Chapter 10

AVL Trees

(Revision Date: 22 March 1999)

©1999, Thomas A. Anastasio. May not be reproduced in any form without written permission of the author.

10.1 Height-Balanced Binary Trees

Most operations on binary trees have asymptotic performance given by $O(h)$, where h is the height of the tree. In compact binary trees, tree height is approximately equal to $\lg n$, where n is the number of nodes in the tree. It is therefore desirable to build trees which are reasonably compact. One way to do this is to require that the tree be *height balanced*.

◇ **Definition**: An empty binary tree is height balanced. If T is a non-empty binary tree with T_L and T_R as left and right subtrees, respectively, then T is height balanced if and only if

1. T_L and T_R are height balanced, and
2. $|h_R - h_L| \leq 1$, where h_R and h_L are the heights of T_R and T_L, respectively.

A less formal, but equivalent, definition of height balance is:

◇ **Definition**: A height-balanced binary tree is one for which **at every node**, the absolute value of the difference in heights of the left and right children is no larger than one.

It is easy to show that every complete binary tree is height-balanced. It is not true that every height-balanced binary tree is complete.

Theorem 10.1 The largest number of nodes in a height-balanced binary tree (HBT) of height h is $2^{h+1} - 1$.

▷ **Proof**: The most compact HBT is a perfect binary tree[1]" and this is the number of nodes in such a tree. ◁

◇ **Definition**: The *balance factor* of a binary tree node is the difference in heights of its two subtrees ($h_R - h_L$). The balance factor of a binary tree is the balance factor of its root node. The balance factor of a height balanced binary tree may take on one of the values $-1, 0, +1$. Any other balance factor indicates an unbalanced tree.

A binary tree is said to be "left-heavy" when bf = -1, "equal-height" when bf = 0, and "right-heavy" when bf = +1.

10.2 Rebalancing AVL Trees

An *AVL tree* is a binary search tree that maintains height balance after every insertion or deletion. Since there will be some insertions or deletions that will destroy the balance property, an AVL tree must sometimes be restructured to restore balance. This restructuring is done by performing one or more "rotations" on parts of the tree.

Of course, not every insertion or deletion will necessarily cause an AVL tree to become unbalanced.

Theorem 10.2 An AVL tree with zero balance factor cannot be unbalanced by the insertion of a single node.

▷ **Proof**: A new node will be added as a leaf to one of the subtrees, increasing that subtree's height by (at most) 1. Therefore, the balance factor of the tree may change to $+1$ or -1, keeping the tree balanced. ◁

An AVL tree with non-zero balance factor *may* become unbalanced (balance factor becomes ± 2) upon insertion of a new node. An AVL tree which becomes unbalanced by insertion of a node can be re-balanced by performing one or more *rotations*. A rotation is a re-structuring of the tree that maintains the binary search tree property. Rotation is said to be done "around a node."

`CCWRotate` means rotate *from* the right *toward* the left (counter-clockwise). The other way for `CWRotate`. For example, Figure 10.1 shows a CW rotation and a CCW rotation around node `X` in a binary tree.

10.2.1 Rotations Are Dictated by a Node's Children

When rebalancing is to be done at a node, the type of rotation depends on the balance factor at the right or left child. There are two rebalancing cases for

[1] A perfect binary is a full binary tree in which all the leaves are on the same level.

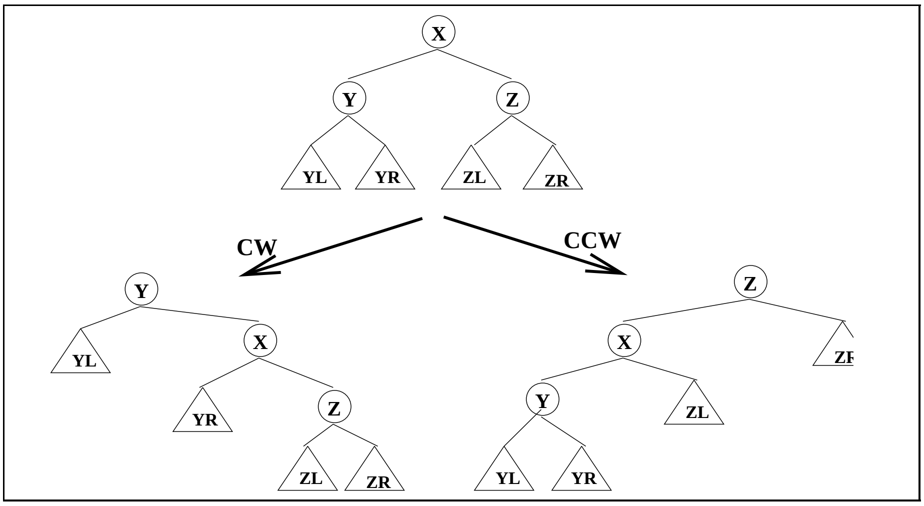

Figure 10.1: Binary Tree CW and CCW Rotations

right child and two cases for left child. Here are the two right child cases (the left child cases are symmetric).

Case 1: Right Child's Balance Factor Becomes +1
Figure 10.2 shows an abstract AVL tree for case 1. In this case, the balance factor of the right subtree of R becomes +1 because Z is added to X_R. The tree rooted at R has balance factor +1 before node Z is inserted and +2 afterwards. To rebalance the tree, a CCWRotate is performed at R. The resulting abstract AVL tree is shown. **Note that the height of the tree after re-balancing is the same as the height before insertion of the new node.**

Case 2: Right Child's Balance Factor Becomes -1
Figure 10.3 shows an abstract AVL tree for case 2. In this case, the balance factor of the right subtree of R becomes -1 because Z is added to X_L. As in case 1, the balance factor at R changes from +1 to +2. To rebalance the tree, two rotations will be necessary. The first rotation will be CWRotate around X; the second rotation will be CCWRotate around R. The figure shows X_L expanded to show its root W and subtrees. There are two possibilities for placement of Z, on W_L or on W_R. Since it doesn't matter in the final analysis, we pursue only one of the two possibilities. Figure 10.4 shows the re-balanced tree after CW rotation about X and CCW rotation about R.

Note that the height of the tree after re-balancing is the same as the height before insertion of the new node.

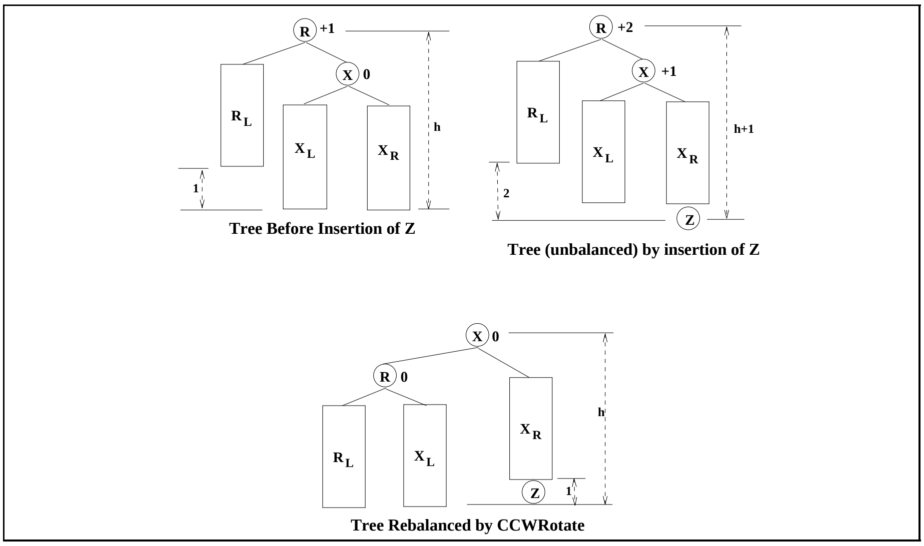

Figure 10.2: Insertion in AVL Tree (Case 1)

10.3 Insertion into an AVL Tree

The `doInsert` procedure for AVL trees does a node insertion just like node insertion in an ordinary binary search tree (`BSTree`). Each recursive call takes you further down the tree until the insertion point is found. In `BSTree`, we are satisfied to simply go back up the recursion call stack without doing any more work. In `AVLTree`, we do some work at each node on the way back up; namely we adjust the node heights (the balance factors) and perhaps do a rebalance.

10.3.1 Important notes on AVL tree insertions

1. Before we start the insertion, the tree is an AVL tree. That means that some nodes may have non-zero balance factors, but no node has a balance factor greater than 1 in absolute value.

2. We have shown (Figures 10.2, 10.3, and 10.4) that the height of the subtree into which the insertion is made is the same after re-balancing as it was before the insertion. This means that we only need to rebalance that subtree, not the entire AVL tree of which it is a part.

3. We find the root of the subtree needing rebalancing as we work our way back up the recursion. It is the *closest* node to the insertion point that gets a balance factor of ± 2.

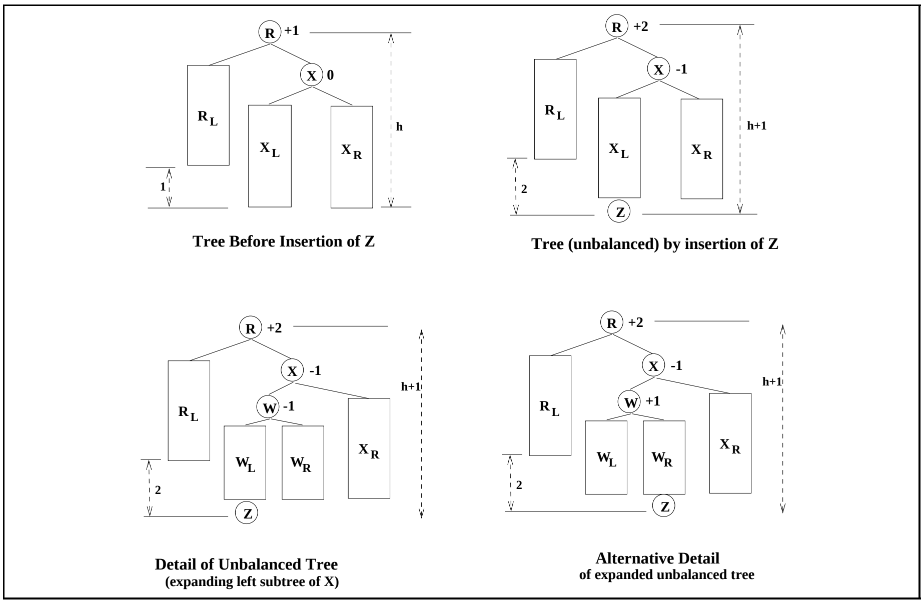

Figure 10.3: Insertion in AVL Tree (Case 2 before rotations)

10.3.2 Rebalancing after an Insertion

The following rules apply when a tree becomes unbalanced after insertion of a new node (the tree balance factor becomes ± 2). The rules apply to the closest unbalanced node above the insertion point.

1. when the subtree root becomes unbalanced to the **right** after insertion of a new node.

```
if root.right is +1 then CCWRotate about root
if root.right is -1 then
    CWRotate about root.right
    CCWRotate about root
```

2. when the subtree root becomes unbalanced to the **left** after insertion of a new node.

```
if root.left is -1 then CWRotate about root
if root.left is +1 then
    CCWRotate about root.left
    CWRotate about root
```

10.3.3 Example of Insertions into AVL Tree

Figure 10.5 shows the growth of an AVL tree in which the values {1, 2, 3, 4, 5, 0, 7, 6} are inserted. The number next to each node is the balance factor

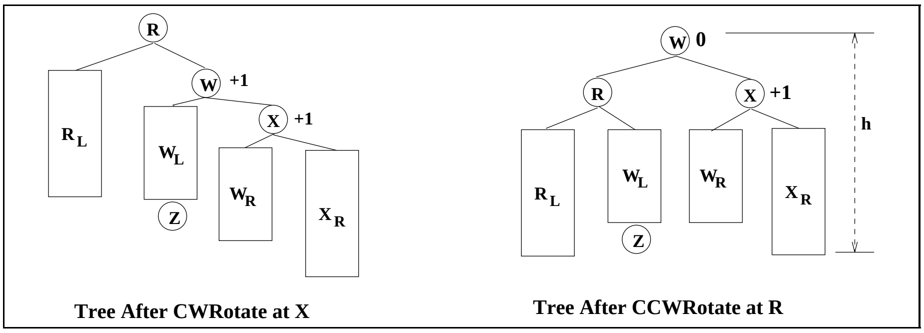

Figure 10.4: Insertion in AVL Tree (Case 2 after rotations)

of the node. At each insertion, the tree is shown after the insertion. When re-balancing is done, the rebalanced tree is also shown.

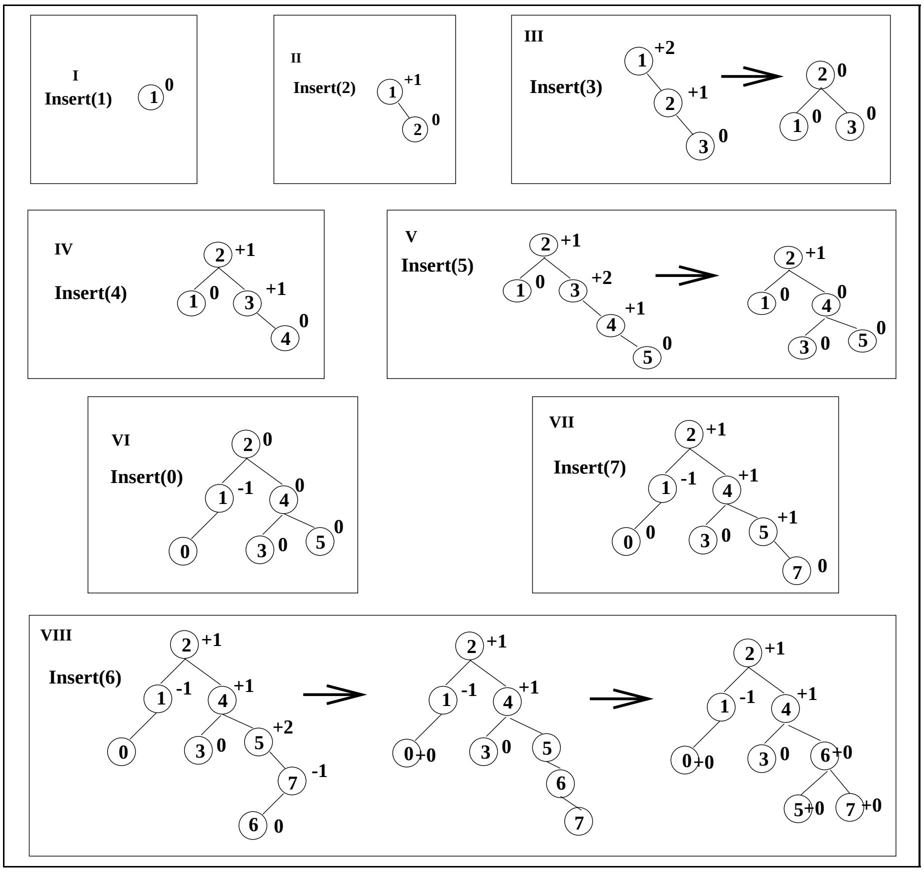

Figure 10.5: Example of Insertion of $\{1, 2, 3, 4, 5, 0, 7, 6\}$ into an AVL Tree

I `doInsert(1)`: The node is inserted into an empty AVL tree. The tree remains an AVL tree.

II `doInsert(2)`: The tree remains AVL.

III `doInsert(3)`: Node `1`'s balance factor becomes `+2`, and its right child's (node `2`) balance factor becomes `+1`. This is "case 1," so we CCWRotate about the unbalanced node (node `1`).

IV `doInsert(4)`: The tree remains AVL.

V `doInsert(5)`: Node `3`'s balance factor becomes `+2`, and its right child's (node `4`) balance factor becomes `+1`. This is "case 1," so CCWRotate about the unbalanced node (node `3`).

VI `doInsert(0)`: The tree remains AVL.

VII `doInsert(7)`: The tree remains AVL.

VIII `doInsert(6)`: Node `5`'s balance factor becomes `+2` and its right child's (node `7`) balance factor becomes `-1`. This is "case 2." Therefore, CWRotate about the child (node `7`) and then CCWRotate about the unbalanced node (node `5`).

10.3.4 Overview of Implementation of Insertion

The following pseudocode gives a skeleton of the way an insertion is done in an AVL tree. `X` is the `Object` being inserted.

```
if X belongs on the left
  if there is no left child,
      make X be the left child
  else recursively insert X in the left subtree
else if X belongs on the right
   if there is no right child
       make X be the right child
   else recursively insertX in the right subtree
else X is a duplicate, don't insert it

Now, rebalance as needed as the recursion unwinds
```

10.3.5 Performance of `doInsert`

It looks like `doInsert` does a lot more work for an AVL tree than for an ordinary binary search tree. First of all, it must do the usual insertion work. Then, working back up the recursion, it must check for balance conditions and rebalance the tree if unbalance is detected. As it turns out, this is asymptotically no more work.

The actual insertion is exactly the same as in an ordinary binary search tree, so it is $O(d)$, where d is the depth of the tree. Since an AVL tree maintains balance, the depth of the tree is bounded by $O(\lg n)$. Since working back up the recursion involves no more steps than the depth of the tree, it is also $O(\lg n)$. The remaining question is how much work is done at each step, going back up the recursion? The re-balancing takes a constant amount of work and occurs just at the node that becomes unbalanced. How about the recomputation of node height? That depends on how the computation is done. Since an `AVLTreeNode` stores its height, any node can compute its own height by checking the heights of its left and right children in constant time.

Insertion of an element into an `AVLTree` involves ordinary `BSTree` insertion followed by rebalancing. The rebalancing is done as the recursion unwinds and occurs only at the node that became unbalanced by the insertion. The reason it occurs only at that one node is that the rebalancing is guaranteed not to change the height of the tree rooted at that node[2], so the balance further up the tree is unchanged. The rules for rebalancing involve looking at the right (or left) child of the unbalanced node. This child node is guaranteed to have a balance factor of ± 1.

10.4 Deletion from an AVL Tree

Deletion of an element from an `AVLTree` is similar to insertion in that ordinary `BSTree` deletion is done. The big difference is that rebalancing may have to be done all the way back to the root. We cannot stop at the first unbalanced node as we can with insertion. We shall also have to introduce another rule for rebalancing because the right (or left) child of the unbalanced node may have a balance factor of `0`, `+1`, or `-1`.

10.5 Overview of Implementation of Deletion

The following pseudocode shows how deletion is done from an AVL tree. `X` is the object to be deleted, `node` is the root of the subtree being currently explored, and `parent` is the parent of `node`.

```
if X belongs on the right of node
   recursively delete X from the right subtree
else if X belongs on the left of node
   recursively delete X from the left subtree
else // X belongs at node
    if node is a leaf
       remove node from the tree and return
    else if node has a right subtree
        set the value at node to the minimum value in
           the right subtree, then recursively delete
           that value from the right subtree
    else // node has left subtree
        set the value at node to the maximum value in
           the left subtree, then recursively delete
           that value from the left subtree

reBalance the tree at node, as needed
```

10.5.1 Special Case for Deletion

Theorem 10.2 guarantees that with insertion we never get an unbalanced node having its relevant child with a balance factor of `0`. No such guarantee can be made in the case of deletion. Figure 10.6 shows an example of how a node can have a balance factor of `+2` after a deletion in a subtree and yet have a right

[2]See the diagrams on page 167

child with a balance factor of `0`. Of course, there is also the symmetric case of a balance factor of `-2` with a left child balance factor of `0`. The rules for rebalancing must be modified to account for these cases.

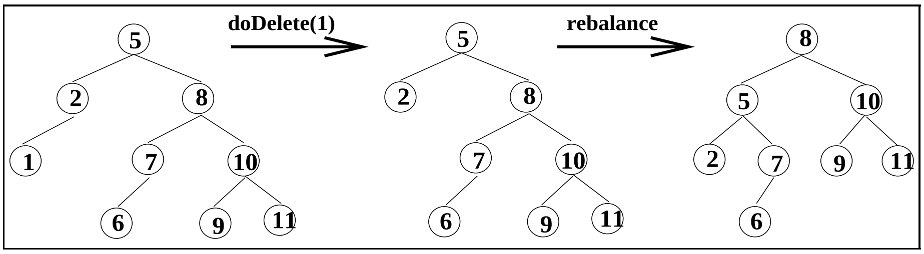

Figure 10.6: Example of Deletion from an AVL Tree

Starting with the `AVLTree` on the left of the figure, we first delete the “`1`” node. This causes the balance factor at node “`2`” to become zero and the balance factor at node “`5`” to become `+2`. Thus, node “`5`” is unbalanced to the right, but its right child (node “`8`”) has zero balance factor. A `CCW` rotation at node “`5`” will rebalance the tree.

10.5.2 New Rules for Rebalancing

The following rules for rebalancing work for both insertion and deletion. The only difference from previous rules in section 10.3.2 is that the relevant child may now have zero balance factor. This situation can only happen with deletion.

1. when the subtree root becomes unbalanced to the **right** after insertion or deletion of a node.

```
if root.right is +1 or 0 then CCWRotate about root
if root.right is -1 then
    CWRotate about root.right
    CCWRotate about root
```

2. when the subtree root becomes unbalanced to the **left** after insertion or deletion of a node.

```
if root.left is -1  or 0 then CWRotate about root
if root.left is +1 then
    CCWRotate about root.left
    CWRotate about root
```