Balanced Binary Search Tree (AVL)

- Improves on the binary search tree by always guaranteeing the tree is height balanced.
  - Allows for more efficient operations.
  - Developed by Adel'son-Velskhii and Landis in 1962.

- balanced – the heights of the left and right subtrees of every node differ by at most 1.
Balance Factor

- Associated with each node and indicates the height difference between the left and right branch.
  - left-high
  - equal-high
  - right-high
AVL Operations

- Search and traversal are the same as with the binary search tree.
- Insertion and deletion must be modified.
  - Maintain the balance property as keys are added and/or removed.
  - Ensures height never exceeds $1.44 \log n$.
  - Provides $O(\log n)$ worst case time.
AVL Insertions

- Begins with the same process used with a BST.
  - Must rebalance the tree if the insertion causes it to become unbalanced.
- Example: insert key 120
AVL Insert Example

- Suppose we add key 28 to the AVL tree.
AVL Rotation

- Multiple subtrees can become unbalanced after an inserting a new key.
  - Limited to the nodes along the insertion path.
  - Balance factors are adjusted during the recursion unwinding.
- **pivot node** – root node of the first out of balance subtree encountered.
AVL Rotation

- An AVL subtree is rebalanced by performing a rotation around the pivot node.
  - Rearrange links of the pivot node, its children and at most one of its grandchildren.
  - There are four possible cases.
AVL Rotation (Case 1)

- The balance factor of the pivot node (P) is left-high before the insertion into the left subtree (C) of P.
AVL Rotation (Case 1)

- The pivot node has to be rotated right over its left child.
  - P becomes the right child of C.
  - Right child of C becomes the left child of P.
AVL Rotation (Case 1)

- Result after the right rotation.
AVL Rotation (Case 2)

- Involves three nodes: pivot (P), left child (C) of P and the right child (G) of C.
  - Balance factor of P is left-high before the insertion.
  - Inserted into either left or right subtree of G.
AVL Rotation (Case 2)

- Requires a double rotation:
  - node C has to be rotated left over node G.
  - pivot node has to be rotated right over its left child.

- Link modifications:
  - right child of G becomes the new left child of P.
  - left child of G becomes the new right child of C.
  - C becomes the new left child of G.
  - P becomes the new right child of G.
AVL Rotation (Case 2)
AVL Rotation (Case 2)

- Result after the two rotations.
AVL Rotation (Case 3)

- A mirror image of the first case.
  - P is right-high.
  - The new key is inserted in the right subtree of C.
AVL Rotation (Case 4)

- A mirror image of the second case.
  - P is right-high.
  - G is the left child of C instead of the right.
New Balance Factors

- The balance factors of the nodes along the insertion path may have to be modified.
  - Performed in reverse order as the recursion unwinds.
- The new balance factor of a node depends on its current factor and the subtree into which the new node was inserted.

<table>
<thead>
<tr>
<th>current factor</th>
<th>left subtree</th>
<th>right subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;&gt;</td>
<td>=</td>
</tr>
<tr>
<td>=</td>
<td>&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>=</td>
<td>&lt;&lt;</td>
</tr>
</tbody>
</table>
New Balance Factors

- The balance factors of the nodes impacted by a rotation have to be modified.
- The new balance factors depend on the case that triggered the rotation.

<table>
<thead>
<tr>
<th>Case</th>
<th>G</th>
<th>new P</th>
<th>new L</th>
<th>new R</th>
<th>new G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>=</td>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>=</td>
<td>=</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>=</td>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>=</td>
<td>=</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>&gt;</td>
<td></td>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>
Building an AVL Tree

- Suppose we want to build an AVL tree from the list.

\[
60 \quad 25 \quad 35 \quad 100 \quad 17 \quad 80
\]

1. Insert 60.
2. Insert 25.
3. Insert 35.
4. Left rotate at 25.
5. Right rotate at 60.
6. Insert 100.
7. Insert 17.
Building an AVL Tree

(h) Insert 80.

(i) Right rotate at 100.

(j) Left rotate at 60.
AVL Deletions

- When an entry is removed, we must ensure the balance property is maintained.
  - Use the same technique as with a BST.
  - After the node is removed, subtrees may have to be rebalanced.