Part 2: Balanced Trees

1 AVL Trees

We could define a perfectly balanced binary search tree with N nodes to be a complete binary search tree, one in which every level except the last is completely full. A perfectly balanced BST would have guaranteed $O(\log N)$ search time. Unfortunately it is too costly to rebalance such a tree after each insertion and deletion. We need a tree with a weaker balancing condition. An AVL tree is such a tree. An ***AVL tree*** (AVL for the names of its inventors, Adelson-Velskii and Landis) is a binary search tree with a balancing condition that is relatively inexpensive to maintain.

Definition 1. An ***AVL tree*** is a binary search tree in which, for every node in the tree, the heights of the left and right subtrees of the node differ by at most 1. This is called the ***balancing condition***.

Note. We will see that a consequence of this definition is that the height of an AVL tree with N nodes, even in the worst case, is proportional to $\log N$.

For the sake of precision, we will define the ***balance factor*** at a node as the ***height of the left subtree of that node minus the height of the right subtree of that node***. Figure 1 has four examples of AVL trees. The smallest has one node whose balance factor is 0. The second has two nodes. The balance factor at the root is +1 and that of the child is 0. In the third tree, the root has a balance factor of 1, because its left tree has height 1 and its right tree has height 0. The node containing 10 has a balance factor of 1, and the other nodes have balance factors of 0. The fourth tree's root has a balance factor of 1, and the node containing 10 has a balance factor of 0.

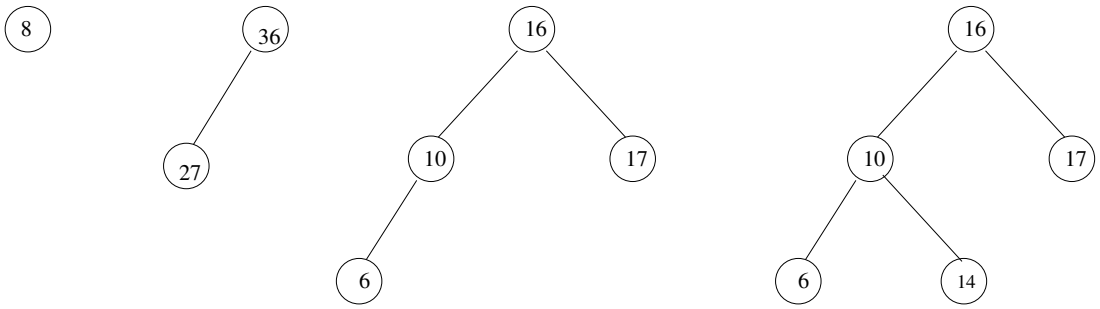

Figure 1: Three small AVL trees

Examples of binary search trees that are not AVL trees are shown in Figure 2.

1.1 AVL Tree Search

A search takes $O(h)$ time, where h is the height of the tree. This is because an AVL tree is a binary search tree and the search algorithm is the same for both trees. This is the easy part. Because the height of an AVL tree is at worst $O(\log N)$[1], searching takes $O(\log N)$ time in the worst case.

[1] We will establish this later in this chapter.

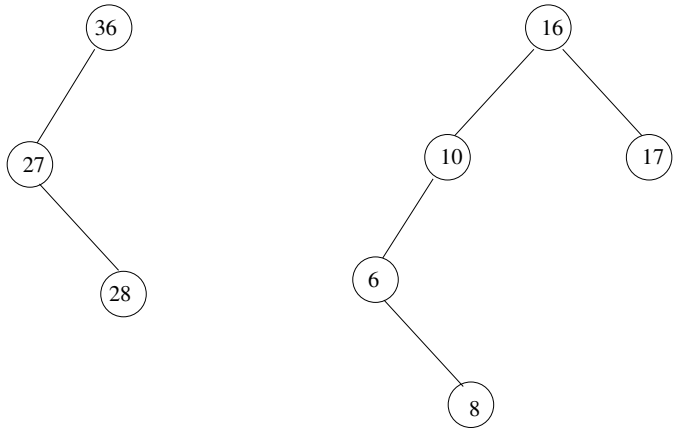

Figure 2: Binary search trees that are not AVL trees

1.2 AVL Tree Insertion

When an element is inserted into a tree, whether as a left child or a right child of some node p, the height of the tree rooted at p may or may not increase. If p had a left child but no right child, and the new node is p's new right child, the tree rooted at p does not change height as a result of the insertion. But if p had no children at all and this is a new child, then p's tree increased height by 1. In this case it is possible that the tree rooted at the parent of p increased in height as well, and that the tree rooted at the grandparent of p increased in height as well. If you look at the trees in Figure 1 and imagine inserting a node containing 4 as the left child of 6 in either of the two rightmost trees, you can see that each of the trees all of the way up the path to the root increased in height. On the other hand, if you insert 18 as the right child of 17, then the tree rooted at 17 increases in height but not that of its parent.

We can conclude that an insertion might increase the height of one or more subtrees by 1. This will change the balance factors of the trees on the path up to the root from the point of insertion. Some of the trees will stay balanced, meaning their balance factors are -1, 0, or 1, whereas others may now have a balance factor of -2 or +2. For example, consider the tree that is the second from the right in Figure 1. If we give a new left child to node 6, then its balance factor changes from 0 to 1 and the balance factor of the node containing 10 changes from 1 to 2.

Consider traveling up the tree from the point of insertion, inspecting the balance condition at each node. The first node found that is out of balance (not -1, 0, or 1) can be out of balance in one of four ways. An analysis shows that of the four different cases, two are symmetric. There are therefore two theoretical cases and their symmetric counterparts. Let A be the node of greatest depth that is out of balance, meaning the first one found while traveling up the path from the point of insertion. The four possibilities are

1. The insertion was in the left subtree of the left child of A
2. The insertion was in the right subtree of the left child of A
3. The insertion was in the left subtree of the right child of A
4. The insertion was in the right subtree of the right child of A

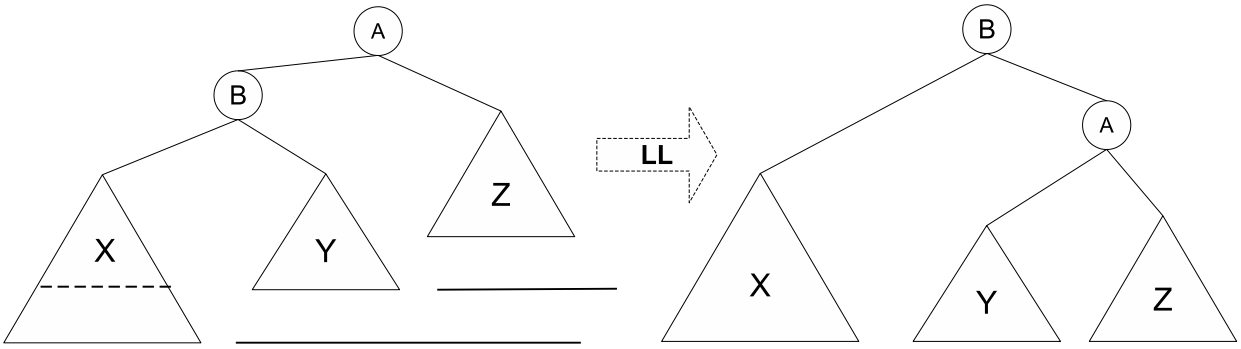

Figure 3: LL Rotation

We will see that there are four different rebalancing algorithms that restore the AVL tree's balance condition at all of its nodes. For now, these algorithms will be known as the LL, RR, RL, and LR rotations. Soon they will be explained.

Cases 1 and 4 are symmetric and are resolved with an LL or RR rebalancing respectively. Cases 2 and 3 are symmetric and are resolved with an LR or RL rebalancing respectively.

1.2.1 The Single Rotations

The LL and RR rotations are called ***single rotations***. They are symmetric to each other. Although the RR is symmetric, this does not mean that it can use the same code; it cannot, as will become evident. The LL rotation is depicted in Figure 3. The LR and RL rotations are called double rotations. We will get to them later.

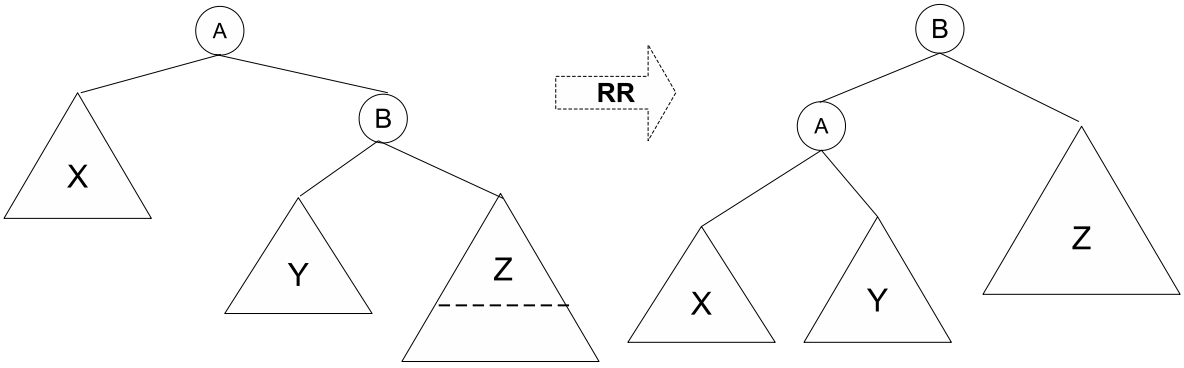

Figure 4: RR Rotation

In the LL rotation in the figure, the left child of A is labeled B. B has two subtrees labeled X and Y, and A has a right child labeled Z. In the LL rotation, B rotates up to take the place of A, and A becomes B's right child. B keeps its left subtree, X, but it gives its right tree to A, which "adopts" it as its left subtree.

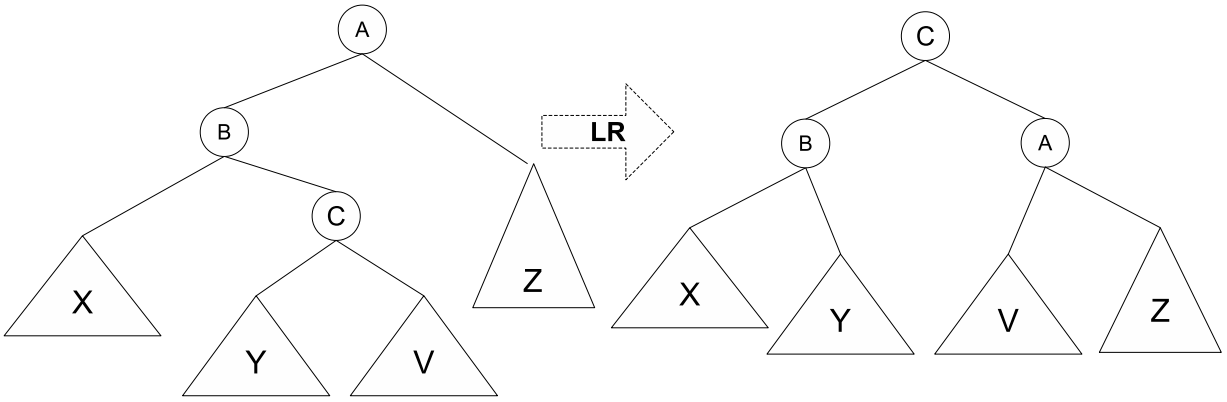

Figure 5: A LR double rotation

We now explain why the LL single rotation does not increase the height of the tree rooted at the unbalanced node, A. To see this you have to observe the following facts. Let h be the height of tree X after the insertion. The tree rooted at A has height $h+2$ because there is an edge from A to B and an edge from B to the root of X.

1. Since all nodes below A are in balance, the tree rooted at B is in balance, which implies that $height(X) - height(Y) < 2$
2. If $height(Y) = height(X)$ after the insertion, then before the insertion, the tree rooted at A would not have been an AVL tree, since the left subtree of B (i.e., X) would have had the same height that it has after the insertion, and it is not an AVL tree now. Therefore, $height(X) = height(Y) + 1$.
3. Since the tree rooted at A is not an AVL tree now, $height(X)$ - $height(Z) \geq 2$. If $height(X)$ - $height(Z) > 2$, then it would not have been an AVL tree before the insertion, so $height(X)$ - $height(Z) = 2$.
4. It follows that $height(Y) = h - 1$ and $height(Z) = h - 1$. They are the same height.
5. After the rotation, the left subtree of B has height $h+1$ relative to B, and the right subtree has height $(h-1+2) = h+1$ relative to B. Thus the tree has height $h+1$, which is the height that the tree was prior to the insertion.
6. This shows that the subtree rooted at B is restored to the height it had prior to the insertion, which implies that no further rebalancing is necessary.

A single rotation is performed once and the tree is restored to an AVL tree at a cost of a few pointer assignments. Balance factors or height information needs to be updated in the nodes along the path to the point of rotation, but this is no more than $O(\log N)$ steps. A full analysis follows after we explain the double rotations.

An example of a function to perform the LL rotation follows, assuming that the `height()` function returns -1 when its argument is `NULL`. It is also assumed that this function is only called if the tree rooted at `k2` needs an LL rotation. Otherwise there is a chance that a null pointer will be

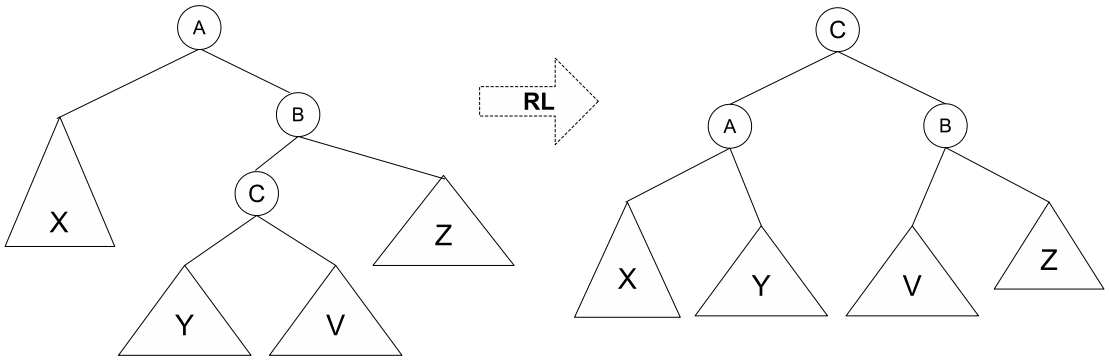

Figure 6: A RL double rotation

dereferenced. If the tree needs an LL rotation at `k2`, then `k2->left` is not null, so `k2->left->right` cannot be a null-pointer dereference. It is also important to observe that the height function used in this code returns a -1 if its argument is a null pointer.

```
void LL_rotation( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```

It is not hard to write the RR rotation after examining the above code.

1.2.2 The Double Rotations

The LR and RL rotations are called ***double rotations*** because they can actually be implemented by two successive single rotations. It is easier to see it though as a single operation, and that is how it is explained in these notes. The double rotations are required because a single rotation would not rebalance in this case, as you can verify with an example.

To perform the double rotation, travel up the tree from the point of insertion until you reach a node that is no longer in balance. Call this node A. All nodes on the path from the point of insertion up to but not including A remained in balance but A is out of balance. Figure 1.2.1 depicts the LR rotation.

The LR rotation is used when the insertion is in the right subtree of the left child of A. Therefore, the right subtree of the left child of A has grown in height by 1. We do not know whether it is the left or right subtree of this tree, i.e., whether it is the subtree labeled Y or V in Figure 1.2.1, but it does not matter. In the figure, the root of the right child of the left child of A is C. To do the double LR rotation, make C the new root, replacing A, make its parent B its left child and make the old root, A, the right child of C. Because C is now the root and it had two children, it must give up its old subtrees for adoption. It gives Y to B and V to A, as illustrated.

The rotation reduces the height of the tree by 1 for similar reasons as above. If h is the height of Y (or V) after the insertion, then $height(X)$ must be h also. $Height(Z)$ must be $h+1$. It cannot be less. Why? After the rebalancing, the tree rooted at B has height $h+2$, which is the height it had prior to the insertion. Thus, the rest of the tree remains in balance and no further actions need to be performed.

1.2.3 Analysis of Insertion Time

If no rotation is performed, the insertion takes $O(h)$ steps. If any rotation is performed, regardless of which it is, the height of the rebalanced subtree is reduced to what it was prior to the insertion. (So then how can a tree grow in height?) This implies that the height of the tree after rebalancing is the height that it was before the insertion. Therefore, since the tree was an AVL tree before the insertion, all ancestor nodes were in balance and they remain in balance after the insertion and rebalance. This implies that once the rebalancing has been done, the insertion is complete. Rebalance takes constant time; therefore insertion takes $O(h)$ steps for the search plus a constant for the rebalancing.

1.2.4 The Insertion Implementation

There are two ways to implement the insertion algorithm. In a sense the simplest is to modify the BST insertion code with the inclusion of a call to a single function to rebalance the tree. This is what the the fourth edition of the Weiss textbook does. The alternative is to embed the rotations in the `insert()` function directly. I include both methods for the purpose of illustration.

The method using a single call to a balance function needs to define a height function that returns -1 if the pointer is `NULL`, and the height stored in the node otherwise. This function and the `balance()` function as well as the version of `insert()` using them is in the following listing.

```
static const int ALLOWED_IMBALANCE = 1;

int height( AvlNode *t ) const
{
    return t == nullptr ? -1 : t->height;
}


// Assume t is balanced or within one of being balanced
void balance( AvlNode * & t )
{
    if ( t == nullptr )
        return;

    if ( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
        if ( height( t->left->left ) >= height( t->left->right ) )
            LL_rotation( t );
        else
            LR_rotation( t );
    else if ( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
        if ( height( t->right->right ) >= height( t->right->left ) )
            RR_rotation( t );
        else
```

```
            RL_rotation( t );

    t->height = max( height( t->left ), height( t->right ) ) + 1;
}

void insert( const Comparable & x, AvlNode * & t )
{
    if ( t == nullptr )
        t = new AvlNode{ x, nullptr, nullptr };
    else if ( x < t->element )
        insert( x, t->left );
    else if ( t->element < x )
        insert( x, t->right );

    balance( t );
}
```

You can verify that the logic of the `balance()` function is correct. Note that because the `insert()` function is recursive, the `balance()` function will be called at each level of recursion. It will not do any work after any of the rotations are called, because these rotations will adjust the heights of the subtrees and later called to check the height balance will find them within the allowed limit.

A version of `insert()` that embeds the rotations directly follows:

```
insert( const Comparable & x, AvlNode* & t ) const
{
    if ( t == NULL )
                t = new AvlNode( x, NULL, NULL );
    else if ( x < t->element ) {
        insert( x, t->left );
        if ( height( t->left ) - height( t->right ) == 2 )
            if ( x < t->left->element )
                LL_rotation( t );
            else
                LR_rotation( t );
    }
    else if ( t->element < x ) {
        insert( x, t->right );
        if ( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                RR_rotation( t );
            else
                RL_rotation( t );
    }
    else
        ;  // Duplicate; do nothing
        t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

1.3 AVL Tree Deletion

Deletion from an AVL tree is more complicated than insertion. First of all, the basic deletion algorithm is the same as that of an unbalanced BST: if a node is a leaf, delete it; if it has one

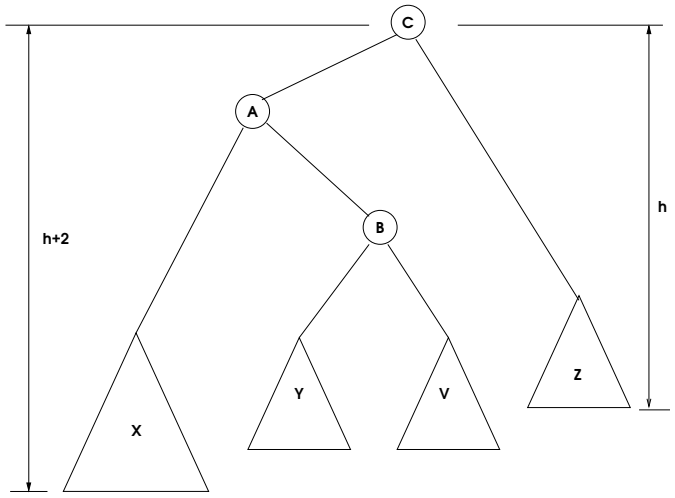

Figure 7: Case 1 AVL tree before rebalancing

subtree, make the subtree the child of the node's parent, and if it has two subtrees, replace it with the in-order successor or predecessor and delete the node that contained that element.

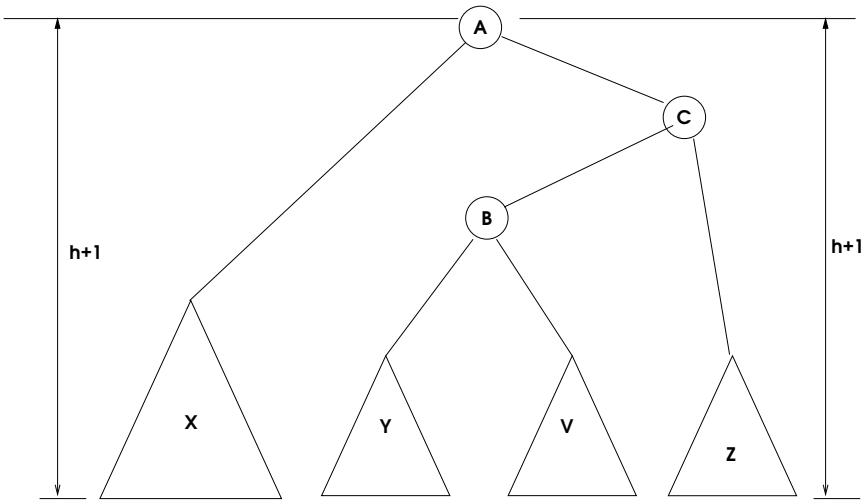

Figure 8: Case 1 AVL tree after rebalancing

Having deleted the node, the height of some subtree has been reduced by 1. This reduction in height might have caused some node on a path from that node back up to the root to lose its balance condition. Consider the first node on the path back to the root whose balance condition is now violated.

It does not matter that the loss of balance condition was caused by a deletion – one of the four rebalancing algorithms will rebalance the tree that is out of balance. The question is, how do we know which to do? The wrong choice will leave the tree unbalanced. We will answer this question soon.

The other problem is that rebalancing may make the height of the tree smaller. (After an insertion, rebalancing always restores the tree to its height before the insertion, but after a deletion, it may not change the height at all!) Unlike the insertion problem, where the height had increased and is

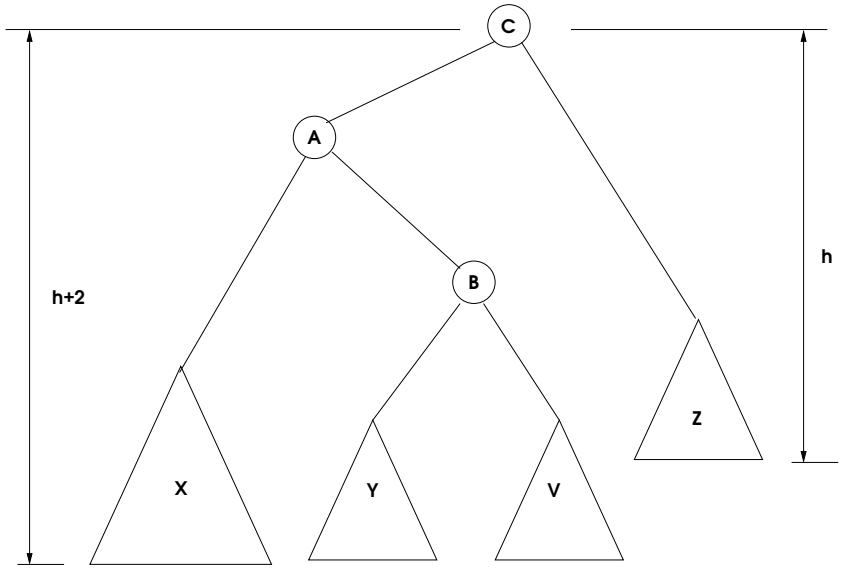

Figure 9: Case 2 AVL tree before rebalancing

now restored to what it was before, in deletion, the height was diminished, and now rebalancing may diminish it again. This implies that the parent of the node whose tree was just rebalanced may become unbalanced, requiring that its tree be rebalanced, and then its parent might need it, and so on until we reach the root. When we finally get to the root, we rebalance and the process stops there. The time for a deletion is therefore $O(h)$ steps to find the node and delete it, and $O(h)$ rebalancing steps in the worst case.

We now prove that the deletion algorithm is correct.

Suppose that we delete a node from the right subtree of a node, C. Assume that the tree rooted at C was balanced prior to the deletion but that after the deletion it is not balanced. Figure 7 illustrates schematically what the tree rooted at C must look like. In the figure, h is the height of the right subtree of C with respect to C after the deletion occurred. Then the left subtree must be of height $h+2$. (It cannot be more, otherwise the tree would have been out of balance before the deletion. It cannot be $h+1$, h, or $h-1$ because the tree rooted at C would still be in balance. It cannot be $h-2$ because that would imply that before the deletion it was out of balance.) Unlike the unbalanced tree that results from an insertion, the resulting unbalanced tree may have more than one leaf node in the left subtree at height $h+2$. The height of the tree labeled Z is $h-1$ because there is one edge from C to its root. We make the following claim:

Lemma 2. *Suppose that C is the root of an AVL tree in whose right subtree a deletion has occurred, and all subtrees of C are balanced. Suppose that t is a pointer to C and that $height(t \rightarrow left)$-$height(t \rightarrow right) = 2$. Then an LL rotation at t will rebalance the tree if and only if $height((t \rightarrow left) \rightarrow left) \geq height((t \rightarrow left) -> right)$ and an LR rotation will rebalance the tree otherwise.*

Proof. First we prove that if $height((t \rightarrow left) \rightarrow left) \geq height((t \rightarrow left) \rightarrow right)$ then an LL rotation will rebalance the tree. We do this in two steps, separating the case of unequal heights from equal heights. Assume that the tree is rooted at C. Assume also that t is a pointer to C, that the height of the right subtree is h, and consequently that the height of the left subtree is $h+2$, as

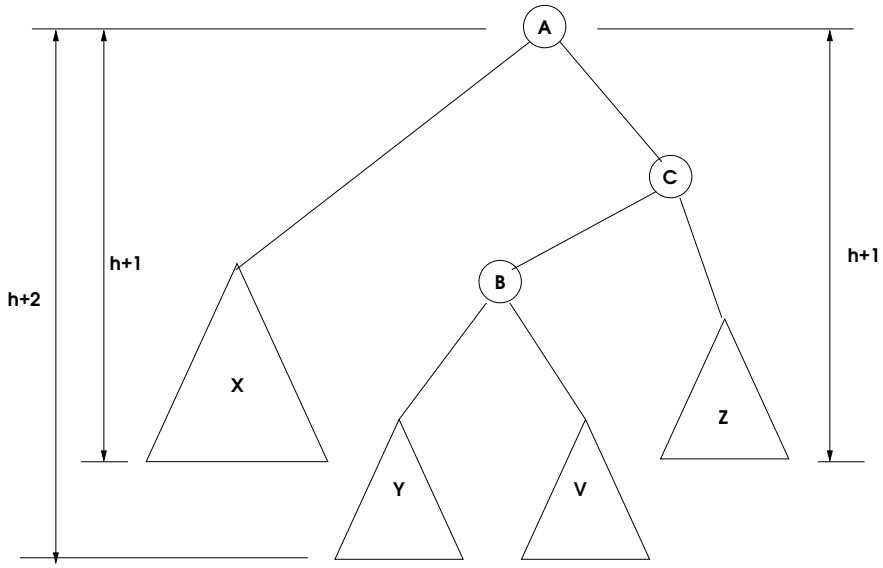

Figure 10: Case 2 AVL tree after rebalancing

depicted in Figure 7 This implies that the height of the tree labeled X is h because there are two edges from C to its root.

Case 1: $height((t \rightarrow left) \rightarrow left) > height((t \rightarrow left) \rightarrow right)$

In this case, the left subtree of A is deeper than the right subtree. Since A is an AVL tree, the height from the root at C to the bottom of one or both of Y or V is exactly $h+1$. (It cannot be h or less because the tree rooted at A would be out of balance. It cannot be more because of the assumption stated in Case 1.) It is possible that either Y or V is one level taller than the other. We can conclude that the height of Y and/or V is $h-1$, and the others height is either $h-1$ or $h-2$. For the sake of simplicity, assume that both are the same height. It makes no difference. Recapping, $height(X) = h$, $height(Z) = h-1$, $height(Y) = height(V) = h-1$.

Figure 8 shows the tree after the LL rotation at the root. The tree now has height $h+1$. It is easily verified by counting the edges from C to the bottoms of all subtrees and considering their heights that the bottoms of X and Z are at the same height, and one or both of Y and V have their bottom nodes at this height as well. This proves that the LL rotation rebalances the tree.

Case 2: $height((t \rightarrow left) \rightarrow left) = height((t \rightarrow left) \rightarrow right)$

If the heights of the left and right subtrees of A are equal, then it means that either Y or V has nodes at depth $h+2$ in the tree, or possibly both. After the LL rotation, the tree will be as shown below. The height of the right subtree of A will still be $h+2$ relative to the pointer t, and the tree remains an AVL tree. This case is illustrated in Figure 9.

After the LL rotation, the tree will be as shown in Figure 10. The height of the right subtree of A will still be $h+2$ relative to the pointer t, and the tree remains an AVL tree. By counting the edges from the new root A to the bottoms of each of the trees X, Y, V, and Z, and accounting for the heights of these trees, you can verify that each of the trees root at A, B, and C is an AVL tree.

Cases 1 and 2 combined establish that the LL rotation will rebalance the tree if the height of the right subtree of the left child is not greater than that of the left subtree of the left child. To prove that the LR rotation will fail if this condition is true, consider the tree in Case 2, but with

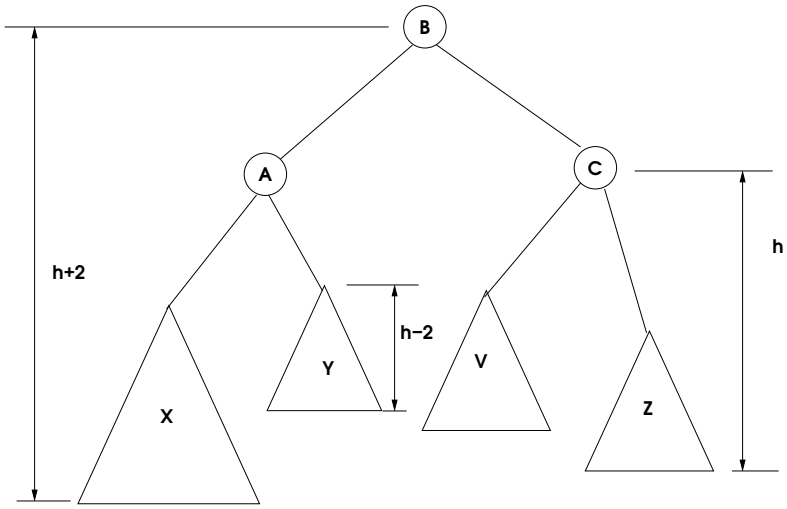

Figure 11: Case 2 but with LR rotation instead of LL rotation.

$height(Y) = height(V) - 1$. Since one of Y or V is the same height as X relative to C, it must be V. Since there are three edges from C to both Y and V, and only two to X, and since the bottom of X is at depth $h+2$ (because of the imbalance at C), we must have that $height(X) = h$, $height(V) = h-1$, and $height(Y) = h-2$. Also, $height(Z) = h-1$. We apply the LR rotation and show the resulting tree in Figure 11. You can see that the height difference between the left and right subtrees X and Y of A is now 2. This shows that an LR fails and that an LL rotation must be applied in this case. An LL must also be applied in Case 1 for the same reason, because the Case 1 tree will have both Y and V of height $h-2$.

Case 3: $height((t \to left) \to left) < height((t \to left) \to right)$

Because the left subtree of A is shorter than its right subtree, but the left subtree of C is of height $h+2$ relative to C, the bottom of X must be at height $h+1$. It could not be less, otherwise A would have been out of balance before the deletion in the right subtree of C. It also implies that the depth of the bottommost nodes of one or both of Y or V is $h+2$, as shown in Figure 12.

After an LR rotation, the tree will be as shown in Figure 13. The bottoms of X and V are at $h+1$, and one or both of Y and V is at $h+1$, and if one of them is not, then it is at height h, since node B was balanced prior to the rotation, implying that the shorter one can be at most one level shorter than the other. All of the heights can be verified, as before, by counting edges from the nodes to the various trees. Thus this rotation leaves the tree balanced.

I leave it as an exercise for you to show that if an LL rotation were tried in this case, the tree would be unbalanced afterward. □

Corollary 3. *Suppose that C is the root of an AVL tree in whose left subtree a deletion has occurred, and all subtrees of C are balanced. Suppose that t is a pointer to C and that $height(t \to right)$-$height(t \to left) = 2$. Then an RR rotation at t will rebalance the tree if and only if $height((t \to right) \to right) \geq height((t \to right) -> left)$ and an RL rotation will rebalance the tree otherwise.*

Proof. The argument is symmetric to the one for Lemma 2. □

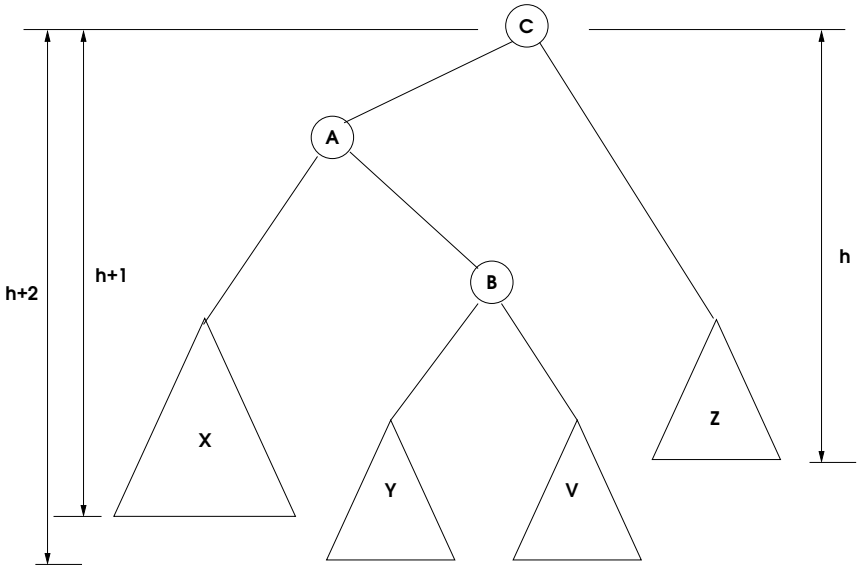

Figure 12: Case 3 AVL tree before LR rotation.

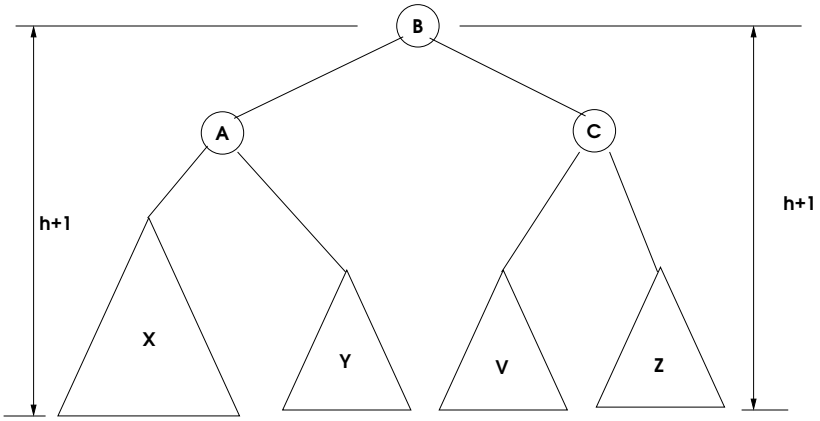

Figure 13: Case 3 AVL tree after LR rotation.

1.3.1 The Deletion Algorithm

The deletion algorithm based on the preceding lemma and its symmetric corollary is described by the listing below. It assumes that `height()` is a function that gets the height of the tree pointed to by its argument. The heights are assumed to be stored in the nodes. This is a version with the rotations called directly. As with insertion, it is possible to modify the BST `remove()` method by including a call to balance the tree at the end.

```
remove( const Comparable & x, AvlNode<Comparable> * & t )
{
    if ( t == NULL )
        // can't delete from an empty tree
        return;
    if ( x < t->element ) {
        // delete from the left subtree
```

```
        remove( x, t->left );
        // check if the heights of the subtrees are now too different
        if ( height( t->right ) - height( t->left ) == 2 ) // unbalanced
            // right subtree too tall relative to left
            // Which rotation to use depends on whether the left subtree
               of the
            // right subtree is larger, or the right of the right is
               larger.
            // If the right is larger we MUST use RR
            if ( height((t->right)->right) >= height((t->right)->left) )
                RR_rotation( t );
            else
                RL_rotation( t );
    }
    else if ( t->element < x ) {
        // delete from the right subtree
        remove( x, t->right );
        if ( height( t->left ) - height( t->right ) == 2 ) // unbalanced
            // left subtree too tall
            if ( height((t->left)->left) >= height((t->left)->right) )
                LL_rotation( t );
            else
                LR_rotation( t );
    }
    else { // delete this node
        if ((t->left != NULL) && (t->right != NULL) ){ // two non-empty
           subtrees
            t->element = findMin(t->right)->element;
            remove(t->element, t->right);
            if ( height( t->left ) - height( t->right ) == 2 ) //
               unbalanced
                // left subtree too tall
                if ( height((t->left)->left) >= height((t->left)->right) )
                    LL_rotation( t );
                else
                    LR_rotation( t );
        }
        else {
            AVLNode<Comparable>* OldNode = t;
            t = (t->left != NULL)? t->left : t->right;
            delete OldNode;
        }
    }
    if ( NULL != t )
        t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

1.4 Worst case AVL Trees (Fibonacci Trees)

Do AVL trees have $O(logN)$ depth in the worst case? To answer this question, we can try to determine the tallest possible AVL trees with a fixed number of nodes. It turns out to be easier to do the converse, namely given an AVL tree of height h, to determine what is the least number of

nodes in it. These ***minimal*** AVL trees are worst case trees, because they contain the fewest nodes possible for their height.

Let $S(h)$ be the number of nodes in a minimal AVL tree of height h. This means that if T is a minimal AVL tree containing n nodes and $n < S(h)$, its height must be less than h. If T is a minimal AVL tree of height h, then its left and right subtrees must also be minimal AVL trees. Furthermore, their heights differ by at most 1. This means that one is of height $h-1$ and the other is of height $h-2$, because the root adds 1 to the height of each. This leads to the recurrence relation:

$S(0) = 1$

$S(1) = 2$

$S(h) = S(h-1) + S(h-2) + 1$

Notice that the recurrence looks similar to the Fibonacci number recurrence relation. If you write out the first few terms you get 1, 2, 4, 7, 12, 20, 33, 54. Let $F(k)$ denote a function of k that returns the k^{th} Fibonacci number. Assume that $F(0) = 0$ and $F(1) = 1$. It is easy to prove by induction that $S(h) = F(h+3)$–1. The Fibonacci recurrence is defined as

$$F(n) = F(n-1) + F(n-2)$$

If we write instead f_k as the k^{th} Fibonacci number, then you can see that the Fibonacci recurrence is of the form

$$f_n = a_1 f_{n-1} + a_2 f_{n-2} + \cdots + a_k f_{n-k} \tag{1}$$

in which $a_1 = a_2 = 1$ and $k = 2$:

$$f_n = f_{n-1} + f_{n-2} \tag{2}$$

which can be rewritten as

$$f_n - f_{n-1} - f_{n-2} = 0 \tag{3}$$

A recurrence in this form is called a ***homogeneous linear recurrence*** with ***constant coefficients***. All such recurrences have solutions of the form

$$F_n = cr^n$$

where c and r are constants. Therefore, the Fibonacci equation is

$$cr^n - cr^{n-1} - cr^{n-2} = 0 \tag{4}$$

which is equivalent to

$$cr^{n-2}(r^2 - r^1 - 1) = 0 \tag{5}$$

As a polynomial in r, this has two non-zero solutions for r, which we can denote ϕ_1 and ϕ_2. Their actual values are $\phi_1 = (1+\sqrt{5})/2$ and $\phi_2 = (1-\sqrt{5})/2$. But in Eq. 3 the solutions are functions, not numbers, that satisfy that equation and equation 5 as well.

Since $\phi_1(n)$ and $\phi_2(n)$ are both solutions to Eq. 5, Eq. 3 can be written as

$$\phi_1(n) = \phi_1(n-1) + \phi_1(n-2)$$

and

$$\phi_2(n) = \phi_2(n-1) + \phi_2(n-2)$$

which means, adding these together, that the function $\phi_1 + \phi_2$ is also a solution, since

$$\begin{aligned}(\phi_1 + \phi_2)(n) = \phi_1(n) + \phi_2(n) &= \phi_1(n-1) + \phi_1(n-2) + \phi_2(n-1) + \phi_2(n-2) \\ &= \phi_1(n-1) + \phi_2(n-1) + \phi_1(n-2) + \phi_2(n-2) \\ &= (\phi_1 + \phi_2)(n-1) + (\phi_1 + \phi_2)(n-2)\end{aligned}$$

Similarly, any linear combination $a\phi_1 + b\phi_2$ of them is a solution, since

$$\begin{aligned}(a\phi_1 + b\phi_2)(n) = a\phi_1(n) + b\phi_2(n) &= a\phi_1(n-1) + a\phi_1(n-2) + b\phi_2(n-1) + b\phi_2(n-2) \\ &= a\phi_1(n-1) + b\phi_2(n-1) + a\phi_1(n-2) + b\phi_2(n-2) \\ &= (a\phi_1 + b\phi_2)(n-1) + (a\phi_1 + b\phi_2)(n-2)\end{aligned}$$

This implies that there are infinitely many solutions to the recurrence relation if they do not have to also satisfy the base case, or initial conditions, and that all solutions are of the form $a\phi_1 + b\phi_2$. To find the exact solution to our particular recurrence relation, we solve for the constants a and b in the two simultaneous equations of the base case:

$$\begin{aligned}F(0) = a\phi_1(0) + b\phi_2(0) = a\phi_1^0 + b\phi_2^0 &= 0 \\ F(1) = a\phi_1(1) + b\phi_2(1) = a\phi_1^1 + b\phi_2^1 &= 1\end{aligned}$$

which becomes

$$\begin{aligned}a + b &= 0 \\ a\phi_1 + b\phi_2 &= 1\end{aligned}$$

From the first, $b = -a$. Substituting $b = -a$ in the second equation, we get

$$a(\phi_1 - \phi_2) = 1$$

from which we conclude that $a = -b = 1/(\phi_1 - \phi_2)$. You can work out that $\phi_1 - \phi_2 = \sqrt{5}$ from which we get the unique solution

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

which is the closed form formula for the n^{th} Fibonacci number. Since $S(h) = F(h+3) - 1$

$$S(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+3} - 1$$

Since $(1-\sqrt{5})/2 \approx -0.618 < 1$, as h increases, the second term approaches 0 and we can approximate $S(h)$ by

$$S(h) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{3}\left(\frac{1+\sqrt{5}}{2}\right)^{h} \tag{6}$$

This is approximately equal to

$$1.8944 \cdot \left(\frac{1+\sqrt{5}}{2}\right)^{h}$$

Since $S(h)$ is the least number of nodes in an AVL tree of height h, if n is the number of nodes in tree T and $n < S(h)$, T must have height strictly less than h. If $n \geq S(h)$ then T can have height h. For a given tree T with n nodes, let h be the smallest number such that $S(h) \leq n < S(h+1)$. From the preceding reasoning, T has height at most h.

Let us now write this h as a function $H(n)$ of n. This amounts to setting the left hand side of Eq. 6 equal to n and solving for n :

$$n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

When we do this we get that $n \approx 1.8944 \cdot 1.618^{h}$. Taking logs and solving for h in terms of n, $H(n) \approx 1.44 \cdot \log n$. This means that the height of an AVL tree with n nodes is at most roughly $1.44 \cdot \log n$. In other words, the height of an AVL tree is about 45% taller than the height of a perfectly balanced full tree with n nodes, in the worst case.

Summarizing, the importance of these Fibonacci trees are that they act as worst case trees: any tree with fewer than $S(h)$ nodes has height less than h. It follows that AVL trees have $O(\log n)$ height in the worst case.