AVL Trees

CSE 373
Data Structures
Lecture 8
Readings

• Reading
  › Section 4.4,
Binary Search Tree - Best Time

- All BST operations are $O(d)$, where $d$ is tree depth
- minimum $d$ is $d = \left\lfloor \log_2 N \right\rfloor$ for a binary tree with $N$ nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is $O(\log N)$
Binary Search Tree - Worst Time

- Worst case running time is $O(N)$
  - What happens when you insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of “balance”:
    - compare depths of left and right subtree
  - Unbalanced degenerate tree
Balanced and unbalanced BST
Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting
Balancing Binary Search Trees

• Many algorithms exist for keeping binary search trees balanced
  › Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  › Weight-balanced trees
  › Splay trees and other self-adjusting trees
  › B-trees and other multiway search trees
Perfect Balance

• Want a complete tree after every operation
  › tree is full except possibly in the lower right
• This is expensive
  › For example, insert 2 in the tree on the left and then rebuild as a complete tree
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node
Height of an AVL Tree

- \( N(h) = \) minimum number of nodes in an AVL tree of height \( h \).
- **Basis**
  - \( N(0) = 1, \ N(1) = 2 \)
- **Induction**
  - \( N(h) = N(h-1) + N(h-2) + 1 \)
- **Solution** (recall Fibonacci analysis)
  - \( N(h) \geq \phi^h \quad (\phi \approx 1.62) \)
Height of an AVL Tree

- \( N(h) \geq \phi^h \) \((\phi \approx 1.62)\)
- Suppose we have \( n \) nodes in an AVL tree of height \( h \).
  - \( n \geq N(h) \)
  - \( n \geq \phi^h \) hence \( \log_\phi n \geq h \) (relatively well balanced tree!!)
  - \( h \leq 1.44 \log_2 n \) (i.e., Find takes \( O(\log n) \))
Node Heights

height of node = h
balance factor = h_{left} - h_{right}
empty height = -1
Node Heights after Insert 7

Tree A (AVL)

- Height of node = h
- Balance factor = $h_{\text{left}} - h_{\text{right}}$
- Empty height = -1

Tree B (not AVL)

- Balance factor $1 - (-1) = 2$
- Balance factor $-1$

1/24/02
AVL Trees - Lecture 8
Insert and Rotation in AVL Trees

• Insert operation may cause balance factor to become 2 or \(-2\) for some node
  › only nodes on the path from insertion point to root node have possibly changed in height
  › So after the Insert, go back up to the root node by node, updating heights
  › If a new balance factor (the difference \(h_{left} - h_{right}\)) is 2 or \(-2\), adjust tree by \textit{rotation} around the node
Single Rotation in an AVL Tree
Insertions in AVL Trees

Let the node that needs rebalancing be $\alpha$.

There are 4 cases:

Outside Cases (require single rotation):
1. Insertion into left subtree of left child of $\alpha$.
2. Insertion into right subtree of right child of $\alpha$.

Inside Cases (require double rotation):
3. Insertion into right subtree of left child of $\alpha$.
4. Insertion into left subtree of right child of $\alpha$.

The rebalancing is performed through four separate rotation algorithms.
AVL Insertion: Outside Case

Consider a valid AVL subtree

```
        j
       / \
      k   Z
     /  / \
    X h  h h
   /  /    \
  Y    Z
```

- **AVL Insertion: Outside Case**
- **Consider a valid AVL subtree**
- Diagram shows a tree structure with nodes labeled j, k, X, Y, and Z, illustrating the outside case for AVL insertion.
AVL Insertion: Outside Case

Inserting into X destroys the AVL property at node j
AVL Insertion: Outside Case

Do a “right rotation”
Single right rotation

Do a “right rotation”
Outside Case Completed

“Right rotation” done! (“Left rotation” is mirror symmetric)

AVL property has been restored!
AVL Insertion: Inside Case

Consider a valid AVL subtree
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does “right rotation” restore balance?
"Right rotation" does not restore balance... now k is out of balance
Consider the structure of subtree Y...
AVL Insertion: Inside Case

$Y = \text{node } i \text{ and subtrees } V \text{ and } W$
AVL Insertion: Inside Case

We will do a left-right “double rotation” . . .
Double rotation: first rotation

left rotation complete
Double rotation: second rotation

Now do a right rotation
Double rotation: second rotation

right rotation complete

Balance has been restored

1/24/02
Implementation

No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don’t perform rotations.

Once you have performed a rotation (single or double) you won’t need to go back up the tree.
Single Rotation

RotateFromRight(n : reference node pointer) {
  p : node pointer;
  p := n.right;
  n.right := p.left;
  p.left := n;
  n := p
}

You also need to modify the heights or balance factors of n and p
Double Rotation

• Implement Double Rotation in two lines.

DoubleRotateFromRight(n : reference node pointer) {
    ???
}
AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.
Pros and Cons of AVL Trees

Arguments for AVL trees:
1. Search is $O(\log N)$ since AVL trees are always balanced.
2. Insertion and deletions are also $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:
1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have $O(N)$ for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).
Double Rotation Solution

DoubleRotateFromRight(n : reference node pointer) {
    RotateFromLeft(n.right);
    RotateFromRight(n);
}

\begin{center}
\begin{tikzpicture}
    \node (n) at (0,0) [circle, draw] {n};
    \node (x) at (-1,-1) [circle, draw] {X};
    \node (v) at (-2,-2) [circle, draw] {V};
    \node (w) at (0,-2) [circle, draw] {W};
    \node (z) at (1,-1) [circle, draw] {Z};
    \draw (n) -- (x);
    \draw (x) -- (v);
    \draw (x) -- (z);
    \draw (z) -- (w);
\end{tikzpicture}
\end{center}