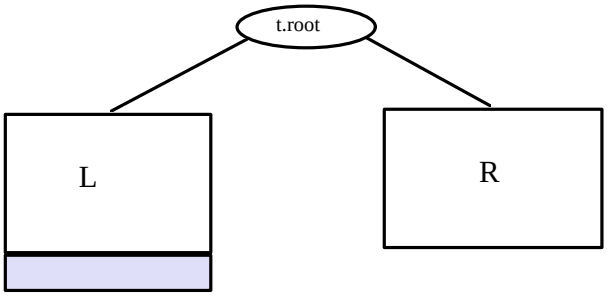
Insertion into an AVL tree

An AVL tree is a Height Balanced Search Tree. We will use recursion to implement the insertion algorithm. In inserting an item in an AVL we may distort the balance, i.e. the heights of the left and right trees may differ by more than one. If tree is balanced after insertion then return this tree.

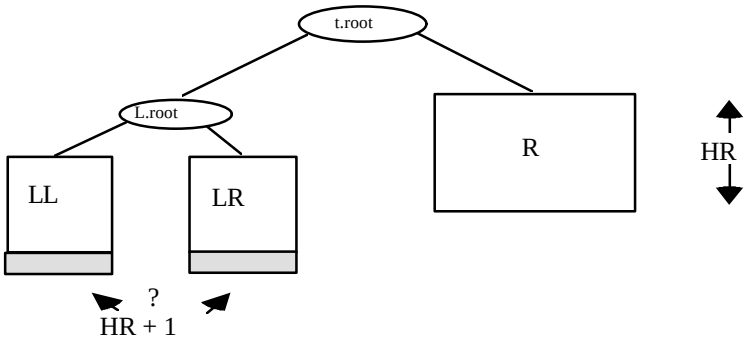
Assume insertion has been in the left (sub)tree of t and assume the balance has being distorted.



HL = HR + 2, HL -- height of Left, HR -- height of R

After insertion | HL - HR | = 2 > 1.

We need to restore the balance by manipulating the tree.

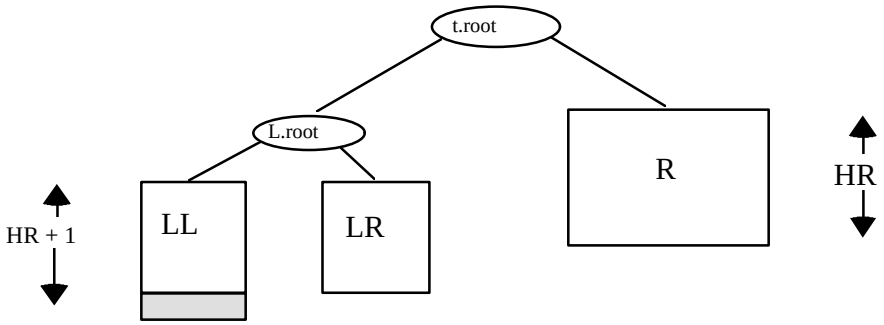


We have 2 cases to consider,

1. Hgt(LL) > Hgt(LR) and Hgt(LL) > Hgt(R),
   in effect, insertion was in LL

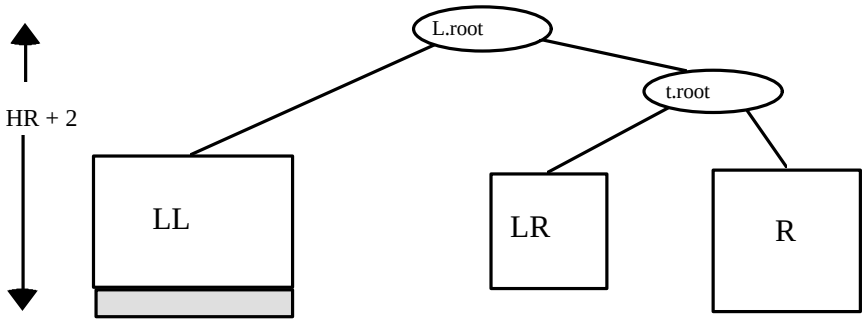
2. Hgt(LR) > Hgt(LL) and Hgt(LR) > Hgt(R),
   in effect insertion was in LR

Case 1. Hgt(LL) > Hgt(LR) and Hgt(LL) > Hgt(R)



We combine sub trees to restore balance but maintain the whole tree as a Search tree.

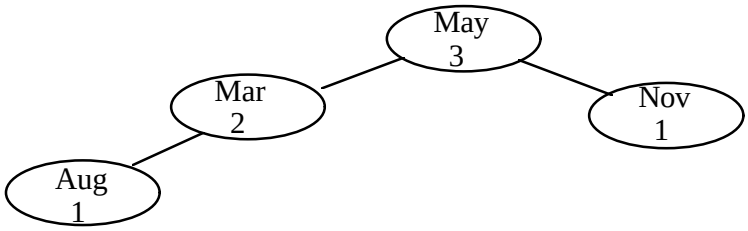
Rotate Right



HL = Max(Hgt(LL), Hgt(t)) + 1,
where t is a reference to original tree.

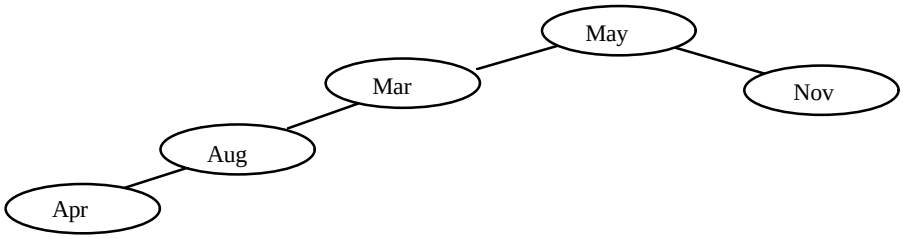
Example: Case 1.

Consider inserting the Months of the year (alphabetical ordering) into a AVL tree.

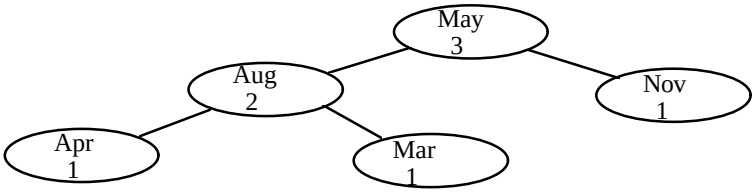


Insert “Apr” into this AVL tree.
Insert Left and then Left -- an LL insertion.

Initially we get,

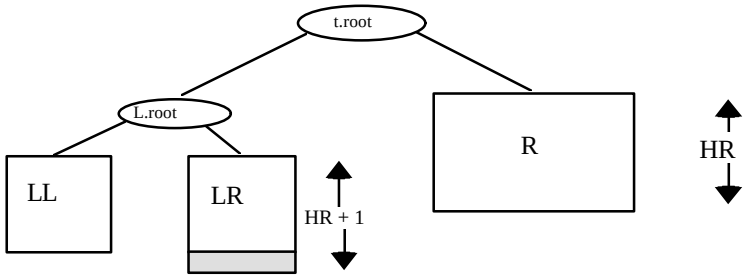


This is not balanced, to restore the balance we Rotate Right about the node ‘Mar’, the lowest node that is unbalanced.

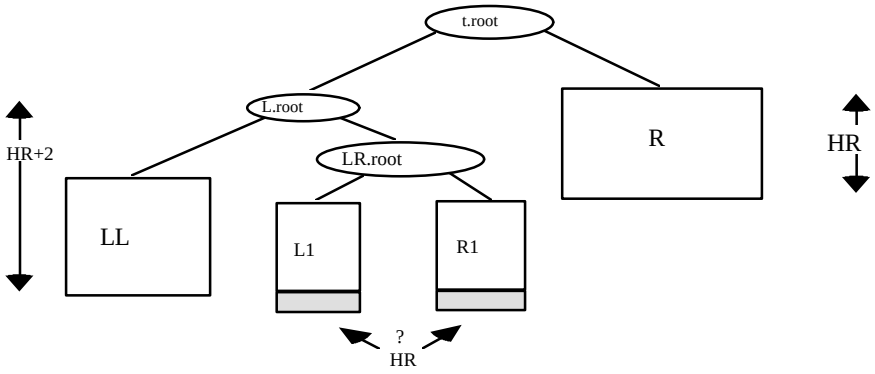


This is now an AVL tree, with height 3 (the same as before insertion)

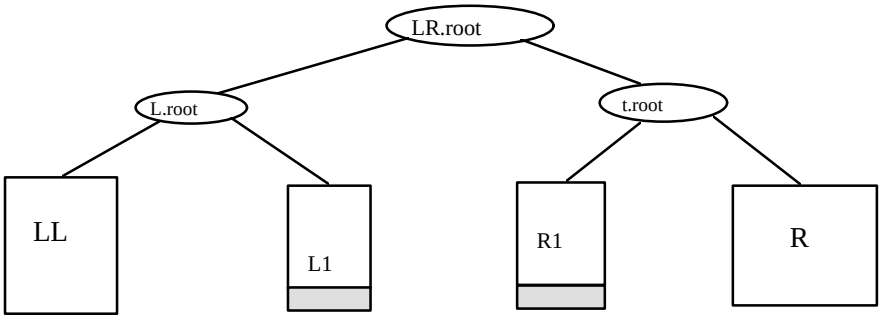
Case 2. Hgt(LR) > Hgt(LL) and Hgt(LR) > Hgt(R)



We need to look at LR and its subtrees



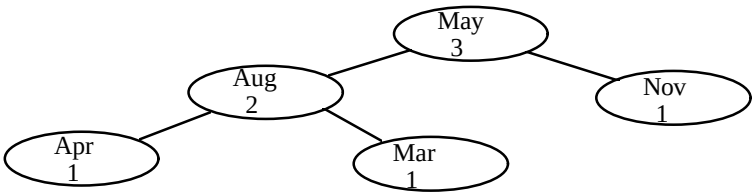
We re-combine subtrees to get,



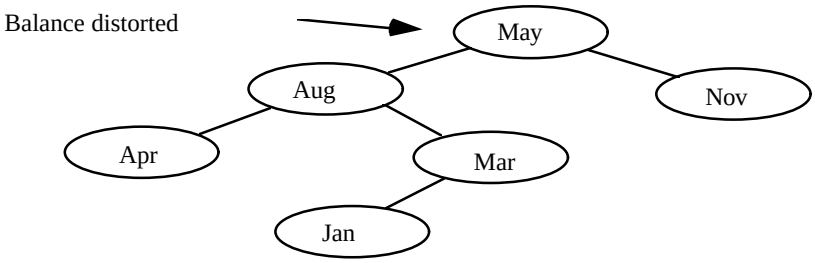
We have assumed that initially insertion is into the left subtree; insertion into right subtree is symmetrically similar.

Example: Case 2.

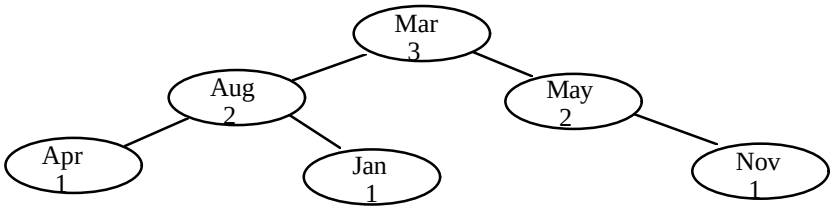
Insert “Jan”; insert Left and then Right -- LR insertion



Initially we get,

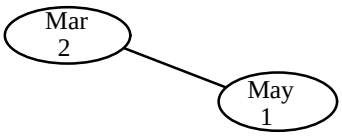


Re-Combine to get

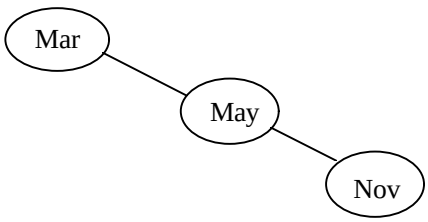


This subtree has same height as tree before insertion.

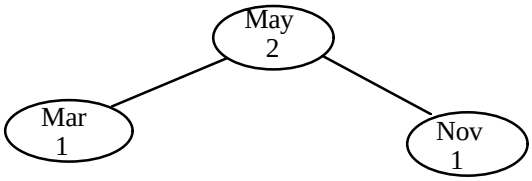
Examples of RR and RL insertions



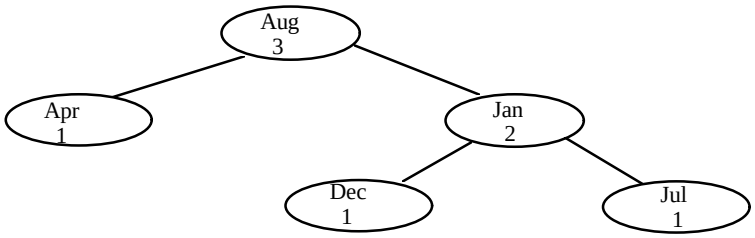
Insert “Nov” to initially get -- RR insertion



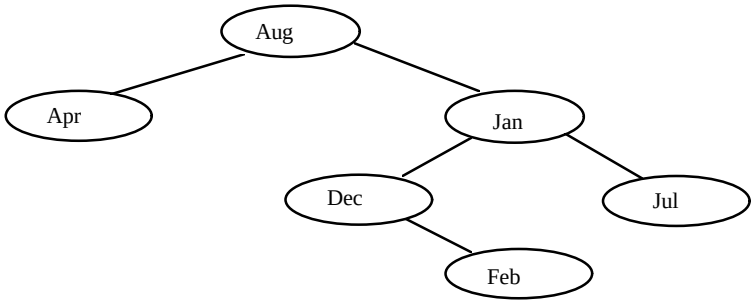
Re-Combining -- **Rotate Left**



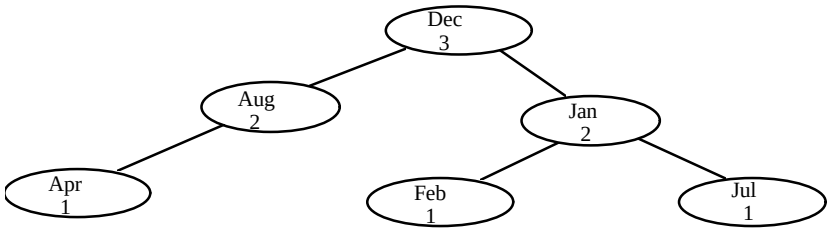
Example Insert into Right of Left subtree -- RL insertion



Insert "Feb" to initially get



Re-Combining we get,



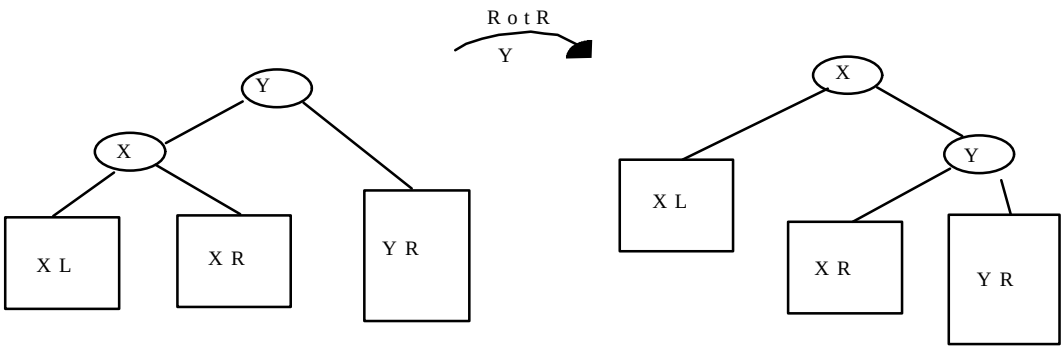
Eiffel Program for Adding item to AVL tree

The critcal function AVL_Update is similar to Insert in Binary Search Trees.

```
AVL_Update(x:G; bt:BIN_NODE[G]) : BIN_NODE[G] is
    local
        t : BIN_NODE[G]
    do
        !!t
        if bt = void then
            t.build(x,void,void)
            result := t
        elseif x < bt.value then
            t.build( bt.value, AVL_Update(x,bt.left), bt.right)
            result := Rebal(t)
        elseif x > bt.value then
            t.build( bt.value, bt.left,  AVL_Update(x,bt.right))
            result := Rebal(t)
        elseif equal(x, bt.value) then
            result := bt
        end
    end -- AVL_Update
```

After a subtree has been recursively built it is rebalanced. To implement Rebal we define a basic functions RotR "Rotate Right" and symmetrically RotL. For example, RotR will be used for an LL insertion and RotL for a RR insertion.

RotR about node y



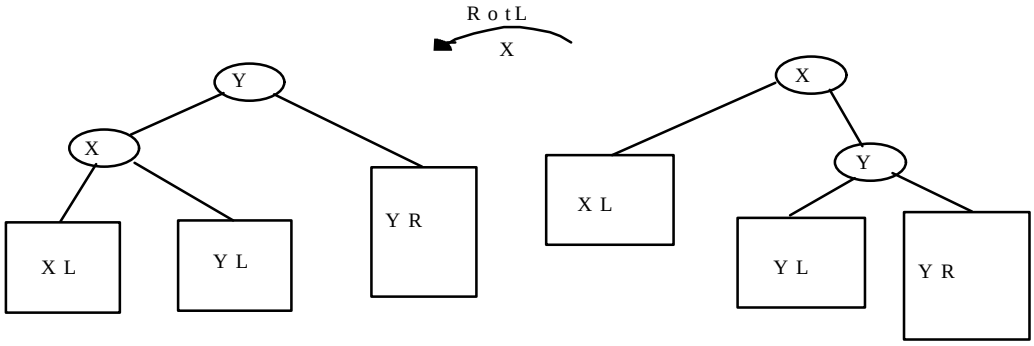
In Eiffel,

```
RotR(t:BIN_NODE[G]):BIN_NODE[G] is
        require
                Non_Void_Left:          t/=void and then t.left/=void
        local
                L,R, new, new_right : BIN_NODE[G]
        do
                L := t.left
                R := t.right
                !!new_right
                new_right.build(t.value, L.right, R)
                !!new
                new.build(L.value, L.left, new_right)
                result := new
        end -- RotR
```

Symmetrically, RotL "Rotate Left"

Read diagram (next page) from right to left i.e. input on right, output on left

RotL undoes what RotR does.

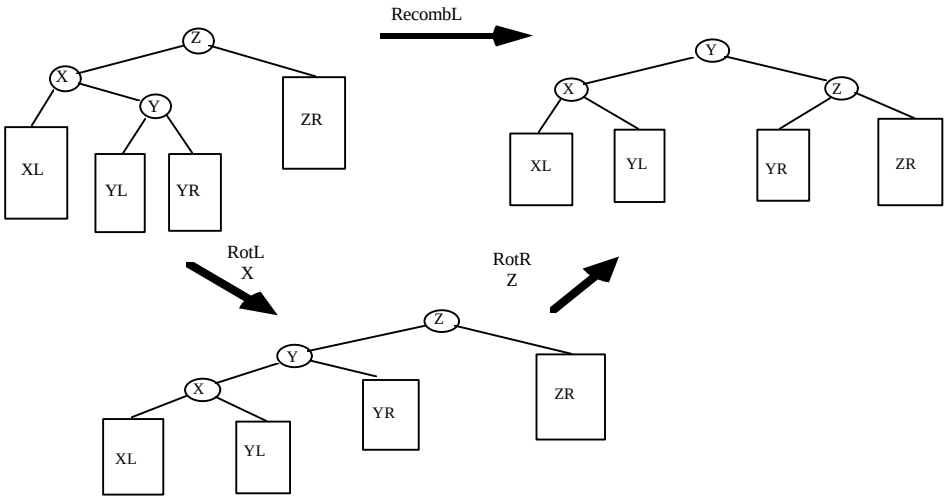


In Eiffel,

```
RotL(t:BIN_NODE[G]):BIN_NODE[G] is
        require
                Non_Void_Right:       t /= void and then t.right /= void
        local
                L,R, new, new_left : BIN_NODE[G]
        do
                L := t.left
                R := t.right
                !!new_left
                new_left.build(t.value, L, R.left)
                !!new
                new.build(R.value, new_left, R.right)
                result := new
        end -- RotL
```

To deal with LR and the symmetric RL insertion, we use functions RecombL "Recombine Left" and RecombR.
These functions will be constructed using RotL and RotR.

RecombL "Recombine Left"



We can write this in Eiffel as:

```
RecombL(t:BIN_NODE[G]):BIN_NODE[G] is
require
        Non_Void_Left:
                t /= void and then t.left /= void
local
        new : BIN_NODE[G]
do
      !!new
      new.build(t.value, RotL(t.left), t.right)
      result := RotR(new)
end -- RecomL
```

RecombR "Recombine Right" is similar,

```
RecombR(t:BIN_NODE[G]) : BIN_NODE[G] is
require
        Non_Void_Left:
             t /= void and then t.right /= void
local
        new : BIN_NODE[G]
do
      !!new
      new.build(t.value, t.left, RotR(t.right))
      result := RotL(new)
end -- RecomR
```

With these basic routines we can write a routine for rebalancing a tree.

```
Rebal(t:BIN_NODE[G]) : BIN_NODE[G] is
require
        Non_Void:  t /= void
do
    If Bal_Factor(t) > 1 then
            result := Balance_Left(t)
    elseif Bal_Factor(t) < -1 then
            result := Balance_Right(t)
    else
            result := t
    end
end -- Rebal
```

The function Bal_Factor returns

Height(Left) - Height(Right)

```
Bal_Factor(t : BIN_NODE[G]) : INTEGER is
require
        Non_Void: t /= void
do
    result:= Height(t.left) - Height(t.right)
end -- Bal_Factor
```

The routines Balance_Left and Balance_Right do the appropriate balancing.

```
Balance_Left(t:BIN_NODE[G]):BIN_NODE[G] is
require
      Non_Void_Left:
            t /= void and then t.left /= void
do
    if Bal_Factor(t.left) = 1 then
            result := RotR(t)
    elseif Bal_Factor(t.left) = -1 then
            result := reCombL(t)
    else
            result := t
    end
end -- Balance_Left
```

Similarly, Balance_Right is

```
Balance_Right(t:BIN_NODE[G]):BIN_NODE[G] is
require
        Non_Void_Right:
                t /= void and then t.right /= void
do
      if Bal_Factor(t.right) = -1 then
                result := RotL(t)
      elseif Bal_Factor(t.right) = 1 then
                result := reCombR(t)
      else
                result := t
      end
end -- Balance_Right
```

Efficiency of AVL trees

If the tree contains n nodes, then the complexity of the operations Search, Add (Update) and Delete are proportional to the height of the tree.

Operation	Time
Search	O(log n)
Add (Update / Insert)	O(log n)
Remove (Delete)	O(log n)

Empirical Results.

Average height ≈ log n + 0.25

Prob. of Rebalance for Update: 0.5 (50%)

Prob. of Rebalnce for Delete: 0.2 (20%)

Sorting

Also, If the items are to be sorted or processed in sorted order, then time complexity is O(n) using Inorder Traversal.