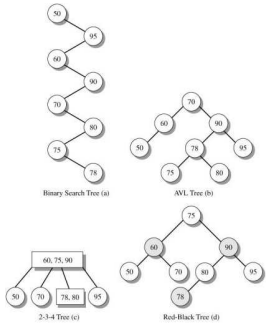
Vyvážené stromy

- AVL

Balanced Binary Search Trees

- Red-black or AVL trees balance a binary search tree so it more nearly resembles a complete tree.
  - An AVL tree maintains height-balance at each node. For each node, the difference in height of its two subtrees is in the range -1 to 1.
  - Red-black trees are a representation of a 2-3-4 tree and feature nodes that have the color attribute BLACK or RED. The tree maintains a measure of balance called the BLACK-height.

Balanced Binary Search Trees (continued)



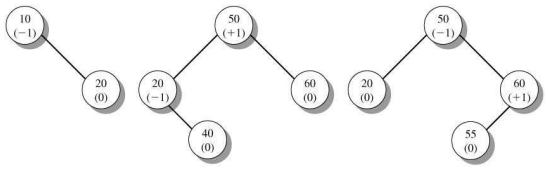
AVL Trees

- For each AVL tree node, the difference between the heights of its left and right subtrees is either -1, 0 or +1.

  balanceFactor = height(left subtree) - height(right subtree)

  - If balanceFactor is positive, the node is "heavy on the left" since the height of the left subtree is greater than the height of the right subtree.
  - With a negative balanceFactor, the node is "heavy on the right."
  - A balanced node has balanceFactor = 0.

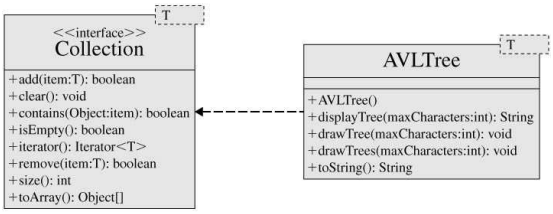
AVL Trees (continued)



AVLTree Class

- The AVLTree class implements the Collection interface and builds an AVL tree. It has the same public methods as the STree class.

AVLTree Class (continued)

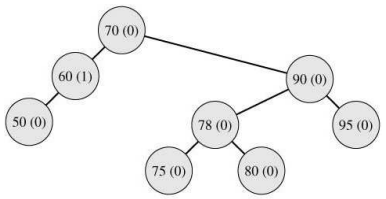


AVLTree Class (continued)

```
String[] stateList = {"NV", "NY", "MA", "CA", "GA"};
      int[] arr = {50, 95, 60, 90, 70, 80, 75, 78};
int i;
// avlTreeA and avlTreeB are empty collections
AVLTree<String> avltreeA = new AVLTree<String>();
      AVLTree<Integer> avltreeB = new AVLTree<Integer>();
for (i = 0; i < stateList.length; i++)
            avltreeB.add(stateList[i]);
for (i = 0; i < arr.length; i++)
            avltreeB.add(arr[i]);

// output list of elements
System.out.println("States: " + avltreeA);
// display the tree
System.out.println(avltreeB.displayTree(2));
avltreeB.drawTree(2);
```

AVLTree Class (continued)

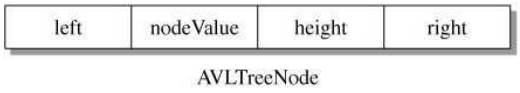
```
Run:
States: [CA, GA, MA, NV, NY]
            70(-1)
        60(1)                   90(1)
   50(0)                78(0)           95(0)
                75(0)       80(0)
```



Implementing the AVLTree Class

- An AVLNode contains the node value, references to the node's children, and the height of the subtree.

height(node) = max (height(node.left), height(node.right)) + 1;



AVLTreeNode

AVLNode Class

```
// declares a binary search tree node object
private static class AVLNode<T>
{
   // node data
   public T nodeValue;

   // child links and link to the node's parent
   public AVLNode<T> left, right;

   // public int height;
   public int height;
```

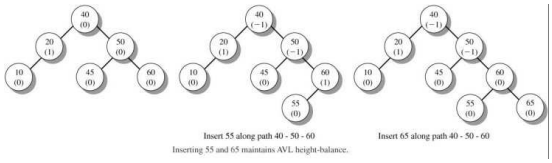
AVLNode Class (concluded)

```
   // constructor that initializes the value,
   // balance factor and parent fields and sets
   // the link fields left and right to null
   public AVLNode (T item)
   {
      nodeValue = item;
      left = null;
      right = null;
      height = 0;
   }
}
```

Implementing the AVLTree Class (continued)

- The recursive addNode() algorithm moves to the insertion point using the usual rules for a binary search tree.
- The addition of an element may cause the tree to be out of balance. The recursive addNode() algorithm reorders nodes as it returns from function calls.

Implementing the AVLTree Class (continued)

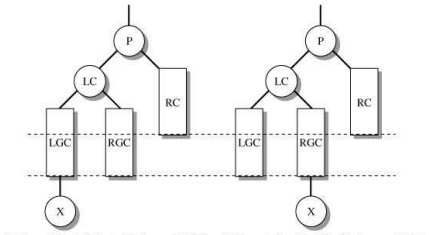


Inserting 55 and 65 maintains AVL height-balance.

Implementing the AVLTree Class (continued)

- Inserting a node on the left subtree of P may cause P to be "heavy on the left" (balance factor +2). The new node is either in the outside or inside grandchild subtree.

Implementing the AVLTree Class (continued)



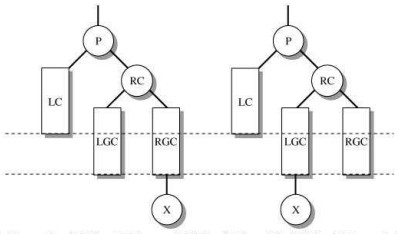
(a) Insert in left (outside) grandchild Left subtree of LC (b) Insert in right (inside) grandchild Right subtree of LC

Inserting X imbalances the parent node P with balance factor 2.

Implementing the AVLTree Class (continued)

- Inserting a node on the right subtree of P may cause P to be "heavy on the right" (balance factor -2). The new node is either in the outside or inside grandchild subtree.

Implementing the AVLTree Class (continued)



(a) Insert in right (outside) grandchild Right subtree of RC (b) Insert in left (inside) grandchild Left subtree of RC

Inserting X imbalances the parent node P with balance factor – 2.

AVL Tree Rotations

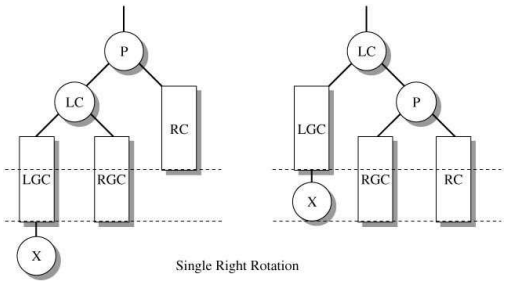
- When correcting an imbalance for a parent node, addNode() uses either a single or a double rotation. The rotation methods need to determine the height of left and right subtrees of a node.

```
private static <T> int height(AVLNode<T> t)
{
   if (t == null)
      return -1;
   else
      return t.height;
}
```

AVL Tree Rotations (continued)

- A *single right rotation* rotates the nodes so that the left child (LC) replaces the parent, which becomes a right child. In the process, the nodes in the right subtree of LC (RGC) are attached as a left child of P. This maintains the search tree ordering since nodes in the right subtree are greater than LC but less than P.

AVL Tree Rotations (continued)



Single Right Rotation

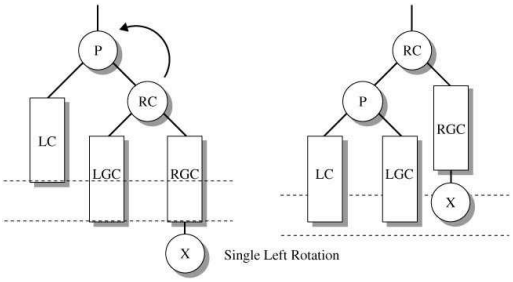
singleRotateRight()

```
// perform a single right rotation for parent p
private static <T> AVLNode<T> singleRotateRight(
AVLNode<T> p)
{
   AVLNode<T> lc = p.left;

   p.left = lc.right;
   lc.right = p;
   p.height = max( height( p.left ),
         height( p.right ) ) + 1;
   lc.height = max( height( lc.left ),
         lc.height ) + 1;

   return lc;
}
```

AVL Tree Rotations (continued)

- A symmetric single left rotation occurs when the new element enters the subtree of the right outside grandchild. The rotation exchanges the parent and right child nodes, and attaches the subtree LGC as a right subtree for the parent node.

AVL Tree Rotations (continued)



Single Left Rotation

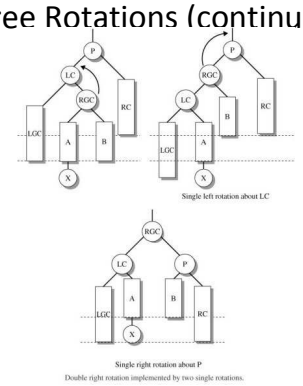
singleRotateLeft()

```
// perform a single left rotation for parent p
private static <T> AVLNode<T> singleRotateLeft(
AVLNode<T> p)
{
   AVLNode<T> rc = p.right;

   p.right = rc.left;
   rc.left = p;
   p.height = max(height(p.left),
         height(p.right)) + 1;
   rc.height = max(height(rc.right),
         rc.height) + 1;

   return rc;
}
```

AVL Tree Rotations (continued)

- When a new item enters the subtree for an inside grandchild, the imbalance is fixed with a double rotation which consists of two single rotations.

AVL Tree Rotations (continued)



doubleRotateRight()

```
// perform a double right rotation for parent p
private static <T> AVLNode<T> doubleRotateRight(
AVLNode<T> p)
{
   p.left = singleRotateLeft(p.left);
   return singleRotateRight(p);
}
```

doubleRotateLeft()

```
// perform a single left rotation for parent p
private static <T> AVLNode<T> doubleRotateLeft(
AVLNode<T> p)
{
   p.right = singleRotateRight(p.right);
   return singleRotateLeft(p);
}
```

addNode()

- Recursive addNode() descends to the insertion point and inserts the node. As it returns, it visits the nodes in reverse order, fixing any imbalances using rotations.

addNode() (continued)

```
private AVLNode<T> addNode(AVLNode<T> t, T item)
{
   if( t == null )
      t = new AVLNode<T>(item);
   else if (((Comparable<T>)item).compareTo(
         t.nodeValue) < 0)
   {
      t.left = addNode( t.left, item);

      if (height(t.left) - height(t.right) == 2 )
         if (((Comparable<T>)item).compareTo(
               t.left.nodeValue) < 0)
            t = singleRotateRight(t);
         else
            t = doubleRotateRight(t);
   }
   else if (((Comparable<T>)item).compareTo(
         t.nodeValue) > 0 )
   {
      t.right = addNode(t.right, item );
```

addNode() (concluded)

```
      if (height(t.left) - height(t.right) == -2)
         if (((Comparable<T>)item).compareTo(
               t.right.nodeValue) > 0)
            t = singleRotateLeft(t);
         else
            t = doubleRotateLeft(t);
   }
   else
      // duplicate; throw IllegalStateException
      throw new IllegalStateException();

   t.height = max( height(t.left),
         height(t.right) ) + 1;

   return t;
}
```

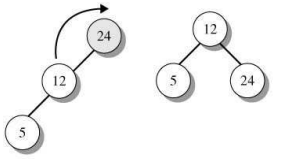
add()

- Method add() assures that item is not in the tree, calls addNode() to insert it, and then increments treeSize and modCount.

add() (concluded)

```
// if item is not in the tree, insert it and
// return true; if item is a duplicate, do not
// insert it and return false
public boolean add(T item)
{
   try
   {
      root = addNode(root, item);
   }

   catch (IllegalStateException ise)
   { return false; }

   // increment the tree size and modCount
   treeSize++;
   modCount++;

   // we added a node to the tree
   return true;
}
```

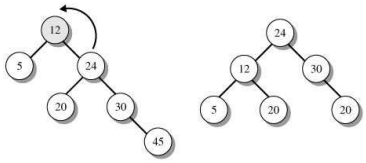
Example - Building an AVL Tree



Insert 24 12 5 Single Rotate Right (P = 24)

Insert the first three elements 24, 12, and 5.
At 5, node 24 has balance factor 2.

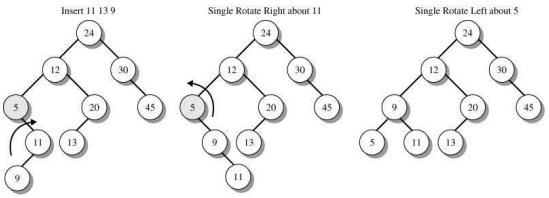
Example - Building an AVL Tree (continued)



Insert 30 20 45 Single Rotate Left (P = 12) attach 20 as right child of 12

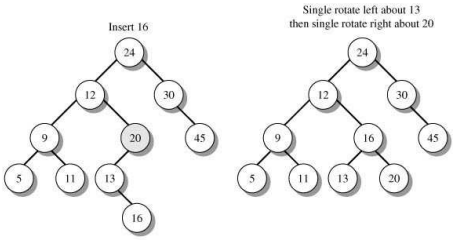
Insert the next three elements 30, 20, and 45.
At 45, node 12 has balance factor -2.

Example - Building an AVL Tree (continued)



Insert the three elements 11, 13, and 9. At 9, node 5 has balance factor −2.

Example - Building an AVL Tree (concluded)



Insert the last element 16. Node 20 has balance factor +2.

Efficiency of AVL Tree Insertion

- A mathematical analysis shows that

  $\text{int}(\log_2 n) \leq \text{height} < 1.4405 \log_2(n+2) - 1.3277$

  indicating that the worst case running time for insertion is $O(\log_2 n)$. The worst case for the difficult deletion algorithm is also $O(\log_2 n)$.