CSE 373

AVL trees

read: Weiss Ch. 4, section 4.1 - 4.4

slides created by Marty Stepp

http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Trees and balance

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of $N$ nodes has a height of $\sim \log_2 N$.
  - A very unbalanced tree can have a height close to $N$.

- The runtime of adding to / searching a BST is closely related to height.
- Some tree collections (e.g. TreeSet) contain code to balance themselves as new nodes are added.
• Observation: The shallower the BST the better.
  ▪ Average case height is $O(\log N)$
  ▪ Worst case height is $O(N)$
  ▪ Simple cases such as adding $(1, 2, 3, ..., N)$, or the opposite order, lead to the worst case scenario: height $O(N)$.

• For binary tree of height $h$:
  ▪ max # of leaves: $2^{h-1}$
  ▪ max # of nodes: $2^h - 1$
  ▪ min # of leaves: 1
  ▪ min # of nodes: $h$
Calculating tree height

- Height is max number of nodes in path from root to any leaf.
  - height(null) = 0
  - height(a leaf) = ?
  - height(A) = ?
  - *Hint*: it's recursive!

- height(a leaf) = 1
- height(A) = 1 + max(height(A.left), height(A.right))
AVL trees

*AVL tree*: a binary search tree that uses modified add and remove operations to stay balanced as its elements change

- one of several kinds of auto-balancing trees (others in book)
- invented in 1962 by two Russian mathematicians
  - (Adelson-Velskii and Landis)
  - A-V & L proved that an AVL tree's height is always $O(\log N)$.

- *basic idea*: When nodes are added to / removed from the tree, if the tree becomes unbalanced, repair the tree until balance is restored.
  - rebalancing operations are relatively efficient ($O(1)$)
  - overall tree maintains a balanced $O(\log N)$ height, fast to add/search
Balance factor

- balance factor, for a tree node $T$:
  - $= \text{height of } T\text{'s right subtree minus height of } T\text{'s left subtree.}$
  - $\text{BF}(T) = \text{Height}(T\text{.right}) - \text{Height}(T\text{.left})$
    - (the tree at right shows BF of each node)

- an AVL tree maintains a "balance factor" in each node of 0, 1, or -1
  - i.e. no node's two child subtrees differ in height by more than 1

- it can be proven that the height of an AVL tree with $N$ nodes is $O(\log N)$
AVL tree examples

- Two binary search trees:
  - (a) an AVL tree
  - (b) not an AVL tree (unbalanced nodes are darkened)
Which are valid AVL trees?

- What is the balance factor of each tree node?
• Many of the AVL tree operations depend on height.
  ▪ Height can be computed recursively by walking the tree; *too slow*.
  ▪ Instead, each node can keep track of its subtree height as a field:

```java
private class TreeNode {
    private E data;
    private int height;
    private TreeNode left;
    private TreeNode right;
}
```
For all AVL operations, we assume the tree was balanced before the operation began.

- Adding a new node begins the same as with a typical BST, traversing left and right to find the proper location and attaching the new node.
- But adding this new node may unbalance the tree by 1:

```java
set.add(49);
```

```
    55
   /   \
  29    87
 /    /  \
-3   42   49
```
AVL add cases

- Consider the lowest node $k_2$ that has now become unbalanced.
  - The new offending node could be in one of the four following grandchild subtrees, relative to $k_2$:

  1) Left-Left,
  2) Left-Right,
  3) Right-Left,
  4) Right-Right
Key idea: rotations

- If a node has become out of balanced in a given direction, rotate it in the opposite direction.
  - rotation: A swap between parent and left or right child, maintaining proper BST ordering.
Right rotation

- **right rotation** (clockwise):  
  \( \text{(fixes Case 1 (LL))} \)
  - left child \( k_1 \) becomes parent
  - original parent \( k_2 \) demoted to right
  - \( k_1 \)'s original right subtree \( B \) (if any) is attached to \( k_2 \) as left subtree
Right rotation example

- What is the balance factor of $k_2$ before and after rotating?
Right rotation steps

1. Detach left child (11)'s right subtree (27) *(don't lose it!)*
2. Consider left child (11) be the new parent.
3. Attach old parent (43) onto right of new parent (11).
4. Attach new parent (11)'s old right subtree (27) as left subtree of old parent (43).
Right rotation code

```java
private TreeNode rightRotate(TreeNode oldParent) {
    // 1. detach left child's right subtree
    TreeNode orphan = oldParent.left.right;

    // 2. consider left child to be the new parent
    TreeNode newParent = oldParent.left;

    // 3. attach old parent onto right of new parent
    newParent.right = oldParent;

    // 4. attach new parent's old right subtree as
    //    left subtree of old parent
    oldParent.left = orphan;

    oldParent.height = height(oldParent); // update nodes'
    newParent.height = height(newParent); // height values

    return newParent;
}
```
private int height(TreeNode node) {
    if (node == null) {
        return 0;
    }
    int left = (node.left == null) ? 0 : node.left.height;
    int right = (node.right == null) ? 0 : node.right.height;
    return Math.max(left, right) + 1;
}
Left rotation

- **left rotation** (counter-clockwise): \( (\text{fixes Case 4 (RR)}) \)
  - right child \( k_2 \) becomes parent
  - original parent \( k_1 \) demoted to left
  - \( k_2 \)'s original left subtree \( B \) (if any) is attached to \( k_1 \) as left subtree
Left rotation steps

1. Detach right child (65)'s left subtree (51) *(don't lose it!)*
2. Consider right child (65) be the new parent.
3. Attach old parent (43) onto left of new parent (65).
4. Attach new parent (65)'s old left subtree (51) as right subtree of old parent (43).
Left rotation code

private TreeNode leftRotate(TreeNode oldParent) {
    // 1. detach right child's left subtree
    TreeNode orphan = oldParent.right.left;

    // 2. consider right child to be the new parent
    TreeNode newParent = oldParent.right;

    // 3. attach old parent onto left of new parent
    newParent.left = oldParent;

    // 4. attach new parent's old left subtree as
    //    right subtree of old parent
    oldParent.right = orphan;

    oldParent.height = height(oldParent); // update nodes'
    newParent.height = height(newParent); // height values

    return newParent;
}
Problem cases

- A single right rotation does not fix Case 2 (LR).

(Similarly, a single left rotation does not fix Case 3 (RL).)
Left-right double rotation

left-right double rotation:  (fixes Case 2 (LR))

- 1) left-rotate $k_3$'s left child ... reduces Case 2 into Case 1
- 2) right-rotate $k_3$ to fix Case 1
Left-right rotation example

- What is the balance factor of $k_1$, $k_2$, $k_3$ before and after rotating?
Right-left double rotation

- **right-left double rotation**:  
  (fixes Case 3 (RL))
  - 1) right-rotate $k_1$'s right child ... reduces Case 3 into Case 4
  - 2) left-rotate $k_1$ to fix Case 4
AVL add example

- Draw the AVL tree that would result if the following numbers were added in this order to an initially empty tree:
  - 20, 45, 90, 70, 10, 40, 35, 30, 99, 60, 50, 80