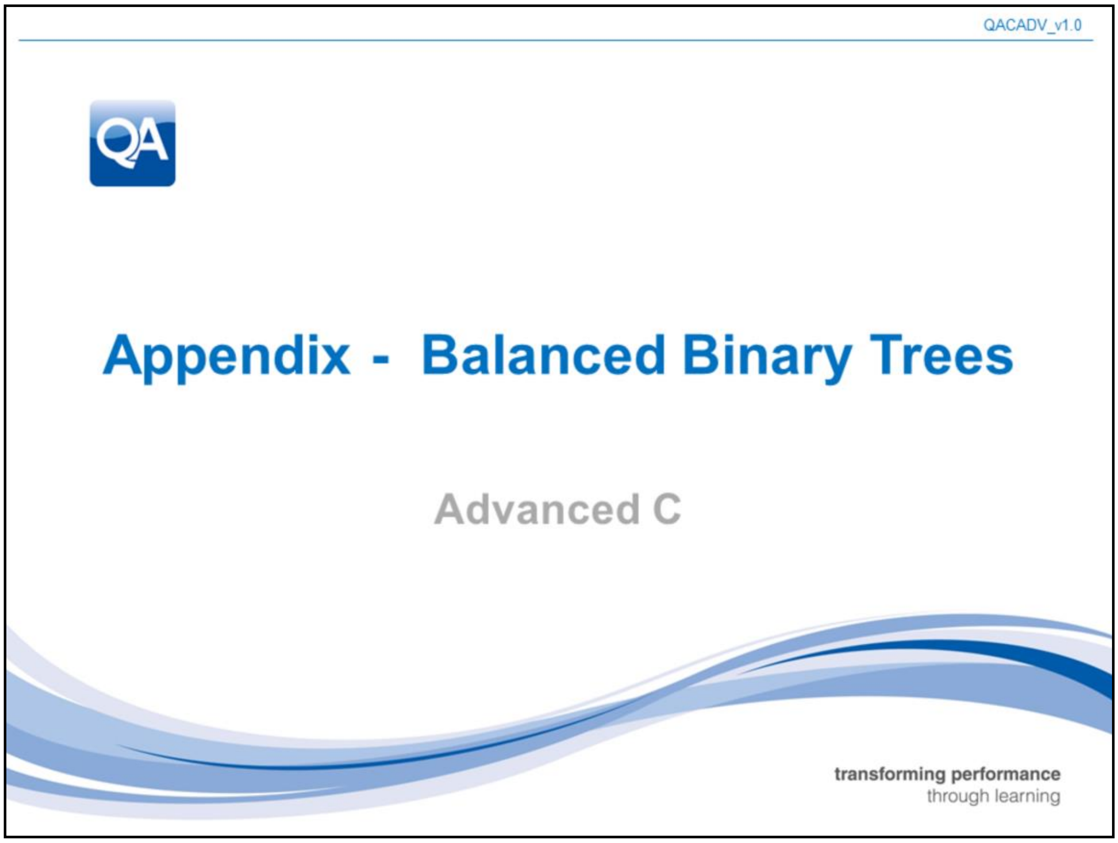
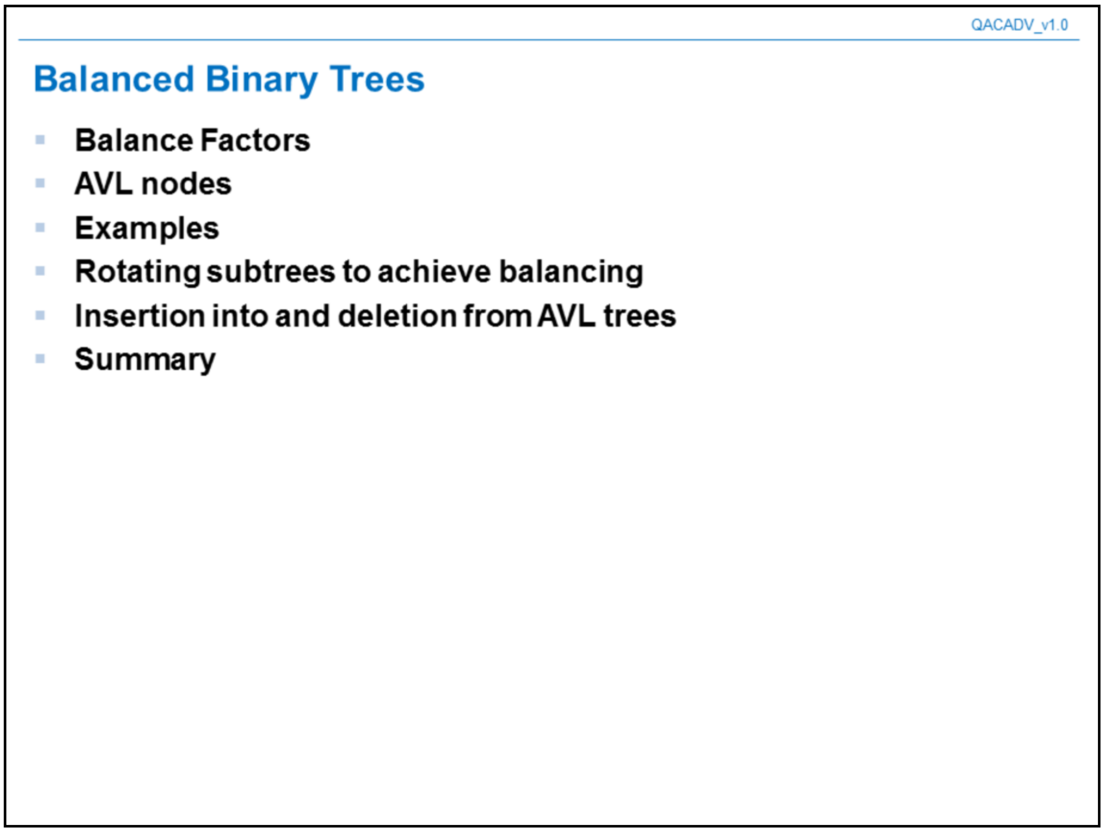


QACADV_v1.0

Appendix - Balanced Binary Trees

Advanced C

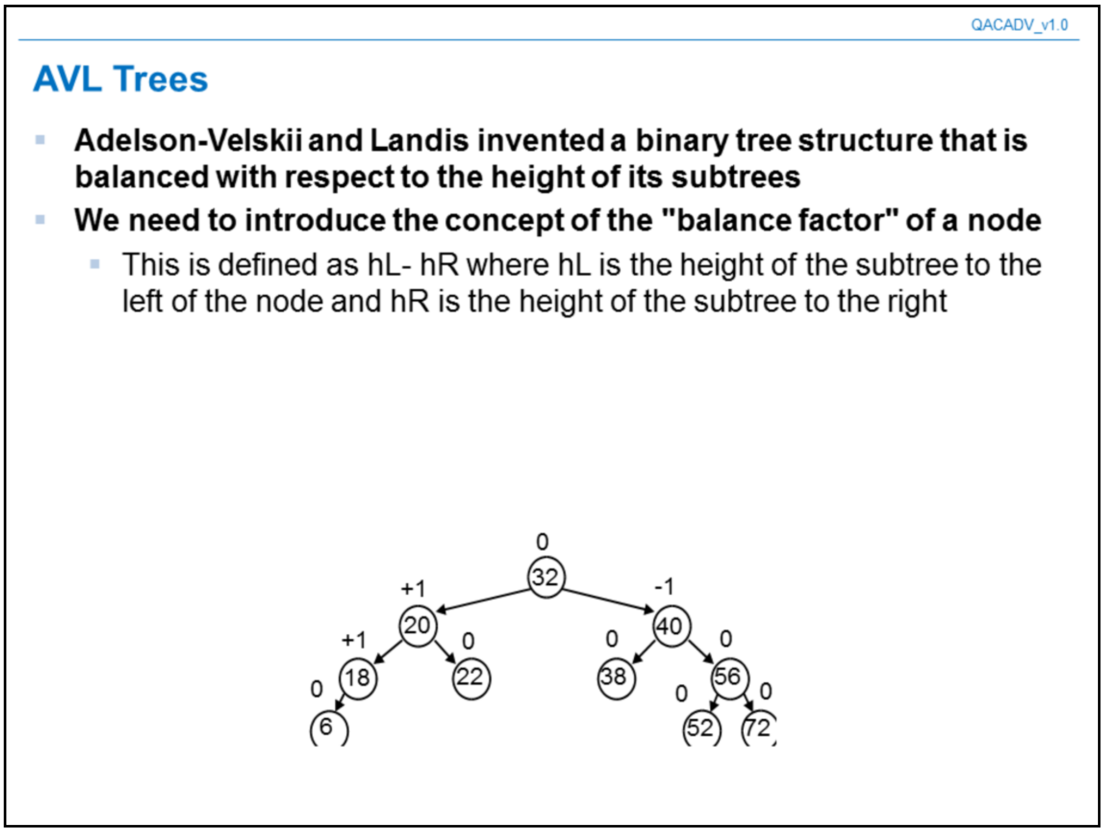
transforming performance
through learning



QACADV_v1.0

Balanced Binary Trees

- Balance Factors
- AVL nodes
- Examples
- Rotating subtrees to achieve balancing
- Insertion into and deletion from AVL trees
- Summary



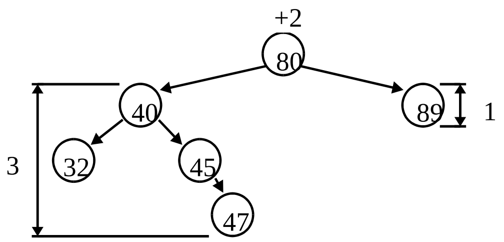
QACADV_v1.0

AVL Trees

- **Adelson-Velskii and Landis invented a binary tree structure that is balanced with respect to the height of its subtrees**
- **We need to introduce the concept of the "balance factor" of a node**
    - This is defined as hL- hR where hL is the height of the subtree to the left of the node and hR is the height of the subtree to the right

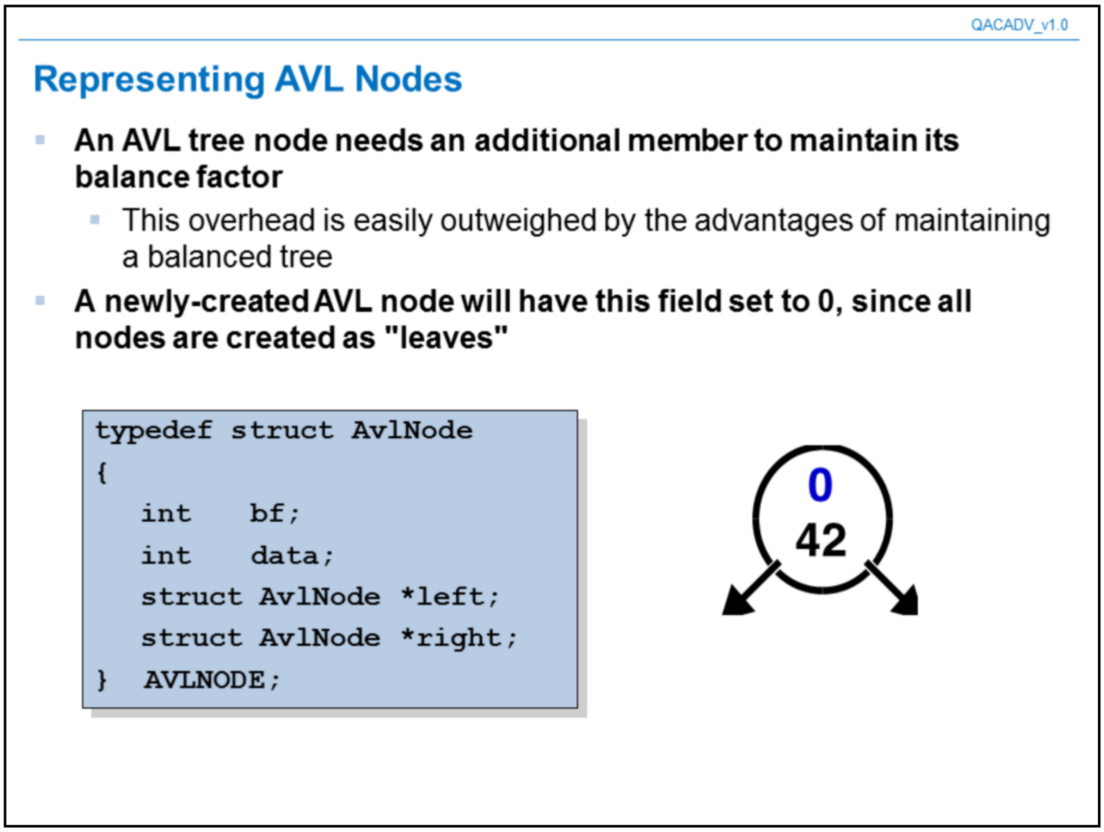
0
32
+1
20
-1
40
+1
18
0
22
0
38
0
56
0
6
0
52
0
72

In 1962, Adelson-Velskii and Landis invented a tree structure, known as an AVL tree, which is permanently balanced. AVL trees use a mechanism called a *balance factor*. The balance factor of a node is the height (or *depth* if you prefer, since the two terms are interchangeable) of the left subtree, minus the height of the right subtree. For example, consider the node containing the value `80` in the following tree:



To its left is a tree of height 3; to its right is a tree of height 1. The *balance factor* of the 80 node is therefore 3 - 1 = 2. For nodes appearing at the end of a branch, such as 89 in this example, the height of the left and right subtrees are both zero; therefore, node 89 has a balance factor of zero. The height of a tree can be found by calling the following function with the parameters int h = height(root, 0):

```
int height (BNODE *start, int depth)
{
   int hLeft, hRight;

   if (start == NULL) return depth;

   hLeft  = height(start->left, depth + 1);
   hRight = height(start->right, depth + 1);
   return (hLeft > hRight) ? hLeft : hRight;
}
```

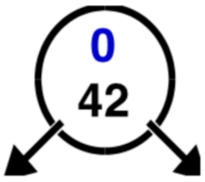


QACADV_v1.0

Representing AVL Nodes

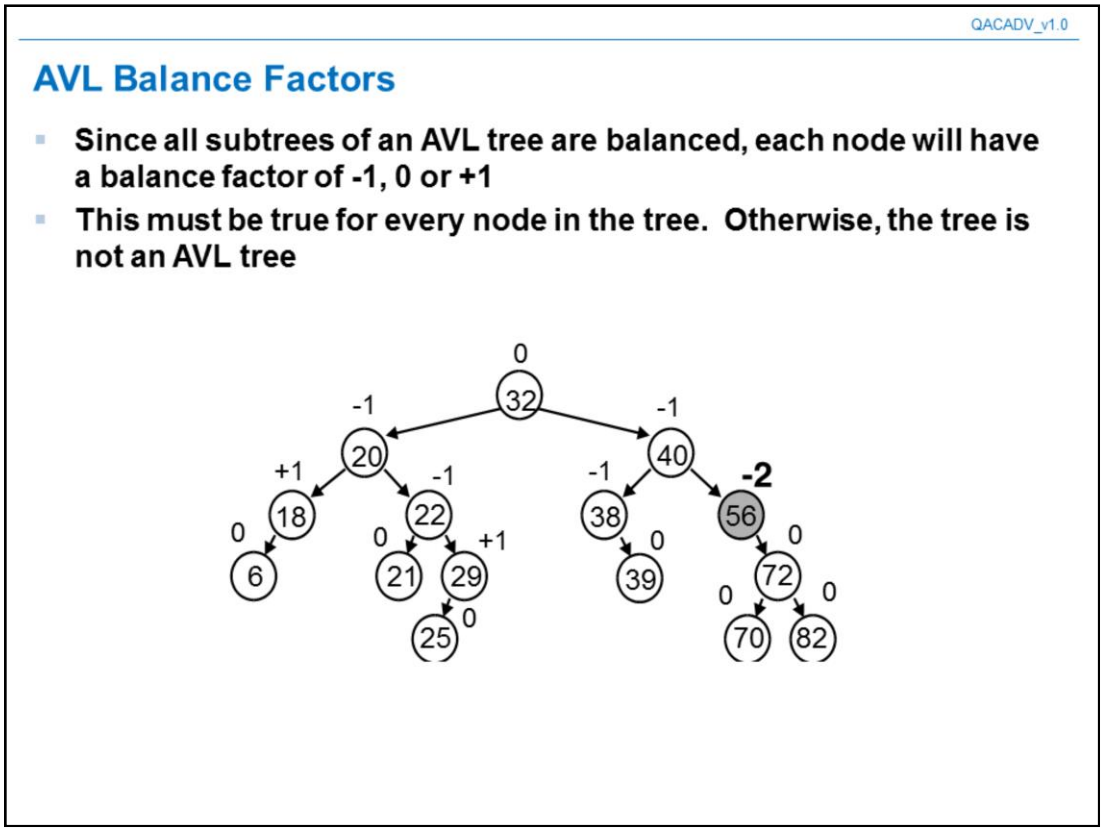
- **An AVL tree node needs an additional member to maintain its balance factor**
  - This overhead is easily outweighed by the advantages of maintaining a balanced tree
- **A newly-created AVL node will have this field set to 0, since all nodes are created as "leaves"**

```
typedef struct AvlNode
{
   int    bf;
   int    data;
   struct AvlNode *left;
   struct AvlNode *right;
}  AVLNODE;
```



In order for a structure to represent a node in an AVL tree, there must be an additional integer member in which the balance factor of the node may be stored. Despite this additional overhead, the advantage of maintaining balanced trees (with *logarithmic* search times) versus linear lists (with *linear* search times) outweighs the disadvantage.

When a node is created, it will be initialised with a *balance factor* (`bf`) of zero. This is because nodes are always created as leaf nodes, appearing at the end of a branch. If a node is added to the subtree to the *left*, the balance factor `bf` is *incremented*. If a node is added to the subtree to the *right,* the balance factor `bf` is *decremented*. If such an insertion causes `bf` to become ±2, the tree is rebalanced immediately using one of the left or right rotations discussed shortly.



QACADV_v1.0

AVL Balance Factors

- Since all subtrees of an AVL tree are balanced, each node will have a balance factor of -1, 0 or +1
- This must be true for every node in the tree. Otherwise, the tree is not an AVL tree

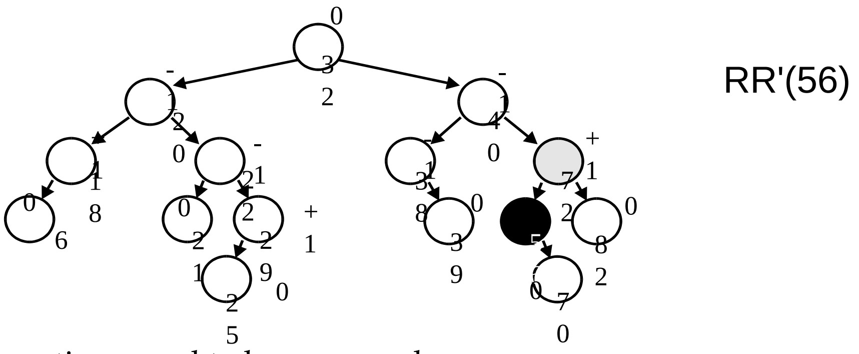
0
32
-1
20
-1
40
+1
18
-1
22
-1
38
-2
56
0
6
0
21
+1
29
0
39
0
72
0
25
0
70
0
82

Earlier in the chapter we introduced the concept of balanced trees in a somewhat informal manner, using phrases such as "Does the tree seem to be nicely spread out". We are now ready to quantify the degree of balance in a more rigorous manner.

For a binary tree to be *balanced*, the balance factor `bf` of every node in the tree must be either +1, 0 or -1. That is to say, the subtrees of each node must be either the *same* height or differ in height *by only 1 node*.

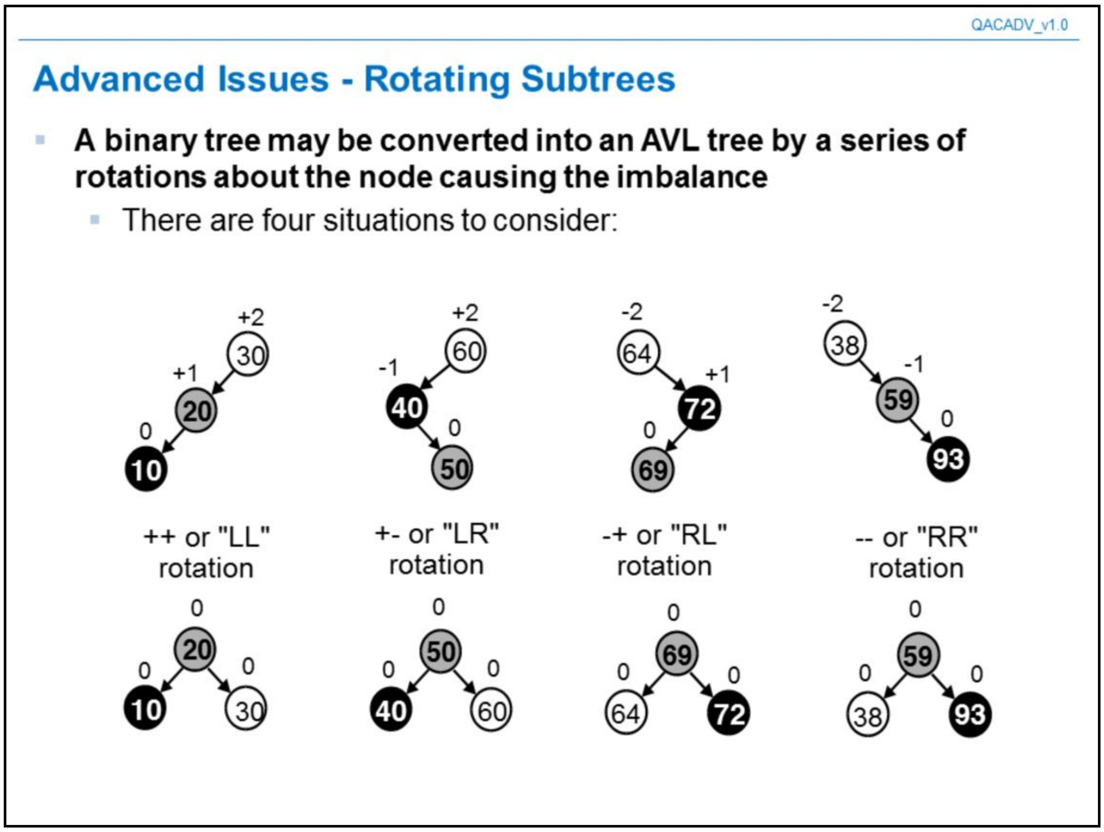
This means that the tree shown above is not a balanced binary tree (and therefore not an AVL tree) because the node containing `56` has a balance factor of -2 (the subtree to its right is two levels deeper than the subtree to its left).

If we could rearrange the tree as shown below, it would then become balanced:



The following questions need to be answered:

- *Which* node needs to be moved, and where should it be moved to?
- *When* should the tree be restructured?
- *What* does RR'(56) mean?


QACADV_v1.0

Advanced Issues - Rotating Subtrees

- **A binary tree may be converted into an AVL tree by a series of rotations about the node causing the imbalance**
  - There are four situations to consider:

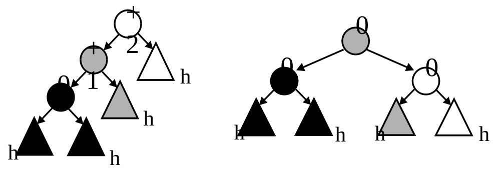
++ or "LL" rotation

+- or "LR" rotation

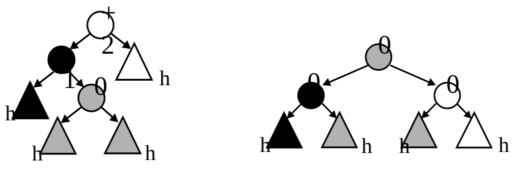
-+ or "RL" rotation

-- or "RR" rotation

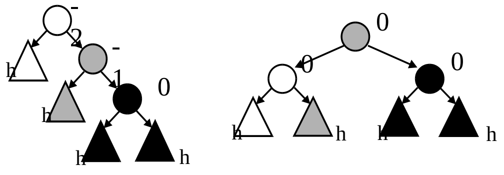
If adding a node imbalances a tree, a *rotation* is made about the node causing the imbalance. *All the rotations shown in the above examples affect the node whose balance factor is* ±2. These examples are rather simplistic. What if the nodes that are being rotated have subtrees? Using "grandparent", "parent" and "child" terminology:



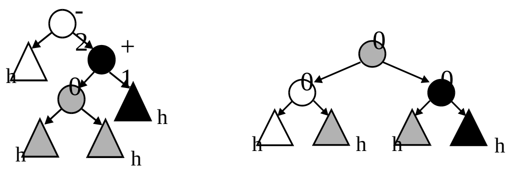
An "LL" rotation causes the *parent* to become the root. Its *right* subtree is inherited to the *left* of the grandparent node.



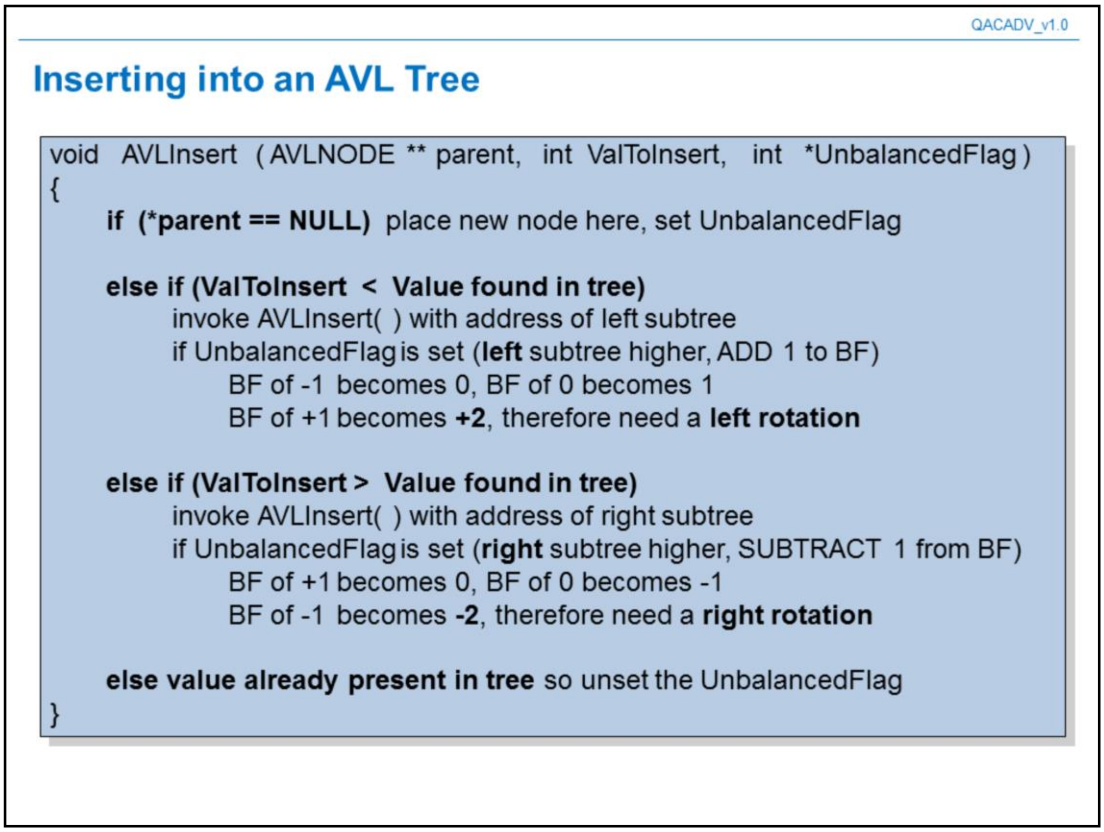
An "LR" rotation causes the *child* to become the root. Its *left* subtree is inherited to the *right* of the parent node, its *right* subtree to the *left* of the grandparent node.



An "RR" rotation causes the *parent* to become the root. Its *left* subtree is inherited to the *right* of the grandparent node.



An "RL" rotation causes the *child* to become the root. Its *left* subtree is inherited to the *right* of the grandparent node, its *right* subtree to the *left* of the parent node.

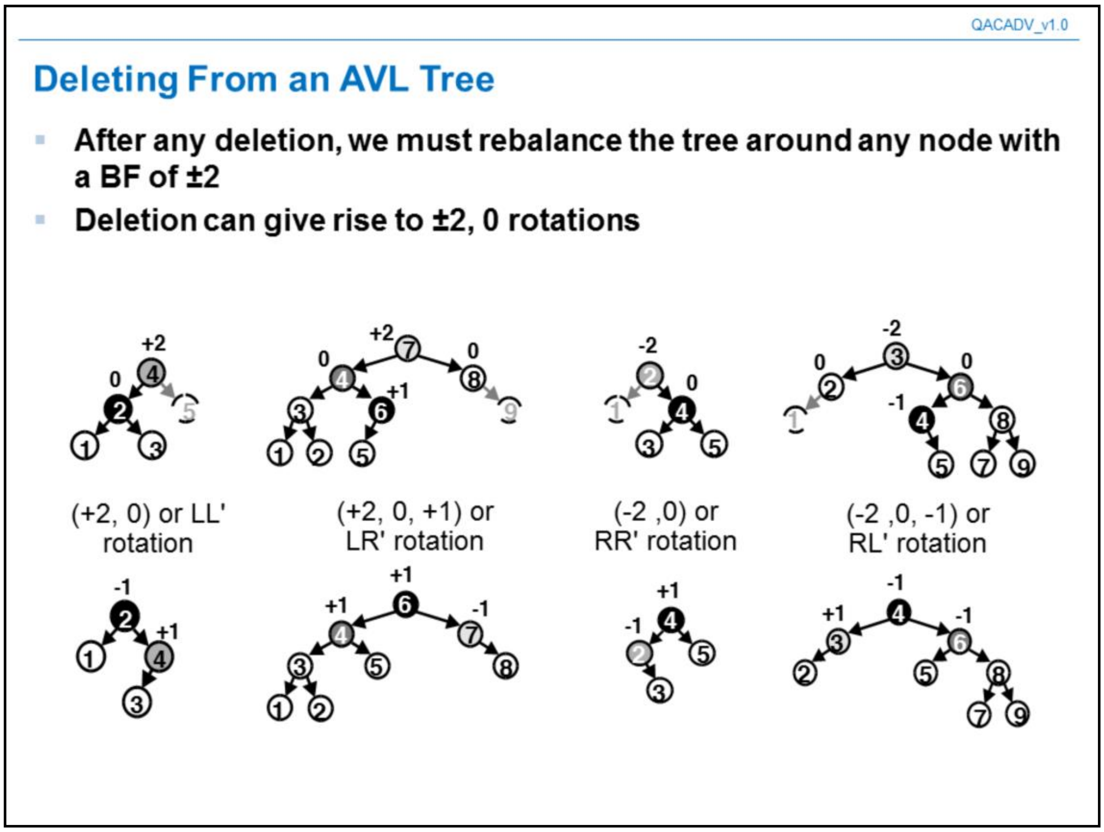

QACADV_v1.0

Inserting into an AVL Tree

```
void  AVLInsert  ( AVLNODE ** parent,  int  ValToInsert,  int  *UnbalancedFlag )
{
      if  (*parent == NULL)  place new node here, set UnbalancedFlag

      else if (ValToInsert  <  Value found in tree)
            invoke AVLInsert( ) with address of left subtree
            if UnbalancedFlag is set (left subtree higher, ADD 1 to BF)
                  BF of -1  becomes 0, BF of 0 becomes 1
                  BF of +1 becomes +2, therefore need a left rotation

      else if (ValToInsert >  Value found in tree)
            invoke AVLInsert( ) with address of right subtree
            if UnbalancedFlag is set (right subtree higher, SUBTRACT  1 from BF)
                  BF of +1 becomes 0, BF of 0 becomes -1
                  BF of -1  becomes -2, therefore need a right rotation

      else value already present in tree so unset the UnbalancedFlag
}
```

The `AvlInsert()` function is shown above in pseudo-code. When a new item is placed in an AVL tree, it may be necessary to rotate various nodes in the tree in order to preserve the balance of the tree.

Full source code for this function is provided on your systems for you to take away at the end of the course.



QACADV_v1.0

Deleting From an AVL Tree

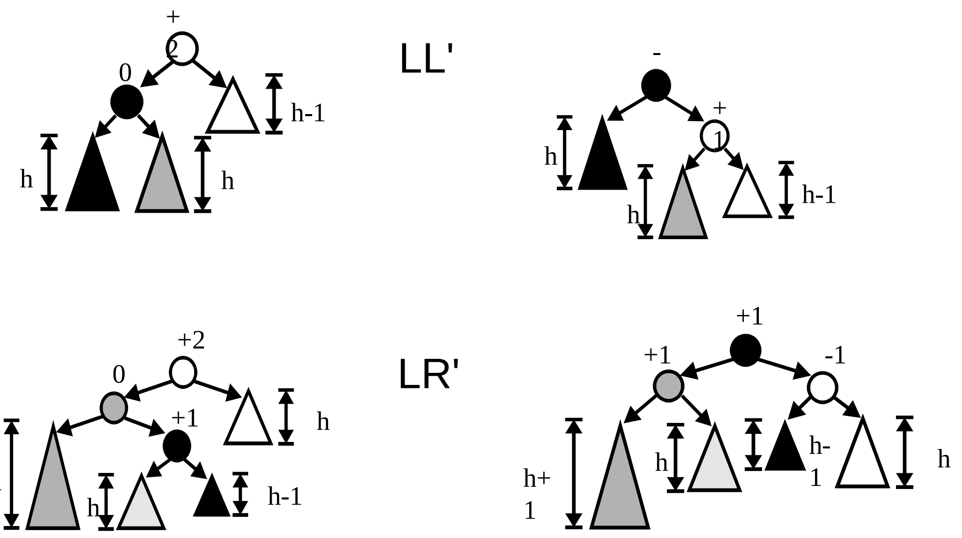
- After any deletion, we must rebalance the tree around any node with a BF of ±2
- Deletion can give rise to ±2, 0 rotations

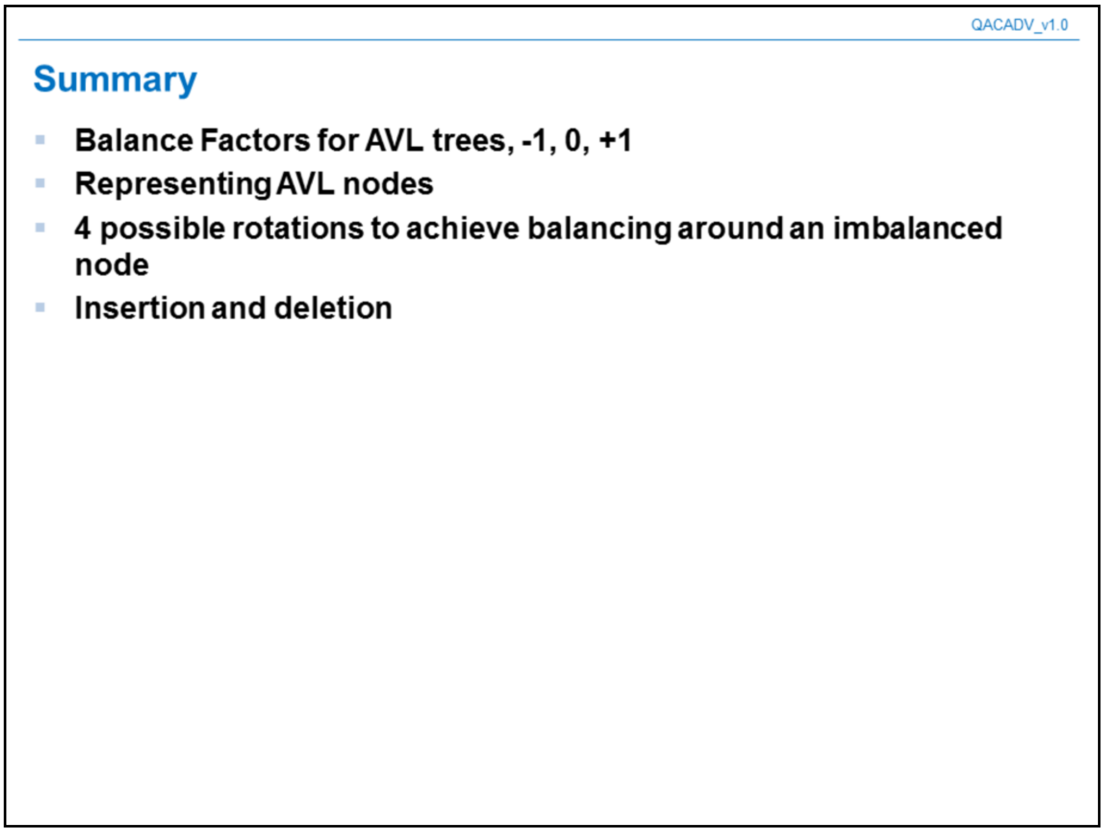
(+2, 0) or LL' rotation
(+2, 0, +1) or LR' rotation
(-2 ,0) or RR' rotation
(-2 ,0, -1) or RL' rotation

Deletion of a node in an AVL tree can cause further imbalances. Rotations must occur around the node causing the imbalance (that is, around the node whose balance factor is +2 or -2). However, we cannot simply use the LL, LR, RR and RL rotations already discussed.

The situation is complicated by so-called "2,0" and "-2,0" rotations. These are caused when a deletion leaves a node with *one* completely balanced subtree. *This situation does not occur during insertion*.

The full picture for the LL' and LR' rotations is shown below (the RR' and RL' rotations are symmetrical):





QACADV_v1.0

Summary

- Balance Factors for AVL trees, -1, 0, +1
- Representing AVL nodes
- 4 possible rotations to achieve balancing around an imbalanced node
- Insertion and deletion