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 NANCY M. AMATO AND JORY DENNY
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 is as in a binary search tree
• Always done by expanding an external node.
• Example insert 54:
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 \(a\))

(Other two cases are symmetrical)
unbalanced...

...balanced
RESTRUCTURING
SINGLE ROTATIONS

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]
\[ a = x \quad b = y \quad c = z \]

**Single Rotation**

\[ \text{single rotation} \]

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]
\[ a = x \quad b = y \quad c = z \]
RESTRUCTURING
DOUBLE ROTATIONS

da = z
b = x
c = y

T_0
T_1
T_2
T_3

da = z
b = x
c = y

double rotation

T_0
T_1
T_2
T_3

da = y
b = x
c = z

T_0
T_1
T_2
T_3

T_3
T_2
T_1

T_0
Algorithm restructure(\(x\))

1. Let \((a,b,c)\) be a left-to-right (inorder) listing of the nodes \(x\), \(y\), and \(z\).
2. Let \((T0,T1,T2,T3)\) be a left-to-right (inorder) listing of the four subtrees of \(x\), \(y\), and \(z\) not rooted at \(x\), \(y\), or \(z\).
3. Replace the subtree rooted at \(z\) with a new subtree rooted at \(b\).
4. Let \(a\) be the left child of \(b\) and let \(T0\) and \(T1\) be the left and right subtrees of \(a\), respectively.
5. Let \(c\) be the right child of \(b\) and let \(T2\) and \(T3\) be the left and right subtrees of \(c\), respectively.
EXERCISE
AVL TREES

- Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
  - 10, 20, 13, 30, 24, 7
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
  - 10, 20, 13, 30, 24, 7
- Now, remove the item with key 24. Draw the resulting tree
- Now, remove the item with key 20. 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

- $n(h) > 2n(h - 2) > 4n(h - 4) > 8n(n - 6), \ldots$ (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 $O(\log n)$
RUNNING TIMES FOR AVL TREES

- A single restructure is $O(1)$ – using a linked-structure binary tree
- $\text{find}(k)$ takes $O(\log n)$ time – height of tree is $O(\log n)$, no restructures needed
- $\text{put}(k, v)$ takes $O(\log n)$ time
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- $\text{erase}(k)$ 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 \( \text{splay}(x) \) 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