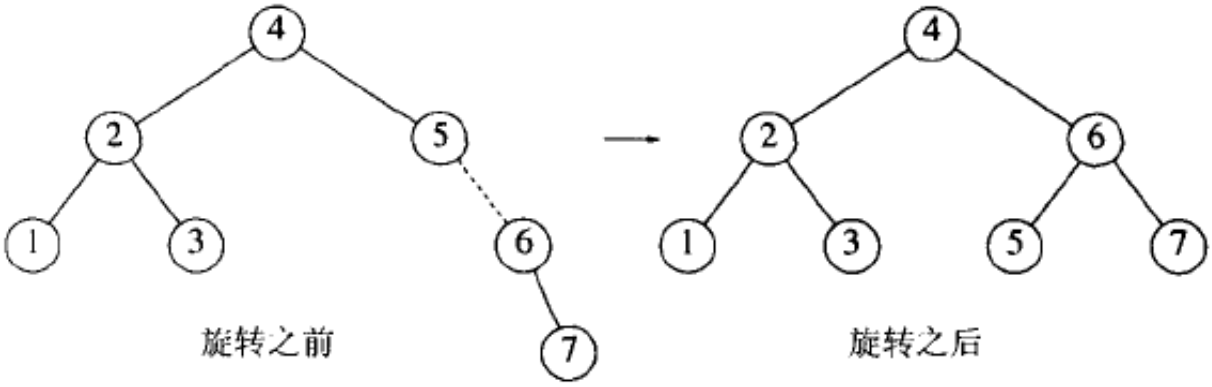

图5 插入7破坏了AVL性质，而后经过单旋转又将AVL性质恢复

1.2双旋转

图6说明了如何通过左-右双旋转修正情形（2），先将以k1为根的子树向左单旋转，再将已k3为根的子树向右单旋转，可以得到右边的树。

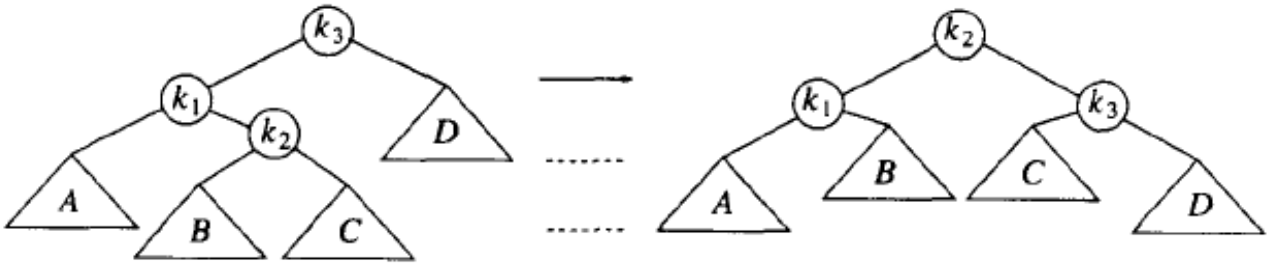

图6 通过左-右双旋转修正情形（2）

图7说明了如何通过右-左双旋转修正情形（4），先将以k3为根的子树向右单旋转，再将已k1为根的子树向左单旋转，可以得到右边的树。

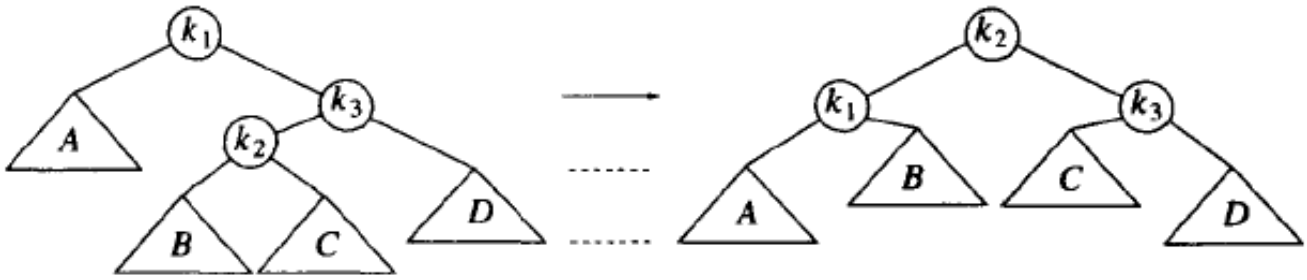

图7 通过右-左双旋转修正情形（4）

图8中，左边的图中，在原来的AVL树中加入结点14，结点6失去平衡，属于情形（4），右边的图式经过右-左双旋转修正之后的树。

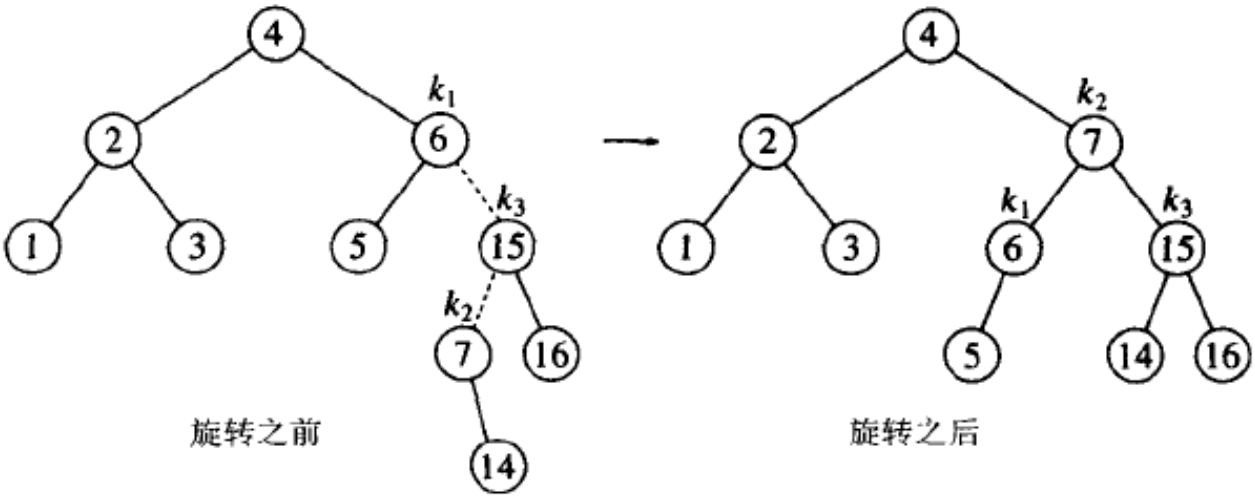

图8 插入结点14，经过右-左双旋转之后的树

3.代码实现

AVL树定义如下：

```
typedef struct AvlNode * AvlTree;//AvlTree
```

AVL树中插入操作如下：

```
AvlTree insert(AvlTree tree, ElemType value)

                tree = (AvlTree)malloc(sizeof(AvlNode));

                tree->left = tree->right = NULL;

        else if(value < tree->elem)//在左子树中添加元素

                tree->left = insert(tree->left, value);//递归插入，类似于二叉
查找树

                if(getHeight(tree->left) - getHeight(tree->right) == 2)//失
去平衡

                        if(value < tree->left->elem)            //情形
（1），向右单旋转

                                tree = singleRotateWithLeft(tree);

                        else                                    //情形
（2），右-左双旋转

                                tree = doubleRotateWithLeft(tree);

        else if(value > tree->elem)//在右子树中添加元素

                tree->right = insert(tree->right, value);

                if(getHeight(tree->right) - getHeight(tree->left) == 2)//失
去平衡

                        if(value > tree->right->elem)           //情形
（4），向左单旋转

                                tree = singleRotateWithRight(tree);

                        else                                    //情形
（3），左-右双旋转

                                tree = doubleRotateWithRight(tree);

        tree->height = max(getHeight(tree->left), getHeight(tree->right))
+ 1;
```

向右单旋转代码如下：

```
AvlTree singleRotateWithLeft(AvlTree k2)

        k2->height = max(getHeight(k2->left), getHeight(k2->right)) + 1;

        k1->height = max(getHeight(k1->left), k2->height) + 1;
```

向左单旋转代码如下：

```
AvlTree singleRotateWithRight(AvlTree k1)

        k1->height = max(getHeight(k1->left), getHeight(k1->right)) + 1;

        k2->height = max(k1->height, getHeight(k2->right)) + 1;
```

左-右双旋转代码如下：

```
AvlTree doubleRotateWithLeft(AvlTree k3)

        singleRotateWithRight(k3->left);

        return singleRotateWithLeft(k3);
```

右-左双旋转代码如下：

```
AvlTree doubleRotateWithLeft(AvlTree k1)

        singleRotateWithLeft(k1->right);

        return singleRotateWithRight(k1);
```