Brad Appleton
Software Tools Developer
E-mail: brad@bradapp.net
WWW: http://www.bradapp.net

AVL Trees: Tutorial and C++ Implementation

The following discussion is part of a freely available public domain AVL tree library written in C++. The full C++ source code distribution may be found in `AvlTrees.tar.gz` (21KB, gzipped tar file). [a plain old K&R C version is available in `libavl.tar.gz` (25KB, gzipped tar file)]

AVL Trees

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. Or to be more formal, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

An AVL tree is a binary tree in which the difference between the height of the right and left subtrees (or the root node) is never more than one.

The idea behind maintaining the "AVL-ness" of an AVL tree is that whenever we insert or delete an item, if we have "violated" the "AVL-ness" of the tree in anyway, we must then restore it by performing a set of manipulations (called "rotations") on the tree. These rotations come in two flavors: single rotations and double rotations (and each flavor has its corresponding "left" and "right" versions).

An example of a single rotation is as follows: Suppose I have a tree that looks like this:

```
    c
   /
  b
```

Now I insert the item "a" and get the resulting binary tree:

```
    c
```

```
      /
     b
    /
   a
```

Now, this resulting tree violates the "AVL criteria", the left subtree has a height of 2 but the right subtree has a height of 0 so the difference in the two heights is "2" (which is greater than 1). SO what we do is perform a "single rotation" (or RR for a single right rotation, or LL for a single left rotation) on the tree (by rotating the "c" element down clockwise to the right) to transform it into the following tree:

```
     b
    / \
   a   c
```

This tree is now balanced.

An example of a "double rotation" (or RL for a double right rotation, or LR for a double left rotation) is the following: Suppose I have a tree that looks like this:

```
   a
    \
     c
```

Now I insert the item "b" and get the resulting binary tree:

```
   a
    \
     c
    /
   b
```

This resulting tree also violates the "AVL criteria" so we fix it by first rotating "c" down to the right (so we get "a-b-c"), and then rotating "a" down to the left so that the tree is transformed into this:

```
     b
    / \
   a   c
```

In order to detect when a "violation" of the AVL criteria occurs we need to have each node keep track of the difference in height between its right and left subtrees. We call this "difference" the "balance" factor and define it to be the height of the right subtree minus the height of the left subtree of a tree. So as long as the "balance" factor of each node is never >1 or <-1 we have an AVL tree. As soon as the balance factor of a node becomes 2 (or -2) we need to perform one or more rotations to ensure that the resultant tree satisfies the AVL criteria.

Implementing AVL Trees in C++

Before we begin our AVL tree implementation in C++, lets assume we have a template class named "Comparable" defined as follows:

```
// cmp_t is an enumeration type indicating the result of a
// comparison.
enum  cmp_t {
   MIN_CMP = -1,   // less than
   EQ_CMP  = 0,    // equal to
```

```
      MAX_CMP = 1     // greater than
   };

   // Class "Comparable" corresponds to an arbitrary comparable element
   // with a keyfield that has an ordering relation. The template parameter
   // KeyType is the "type" of the keyfield
   //
   template <class KeyType>
   class Comparable {
   private:
      KeyType  myKey;

   public:
      Comparable(KeyType  key) : myKey(key) {};

      // Use default copy-ctor, assignment, & destructor

         // Compare this item against the given key & return the result
      cmp_t Compare(KeyType  key) const;

        // Get the key-field of an item
      KeyType Key() const { return  myKey; }
   };
```

Like the "Comparable" class, our AVL tree will also be a template class parameterized by a KeyType:

```
   // Class AvlNode represents a node in an AVL tree. The template parameter
   // KeyType is the "type" of the keyfield
   //
   template <class KeyType>
   class AvlNode {
   private:
      Comparable<KeyType> * myData;         // Data field
      AvlNode<KeyType>    * mySubtree[2];   // Subtree pointers
      short                 myBal;          // Balance factor

      // ... many details omitted
   };
```

Calculating New Balances After a Rotation

To calculate the new balances after a single left rotation; assume we have the following case:

```
              A                                     B
             / \                                   / \
            /   \                                 /   \
           a     B           ==>                 A     c
                / \                             / \
               /   \                           /   \
              b     c                         a     b
```

The left is what the tree looked like BEFORE the rotation and the right is what the tree looks like after the rotation. Capital letters are used to denote single nodes and lowercase letters are used to denote subtrees.

The "balance" of a tree is the height of its right subtree less the height of its left subtree. Therefore, we can calculate the new balances of "A" and "B" as follows (*ht* is the height function):

```
NewBal(A) = ht(b) - ht(a)

OldBal(A) = ht(B) - ht(a)
```

```
        = ( 1 + max (ht(b), ht(c)) ) - ht(a)
```

subtracting the second equation from the first yields:

```
NewBal(A) - OldBal(A) = ht(b) - ( 1 + max (ht(b), ht(c)) )
                        + ht(a) - ht(a)
```

canceling out the ht(a) terms and adding OldBal(A) to both sides yields:

```
NewBal(A) = OldBal(A) - 1 - (max (ht(b), ht(c)) - ht(b) )
```

Noting that max(x, y) - z = max(x-z, y-z), we get:

```
NewBal(A) = OldBal(A) - 1 - (max (ht(b) - ht(b), ht(c) - ht(b)) )
```

But ht(c) - ht(b) is OldBal(B) so we get:

```
NewBal(A) = OldBal(A) - 1 - (max (0, OldBal(B)) )
          = OldBal(A) - 1 -  max (0, OldBal(B))
```

Thus, for A, we get the equation:

```
NewBal(A) = OldBal(A) - 1 - max (0, OldBal(B))
```

To calculate the Balance for B we perform a similar computation:

```
NewBal(B) = ht(c) - ht(A)
          = ht(c) - (1 + max(ht(a), ht(b)) )

OldBal(B) = ht(c) - ht(b)
```

subtracting the second equation from the first yields:

```
NewBal(B) - OldBal(B) = ht(c) - ht(c)
                        + ht(b) - (1 + max(ht(a), ht(b)) )
```

canceling, and adding OldBal(B) to both sides gives:

```
NewBal(B) = OldBal(B) - 1 - (max(ht(a), ht(b)) - ht(b))
          = OldBal(B) - 1 - (max(ht(a) - ht(b), ht(b) - ht(b))
```

But ht(a) - ht(b) is - (ht(b) - ht(a)) = -NewBal(A), so ...

```
NewBal(B) = OldBal(B) - 1 - max( -NewBal(A), 0)
```

Using the fact that min(x,y) = -max(-x, -y) we get:

```
NewBal(B) = OldBal(B) - 1 + min( NewBal(A), 0)
```

So, for a single left rotation we have shown the the new balances for the nodes A and B are given by the following equations:

```
NewBal(A) = OldBal(A) - 1 - max(OldBal(B), 0)
NewBal(B) = OldBal(B) - 1 + min(NewBal(A), 0)
```

Now let us look at the case of a single right rotation. The case we will use is the same one we used for the single left rotation only with all the left and right subtrees switched around so that we have the mirror image of the case we used for our left rotation.

```
         A                                     B
```

```
          / \                              / \
         /   \                            /   \
        B     a          ==>             c     A
       / \                                    / \
      /   \                                  /   \
     c     b                                b     a
```

If we perform the same calculations that we made for the left rotation, we will see that the new balances for a single right rotation are given by the following equations:

```
NewBal(A) = OldBal(A) + 1 - min(OldBal(B), 0)
NewBal(B) = OldBal(B) + 1 + max(NewBal(A), 0)
```

Hence, C++ code for single left and right rotations would be:

```
// Indices into a subtree array
enum  dir_t { LEFT = 0, RIGHT = 1 };

   // Return the minumum of two numbers
inline int
MIN(int a, int b) { return  (a < b) ? a : b; }

   // Return the maximum of two numbers
inline int
MAX(int a, int b) { return  (a > b) ? a : b; }

   // Note that RotateLeft and RotateRight are *static* member
   // functions because otherwise they would have to re-assign
   // to the "this" pointer.

template <class KeyType>
void
AvlNode<KeyType>::RotateLeft(AvlNode<KeyType> * & root) {
  AvlNode<KeyType> * oldRoot = root;

        // perform rotation
  root = root->mySubtree[RIGHT];
  oldRoot->mySubtree[RIGHT] = root->mySubtree[LEFT];
  root->mySubtree[LEFT] = oldRoot;

        // update balances
  oldRoot->myBal -=  (1 + MAX(root->myBal, 0));
  root->myBal    -=  (1 - MIN(oldRoot->myBal, 0));
}


template <class KeyType>
void
AvlNode<KeyType>::RotateRight(AvlNode<KeyType> * & root) {
  AvlNode<KeyType> * oldRoot = root;

        // perform rotation
  root = root->mySubtree[LEFT];
  oldRoot->mySubtree[LEFT] = root->mySubtree[RIGHT];
  root->mySubtree[RIGHT] = oldRoot;

        // update balances
  oldRoot->myBal +=  (1 - MIN(root->myBal, 0));
  root->myBal    +=  (1 + MAX(oldRoot->myBal, 0));
}
```

We can make this code more compact however by using only ONE rotate method which takes an additional parameter: the direction in which to rotate. Notice that I have defined LEFT, and RIGHT to be mnemonic constants to index into an array of subtrees. I can pass the constant LEFT or RIGHT to the rotation method and it can calculate the direction opposite the given direction by subtracting the given direction from the number one.

It does not matter whether LEFT is 0 or RIGHT is 0 as long as one of them is 0 and the other is 1. If this is the case, then:

```
1 - LEFT  = RIGHT
```

and

```
1 - RIGHT = LEFT
```

Using this and the same type definitions as before (and the same inline functions), the C++ code for a single rotation becomes:

```
inline  dir_t
Opposite(dir_t dir) { return dir_t(1 - int(dir)); }

// RotateOnce -- static member function that performs a single
//               rotation for the given direction.
//
template <class KeyType>
void
AvlNode<KeyType>::RotateOnce(AvlNode<KeyType> * & root, dir_t  dir) {
  AvlNode<KeyType> * oldRoot  = root;
  dir_t              otherDir = Opposite(dir);

        // rotate
  root = tree->mySubtree[otherDir];
  oldRoot->mySubtree[otherDir] = tree->mySubtree[dir];
  root->mySubtree[dir] = oldRoot;

        // update balances
  if (dir == LEFT)  {
     oldRoot->myBal -=  (1 + MAX(root->myBal, 0));
     root->myBal    -=  (1 - MIN(oldRoot->myBal, 0));
  } else  /* dir == RIGHT */  {
     oldRoot->myBal +=  (1 - MIN(root->myBal, 0) );
     root->myBal    +=  (1 + MAX(oldRoot->myBal, 0));
  } //else
}
```

We can compact this code even further if we play around with the equations for updating the balances. Let us use the fact that max(x,y) = -min(-x,-y):

for a left rotation

```
oldRoot->myBal -=  (1 + MAX(tree->myBal, 0));
tree->myBal    -=  (1 - MIN(oldRoot->myBal, 0));
```

for a right rotation

```
oldRoot->myBal +=  (1 - MIN(tree->myBal, 0));
tree->myBal    +=  (1 + MAX(oldRoot->myBal, 0));
```

Using the above rule to change all occurrences of "MIN" to "MAX" these equations become:

for a left rotation

```
oldRoot->myBal -=  (1 + MAX( +(tree->myBal), 0));
tree->myBal    -=  (1 + MAX( -(oldRoot->myBal), 0));
```

for a right rotation

```
oldRoot->myBal +=  (1 + MAX( -(tree->myBal), 0));
tree->myBal    +=  (1 + MAX( +(oldRoot->myBal), 0));
```

Note that the difference between updating the balances for our right and left rotations is only the occurrence of a '+' where we would like to see a '-' in the assignment operator, and the sign of the first argument to the MAX function. If we had a function that would map LEFT to +1 and RIGHT to -1 we could multiply by the result of that function to update our balances. Such a function is

```
f(x) = 1 - 2x
```

"f" maps 0 to 1 and maps 1 to -1. This function will **not** map LEFT and RIGHT to the same value regardless of which is 1 and which is 0 however. If we wish our function to have this property then we can multiply (1 - 2x) by (RIGHT - LEFT) so that the result "switches" signs accordingly depending upon whether LEFT is 0 or RIGHT is 0. This defines a new function "g":

```
g(x) = (1 - 2x)(RIGHT - LEFT)
```

If LEFT = 0 and RIGHT = 1 then:

```
g(LEFT)  = (1 - 2*0)(1 - 0) =  1*1    = 1
g(RIGHT) = (1 - 2*1)(1 - 0) = (-1)*1  = -1
```

If LEFT = 1 and RIGHT = 0 then:

```
g(LEFT)  = (1 - 2*1)(0 - 1) = (-1)*(-1)  = 1
g(RIGHT) = (1 - 2*0)(0 - 1) =  1*(-1)    = -1
```

So, as desired, the function "g" maps LEFT to +1 and RIGHT to -1 regardless of which is 0 and which is 1.

Now, if we introduce a new variable called "factor" and assign it the value "g(dir)", we may update the balances in our rotation method without using a conditional statement:

for a rotation in the `dir` direction

```
oldRoot->myBal -=  factor * (1 + MAX(factor * tree->myBal, 0));
tree->myBal    +=  factor * (1 + MAX(factor * oldRoot->myBal, 0));
```

Using this, the new code for our rotation method becomes:

```
// RotateOnce -- static member function that performs a single
//               rotation for the given direction.
//               Return 1 if the tree height changes due to rotation,
//               otherwise return 0.
//
template <class KeyType>
void
AvlNode<KeyType>::RotateOnce(AvlNode<KeyType> * & root, dir_t  dir) {
  AvlNode<KeyType> * oldRoot  = root;
  dir_t     otherDir = Opposite(dir);
  short     factor   = (RIGHT - LEFT) * (1 - (2 * dir));

        // rotate
  root = tree->mySubtree[otherDir];
  oldRoot->mySubtree[otherDir] = tree->mySubtree[dir];
  root->mySubtree[dir] = oldRoot;

        // update balances
  oldRoot->myBal -=  factor * (1 + MAX(factor * root->myBal, 0));
  root->myBal    +=  factor * (1 + MAX(factor * oldRoot->myBal, 0));
}
```

However, although this second version of "rotate" is more compact and doesn't require the use of a conditional test on the variable "dir", It may actually run slower than our first version of "rotate" because the time required to make the "test" may well be less than the time required to perform the additional multiplications and subtractions.

Now a double rotation can be implemented as a series of single rotations:

```
// RotateTwice -- static member function to rotate a given node
//                for the given direction and then the opposite
//                direction to restore the balance of an AVL tree
//                Return 1 if the tree height changes due to rotation,
//                otherwise return 0.
//
template <class KeyType>
void
AvlNode<KeyType>::RotateTwice(AvlNode<KeyType> * & root, dir_t  dir) {
    dir_t   otherDir = Opposite(dir);
    RotateOnce(root->mySubtree[otherDir], otherDir);
    RotateOnce(root, dir);
}
```

another Method for Calculating Balances After Rotation

One may use a different method than the one described above which is perhaps simpler. Note however that the method for updating balances described above works regardless of what numbers the balance factor may contain (as long as they are correct -- it works, no matter how imbalanced). If we take into account some of the conditions that cause a rotation, we have more information to work with (like that the node to be rotated has a balance of +2 or -2 etc..)

For a single LL rotation we have one of two possibilities:

```
         A                                        B
        / \                                      / \
       /   \                                    /   \
      a     B               ==>                A     c
           / \                                / \
          /   \                              /   \
         b     c                            a     b
```

Balance Factors

	Before Rotation		**After Rotation**	
case 1:	A = +2	B = +1	A = 0	B = 0
case 2:	A = +2	B = 0	A = +1	B = -1

so in either case NewB = OldB -1 and newA = -newB so we get A = - (--B) for a single left rotation.

For a single RR rotation the possibilities are (The picture is a mirror image of the LL one -- swap all right and left kids of each node)

Balance Factors

	Before Rotation		**After Rotation**	
case 1:	A = -2	B = -1	A = 0	B = 0

case 2:	A = -2	B = 0	A = -1	B = +1

so in either case NewB = OldB +1 and newA = -newB so we get A = - (++B) for a single left rotation.

This means that we can use the following to update balances:

```
      // Use mnemonic constants for indicating a change in height
   enum height_effect_t { HEIGHT_NOCHANGE = 0, HEIGHT_CHANGE = 1 };

   // RotateOnce -- static member function that performs a single
   //               rotation for the given direction.
   //               Return 1 if the tree height changes due to rotation,
   //               otherwise return 0.
   //
   template <class KeyType>
   int
   AvlNode<KeyType>::RotateOnce(AvlNode<KeyType> * & root, dir_t dir)
   {
      dir_t  otherDir = Opposite(dir);
      AvlNode<KeyType> * oldRoot = root;

         // See if otherDir subtree is balanced. If it is, then this
         // rotation will *not* change the overall tree height.
         // Otherwise, this rotation will shorten the tree height.
      int  heightChange = (root->mySubtree[otherDir]->myBal == 0)
                             ? HEIGHT_NOCHANGE
                             : HEIGHT_CHANGE;

         // assign new root
      root = oldRoot->mySubtree[otherDir];

         // new-root exchanges it's "dir" subtree for it's parent
      oldRoot->mySubtree[otherDir] = root->mySubtree[dir];
      root->mySubtree[dir] = oldRoot;

         // update balances
      oldRoot->myBal = -((dir == LEFT) ? --(root->myBal) : ++(root->myBal));

      return  heightChange;
   }
```

We get an even nicer scenario when we look at LR and RL rotations. For a double LR rotation we have one of three possibilities:

```
              A                                         B
             / \                                       / \
            /   \                                     /   \
           a     C                ==>                A     C
                / \                                 / \   / \
               /   \                               /   | |   \
              B     c                             a   b1  b2  c
             / \
            /   \
           b1   b2
```

Balance Factors

	Before Rotation			**After Rotation**		
case 1:	A = +2	B = +1	C = -1	A = -1	B = 0	C = 0
case 2:	A = +2	B = 0	C = -1	A = 0	B = 0	C = 0
case 3:	A = +2	B = -1	C = -1	A = 0	B = 0	C = +1

So we get, in all three cases:

```
newA = -max( oldB, 0 )
newC = -min( oldB, 0 )
newB = 0
```

Now for a double RL rotation we have the following possibilities (again, the picture is the mirror image of the LR case):

Balance Factors

	Before Rotation			**After Rotation**		
case 1:	A = -2	B = +1	C = +1	A = 0	B = 0	C = -1
case 2:	A = -2	B = 0	C = +1	A = 0	B = 0	C = 0
case 3:	A = -2	B = -1	C = +1	A = +1	B = 0	C = 0

So we get, in all three cases:

```
newA = -min( oldB, 0 )
newC = -max( oldB, 0 )
newB = 0
```

*This is exactly the **mirror image** of what we had for the LR case:* The nodes `A` and `C` in the newly rotated result simply *exchanged balance factors* with one another between the RL case and the LR case. What this means is that in each case, the new balance factor of the new *left* subtree is the same, and the new balance factor of the new *right* subtree is the same:

```
new(left)  = -max( oldB, 0 )
new(right) = -min( oldB, 0 )
new(root)  = 0
```

So now we can write the code for a double rotation as follows:

```
// RotateTwice -- static member function to rotate a given node
//                twice for the given direction in order to
//                restore the balance of an AVL tree.
//                Return 1 if the tree height changes due to rotation,
//                otherwise return 0.
//
template <class KeyType>
int
AvlNode<KeyType>::RotateTwice(AvlNode<KeyType> * & root, dir_t dir)
{
   dir_t  otherDir = Opposite(dir);
   AvlNode<KeyType> * oldRoot = root;
   AvlNode<KeyType> * oldOtherDirSubtree = root->mySubtree[otherDir];

      // assign new root
   root = oldRoot->mySubtree[otherDir]->mySubtree[dir];

      // new-root exchanges it's "dir" subtree for it's grandparent
   oldRoot->mySubtree[otherDir] = root->mySubtree[dir];
   root->mySubtree[dir] = oldRoot;

      // new-root exchanges it's "other-dir" subtree for it's parent
   oldOtherDirSubtree->mySubtree[dir] = root->mySubtree[otherDir];
   root->mySubtree[otherDir] = oldOtherDirSubtree;

      // update balances
   root->mySubtree[LEFT]->myBal  = -MAX(root->myBal, 0);
   root->mySubtree[RIGHT]->myBal = -MIN(root->myBal, 0);
   root->myBal = 0;

      // A double rotation always shortens the overall height of the tree
```

```
    return  HEIGHT_CHANGE;
}
```

Now that we have the rotation methods written, we just need to worry about when to call them. One helpful item is a method called `balance()` which is called when a node gets too heavy on a particular side:

```
   // Use mnemonic constants for valid balance-factor values
enum balance_t { LEFT_HEAVY = -1, BALANCED = 0, RIGHT_HEAVY = 1 };

   // Return true if the tree is too heavy on the left side
inline static int
LEFT_IMBALANCE(short bal) { return (bal < LEFT_HEAVY); }

   // Return true if the tree is too heavy on the right side
inline static int
RIGHT_IMBALANCE(short bal) { return (bal > RIGHT_HEAVY); }

// Rebalance -- static member function to rebalance a (sub)tree
//              if it has become imbalanced.
//              Return 1 if the tree height changes due to rotation,
//              otherwise return 0.
template <class KeyType>
int
AvlNode<KeyType>::ReBalance(AvlNode<KeyType> * & root) {
   int  heightChange = HEIGHT_NOCHANGE;

   if (LEFT_IMBALANCE(root->myBal)) {
         // Need a right rotation
      if (root->mySubtree[LEFT]->myBal  ==  RIGHT_HEAVY) {
            // RL rotation needed
         heightChange = RotateTwice(root, RIGHT);
      } else {
            // RR rotation needed
         heightChange = RotateOnce(root, RIGHT);
      }
   } else if (RIGHT_IMBALANCE(root->myBal)) {
         // Need a left rotation
      if (root->mySubtree[RIGHT]->myBal  ==  LEFT_HEAVY) {
            // LR rotation needed
         heightChange = RotateTwice(root, LEFT);
      } else {
            // LL rotation needed
         heightChange = RotateOnce(root, LEFT);
      }
   }

   return  heightChange;
}
```

This method helps but now comes the hard part (in my humble opinion), figuring out when the height of the current subtree has changed.

Determining When the Height of the Current Subtree has Changed

After we have inserted or deleted a node from the current subtree, we need to determine if the total height of the current tree has changed so that we may pass the information up the recursion stack to previous instantiations of the insertion and deletion methods. Let us first consider the case of an insertion. The simplest case is at the point where the insertion occurred. Since we have created a node where one did not previously exist, we have increased the height of the inserted node from 0 to 1. Therefore we need to pass the value 1 (I will use "1" for TRUE and "0" for FALSE) back to the previous level of recursion to indicate the increase in the height of the current subtree.

```
       |            after insertion                |
      NULL         ================>               |
                                                   A
```

The remaining cases for an insertion are almost as simple. If a 0 (FALSE) was the "height-change-indicator" passed back by inserting into a subtree of the current level, then there is no height change at the current level. It is true that the structure of one of the subtrees may have changed due to an insertion and/or rotation, but since the height of the subtree did not change, neither did the height of the current level.

```
       |            after insertion                |
       |           ================>               |
       A                                           A
      / \                                         / \
     /   \                                       /   \
    b     c                                     b     d
```

If the current level is balanced after inserting the node (but before attempting any rotations) then we just made one subtree equal in height to the other. Therefore the overall height of the current level remains unchanged and a 0 is returned.

```
       |            after insertion                |
       |           ================>               |
       A                                           A
      /                                           / \
     /                                           /   \
    b                                           b     c
```

Before we write the code for an insertion, we still need a method to compare items while we traverse the tree. Normally, we expect this `Compare()` method to return a `strcmp()` type result (<0, ==0, or >0 for <,==,> respectively). We will be a little sneaky and write our own `Compare()` method which will use the `Compare()` method of the supplied `KeyType`, and take an additional parameter describing whether we want to actually compare the values of the two items, or if we just want to traverse towards the maximal or minimal element of the tree. We can use the enumeration values of the `cmp_t` type (`EQ_CMP`, `MIN_CMP`, `MAX_CMP`) to indicate the type of comparison that is desired. This extra `Compare()` method of ours doesnt help much for insertion, but it will be a *big* help for deletion (or searching) when we need to find the minimal or maximal element in a subtree:

```
// Compare -- Perform a comparison of the given key against the given
//            item using the given criteria (min, max, or equivalence
//            comparison). Returns:
//               EQ_CMP if the keys are equivalent
//               MIN_CMP if this key is less than the item's key
//               MAX_CMP if this key is greater than item's key
//
template <class KeyType>
cmp_t
AvlNode<KeyType>::Compare(KeyType key, cmp_t cmp) const
{
   switch (cmp) {
      case EQ_CMP :  // Standard comparison
         return  myData->Compare(key);

      case MIN_CMP :  // Find the minimal element in this tree
         return  (mySubtree[LEFT] == NULL) ? EQ_CMP : MIN_CMP;

      case MAX_CMP :  // Find the maximal element in this tree
         return  (mySubtree[RIGHT] == NULL) ? EQ_CMP : MAX_CMP;
   }
```

```
}
```

We are now ready to write the insertion method for our AVL tree:

```
// Insert -- Insert the given key into the given tree. Return the
//           node if it already exists. Otherwise return NULL to
//           indicate that the key was successfully inserted.
//           Upon return, the "change" parameter will be '1' if
//           the tree height changed as a result of the insertion
//           (otherwise "change" will be 0).
//
template <class KeyType>
Comparable<KeyType> *
AvlNode<KeyType>::Insert(Comparable<KeyType> *   item,
                         AvlNode<KeyType>    * & root,
                         int                   & change)
{
      // See if the tree is empty
   if (root == NULL) {
         // Insert new node here
      root = new AvlNode<KeyType>(item);
      change =  HEIGHT_CHANGE;
      return  NULL;
   }

      // Initialize
   Comparable<KeyType> * found = NULL;
   int  increase = 0;

      // Compare items and determine which direction to search
   cmp_t  result = root->Compare(item->Key());
   dir_t  dir = (result == MIN_CMP) ? LEFT : RIGHT;

   if (result != EQ_CMP) {
         // Insert into "dir" subtree
      found = Insert(item, root->mySubtree[dir], change);
      if (found)  return  found;   // already here - dont insert
      increase = result * change;  // set balance factor increment
   } else  {   // key already in tree at this node
      increase = HEIGHT_NOCHANGE;
      return  root->myData;
   }

   root->myBal += increase;    // update balance factor

  // --------------------------------------------------------------------
  // re-balance if needed -- height of current tree increases only if its
  // subtree height increases and the current tree needs no rotation.
  // --------------------------------------------------------------------

   change =  (increase && root->myBal)
                ? (1 - ReBalance(root))
                : HEIGHT_NOCHANGE;
   return  NULL;
}
```

Deletion is more complicated than insertion. The height of the current level may decrease for two reasons: either a rotation occurred to decrease the height of a subtree (and hence the current level), or a subtree shortened in height resulting in a now balanced current level (subtree was "trimmed down" to the same size as the other). Just because a rotation has occurred however, does not mean that the subtree height has decreased. There is a special case where rotating preserves the current subtree height.

Suppose I have a tree as follows:

```
                          C
                         / \
```

```
A   E
   / \
  D   F
```

Deleting "A" results in the following (imbalanced) tree:

```
C
 \
  E
 / \
D   F
```

This type of imbalance cannot occur during insertion, only during deletion. Notice that the root has a balance of 2 but its heavy subtree has a balance of zero (the other case would be a -2 and a 0). Performing a single left rotation to restore the balance results in:

```
  E
 / \
C   F
 \
  D
```

This tree has the same height as it did before it was rotated. Hence, we may determine if deletion caused the subtree height to change by seeing if one of the following occurred:

1. If the new balance (after deletion) is zero and NO rotation took place.
2. If a rotation took place but was **not** one of the special rotations mentioned above (a -2:0 or a 2:0 rotation).

For insertion, we only needed to check if a rotation occurred to see if the subtree height had changed. But for deletion we need to check all of the above. So for deletion of a node we have:

```
// Delete -- delete the given key from the given tree. Return NULL
//           if the key is not found in the tree. Otherwise return
//           a pointer to the node that was removed from the tree.
//           Upon return, the "change" parameter will be '1' if
//           the tree height changed as a result of the deletion
//           (otherwise "change" will be 0).
//
template <class KeyType>
Comparable<KeyType> *
AvlNode<KeyType>::Delete(KeyType              key,
                         AvlNode<KeyType> * & root,
                         int                & change,
                         cmp_t                cmp)
{
      // See if the tree is empty
   if (root == NULL) {
         // Key not found
      change = HEIGHT_NOCHANGE;
      return  NULL;
   }

      // Initialize
   Comparable<KeyType> * found = NULL;
   int  decrease = 0;

      // Compare items and determine which direction to search
   cmp_t  result = root->Compare(key, cmp);
   dir_t  dir = (result == MIN_CMP) ? LEFT : RIGHT;

   if (result != EQ_CMP) {
         // Delete from "dir" subtree
      found = Delete(key, root->mySubtree[dir], change, cmp);
```

```
      if (! found)  return  found;   // not found - can't delete
      decrease = result * change;    // set balance factor decrement
   } else  {   // Found key at this node
      found = root->myData;  // set return value

      // --------------------------------------------------------------
      // At this point we know "result" is zero and "root" points to
      // the node that we need to delete.  There are three cases:
      //
      //    1) The node is a leaf.  Remove it and return.
      //
      //    2) The node is a branch (has only 1 child). Make "root"
      //       (the pointer to this node) point to the child.
      //
      //    3) The node has two children. Swap items with the successor
      //       of "root" (the smallest item in its right subtree) and
      //       delete the successor from the right subtree of "root".
      //       The identifier "decrease" should be reset if the subtree
      //       height decreased due to the deletion of the successor of
      //       "root".
      // --------------------------------------------------------------

      if ((root->mySubtree[LEFT] == NULL) &&
          (root->mySubtree[RIGHT] == NULL)) {
             // We have a leaf -- remove it
         delete  root;
         root = NULL;
         change = HEIGHT_CHANGE;    // height changed from 1 to 0
         return  found;
      } else if ((root->mySubtree[LEFT] == NULL) ||
                 (root->mySubtree[RIGHT] == NULL)) {
            // We have one child -- only child becomes new root
         AvlNode<KeyType> * toDelete = root;
         root = root->mySubtree[(root->mySubtree[RIGHT]) ? RIGHT : LEFT];
         change = HEIGHT_CHANGE;    // We just shortened the subtree
            // Null-out the subtree pointers so we dont recursively delete
         toDelete->mySubtree[LEFT] = toDelete->mySubtree[RIGHT] = NULL;
         delete  toDelete;
         return  found;
      } else {
            // We have two children -- find successor and replace our
            // current data item with that of the successor
         root->myData = Delete(key, root->mySubtree[RIGHT],
                               decrease, MIN_CMP);
      }
   }

   root->myBal -= decrease;       // update balance factor

   // ------------------------------------------------------------------
   // Rebalance if necessary -- the height of current tree changes if one
   // of two things happens: (1) a rotation was performed which changed
   // the height of the subtree (2) the subtree height decreased and now
   // matches the height of its other subtree (so the current tree now
   // has a zero balance when it previously did not).
   // ------------------------------------------------------------------
   //change = (decrease) ? ((root->myBal) ? ReBalance(root)
   //                                     : HEIGHT_CHANGE)
   //                    : HEIGHT_NOCHANGE ;
   if (decrease) {
      if (root->myBal) {
         change = ReBalance(root);  // rebalance and see if height changed
      } else {
         change = HEIGHT_CHANGE;    // balanced because subtree decreased
      }
   } else {
      change = HEIGHT_NOCHANGE;
   }

   return  found;
}
```

Note how in the case of both subtrees of the deleted item being non-null, I only need one statement. This is due to the way `AvlNode::Delete` sets its parameters. The data pointer passed on entrance points to the deleted node's data on exit. So I just delete the minimal element of the right subtree, and steal its data as my-own (returning my former data item on exit).

And there we have it, the maintenance of AVL tree manipulations, the brunt of which is covered in 5 methods, none of which (except for delete which is about 1.5 pages) is greater than 1 normal page in length, including comments (and there are a lot). The main methods are:

```
RotateOnce(), RotateTwice(), ReBalance(), Insert(), Delete().
```

All other methods are very small and easy to code. The only method still missing is the `Search()` method, and that is no different from a normal binary tree search:

```
// Search -- Look for the given key using the given comparison criteria,
//           return NULL if not found, otherwise return the item address.
template <class KeyType>
Comparable<KeyType> *
AvlNode<KeyType>::Search(KeyType            key,
                         AvlNode<KeyType> * root,
                         cmp_t              cmp)
{
   cmp_t result;
   while (root  &&  (result = root->Compare(key, cmp))) {
      root = root->mySubtree[(result < 0) ? LEFT : RIGHT];
   }
   return  (root) ? root->myData : NULL;
}
```

And lets not forget the constructor and destructor:

```
template <class KeyType>
AvlNode<KeyType>::AvlNode(Comparable<KeyType> * item)
   : myData(item), myBal(0)
{
   myBal = 0 ;
   mySubtree[LEFT] = mySubtree[RIGHT] = NULL ;
}

template <class KeyType>
AvlNode<KeyType>::~AvlNode(void) {
   if (mySubtree[LEFT])  delete  mySubtree[LEFT];
   if (mySubtree[RIGHT]) delete  mySubtree[RIGHT];
}
```

Now that we have implemented most of the methods for AVL tree manipulations, we should probably finish the declaration that we started near the beginning of this discussion:

```
#include "Comparable.h"

// Indices into a subtree array
//     NOTE: I would place this inside the AvlNode class but
//           when I do, g++ complains when I use dir_t. Even
//           when I prefix it with AvlNode:: or AvlNode<KeyType>::
//           (If you can get this working please let me know)
//
enum  dir_t { LEFT = 0, RIGHT = 1 };

// AvlNode -- Class to implement an AVL Tree
//
```

```
template <class KeyType>
class AvlNode {
public:
      // Max number of subtrees per node
   enum  { MAX_SUBTREES = 2 };

   static  dir_t
   Opposite(dir_t dir) {
     return dir_t(1 - int(dir));
   }

   // ----- Constructors and destructors:

   AvlNode(Comparable<KeyType>  * item=NULL);
   virtual ~AvlNode(void);

   // ----- Query attributes:

      // Get this node's data
   Comparable<KeyType> *
   Data() const { return  myData; }

      // Get this node's key field
   KeyType
   Key() const { return  myData->Key(); }

      // Query the balance factor, it will be a value between -1 .. 1
      // where:
      //     -1 => left subtree is taller than right subtree
      //      0 => left and right subtree are equal in height
      //      1 => right subtree is taller than left subtree
   short
   Bal(void) const { return  myBal; }

      // Get the item at the top of the left/right subtree of this
      // item (the result may be NULL if there is no such item).
      //
   AvlNode *
   Subtree(dir_t dir) const { return  mySubtree[dir]; }

   // ----- Search/Insert/Delete
   //
   //   NOTE: These are all static functions instead of member functions
   //         because most of them need to modify the given tree root
   //         pointer. If these were instance member functions than
   //         that would correspond to having to modify the 'this'
   //         pointer, which is not allowed in C++. Most of the
   //         functions that are static and which take an AVL tree
   //         pointer as a parameter are static for this reason.

      // Look for the given key, return NULL if not found,
      // otherwise return the item's address.
   static Comparable<KeyType> *
   Search(KeyType key, AvlNode<KeyType> * root, cmp_t cmp=EQ_CMP)

      // Insert the given key, return NULL if it was inserted,
      // otherwise return the existing item with the same key.
   static Comparable<KeyType> *
   Insert(Comparable<KeyType> * item, AvlNode<KeyType> * & root) {
      int  change;
      return  Insert(item, root, change);
   }

      // Delete the given key from the tree. Return the corresponding
      // node, or return NULL if it was not found.
   static Comparable<KeyType> *
   Delete(KeyType key, AvlNode<KeyType> * & root, cmp_t cmp=EQ_CMP) {
      int  change;
      return  Delete(key, root, change, cmp);
   }

private:
```

```
    // ----- Private data

    Comparable<KeyType> * myData;  // Data field
    AvlNode<KeyType>    * mySubtree[MAX_SUBTREES];   // Subtree pointers
    short                 myBal;   // Balance factor

    // ----- Routines that do the *real* insertion/deletion

       // Insert the given key into the given tree. Return the node if
       // it already exists. Otherwise return NULL to indicate that
       // the key was successfully inserted.  Upon return, the "change"
       // parameter will be '1' if the tree height changed as a result
       // of the insertion (otherwise "change" will be 0).
    static Comparable<KeyType> *
    Insert(Comparable<KeyType> *   item,
           AvlNode<KeyType>    * & root,
           int                   & change);

       // Delete the given key from the given tree. Return NULL if the
       // key is not found in the tree. Otherwise return a pointer to the
       // node that was removed from the tree.  Upon return, the "change"
       // parameter will be '1' if the tree height changed as a result
       // of the deletion (otherwise "change" will be 0).
    static Comparable<KeyType> *
    Delete(KeyType              key,
           AvlNode<KeyType> * & root,
           int                & change,
           cmp_t                cmp=EQ_CMP);

    // Routines for rebalancing and rotating subtrees

       // Perform an XX rotation for the given direction 'X'.
       // Return 1 if the tree height changes due to rotation,
       // otherwise return 0.
    static int
    RotateOnce(AvlNode<KeyType> * & root, dir_t dir);

       // Perform an XY rotation for the given direction 'X'
       // Return 1 if the tree height changes due to rotation,
       // otherwise return 0.
    static int
    RotateTwice(AvlNode<KeyType> * & root, dir_t dir);

       // Rebalance a (sub)tree if it has become imbalanced
    static int
    ReBalance(AvlNode<KeyType> * & root);

       // Perform a comparison of the given key against the given
       // item using the given criteria (min, max, or equivalence
       // comparison). Returns:
       //    EQ_CMP if the keys are equivalent
       //    MIN_CMP if this key is less than the item's key
       //    MAX_CMP if this key is greater than item's key
    cmp_t
    Compare(KeyType key, cmp_t cmp=EQ_CMP) const;

 private:
       // Disallow copying and assignment
    AvlNode(const AvlNode<KeyType> &);
    AvlNode & operator=(const AvlNode<KeyType> &);
 };
```

---

back to [Brad Appleton's Home Page]()