Today’s Outline

• Announcements
  – Assignment #2 due Fri, April 19 at the BEGINNING of lecture

• Today’s Topics:
  – Binary Search Trees (Weiss 4.1-4.3)
  – AVL Trees (Weiss 4.4)

Trivia: AVL stands for Adelson-Velskii and Landis.

The AVL Balance Condition

Left and right subtrees of every node have equal heights differing by at most 1

Define: \( \text{balance}(x) = \text{height}(x.\text{left}) – \text{height}(x.\text{right}) \)

AVL property: \(-1 \leq \text{balance}(x) \leq 1, \text{ for every node } x\)

• Ensures small depth
  – Roughly height \( h \) for size \( 2^h \)
• Easy to maintain
  – Using single and double rotations that just add some constant work to insertions and deletions.

The AVL Tree Data Structure

Structural properties
1. Binary tree property
   (0,1, or 2 children)
2. Heights of left and right subtrees of every node differ by at most 1

Result:
  Worst case depth of any node is: \( O(\log n) \)

Ordering property
  – Same as for BST

Is this an AVL Tree?

NULLs have height \(-1\)

Student Activity: If not AVL, put a box around nodes where AVL property is violated.
Proving Shallowness Bound

Let $S(h)$ be the min # of nodes in an AVL tree of height $h$

Claim: $S(h) = S(h-1) + S(h-2) + 1$

Solution of recurrence: $S(h) = \Theta(2^h)$

(like Fibonacci numbers; see Ch. 2)

So for a tree of size $n$ nodes, the height is approximately $\log_2 n$.

Weiss’s AvlNode

```java
private static class AvlNode<AnyType> {
    // Constructors
    AvlNode(AnyType theElement) {
        this( theElement, null, null );    }
    AvlNode(AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt) {
        element  = theElement;
        left     = lt;
        right    = rt;
        height   = 0;
    }
    AnyType element;      // The data in the node
    AvlNode<AnyType> left;         // Left child
    AvlNode<AnyType> right;        // Right child
    int height;       // Height
}
```

AVL trees: find, insert

- **AVL find:**
  - same as BST find.

- **AVL insert:**
  - same as BST insert, except may need to “fix” the AVL tree after inserting new value.

AVL tree insert

Let $x$ be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of $x$. (ie. left of left)
2. right subtree of the left child of $x$.
3. left subtree of the right child of $x$.
4. right subtree of the right child of $x$.

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.

AVL Insert: detect & fix imbalances

1. Insert the new node just as you would in a BST (as a new leaf)
2. For each node on the path from the inserted node up to the root, the insertion may (or may not) have changed the node’s height
3. So after recursive insertion in a subtree, check for height imbalance at each of these nodes and perform a rotation to restore balance at that node if needed

All the action is in defining the correct rotations to restore balance

Fact that makes it a bit easier:
- There must be a deepest node that is imbalanced after the insert (all descendants still balanced)
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced
Bad Case #1

Insert(6)
Insert(3)
Insert(1)

Bad Case #1: Example

Insert(6)
Insert(3)
Insert(1)

Third insertion violates balance property
• happens to be at the root

What is the only way to fix this?
• Which of the 4 cases?

Fix: Apply “Single Rotation”

• Single rotation: The basic operation we’ll use to rebalance
  – Move child of unbalanced node into parent position
  – Parent becomes the “other” child (always okay in a BST!)
  – Other subtrees move in only way BST allows (next slide)

AVL Property violated at this node (“x”)

AVL Trees

Generalized left-left case

• Node a imbalanced due to insertion somewhere in left-left grandchild increasing height of left subtree.
  – 1 of 4 possible imbalance causes (other three coming)
• First we did the insertion, which makes a imbalanced:

Before insertion – balanced.

After single rotation – balanced!

Generalized left-left case (cont.)

• So we rotate at a, using BST facts: X < b < Y < a < Z
• A single rotation to the right restores balance at the node
  – To same height as before insertion (so ancestors now balanced)

Single rotation example: insert (1)

• Do the rotation at the deepest node that became unbalanced.
• What node is that?
The general right-right case

- Mirror image to left-left case, so you rotate the other way
  - Single rotation to the left
  - Exact same concept, but slightly different code

Bad Case #3

Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: \texttt{insert(1)}, \texttt{insert(6)}, \texttt{insert(3)}

- First wrong idea: single rotation like we did for left-left

Bad Case #3: Wrong Solution #1

Simple example: \texttt{insert(1)}, \texttt{insert(6)}, \texttt{insert(3)}

- Second wrong idea: single rotation on the child of the unbalanced node
Bad Case #3: Correct Solution: Double Rotation

AVL Property violated at this node ("x")

Double Rotation:
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

The general right-left case (cont.)
• Like in the left-left and right-right cases, the height of the subtree after rebalancing is the same as before the insert
  – So no ancestor in the tree will need rebalancing
• Does not have to be implemented as two rotations; can just do:

Double rotation: insert(5), step 1

The last case: left-right
• Mirror image of right-left – double rotation
  – Again, no new concepts, only new code to write

Double rotation: insert(5), step 2
AVL Insert - Summary

• Insert as in a BST
• Check back up path for imbalance, which will be 1 of 4 cases:
  – node’s left-left grandchild is too tall
  – node’s left-right grandchild is too tall
  – node’s right-left grandchild is too tall
  – node’s right-right grandchild is too tall
• Only one case occurs because tree was balanced before insert
• After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  – So all ancestors are now balanced

Imbalance at node X

Single Rotation
1. Rotate between x and child

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

Insert into an AVL tree: a b e c d

Single and Double Rotations:
Inserting what integer values would cause the tree to need a:
1. single rotation?
2. double rotation?
3. no rotation?

Insert 3

Insert(3)

Unbalanced?

Insert 33

Insert(33)

Unbalanced?

How to fix?
AVL Trees

- Balance condition:
  - For every node \( x \), \(-1 \leq \text{balance}(x) \leq 1\)
  - Strong enough: Worst case depth is \( O(\log n) \)
  - Easy to maintain: one single or double rotation

- Guaranteed \( O(\log n) \) running time for
  - Find ?
  - Insert ?
  - Delete ?
  - buildTree ?

- What extra info do we maintain in each node?

- Where were rotations performed? What complexity for doing a rotation?

- How do we locate this node? What complexity?