Logistics

Reading

- Weiss Ch 19.1-3 BSTs
- Weiss Ch 19.4: AVL Trees
- Weiss Ch 21: Binary Heap

Today

- Tree Rotations: Balancing via pointer manipulation
- AVL Trees
Q: Why worry about insert/remove messing with the tree? What affect can it have on the performance of future ops on the tree?

Q: What property of a tree dictates the runtime complexity of its operations?

Recall from our practice footnote

Build and draw a BST by inserting these numbers in order: 5, 13, 18, 21, 23, 31, 5, 7, 89, 130

Build and draw a BST by inserting these numbers in order: 31, 18, 130, 89, 21, 5, 57, 13, 23
Balancing Trees

- add/remove/find complexity $O(\text{height}(t))$
- Degenerate tree has height $N$: a linked list
- Prevent this by re-balancing on insert/remove
- Several kinds of trees do this
  - AVL: left/right subtree height differ by max 1
  - Red-black: preserve 4 red/black node properties
  - AA: red-black tree + all left nodes black
  - Splay: amortized bound on ops, very different
The AVL Tree

The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and E. M. Landis, who published it in their 1962 paper "An algorithm for the organization of information".
– Wikip: AVL Tree

- A self-balancing tree
- Operations
- Proof of logarithmic height
AVL Balance Property

$T$ is an AVL tree if and only if

- $T\.left$ and $T\.right$ differ in height by at most 1
- AND $T\.left$ and $T\.right$ are AVL trees
T is an AVL tree if and only if

- T.left and T.right differ in height by at most 1
- AND T.left and T.right are AVL trees

1 Not AVL

2 AVL

3 Not AVL

4 AVL

5 Not AVL

6 AVL

7 Not AVL

8 AVL

80 not AVL
Nodes and Balancing in AVL Trees

Track Balance Factor of trees
- balance = height(t.left) - height(t.right);
- Must be -1, 0, or +1 for AVL
- If -2 or +2, must fix

class Node<T>{
    Node<T> left, right;
    T data;
    int height;
}

Don’t explicitly calculate height
- Adjust balance factor on insert/delete
- Recurse down to add/remove node
- Unwind recursion up to adjust balance of ancestors
- When unbalanced, rotate to adjust heights
- Single or Double rotation can always adjust heights by 1
Re-balancing usually involves

- Drill down during insert/remove
- Follow path back up to make adjustments
- Adjustments even out height of subtrees
- Adjustments are usually rotations
- Rotation changes structure of tree without affecting ordering
Single Rotation Basics

Right Rotation
Rotation node becomes the right subtree

Left Rotation
Rotation node becomes the left subtree
Fixing an Insertion with a Single Rotation

Insert 1, perform rotation to balance heights

- Right rotation at 8

(a) Before rotation

(b) After rotation
Single Rotation Practice

Problem 1
- 40 was just inserted
- Rebalance tree rooted at 16
- Left-rotate 16

Problem 2
- 85 is being removed
- Rebalance tree rooted at 57
- Right rotate 57
Single Rotations Aren’t Enough

Can we fix the following with a single rotation?

(a) Before rotation

(b) After rotation
Example: Can’t fix this with single rotation
Double Rotation Overview

**Left-Right**
- Left Rotate at $k_1$
- Right-rotate at $k_3$

(a) Before rotation  
(b) After rotation

**Right-Left**
- Right Rotate at $k_3$
- Left Rotate at $k_1$

(a) Before rotation  
(b) After rotation
Fixing an Insertion with a Double Rotation

Insert 5, perform two rotations to balance heights
- Problem is at 8: left height 3, right height 1
- Left rotate 4 (height imbalance remains)
- Right rotate 8 (height imbalance fixed)
Problem 3

- 35 was just inserted
- Rebalance the tree rooted at 36
- Use two rotations, at 33 and 36
- 36 should move
class Node<T>{
    Node<T> left, right;
    T data;
    int height;
}

Write the following codes for single/double rotations:

    // Single Right rotation
    // t becomes right child, t.left becomes new
    // root which is returned
    Node<T> rightRotate( Node<T> t ) { ... }

    // Left-Right Double Rotation:
    // left-rotate t.left, then right-rotate t
    Node<T> leftRightRotate( Node<T> t ){ ... }
// Single Right rotation
Node<T> rightRotate( Node<T> t ) {
    Node<T> newRoot = t.left;
    t.left = newRoot.right;
    newRoot.right = t;
    t.height = Math.max(t.left.height,
                        t.right.height)+1;
    newRoot.height = Math.max(newRoot.left.height,
                              newRoot.right.height)+1;
    return newRoot;
}

// Left-Right Double Rotation:
// left-rotate t.left, then right-rotate t
Node<T> leftRightRotate( Node<T> t ){
    t.left = leftRotate(t.left);
    return rightRotate(t);
}

Computational complexities of these methods?
Insersion works by first recursively inserting new data as a leaf

Tree is “unstitched” - waiting to assign left/right branches of intermediate nodes to answers from recursive calls

Before returning, check height differences and perform rotations if needed

Allows left/right branches to change the nodes to which they point
Identify subtree height differences to determine rotations
Useful in removal as well

private AvlNode insert( Comparable x, AvlNode t ){
    if( t == null ){ // Found the spot to insert
        t = new AvlNode( x, null, null ); // return new node with data
    }
    else if( x.compareTo( t.element ) < 0 ) { // Head left
        t.left = insert( x, t.left ); // Recursively insert
    } else{ // Head right
        t.right = insert( x, t.right ); // Recursively insert
    }
    if(height(t.left) - height(t.right) == 2){ // t.left deeper than t.right
        if(height(t.left.left) > t.left.right) { // outer tree unbalanced
            t = rightRotate( t ); // single rotation
        } else { // x went left-right:
            t = leftRightRotate( t ); // double rotation
        }
    }
    else{ ... } // Symmetric cases for t.right deeper than t.left
    return t;
}
Inserted 51
Which node is unbalanced?
Which rotation(s) required to fix?
Rebalancing Answer

Insert 51

35 Unbalanced

After right rotate at 57

After left rotate at 35
Proposition: Maintaining the AVL Balance Property during insert/remove will yield a tree with \( N \) nodes and height \( O(\log N) \)

Prove it: What do AVL trees have to do with rabbits?
AVL Properties Give $\log(N)$ height

**Lemma (little theorem) (Thm 19.3 in Weiss, pg 708, adapted)**

An AVL Tree of height $H$ has at least $F_{H+2} - 1$ nodes where $F_i$ is the $ith$ Fibonacci number.

### Definitions

- $F_i$: $ith$ Fibonacci number $(0,1,1,2,3,5,8,13,\ldots)$
- $S$: size of a tree
- $H$: height (assume roots have height 1)
- $S_H$ as smallest size AVL Tree with height $H$

### Proof by Induction: Base Cases True

<table>
<thead>
<tr>
<th>Tree</th>
<th>height $H$</th>
<th>Min Size $S$</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>$H = 0$</td>
<td>$S_0$</td>
<td>$F_{(0+2)} - 1 = 1 - 1 = 0$</td>
</tr>
<tr>
<td>root</td>
<td>$H = 1$</td>
<td>$S_1$</td>
<td>$F_{(1+2)} - 1 = 2 - 1 = 1$</td>
</tr>
<tr>
<td>root+(left or right)</td>
<td>$H = 2$</td>
<td>$S_2$</td>
<td>$F_{(2+2)} - 1 = 3 - 1 = 2$</td>
</tr>
</tbody>
</table>
Consider an Arbitrary AVL tree $T$

- $T$ has height $H$
- $S_H$ smallest size for tree $T$
- Show that the smallest size $S_H = F_{H+2} - 1$
- Assume equation true for smaller trees
  - Left/Right are smaller AVL trees
  - Left/Right differ in height by at most 1
Induction Part 2

- $T$ has height $H$
- Assume for height $h < H$, smallest size of $T$ is $S_h = F_{h+2} - 1$
- Suppose Left is 1 higher than Right
- Left Height: $h = H - 1$
- Left Size: $F_{(H-1)+2} - 1 = F_{H+1} - 1$
- Right Height: $h = H - 2$
- Right Size: $F_{(H-2)+2} - 1 = F_H - 1$

\[
S_H = size(Left) + size(Right) + 1
= (F_{H+1} - 1) + (F_H - 1) + 1
= F_{H+1} + F_H - 1
= F_{H+2} - 1 \quad \blacksquare
\]
AVL Tree of with height $H$ has at least $F_{H+2} - 1$ nodes.

- How does $F_H$ grow wrt $H$?
- Exponentially:
  \[ F_H \approx \phi^H = 1.618^H \]
- $\phi$: The Golden Ratio
- So, \( \log(F_H) \approx H \log(\phi) \)
- Or, \( \log(N) \approx \text{height} \times \phi \)
- Or, \( \log(\text{size}) \approx \text{height} \times \text{constant} \)