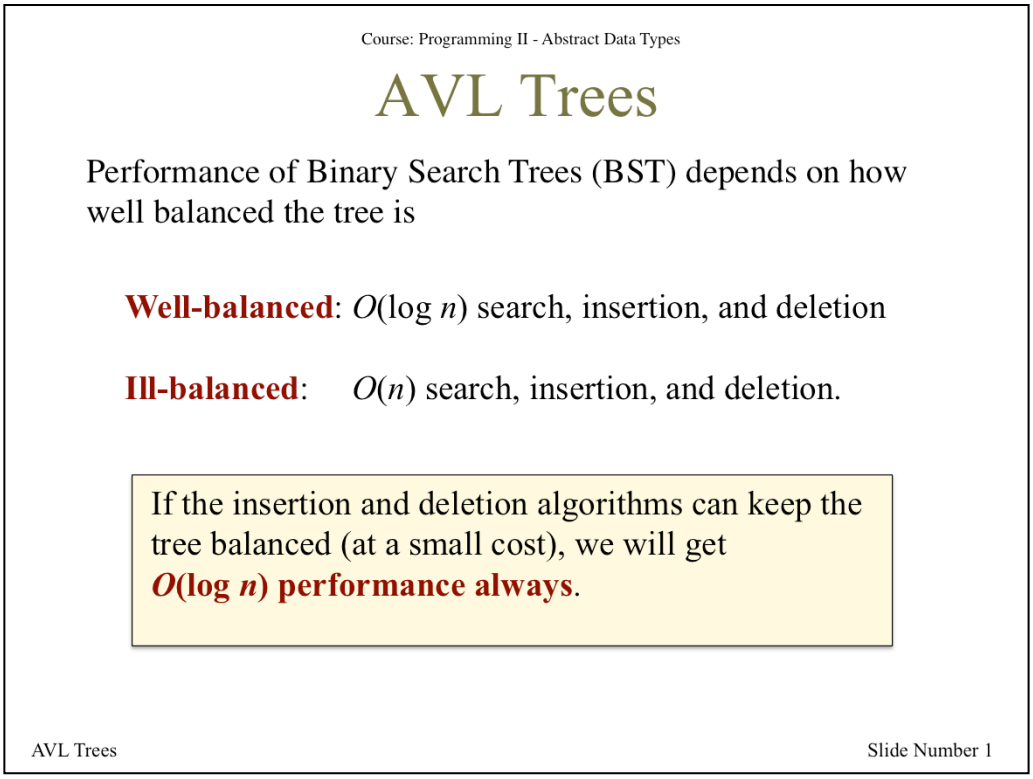
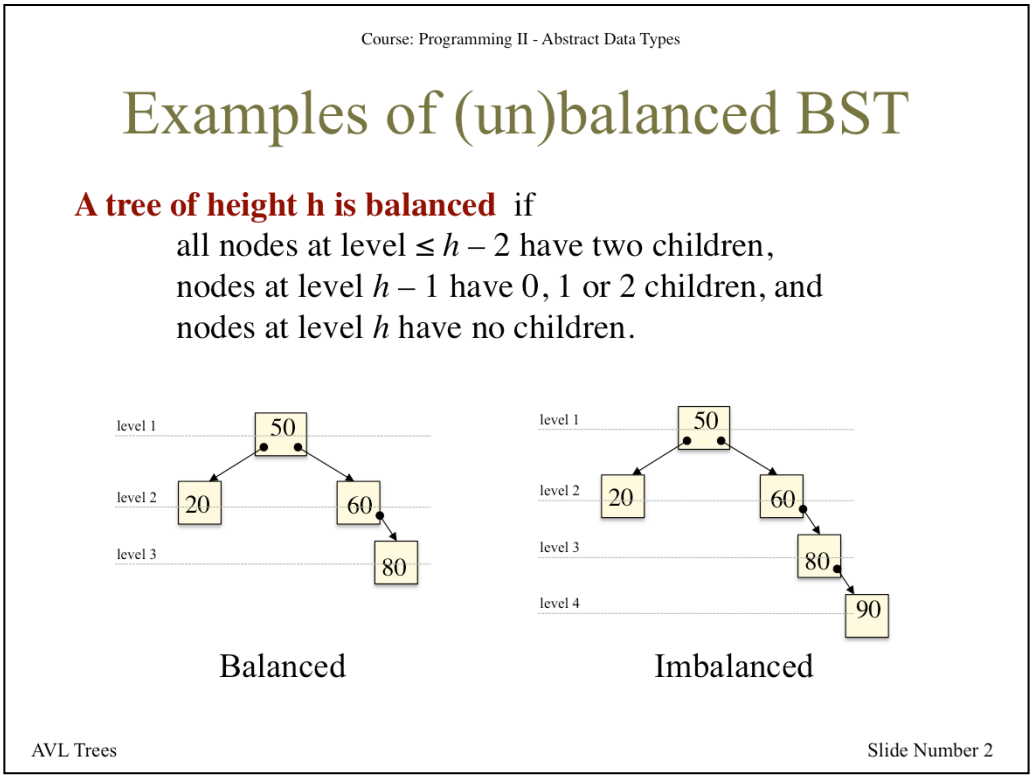


In our lecture about Binary Search Trees (BST) we have seen that the operations of searching for and inserting an element in a BST can be of the order O(log n) (in the best case) and of the order O(n) (in the worst case) when the tree is ill-balanced (i.e. looks like an ordered list).

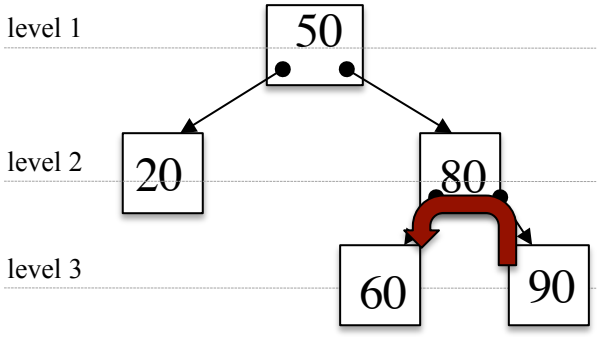
To keep the efficiency of the insertion and retrieval operations closer to O(log n) we need to keep our tree balanced. There are various types of balanced search trees, and each of them include specific operations for insertion and deletion of elements. These include AVL trees, (which are balanced binary search trees), 2-3 trees, 2-4, red-black trees and B trees. We will consider in this lecture the AVL trees, which is the closest to the notion of binary search trees. In the next lectures we will introduce red-black trees and discuss differences between these two types of self-balancing binary search trees. The 2-3 and 2-4 trees are also interesting but they are general search trees where nodes may have more than two children arranged according to a more general ordering relation than the binary search trees. You will hopefully cover the other types of tree structures and algorithms in future courses.

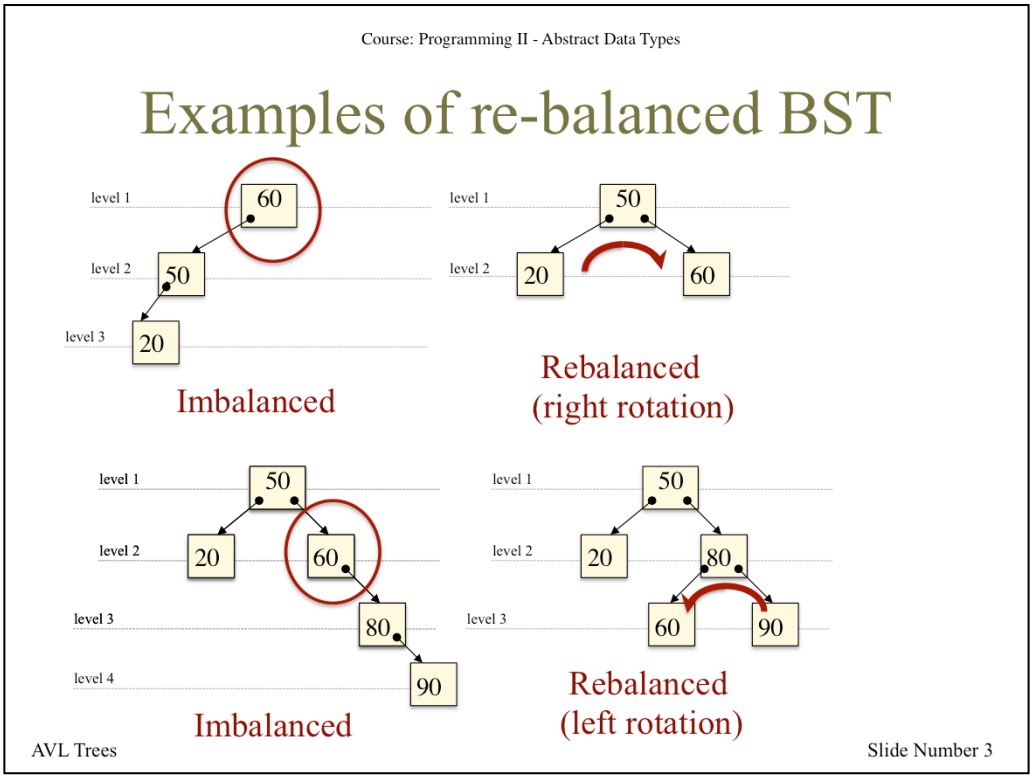
We will begin with a definition of what a balanced search tree is and then we will see how insertion is performed in an AVL tree so to guarantee that the tree after the operation is not only a binary search tree but also balanced. We leave out the case of deletion of a node since it follows pretty much the same strategy we have seen for binary search trees with the additional step of rebalancing the nodes that are modified during the operation.



Given a collection of comparable elements their insertion into a binary search tree can lead to different structures of binary search trees, depending of the ordering in which the elements have been inserted. Not all of these so generated binary search trees are balanced. Recall the definition of height and levels given in the lecture on binary trees. A binary tree of height h is balanced if every node from level 1 to level h-2 of the tree have two children, every node on level h-1 of the tree may have 0, 1 or 2 children and all the node on level h have no children. This is an intuitive definition for any tree to be balanced but it's not fully correct for AVL trees. In the case of AVL trees, **a balanced AVL tree** is one where difference in height of the sub-trees of each node is at most 1. This will become clearer later.

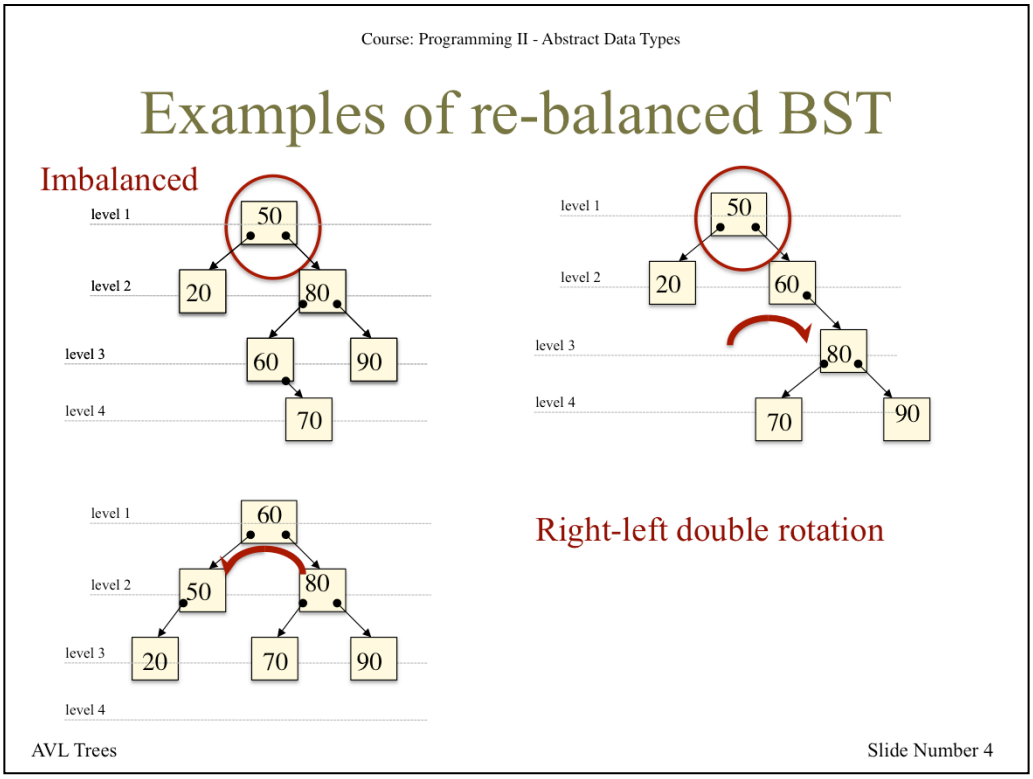
In the above example the balanced tree has height 3 and indeed the root node (node on level h-2) has both children. Nodes on level 2 (h-1) have either zero or 1 child, and the node on level 3 has no children. Also, for each node the difference in height of the sub-trees is no more than 1. The imbalanced tree instead has height 4 and it's imbalanced because it has one node on level 2 (the one including the value 60) that although it's on level h-2 it has only one child. Node 20, on the same level, has no child instead of having both children. The node 60 is the only node in the tree that causes the tree to be imbalanced. The idea of AVL trees is that despite the ordering in which we insert the elements in a binary search tree the insertion operation has to check whether the tree has become imbalanced and then rearrange the nodes in the tree in order to make it balanced again. The AVL algorithm was first developed in 1962 by two mathematicians (Adel'son-Vel'skii and Ladis) hence the name AVL. Applying the AVL re-balancing algorithm to the imbalanced binary search tree given in the above slide would give a re-arranged tree of the form:





To have an idea of how the rebalance could be performed, let's consider the following animation of insertion of elements in a binary search tree. Binary search trees normally include any comparable object but for simplicity we consider the case of tree of numbers.

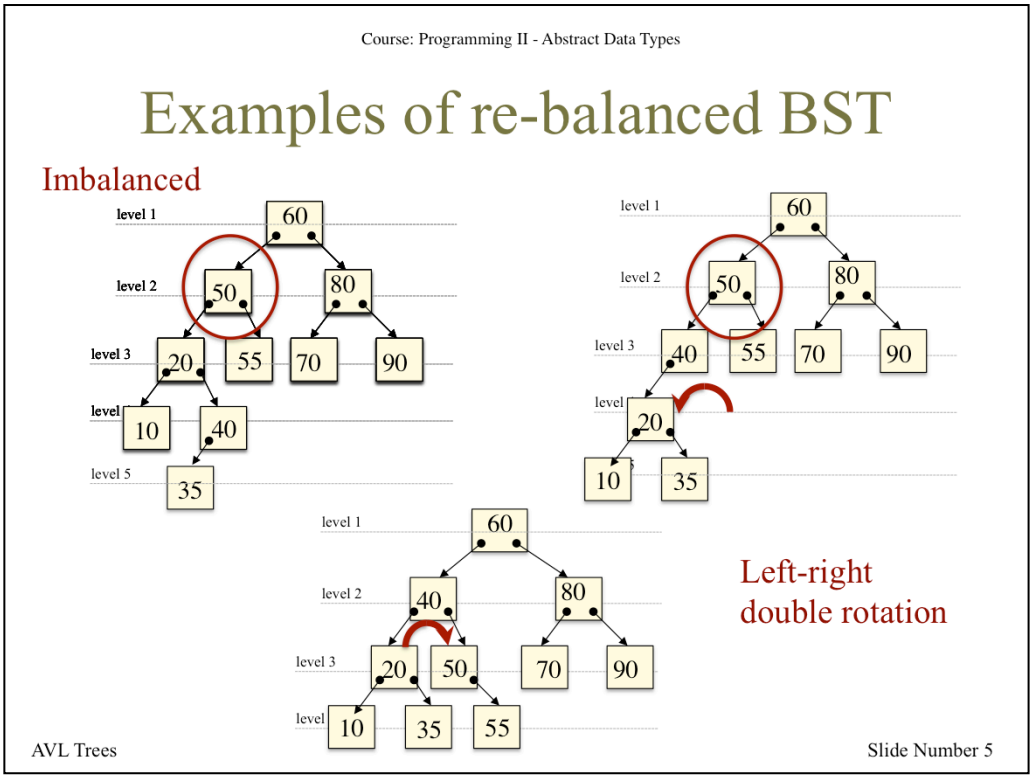
Starting from an empty tree we insert first the number 60 which gives as a tree with just the root. This is clearly a balanced tree. We then insert the node 50. This is added at the bottom of three as we have seen in the implementation of a Binary Search Tree and it is still balanced, then we add the node 20. According to the insert algorithm we have seen in the BST lecture this is added at the bottom of the left subtree as the left most leaf (since it is indeed the smallest number in the tree so far). But now the tree is no longer balanced because the node 60 on level 1 (= 3-2) does not have two children. The node 60 is said to be **not a balanced node** (i.e. the root of an imbalanced BST). The rebalancing could be done by performing a "right rotation" about 50, since the left child of 50 is smaller and it's parent is bigger given the ordering of the BST. If we make 50 to replace 60, and add 60 to it as its right child leaving the left subtree of 50 untouched the BST ordering will be preserved and the tree will become now balanced again. Continuing our example, let's assume that we add now 80 to the (rebalanced tree), which will cause no problem and then add 90. At this point the tree becomes again imbalanced. The non balanced node is now 60 since it is on level 2 (= 4-2) with just one child instead of 2 children. In this case the rebalancing can be achieved by performing a left rotation on the right child of the imbalanced node. The node 80 can replace 60 and 60 can be added to 80 as its left child leaving the right subtree of 80 untouched. This again will rebalance the tree and still preserve the BST ordering.



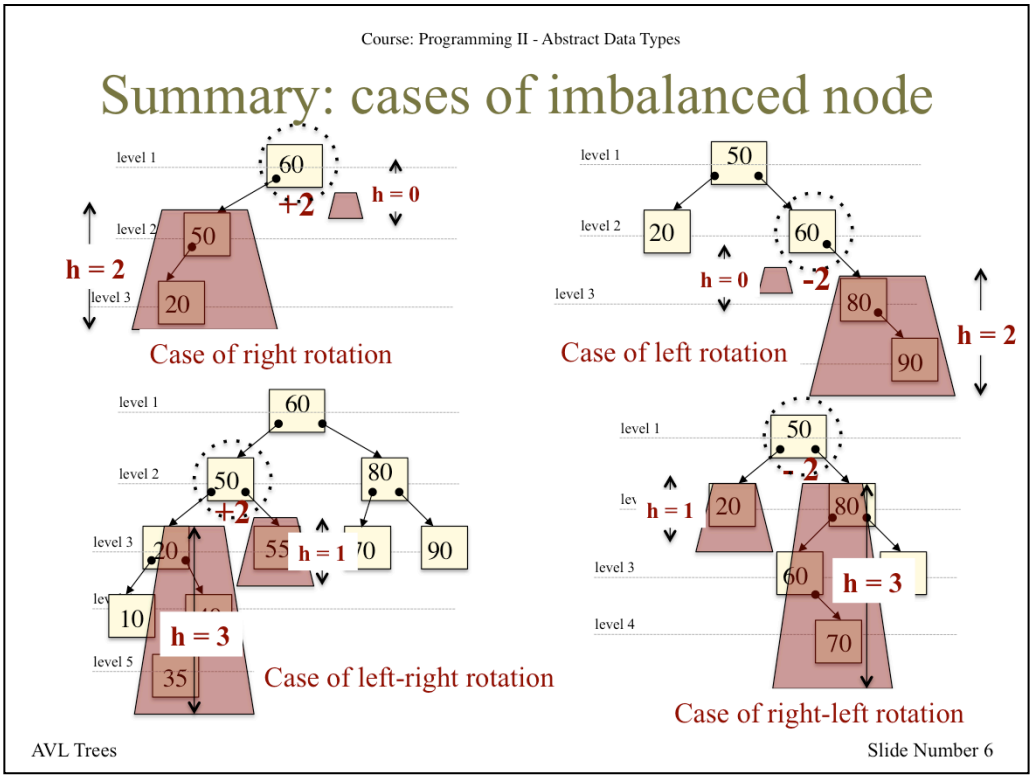
The single right and left rotation mechanisms do not resolve all the possible ways in which a tree can become imbalanced. In the previous slide the node containing 60 becomes imbalance after the addition of a new element to the left sub-tree of its left child. Similarly, it becomes imbalanced after the addition of an element to the right sub-tree of its right child. In these cases single rotation mechanism is sufficient provided that the rotation is applied to the imbalanced node that is closer to the point of the insertion.

This slide shows other two cases of imbalanced nodes. In the top part of the slide the node containing 50 becomes imbalanced after 70 is added to the left sub-tree of its right child. In this case two steps rotation are needed. The first is a right rotation in the attempt to re-establish the balance on its right child node. And then a left rotation is applied since it's right sub-tree is still imbalance. Note that each of these steps preserves the ordering of the binary search tree.

Next slide shows the last case.



In this case, the top left tree has node containing 50 that becomes imbalanced after the addition of the node containing 35. This is added to the right sub-tree of the left child of the imbalanced node (i.e. 50.). Also in this case two steps rotation are needed, but this time the steps are left rotation (which produces the BST on the top right) followed by a right rotation (which produces the bottom BST). Again, each of these steps preserves the ordering of the binary search tree.



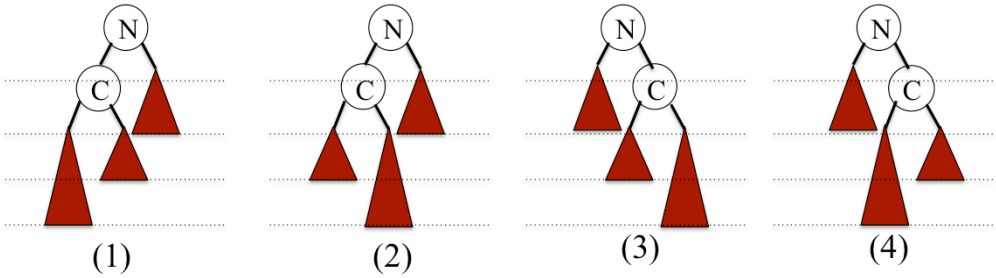
In each of these cases the difference between the heights of the right subtree and that of the left subtree of the imbalanced node is more than 1. This is when we say that the node is imbalance and rebalancing is needed. For instance in the top left tree the height of the left subtree of node 60 is 2 and the height of the right subtree of node 60 is 0. So their difference is 2. However, to distinguish these four cases we need to have a way of distinguishing in which subtree the insertion has been performed in order to decide whether to perform a right(left double) rotation or a left(righ double) routation. To do this we assume that we are performing always the difference between height of the left subtree and height of the right subtree. When the difference is positive and more than 1 then the insertion has been done on the left subtree of the imbalanced node. When the difference is negative and less than -1 then the insertion has been made on the right subtree of the imbalanced node. In the first case we need to make either a right rotation or a left right rotation. In the second case we need to make a left rotation or a right left rotation.

Let consider the first case: the left subtree of the imbalanced node is taller than the right subtree. The the addition has been made in the left subree, i.e. in the subtree with root the left child of the imbalanced node. But if we consider the left subtree, again we don't know whether the addition has been made in its left subtree or in its right subtree. So we need to check whether the height of the **left subtree of the left child of the imbalanced node** is taller than the right subtree of the left child of the imbalanced node. If it is so, than the addition has been made in the left subtree of the left child, so we need just right rotation. If the right subtree of the left child of the imbalanced node is taller than its left tree than the addition has been made on the right subtree of the left child of the imbalanced node, and we need a left-right double rotation. Similarly for the other two cases.

Rebalancing

To correct an imbalanced node N in an AVL tree:

1) right rotation: if addition is in left subtree of N's left child
2) left-right rotation: if addition is in right subtree of the N's left child
3) left rotation: if addition is in right subtree of N's right child
4) right-left rotation: if addition is in left subtree of N's right child

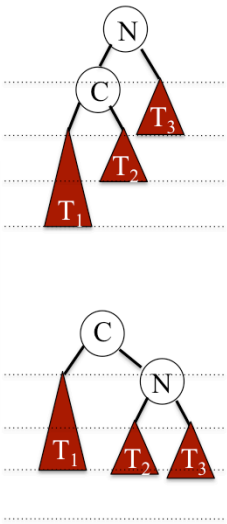


This slide summarises the four cases of rotation and when they should occur. An abstraction of the AVL tree structure that shows the four cases of imbalance is given here. Note that node N here denotes the imbalanced node closest to the insertion point. Don't assume N to be the root of the initial AVL tree. It can be any node in the BST.

Right rotation

Correct imbalance at given nodeN caused by addition in the left subtree of nodeN's left child

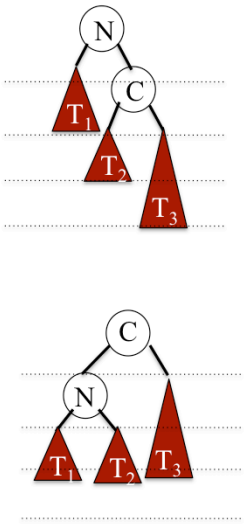
```
rotateRight(TreeNode nodeN){
      nodeC = nodeN's leftchild
      set nodeN's left child to nodeC's
          right child
      set nodeC's right child to nodeN
      return nodeC
}
```



Left rotation

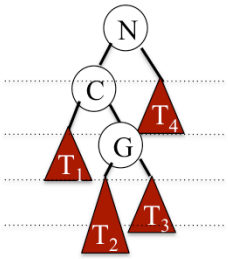
Correct imbalance at given nodeN caused by addition in the right subtree of nodeN's right child

```
rotateLeft(TreeNode nodeN){
    nodeC = nodeN's rightchild
    set nodeN's right child to nodeC's
          left child
      set nodeC's left child to nodeN
      return nodeC
}
```

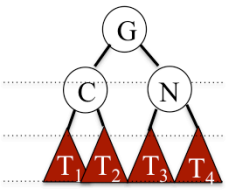


Left-Right rotation

Correct imbalance at given nodeN caused by addition in the right subtree of nodeN's left child



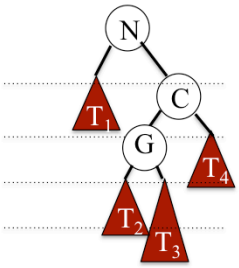

```
rotateLeftRight(TreeNode nodeN){
   nodeC = nodeN's leftchild;
   newLeft = rotateLeft(nodeC);
   set nodeN's left child to newLeft;
   return rotateRight(nodeN);
}
```

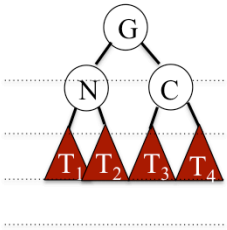


Right-Left rotation

Correct imbalance at given nodeN caused by addition in the left subtree of nodeN's right child

```
rotateRightLeft(TreeNode nodeN){
   nodeC = nodeN's rightchild;
   newRight = rotateRight(nodeC);
   set nodeN's right child to newRight;
   return rotateLeft(nodeN);
}
```





…Putting all together

```
reBalance(TreeNode nodeN){
  if (nodeN's left subtree is taller than right
          subtree by more than 1){
     if (nodeN's leftchild has left subtree
                taller than its right subtree){
          rotateRight(nodeN);
     else rotateLeftRight(nodeN);
  else{ if (nodeN's right subtree is taller than
           left subtree by more than 1){
         if (nodeN's rightchild has right subtree
                taller than its left subtree){
             rotateLeft(nodeN);
         else rotateRightLeft(nodeN);
        } // else nodeN is balanced
   }
  return nodeN.
}
```


This is the procedure that puts together all the four cases of rebalancing discussed so far. This method is called by the `inserElem` method (see slide 16). It checks that the given nodeN is imbalanced. This can be because the height of the left subtree is more than 1 taller than the height of the right subtree (first if case). In this case the insertion has been made by the `inserElem` method in the left subtree of nodeN. But we need to see whether it has been inserted in the left or right subtree of the left child of nodeN. This is the second if case.

The other situation (external else case) is when the insertElem has inserted the new node in the right subtree of NodeN. Again this case we need also to distinguish the cases when it has been inserted in the left subtree or in the right subtree of the right child of nodeN. This corresponds to the other if-then-else statement inside the external else case.

If none of the two cases applies than nodeN is not imbalanced and this procedure returns nodeN unmodified. In the next slide we discuss how to check the height difference.

Computing the height of the subtrees

A key operation in the rebalancing algorithm is computing the difference of the heights of the left and right subtrees

Define an auxiliary method:

```
private int getHeightDifference( )
//post: Returns a number greater than 1 when the height of the left subtree is
//post: more than 1 taller than the height of the right subtree.
//post: Returns a number less than -1 when the height of the right subtree is
//post: more than 1 taller than the height of the left subtree.
//post Returns 0 when the heights of the left and  right subtrees are equal
```

- as auxiliary method in AVL tree, or
- as auxiliary method of the class TreeNode, but node needs to keep track of its height.

As shown in the previous slide of the rebalancing algorithm, a key operation is computing the difference of the heights of the left and right subtrees of the imbalanced node. This will first tell us if the node in question in actually imbalanced and also it should tell us which of its subtree is higher in order to choose which of the four cases of rebalance to apply.

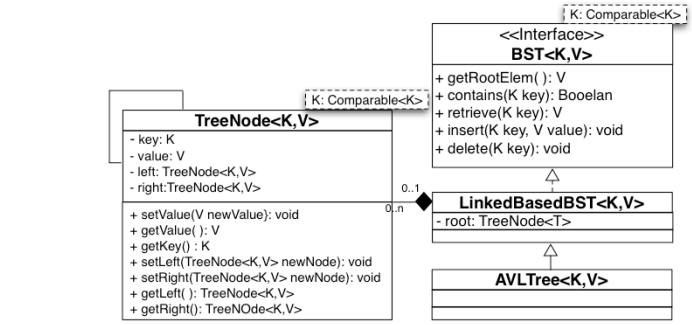
You can define your own access procedure for getting the height of the two subtrees of a given node. This resembles somehow to the notion of a compareTo method that according to the sign of the method returns it can tell us which of the two subtrees is higher.

In this slide I have given an example of how you could specify such a method `getHeightDifference`. You can assume that you are computing always the height of the left subtree minus the height of the right subtree. If the result is 0 then the two subtres have the same height. If the result is greater than 1 than the left subtree is taller and the right and the current node is imbalanced. If the result is less than -1 then the right subtree is taller than the left subtree and the current node is imbalanced.

The other question you need to decide is where to include such an auxiliary method. You can either include it in the implementation of the AVL tree but then each time you go through the recursive call for inserting a node you need to re-compute it for each node you visit to decide whether it is imbalanced or not. A more efficient approach would be to include the `getHeightDifference` method in the class TreeNode. This requires the use of an additional attribute in this class which is the height of the node. Remember that the operations of `setLeft()` and `setRight()` will have to update the height of the node when a new left child or a new right child is set (respectively).

Course: Programming II - Abstract Data Types

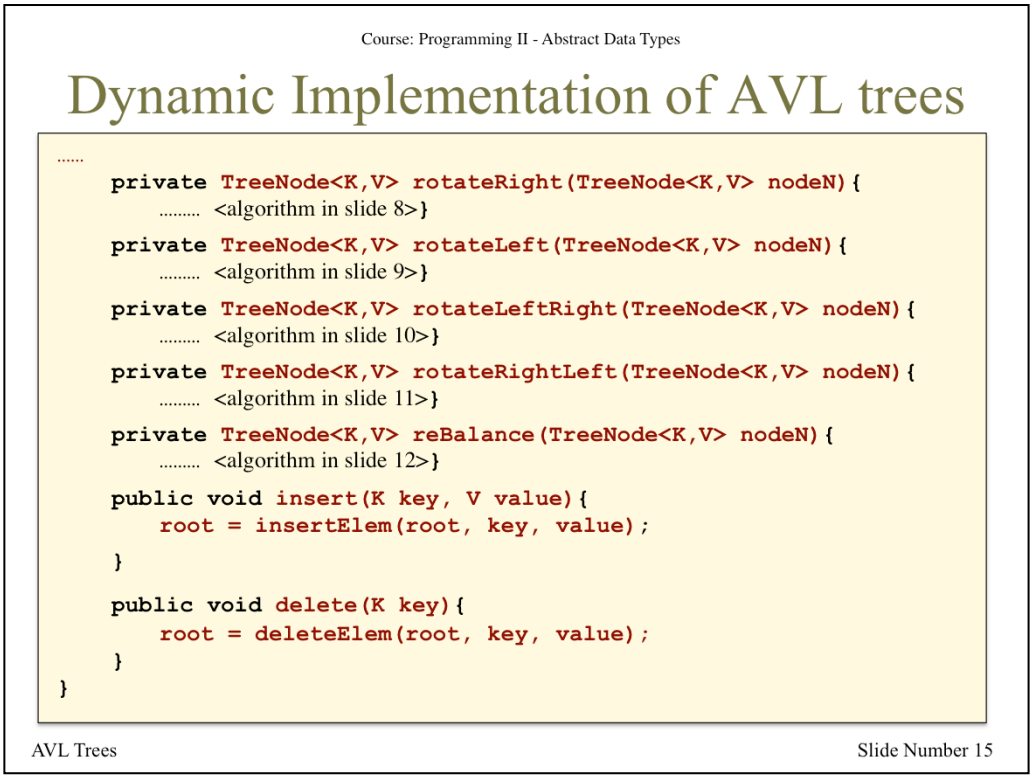
Dynamic Implementation of AVL trees




```
class AVLTree<K extends Comparable<K>,V>
                                    extends LinkedBasedBST<K,V>{
    public AVLTree(){
        super();
    }
    public AVLTree(K key, V, Value){
        super(key, value);
    }
}                                                    continue.....
```

AVL Trees Slide Number 14

You can think of implementing an AVL tree as a subclass of a Binary Search Tree object. The interface for the AVL tree is the same as that of a Binary Search Tree, the only difference is that its methods `insert(K key, V value)` and `delete(K key)` are different as they will need to rebalance the tree and they will override the implementation of their respecting procedures in the superclass `LinkedBasedBST`.



You should be able to implement these auxiliary methods from the algorithms described in the previous slides. You are asked to implement the full class of an AVL tree in your Tutorial 4. We'll see what happens with the auxiliary methods `insertElem`. The method `delete` follows the same strategy as described in the Binary Search with the difference that you have to rebalance a node (root of a subtree) whenever you modify it and all the way up from this node to its parent node and so on until you reach the root of the initial given tree.

Implementing `insertElem`

```
TreeNode<K,V> insertElem(TreeNode<K,V> node, K key, V newValue)
   if (node is null) {
      Create a new node and let node reference it;
      Set the value in the new node to be equal to newValue;
      Set the references in the new node to null;
      Set the height of the node to be 1;
   }
  else if (key == node.getKey( )){
          replace the value in node with newValue;}
       else if (key < node.getKey( )){
              TreeNode newLeft = insertElem(node.getLeft( ), key,
                                                    newValue)
              node.setLeft(reBalance(newLeft)); }
            else {
              TreeNode newRight = insertElem(node.getRight( ), key,
                                                    newValue)
              node.setRight(reBalance(newRight)); }
  return node;
}
```


Inserting an element into an AVL tree is just like inserting an element in a BST but with a rebalancing step. This private auxiliary procedure `insertElem` is in charge of locating the place where to insert a node given its key. It follows the same algorithm of searching in a Binary Search Tree that we have seen in Unit 5, but it has also to perform rebalancing when the node is inserted. Because of the recursive call mechanism the rebalancing operation is call at each level starting from the parent of the node that is inserted. Note that changes will only take place when the node in question is imbalanced. Otherwise the rebalancing method leaves the node as it is.

The final return node object of this procedure is set to be the new root of the AVL tree object, and itself needs to be checked whether it has to be rebalanced. But this is done in the caller method `insert`.

Implementing `deleteElem`

```
TreeNode<K,V> deleteElem(TreeNode<K,V> node, K key)
// post: deletes the node from the BST, whose key is equal key and returns the
// post: root node of the resulting tree after being rebalanced.
 if (node is null)
      throw TreeException;
 else if (node.getKey( ) == key)
        node = deleteNode(node);
        return node;
     else if (node.getKey( ) > key){
       TreeNode<K,V> newLeft=deleteElem(node.getLeft( ),key)
       node.setLeft(newLeft);
       return reBalance(node);}
          else
       TreeNode<K,V> newRight=deleteElem(node.getRight( ),key)
       node.setRight(newRight);
       return reBalance(node);
}
```


Note again, this procedure of deleting a node in an AVL tree follows the same method used in a Binary Search Tree. The only difference is that you need to rebalance a node once it has been modified with a `setLeft()` or a `setRight()` access procedure. This is because its respective right or left subtree might have become imbalanced after the deletion of the node.

In the case when the node is found, its deletion depends on the function `deleteNode`. This is again the same method we have discussed in the Binary Search Tree lecture. Since it also modifies subtrees (because it deletes the left most leaf of the right tree of the given found node, node itself can have become imbalance and so each nodes within the the path that goes from it to the deleted leaf may need to be rebalanced. So `deleteNode` also uses a rebalance call. This is shown in the next slide.

Implementing "deleteNode"

```
TreeNode<K,V> deleteNode(TreeNode<K,V> node)
// post: delete the given node and returns the modified tree rebalanced
 if (node is a leaf)
      return null;
 else{ if (node has only left child)
           return node.getLeft( );
       else if (node has only right child)
                return node.getRight( );
            else{
                 replacementNode = findLeftMost(node.getRight( ));
                 newRight = deleteLeftMost(node.getRight( ));
                 replacementNode.setRight(newRight);
                 replacementNode.setLeft(node.getLeft( ));
                 return reBalance(replacementNode);
           }
```

Implementing "deleteLeftMost"

```
TreeNode<K,V> deleteLeftMost(TreeNode<K,V> node) {
// post: deletes the left most node in the tree rooted at node.
// post: returns the root of the modified sub-tree, rebalanced
   if (node.getLeft( ) is null)
         return node.getRight( );
   else { newChild = deleteLeftMost(node.getLeft( ));
          node.setleft(newChild);
          return reBalance(node);
   }
}
```

The auxiliary method **`findLeftMost`** is identical to the one for Binary Search Tree.

Summary

- AVL trees are special binary search trees that guarantees the tree to be balanced after each insertion and deletion of a node, whilst preserving the ordering over the keys of the nodes in the tree.
- Althought the rebalance procedure is called at each level of the recursion (when inserting a new node), the rebalancing of the tree occurs at most once. Most calls to `rebalance` simply check whether rebalancing is needed.
- The computational time of `insert` in an AVL tree is approximately closed to the computational time of `insert` in an Binary Search tree.