cs261 Recitation7 - AVL trees

Spring 2011
Binary Search Tree: **Balance**

- Complexity of BST operations: proportional to the length of the path from the root to the node being manipulated.

- In a well balanced tree, the length of the longest path is roughly $\log n$.
  - E.g.: 1 million entries $\rightarrow$ longest path is $\log_2 1,048,576 = 20$.

- For a thin, unbalanced tree, operations become $O(n)$:
  - E.g.: elements are added to tree in sorted order.

- **BALANCE IS IMPORTANT!**
AVL trees

• The longest path in a complete binary tree with $n$ elements is guaranteed to be no longer than ceiling(log $n$)

• Recall the operations (e.g. contains, add, remove) of a BST, all $O(h)$ where $h$ represents the height of the tree

• AVL trees are binary search trees that balances itself automatically every time an element is inserted or deleted.

• Each node of an AVL tree has the property that the heights of the sub-tree rooted at its children differ by at most one.
Upper Bound of AVL Tree Height

• Show AVL tree with $n$ nodes has $O(\log n)$ height
• Let $N(h)$ represent the minimum number of nodes that can form an AVL tree of height $h$.

\[
\begin{align*}
N_h &= N_{h-1} + N_{h-2} + 1 \\
N_{h-1} &= N_{h-2} + N_{h-3} + 1 \\
N_h &= (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \\
N_h &> 2N_{h-2} \\
N_h &> 2^\frac{h}{2} \\
\log N_h &> \log 2^\frac{h}{2} \\
2 \log N_h &> h \\
h &= O(\log N_h)
\end{align*}
\]
AVL Rotation/Balancing

• Making a rotation requires re-assigning left, right, and parent of a few nodes, but nothing more than that.

• Rotations are $O(1)$ time operations.

Demo:  http://people.ksp.sk/~kuko/bak/big/
AVL Trees: Balancing Pseudocode

Balancing pseudocode (to rebalance an unbalanced node):

If left child is tallest:

   If left child is heavy on the right side:  // Double rotation needed.
      Rotate the left child to the left
      Rotate unbalanced ("top") node to the right

Else:  // Right child is the tallest.

   If right child is heavy on the left side:  // Double rotation needed.
      Rotate the right child to the right
      Rotate unbalanced ("top") node to the left

Return new "top" node
ROTATIONS

AVL Tree before insertion/deletion

Three possible cases

AVL invariant violated!
Consider a violation $V_2$

Case 1: goes to $C$

Case 2: goes to $B$
Case 1
• AVL insertion is BST insertion plus at most two rotations. \(O(h) + 2O(1) = O(h) = O(\log n)\)
• AVL insertion take \(O(\log n)\) time
Insertion/Add implementation

• In your worksheet 31
• node * AVLnodeAdd(node* current, newValue)

```c
{
    if current == 0
        build new node with the given value
        return balance(newnode)
    else if newValue less than current value
        left = AVLnodeAdd(left, newValue)
    else
        right = AVLnodeAdd(right, newValue)
    return balance(current)
}
```

function balance is used to rotate the tree to keep it balanced
AVL Deletion

- Not too different from insertion. The key difference is that insertion can only imbalance one node at a time, a deletion of a node may imbalance several of its ancestors.

- When delete a node and cause an imbalance of the node’s parent, we need to make necessary rotation on its parent, and also we have to traverse up the ancestry line, checking the balance, and make possibly more rotations to fix the tree.

- Fix AVL tree after a deletion may require $O(\log n)$ more rotations. Rotation is $O(1)$, so deletion is still $O(\log n)$