Lecture 17 – self-balancing trees 1

If the input comes into a binary search tree (BST) *presorted*, then a series of inserts will take quadratic time and give a very expensive implementation of a linked list, since the tree will consist only of nodes with no left children. One solution to the problem is to insist on an extra structural condition called *balance*: No node is allowed to get too deep. There are quite a few general algorithms to implement balanced trees. Most are quite a bit more complicated than a standard binary search tree, and all take longer on average for updates. They do, however, provide protection against the embarrassingly simple cases. Below, we will sketch one of the oldest forms of balanced search trees, the AVL tree.
A second method is to forgo the balance condition and allow the tree to be arbitrarily deep, but after every operation, a restructuring rule is applied that tends to make future operations efficient. These types of data structures are generally classified as self-adjusting. In the case of a binary search tree, we can no longer guarantee an $O(\log N)$ bound on any single operation but can show that any sequence of M operations takes total time $O(M \log N)$ in the worst case. This is generally sufficient protection against a bad worst case. The data structure we will discuss is known as a splay tree; its analysis is fairly intricate and is discussed after AVL trees, below.

AVL trees

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log N)$. The simplest idea is to require that the left and right subtrees have the same height. As Figure 4.31 shows, this idea does not force the tree to be shallow:

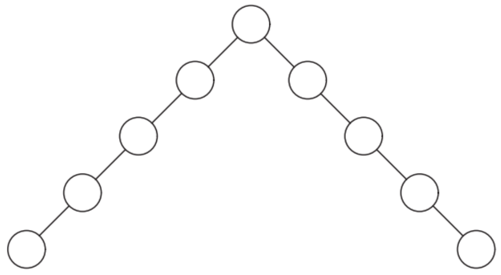


Figure 4.31 A bad binary tree. Requiring balance at the root is not enough.

Another balance condition would insist that every node must have left and right subtrees of the same height. If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of $2^k - 1$ nodes would satisfy this criterion. Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.
An **AVL tree** is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be −1.) In Figure 4.32 the tree on the left is an AVL tree but the tree on the right is not:

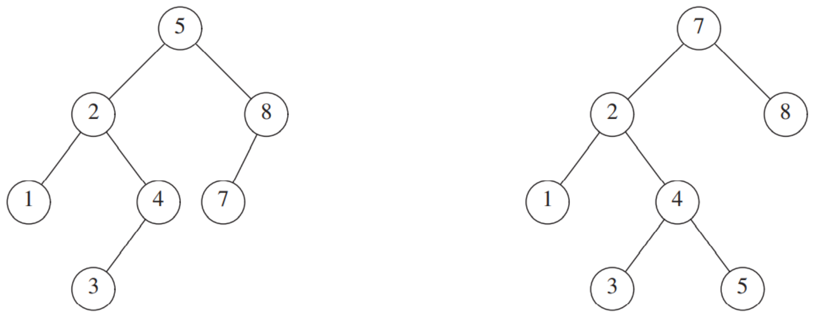


Figure 4.32 Two binary search trees. Only the left tree is AVL.

Height information is kept for each node (in the node structure). It can be shown that the height of an AVL tree is at most roughly $1.44 \log(N + 2) - 1.328$, but in practice it is only slightly more than $\log N$. As an example, the AVL tree of height 9 with the fewest nodes (143) is shown in Figure 4.33:

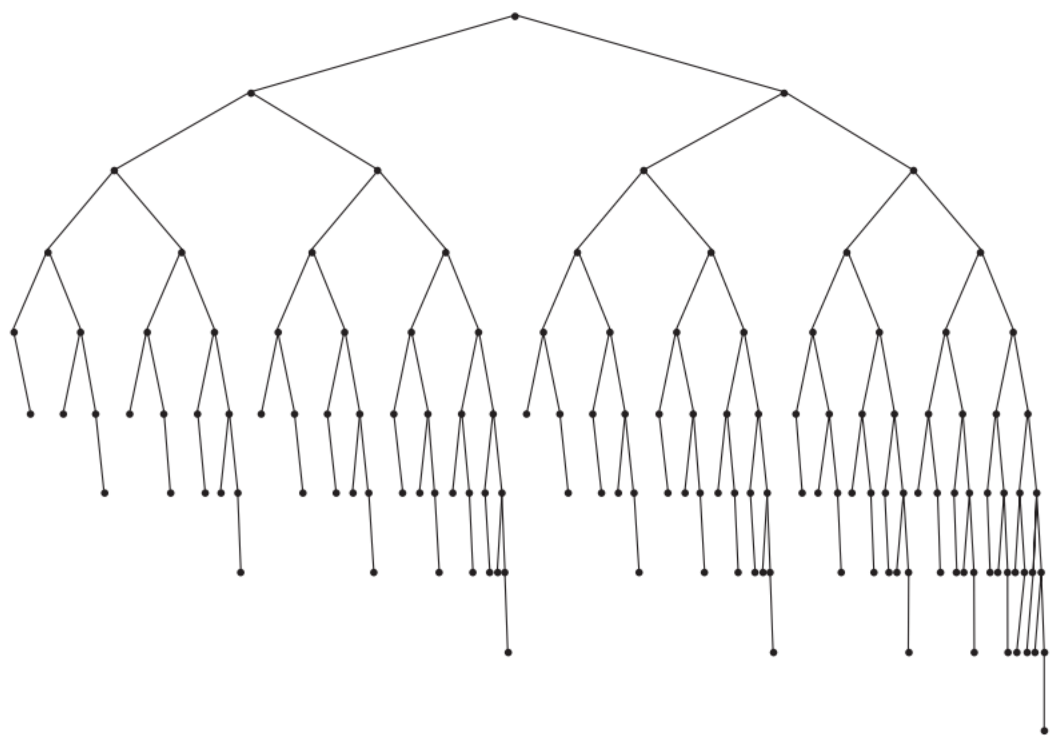


Figure 4.33 Smallest AVL tree of height 9

This tree has as a left subtree an AVL tree of height 7 of minimum size. The right subtree is an AVL tree of height 8 of minimum size. This tells us that the minimum number of nodes, $S(h)$, in an AVL tree of height h is given by $S(h) = S(h-1) + S(h-2) + 1$. For $h = 0$, $S(h) = 1$. For $h = 1, S(h) = 2$. The function $S(h)$ is closely related to the Fibonacci numbers, from which the bound claimed above on the height of an AVL tree follows.

Thus, all the tree operations can be performed in $O(\log N)$ time, except possibly insertion and deletion. When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6 into the AVL tree in Figure 4.32 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a **rotation**.

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition. We will show how to rebalance the tree at the first (i.e., deepest) such node, and we will prove that this rebalancing guarantees that the entire tree satisfies the AVL property.

Let us call the node that must be rebalanced α. Since any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

1) An insertion into the left subtree of the left child of α
2) An insertion into the right subtree of the left child of α
3) An insertion into the left subtree of the right child of α
4) An insertion into the right subtree of the right child of α

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 3. Consequently, as a matter of theory, there are two basic cases. From a programming perspective, of course, there are still four cases.

The first case, in which the insertion occurs on the "outside" (i.e., left–left or right– right), is fixed by a **single rotation** of the tree. The second case, in which the insertion occurs on the "inside" (i.e., left–right or right–left) is handled by the slightly more complex **double rotation**. These are fundamental operations on the tree that we'll see used several times in balanced-tree algorithms. The remainder of this section describes these rotations, proves that they suffice to maintain balance, and gives a casual implementation of the AVL tree. Later, we describe other balanced-tree methods with an eye toward a more careful implementation.

Single rotation

Figure 4.34 shows the single rotation that fixes case 1: (An insertion into the left subtree of the left child of α)

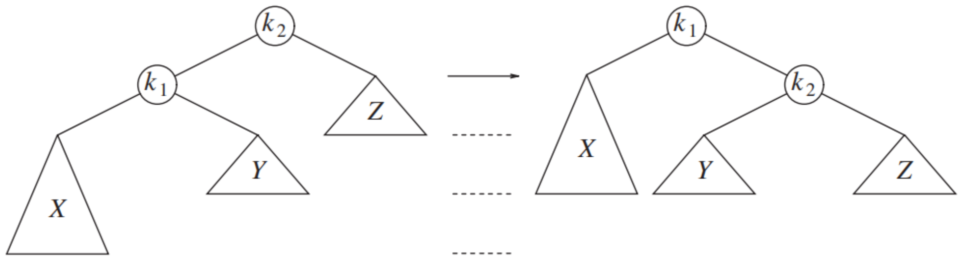


Figure 4.34 Single rotation to fix case 1

The before picture is on the left and the after is on the right. Let us analyze carefully what is going on. Node k_2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines in the middle of the diagram mark the levels). The situation depicted is the only possible case 1 scenario that allows k_2 to satisfy the AVL property before an insertion but violate it afterwards. Subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. Y cannot be at the same level as the new X because then k_2 would have been out of balance before the insertion, and Y cannot be at the same level as Z because then k_1 would be the first node on the path toward the root that was in violation of the AVL balancing condition.
To ideally rebalance the tree, we would like to move X up a level and Z down a level. Note that this is actually more than the AVL property would require. To do this, we rearrange nodes into an equivalent tree as shown in the second part of Figure 4.34. Here is an abstract scenario: Visualize the tree as being flexible, grab the child node k_1, close your eyes, and shake it, letting gravity take hold. The result is that k_1 will be the new root. The binary search tree property tells us that in the original tree $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. X and Z remain as the left child of k_1 and right child of k_2, respectively. Subtree Y, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.
As a result of this work, which requires only a few pointer changes, we have another binary search tree that is an AVL tree. This happens because X moves up one level, Y stays at the same level, and Z moves down one level. k_2 and k_1 not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height. Furthermore, the new height of the entire subtree is exactly the same as the height of the original subtree prior to the insertion that caused X to grow. Thus no further updating of heights on the path to the root is needed, and consequently *no further rotations are needed*. Figure 4.35 shows that after the insertion of 6 into the original AVL tree on the left, node 8 becomes unbalanced:

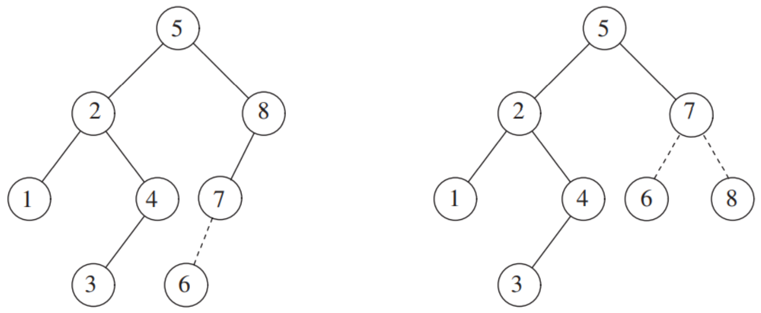


Figure 4.35 AVL property destroyed by insertion of 6, then fixed by a single rotation

Thus, we do a single rotation between 7 and 8, obtaining the tree on the right. As we mentioned earlier, case 4 represents a symmetric case. Figure 4.36 shows how a single rotation is applied:

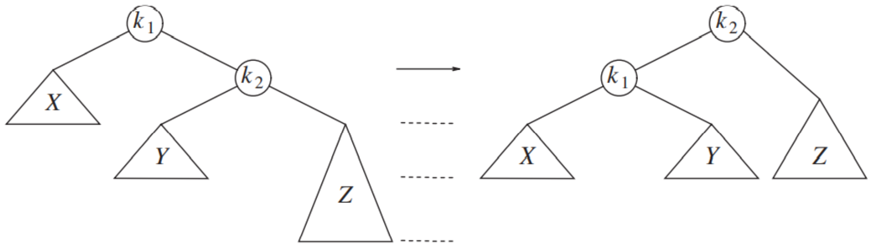
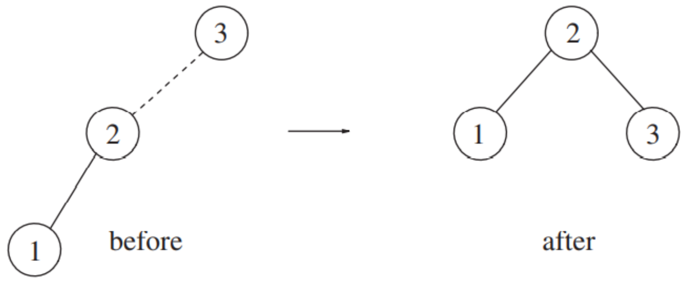
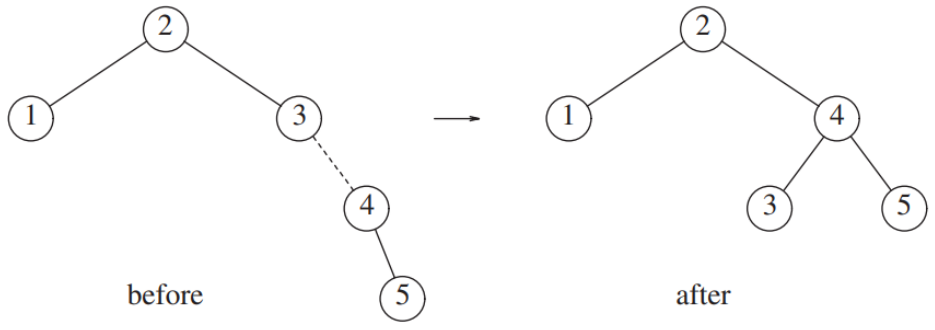


Figure 4.36 Single rotation fixes case 4

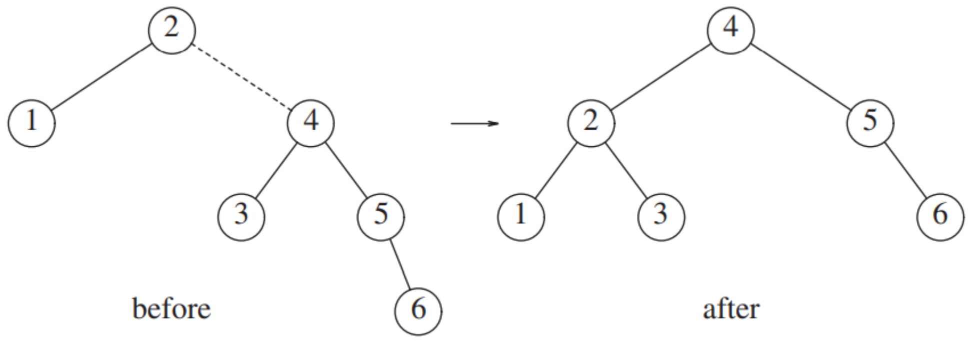
Let us work through a rather long example. Suppose we start with an initially empty AVL tree and insert the items 3, 2, 1, and then 4 through 7 in sequential order. The first problem occurs when it is time to insert item 1 because the AVL property is violated at the root. We perform a single rotation between the root and its left child to fix the problem. Here are the before and after trees:



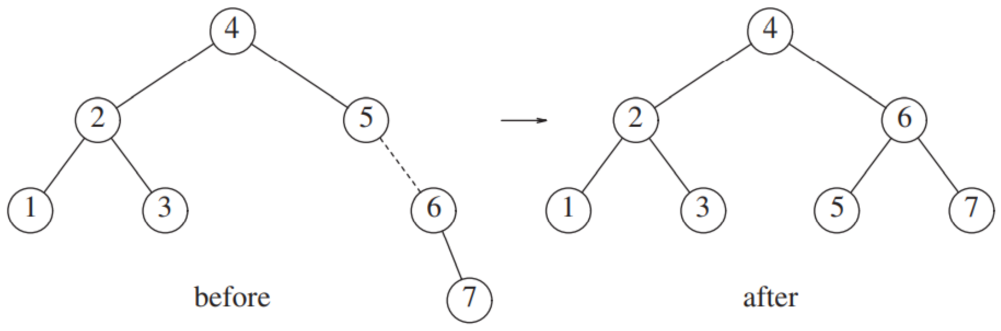
A dashed line joins the two nodes that are the subject of the rotation. Next we insert 4, which causes no problems, but the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. Besides the local change caused by the rotation, the programmer must remember that the rest of the tree has to be informed of this change. Here this means that 2's right child must be reset to link to 4 instead of 3. Forgetting to do so is easy and would destroy the tree (4 would be inaccessible).



Next we insert 6. This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.



The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every item in this subtree must lie between 2 and 4, so this transformation makes sense. The next item we insert is 7, which causes another rotation:



double rotation

The algorithm described above has one problem: As Figure 4.37 shows, it does not work for cases 2 or 3:

Case 2: An insertion into the right subtree of the left child of α

Case 3: An insertion into the left subtree of the right child of α

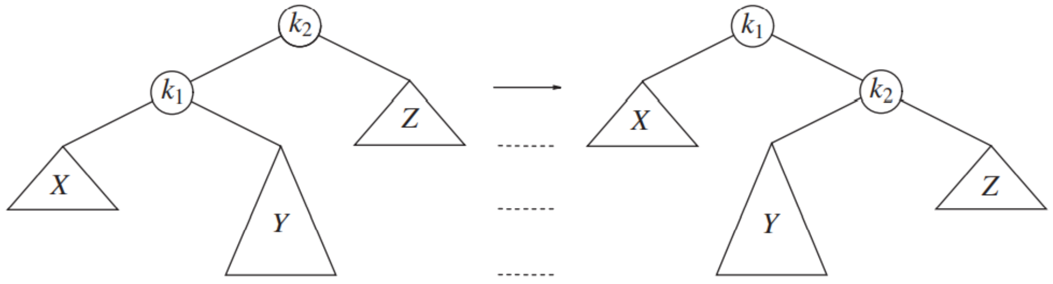


Figure 4.37 Single rotation fails to fix case 2

The problem is that subtree Y is too deep, and a single rotation does not make it any less deep. The double rotation that solves the problem is shown in Figure 4.38:

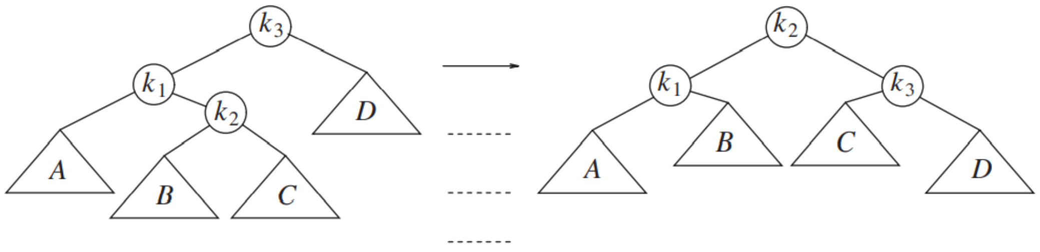


Figure 4.38 Left–right double rotation to fix case 2

The fact that subtree Y in Figure 4.37 has had an item inserted into it guarantees that it is nonempty. Thus, we may assume that it has a root and two subtrees. Consequently, the tree may be viewed as four subtrees connected by three nodes. As the diagram suggests, exactly one of tree B or C is two levels deeper than D (unless all are empty), but we cannot be sure which one. It turns out not to matter; in Figure 4.38, both B and C are drawn at $1\frac{1}{2}$ levels below D.

To rebalance, we see that we cannot leave k_3 as the root, and a rotation between k_3 and k_1 was shown in Figure 4.37 to not work, so the only alternative is to place k_2 as the new root. This forces k_1 to be k_2's left child and k_3 to be its right child, and it also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL tree property, and as was the case with the single rotation, it restores the height to what it was before the insertion, thus guaranteeing that all rebalancing and height updating is complete. Figure 4.39 shows that the symmetric case 3 can also be fixed by a double rotation:

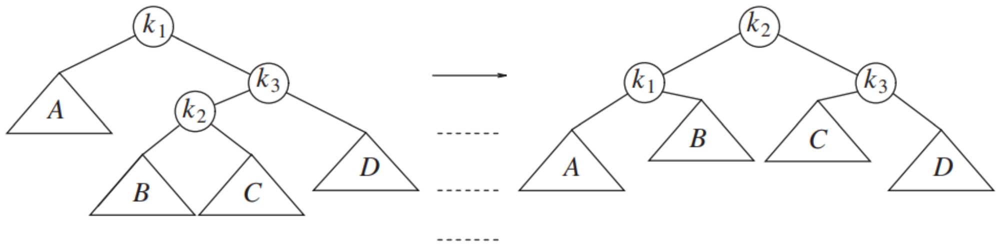
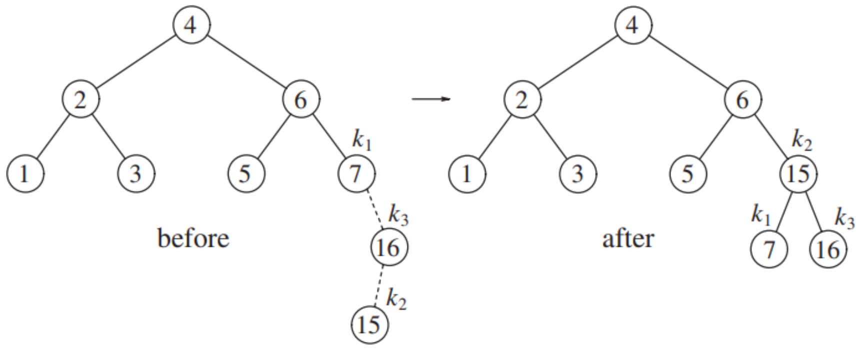
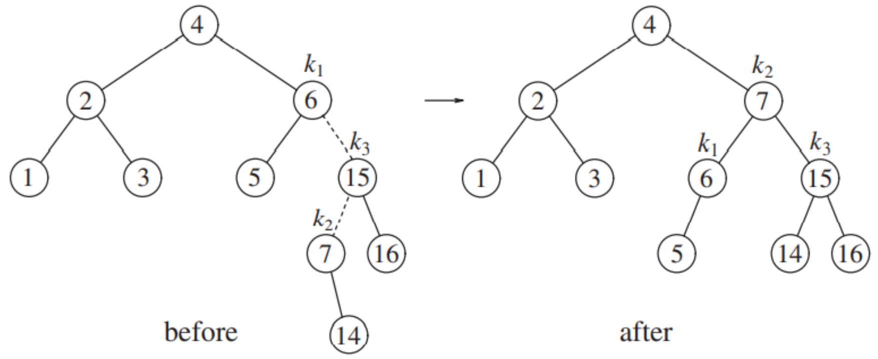


Figure 4.39 Right–left double rotation to fix case 3

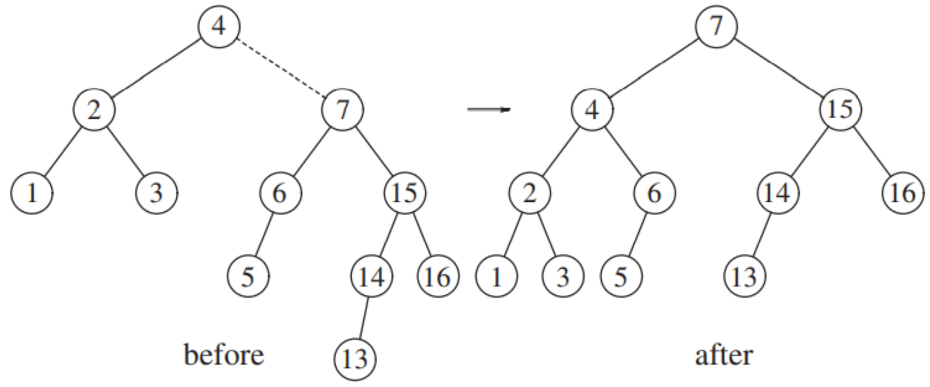
In both cases the effect is the same as rotating between α's child and grandchild, and then between α and its new child. We will continue our previous example by inserting 10 through 16 in reverse order, followed by 8 and then 9. Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7. This is case 3, which is solved by a right–left double rotation. In our example, the right–left double rotation will involve 7, 16, and 15. In this case, k1 is the node with item 7, k3 is the node with item 16, and k2 is the node with item 15. Subtrees A, B, C, and D are empty.



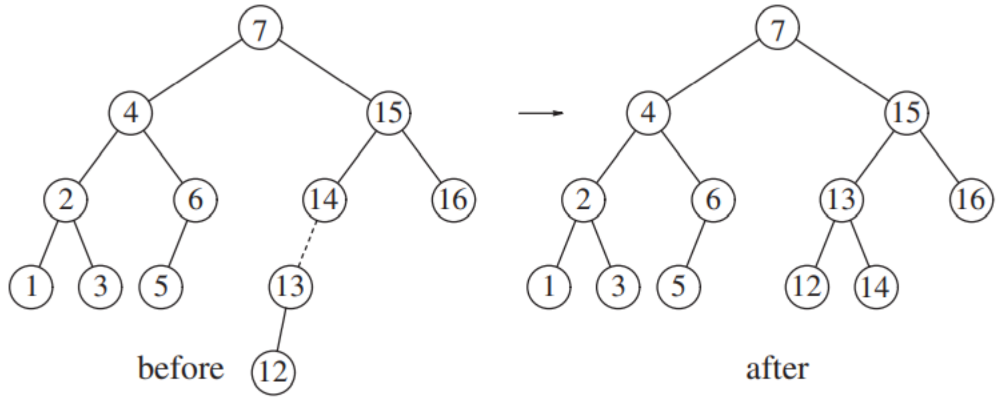
Next we insert 14, which also requires a double rotation. Here the double rotation that will restore the tree is again a right–left double rotation that will involve 6, 15, and 7. In this case, k_1 is the node with item 6, k_2 is the node with item 7, and k_3 is the node with item 15. Subtree A is the tree rooted at the node with item 5; subtree B is the empty subtree that was originally the left child of the node with item 7, subtree C is the tree rooted at the node with item 14, and finally, subtree D is the tree rooted at the node with item 16.



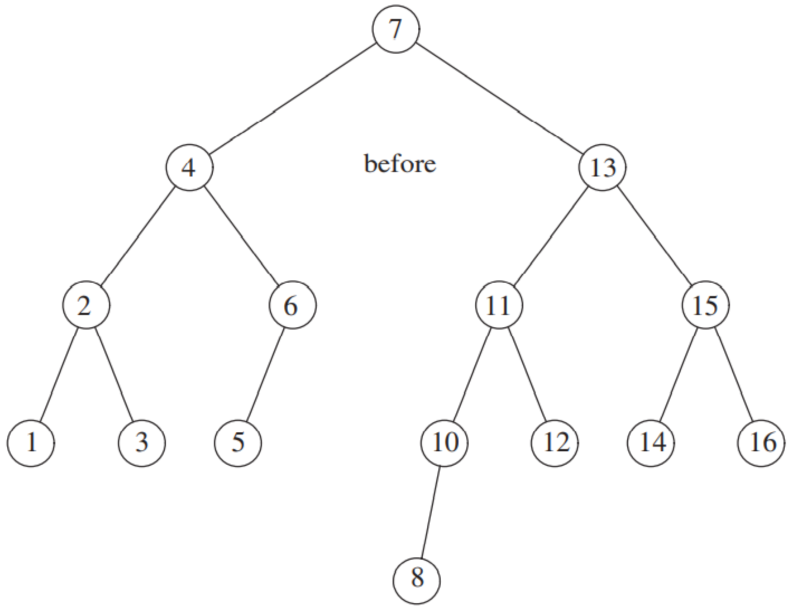
If 13 is now inserted, there is an imbalance at the root. Since 13 is not between 4 and 7, we know that the single rotation will work.



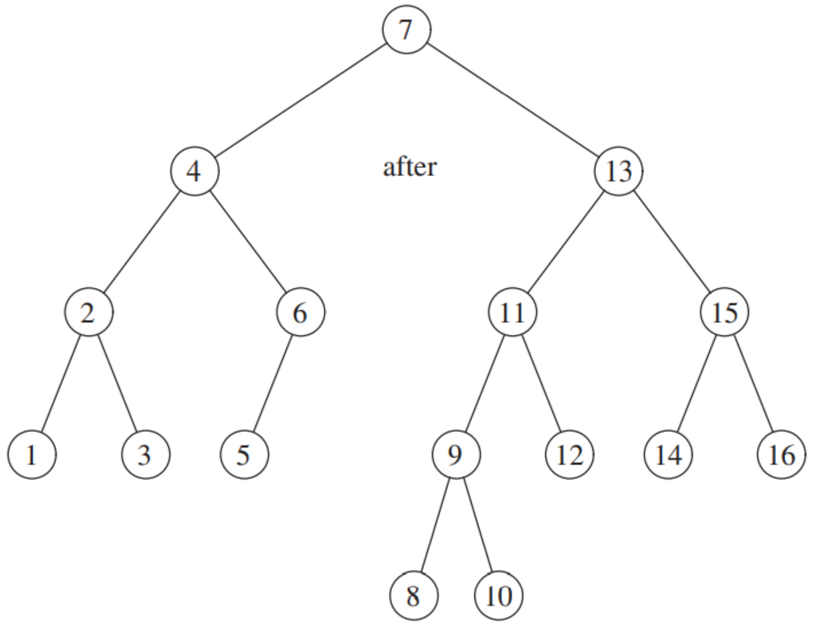
Insertion of 12 will also require a single rotation:



To insert 11, a single rotation needs to be performed, and the same is true for the subsequent insertion of 10. We insert 8 without a rotation, creating an almost perfectly balanced tree:



Finally, we will insert 9 to show the symmetric case of the double rotation. Notice that 9 causes the node containing 10 to become unbalanced. Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:



Let us summarize what happens. The programming details are fairly straightforward except that there are several cases. To insert a new node with item X into an AVL tree T, we recursively insert X into the appropriate subtree of T (let us call this T_{LR}). If the height of T_{LR} does not change, then we are done. Otherwise, if a height imbalance appears in T, we do the appropriate single or double rotation depending on X and the items in T and T_{LR}, update the heights (making the connection from the rest of the tree above), and we are done. Since one rotation always suffices, a carefully coded nonrecursive version generally turns out to be faster than the recursive version, but on modern compilers the difference is not as significant as in the past. However, nonrecursive versions are quite difficult to code correctly, whereas a casual recursive implementation is easily readable.
Another efficiency issue concerns storage of the height information. Since all that is really required is the difference in height, which is guaranteed to be small, we could get by with two bits (to represent +1, 0, −1) if we really try. Doing so will avoid repetitive calculation of balance factors but results in some loss of clarity. The resulting code is somewhat

more complicated than if the height were stored at each node. If a recursive routine is written, then speed is probably not the main consideration. In this case, the slight speed advantage obtained by storing balance factors hardly seems worth the loss of clarity and relative simplicity. Furthermore, since most machines will align this to at least an 8-bit boundary anyway, there is not likely to be any difference in the amount of space used. An 8-bit (signed) char will allow us to store absolute heights of up to 127. Since the tree is balanced, it is inconceivable that this would be insufficient.

With all this, we are ready to write the AVL routines. First, we need the AvlNode class. This is given in Figure 4.40:

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ ele }, left{ lt }, right{ rt }, height{ h } { }

    AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
};
```

Figure 4.40 Node declaration for AVL trees

We also need a quick function to return the height of a node. This function is necessary to handle the annoying case of a nullptr pointer. This is shown in Figure 4.41:

```
/**
 * Return the height of node t or -1 if nullptr.
 */
int height( AvlNode *t ) const
{
    return t == nullptr ? -1 : t->height;
}
```

Figure 4.41 Function to compute height of an AVL node

The basic insertion routine (see Figure 4.42) adds only a single line at the end that invokes a balancing method.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
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 );
}

static const int ALLOWED_IMBALANCE = 1;

// 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 ) )
            rotateWithLeftChild( t );
        else
            doubleWithLeftChild( t );
    else
    if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
        if( height( t->right->right ) >= height( t->right->left ) )
            rotateWithRightChild( t );
        else
            doubleWithRightChild( t );

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

Figure 4.42 Insertion into an AVL tree

The balancing method applies a single or double rotation if needed, updates the height, and returns the resulting tree. For the trees in Figure 4.43, `rotateWithLeftChild` converts the tree on the left to the tree on the right, returning a pointer to the new root. `rotateWithRightChild` is symmetric.

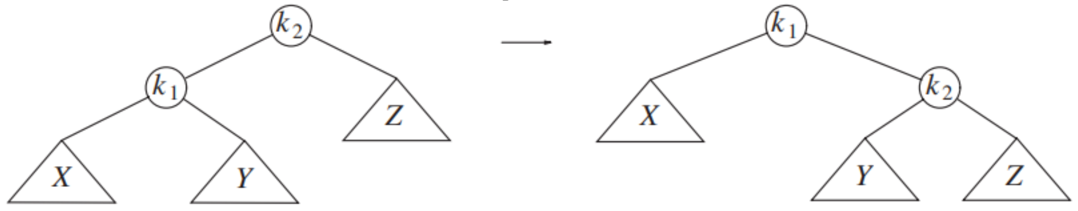


Figure 4.43 Single rotation

The code is shown in Figure 4.44:

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( 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;
}
```

Figure 4.44 Routine to perform single rotation

Similarly, the double rotation pictured in Figure 4.45 can be implemented by the code shown in Figure 4.46.

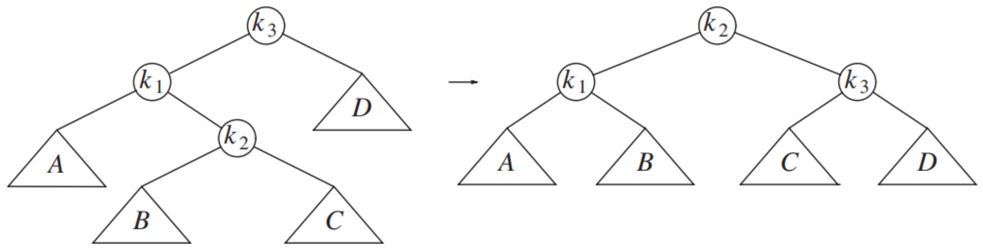


Figure 4.45 Double rotation

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Figure 4.46 Routine to perform double rotation

Since deletion in a binary search tree is somewhat more complicated than insertion, one can assume that deletion in an AVL tree is also more complicated. In a perfect world, one would hope that the deletion routine in Figure 4.26 could easily be modified by changing the last line to return after calling the balance method, as was done for insertion. This would yield the code in Figure 4.47. This change works! A deletion could cause one side of the tree to become two levels shallower than the other side. The case-by-case analysis is similar to the imbalances that are caused by insertion, but not exactly the same. For instance, case 1 in Figure 4.34, which would now reflect a deletion from tree Z (rather than an insertion into X), must be augmented with the possibility that tree Y could be as deep as tree X. Even so, it is easy to see that the rotation rebalances this case and the symmetric case 4 in Figure 4.36. Thus the code for balance in Figure 4.42 lines 28 and 34 uses >= instead of > specifically to ensure that single rotations are done in these cases rather than double rotations. We leave verification of the remaining cases as an exercise.

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Figure 4.26 Deletion routine for binary search trees

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, AvlNode * & t )
{
   if( t == nullptr )
       return;   // Item not found; do nothing

   if( x < t->element )
       remove( x, t->left );
   else if( t->element < x )
       remove( x, t->right );
   else if( t->left != nullptr && t->right != nullptr )  // Two children
   {
       t->element = findMin( t->right  )->element;
       remove( t->element, t->right );
   }
   else
   {
       AvlNode *oldNode = t;
       t = ( t->left != nullptr ) ? t->left : t->right;
       delete oldNode;
   }

    balance( t );
}
```

Figure 4.47 Deletion in an AVL tree