Balanced Search Trees

AVL Trees
Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees
AVL Trees

• An **AVL tree** is a binary search tree that either is empty or in which:
  – Heights between left and right subtrees differs by only 1
  – The left and right subtrees are also AVL trees.
AVL Trees

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

  \[ \text{balanceFactor} = \text{height(left subtree)} - \text{height(right subtree)} \]

  – 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.
Balance can be maintained through rotations

- Rotation: an adjustment to the tree, around an element, that maintains the required ordering of elements

![Diagram of right rotation around element 90]

```plaintext
Right rotation around element 90
```

```plaintext
20
  45
  90
```

```plaintext
20
  45
  90
```
AVL Trees (continued)
AVLTree

Insert 55 along path 40 - 50 - 60
Inserting 55 and 65 maintains AVL height-balance.
Balanced Binary Search Trees
For any right rotation around element \( x \), the right subtree of \( x \)'s left child becomes the left subtree of \( x \).

Rotate right around 100:
For any right rotation around element \( x \), the right subtree of \( x \)’s left child becomes the left subtree of \( x \).

Here is a right rotation around 100:

Notice that 90 is now in the right subtree.
There are four kinds of rotation:

1. A left rotation;

2. A right rotation;

3. A left rotation around the left child of an element, followed by a right rotation around the element itself;

4. A right rotation around the right child of an element, followed by a left rotation around the element itself.
Implementing the AVLTree Class

(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)

(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.
• 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
// 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
• 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)

![Diagrams showing AVL tree rotations](image)

- Single left rotation about LC
- Single right rotation about P
- Double right rotation implemented by two single rotations.
Rotations summary

- Elements not in the subtree of the element rotated about are unaffected by the rotation.
- A rotation takes constant time.
- Before and after a rotation, the tree is still a binary search tree.
- The code for a left rotation is symmetric to the code for a right rotation: Simply swap “left” and “right.”
# Summary Table Of Rotations

<table>
<thead>
<tr>
<th>Type of imbalance</th>
<th>Balance factor of parent</th>
<th>Balance factor of child</th>
<th>Direction of 1st rotation</th>
<th>Direction of 2nd Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-left</td>
<td><img src="image1" alt="Diagram" /></td>
<td>-2</td>
<td>-1</td>
<td>Right</td>
</tr>
<tr>
<td>Right-right</td>
<td><img src="image2" alt="Diagram" /></td>
<td>+2</td>
<td>+1</td>
<td>Left</td>
</tr>
<tr>
<td>Left-right</td>
<td><img src="image3" alt="Diagram" /></td>
<td>-2</td>
<td>+1</td>
<td>Left (child)</td>
</tr>
<tr>
<td>Right-left</td>
<td><img src="image4" alt="Diagram" /></td>
<td>+2</td>
<td>-1</td>
<td>Right (child)</td>
</tr>
</tbody>
</table>
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 11 13 9

Insert the three elements 11, 13, and 9. At 9, node 5 has balance factor -2.
Example - Building an AVL Tree (concluded)

Insert 16

Insert the last element 16. Node 20 has balance factor +2.
Efficiency of AVL Tree Insertion

- A mathematical analysis shows that the worst case running time for insertion is $O(\log_2 n)$. The worst case for the difficult deletion algorithm is also $O(\log_2 n)$. 