CHAPTER 10
AVL TREES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM JORY DENNY AND MUKULIKA GHOSH
AVL trees are balanced

- An AVL Tree is a binary search tree such that for every internal node $v$ of $T$, the heights of the children of $v$ can differ by at most 1

An example of an AVL tree where the heights are shown next to the nodes:
INSERTION IN AN AVL TREE

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example insert 54:

Before Insertion

After Insertion
TRINODE RESTRUCTURING

- let \((a, b, c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

**Case 1:** single rotation (a left rotation about \(a\))

**Case 2:** double rotation (a right rotation about \(c\), then a left rotation about \(a\))
unbalanced...

...balanced
Restructuring
Single Rotations
RESTRUCTURING DOUBLE ROTATIONS

double rotation

\[ a = z \]
\[ b = x \]
\[ c = y \]

\[ T_0 \]
\[ T_1 \]
\[ T_2 \]
\[ T_3 \]

double rotation

\[ a = z \]
\[ b = x \]
\[ c = y \]

\[ T_0 \]
\[ T_1 \]
\[ T_2 \]
\[ T_3 \]

double rotation

\[ a = y \]
\[ b = x \]
\[ c = z \]

\[ T_0 \]
\[ T_1 \]
\[ T_2 \]
\[ T_3 \]
EXERCISE
AVL TREES

- Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
  - 30, 40, 24, 58, 48, 26, 11, 13
- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, \( w \), may cause an imbalance.

- Example:
Let $z$ be the first unbalanced node encountered while travelling up the tree from $w$ (parent of removed node). Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.

- We perform $\text{restructure}(x)$ to restore balance at $z$. 

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.

This can happen at most $O(\log n)$ times. Why?
EXERCISE
AVL TREES

- Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
  - 30, 40, 24, 58, 48, 26, 11, 13
- Now, remove the item with key 48. Draw the resulting tree
- Now, remove the item with key 58. Draw the resulting tree
Fact: The height of an AVL tree storing \( n \) keys is \( O(\log n) \).

Proof: Let us bound \( n(h) \): the minimum number of internal nodes of an AVL tree of height \( h \).

We easily see that \( n(1) = 1 \) and \( n(2) = 2 \).

For \( n > 2 \), an AVL tree of height \( h \) contains the root node, one AVL subtree of height \( h - 1 \) and another of height \( h - 2 \).

That is, \( n(h) = 1 + n(h-1) + n(h-2) \).

Knowing \( n(h-1) > n(h-2) \), we get \( n(h) > 2n(h-2) \). So

- \( n(h) > 2n(h-2) > 4n(h-4) > 8n(h-6) \), ... (by induction),
- \( n(h) > 2^i n(h-2i) \)

Solving the base case we get: \( n(h) > 2^{\frac{h}{2} - 1} \)

Taking logarithms: \( h < 2 \log n(h) + 2 \)

Thus the height of an AVL tree is \( O(\log n) \).
A single restructure is $O(1)$ – using a linked-structure binary tree

find($k$) takes $O(\log n)$ time – height of tree is $O(\log n)$, no restructures needed

put($k$, $v$) takes $O(\log n)$ time
- Initial find is $O(\log n)$
- Restructuring up the tree, maintaining heights is $O(\log n)$

erase($k$) takes $O(\log n)$ time
- Initial find is $O(\log n)$
- Restructuring up the tree, maintaining heights is $O(\log n)$
OTHER TYPES OF SELF-BALANCING TREES

- **Splay Trees** – A binary search tree which uses an operation splay ($x$) to allow for amortized complexity of $O(\log n)$.
- **(2, 4) Trees** – A multiway search tree where every node stores internally a list of entries and has 2, 3, or 4 children. Defines self-balancing operations.
- **Red-Black Trees** – A binary search tree which colors each internal node red or black. Self-balancing dictates changes of colors and required rotation operations.