DEFINITION OF AN AVL TREE

- Any binary search tree $T$ that satisfies the height-balance property is said to be an **AVL tree**, named after the initials of its inventors: Adel’son-Vel’skii and Landis.

- **Height-Balance Property.** For every internal position $p$ of $T$, the heights of the children of $p$ differ by at most 1.
PROPERTIES OF AVL TREE

- height-balance property allows
  - subtree of an AVL tree is itself an AVL tree.
  - The height of an AVL tree storing \( n \) entries is \( O(\log n) \).
    (view 11.3 for the proof)

- height-balance property characterizing AVL trees is equivalent to saying that every position is balanced.

- Given a binary search tree \( T \), we say that a position is balanced if the absolute value of the difference between the heights of its children is at most 1,

- **AVL tree** guarantees worst-case logarithmic running time for all the fundamental map operations.
The insertion and deletion operations start off with corresponding operations of (standard) binary search trees, but with post-processing for each operation to restore the balance.

- After insertion, the height-balance property may be violated.
- Restructure $T$ to fix any unbalance with a “search-and-repair” strategy.

Any ancestor of $z$ that became temporarily unbalanced becomes balanced again, and this one restructuring restores the height-balance property **globally**.
• Let $z$ be the first position we encounter in going up from $p$ toward the root of $T$ such that $z$ is unbalanced
• Let $y$ denote the child of $z$ with greater height
• Let $x$ be the child of $y$ with greater height (there cannot be a tie)
• Perform $\text{restructure}(x)$

before the insertion

after an insertion in subtree $T_3$ causes imbalance at $z$

after restoring balance with trinode restructuring
insertion of an entry with key 54 in the AVL tree

after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced;

a trinode restructuring restores the height-balance property
UPDATE OPERATIONS: DELETION

- As with insertion, we use trinode restructuring to restore balance in the tree $T$ after deletion.
- Let $z$ be the first unbalanced position encountered going up from $p$ toward the root of $T$.
- Let $y$ be that child of $z$ with greater height.
- Let $x$ be the child of $y$ defined as follows:
  - If one of the children of $y$ is taller than the other, let $x$ be the taller child of $y$;
  - Else (both children of $y$ have the same height), let $x$ be the child of $y$ on the same side as $y$.
- Run restructure($x$) operation.
- After rebalancing $z$, we continue walking up $T$ looking for unbalanced positions.
  - The height-balance property is guaranteed to be locally restored within the subtree of $b$ but not globally.
Deletion of the entry with key 32 from the AVL tree

after removing the node storing key 32, the root becomes unbalanced

A trinode restructuring of $x$, $y$, and $z$ restores the height-balance property.
the height of an AVL tree with \( n \) entries is guaranteed to be \( O(\log n) \).

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>get, put, remove</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>firstEntry, lastEntry</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>ceilingEntry, floorEntry, lowerEntry, higherEntry</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>subMap</td>
<td>( O(s + \log n) )</td>
</tr>
<tr>
<td>entrySet, keySet, values</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>
AVLTreeMap uses the node’s auxiliary balancing variable to store the height of the subtree rooted at that node, with leaves having a balance factor of 0 by default.
/** Returns a child of p with height no smaller than that of the other child. */
protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
    if (height(left(p)) > height(right(p))) return left(p);  // clear winner
    if (height(left(p)) < height(right(p))) return right(p);  // clear winner
    // equal height children; break tie while matching parent's orientation
    if (isRoot(p)) return left(p);  // choice is irrelevant
    if (p == left(parent(p))) return left(p);  // return aligned child
    else return right(p);
}

/** Overrides the TreeMap rebalancing hook that is called after an insertion. */
protected void rebalanceInsert(Position<Entry<K,V>> p) {
    rebalance(p);
}

/** Overrides the TreeMap rebalancing hook that is called after a deletion. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
    if (lisRoot(p))
        rebalance(parent(p));
}
JAVA IMPLEMENTATION OF AVL TREE 3

```java
/**
 * Utility used to rebalance after an insert or removal operation. This traverses the
 * path upward from p, performing a trinode restructuring when imbalance is found,
 * continuing until balance is restored.
 */
protected void rebalance(Position<Entry<K,V>> p) {
    int oldHeight, newHeight;
    do {
        oldHeight = height(p);                // not yet recalculated if internal
        if (!isBalanced(p)) {                 // imbalance detected
            // perform trinode restructuring, setting p to resulting root,
            // and recompute new local heights after the restructuring
            p = restructure(tallerChild(tallerChild(p)));
            recomputeHeight(left(p));
            recomputeHeight(right(p));
        }
        recomputeHeight(p);
        newHeight = height(p);
        p = parent(p);
    } while (oldHeight != newHeight && p != null);
}
```