CS 151
AVL Trees
Announcements

Test #1 on Friday in class. If you need special considerations (extra time, quiet space, different time, etc.) talk to me today, either via email or in office hours.

No lab this week! Spend this time reviewing your prelabs, labs, and course notes. I have office hours during part of both lab sections, so you can stop by.

Lab 4 grades just sent out at 9:30 this morning.

Test topics:

• advanced Java:
  • abstract classes and interfaces, inheritance, exceptions, javadoc, jUnit

• algorithms analysis:
  • definition of big-Oh, big-Omega, little-Oh, little-omega, and big Theta
  • determine the running time of various algorithms (including log n ones)

• linear and binary search, selection, insertion and merge sort

• data structures: operations, implementations, running times.
  • arraylists, stacks, queues, linked lists, doubly linked lists, binary trees
Review of AVL Trees

Many BST operations are $O(\text{height of tree})=O(n)$. Want height=$\log n$

An **AVL tree** is a binary search tree such that for every node $r$,
• the heights of the left and right subtrees differ by at most 1,
• the left child is an AVL tree, and
• the right child is an AVL tree.

So an AVL tree is just a BST with an additional height restriction on its subtrees. We will show that this height restriction guarantees that the height of the whole tree is $O(\log n)$. (This will take a bit of proof-work, but it’s true.)

The downside is that our implementation will have to maintain the new property. But we’ll ensure this maintenance doesn’t take longer than $O(\log n)$, either...

Thus, the RT of all the BST operations with an AVL tree will be $O(\log n)$, yay!

In particular, we need to re-implement our adds and removes, because they are the methods that mess with our tree, and can potentially unbalance it.

The idea is: do our usual insert (as a leaf), or remove. If the new tree is still balanced, great! Otherwise we’ve unbalanced it, and we need to **rebalance**.
To rebalance an AVL tree $T$, do the following steps:

- Find the smallest unbalanced subtree in $T$, and label its root $Z$
- (Note: find $Z$ by starting at the leaf we inserted or deleted, and move upwards towards the root, checking unbalanced-ness as we go.)
- Since $Z$ is the smallest unbalanced tree, its children are balanced AVL trees, but their heights differ by $>1$.
- Let $Y$ be $Z$’s tallest child.
- Let $X$ be $Y$’s tallest child.
- Let $t_0$, $t_1$, $t_2$, and $t_3$ be the subtrees of $X,Y,Z$, labeled from left to right.
- Let $a,b,c$ be data of $X,Y,Z$ in increasing order.
- Replace the subtree at $Z$ with the subtree at $b$ drawn below:

```
  b
 /\  \
 a  c
/        \         \t0\t1\t2\t3
```

```
4
```
Starting from the empty tree, draw the trees after the following operations:

insert 50, 25, 75, 60, then 80.

insert 90. (requires a rotation)

remove 90.

insert 65. (requires a rotation)

Notes on rebalancing:

• Inserts and removes always happen at the leaves.
• If a tree becomes imbalanced, it’s at some node along the leaf-root path.
• The length of this path is $O(\log n)$
• Each rebalance is an $O(1)$ operation
• We may have to rebalance more than once along the $O(\log n)$ path
To prove that we have correctly rebalanced the tree rooted at Z:

**case 1: val(X) < val(Y) < val(Z)  [val(X) > val(Y) > val(Z) is symmetric]**

Before the rotation: X and Y were balanced but Z was not
- Z balanced before the add/remove: h(t2)={h(t3), h(t3)+1}
- X is Y’s tallest child, so h(t2) = {h(X), h(X)-1}

After the rotation:
- t0, t1, t2, t3, and X have the same children before, so all still balanced
- Z has children of height h(t2) and h(t3), which are within one, so balanced
- Y has children X and Z, of heights h(X) and 1+max{h(t2), h(t3)}
- But 1 + max{h(t2), h(t3)} = 1 + h(t2) = {h(X)+1, h(X)}, so Y is balanced.

Finally, why is does this rebalancing maintain the binary search-order property?
- X unchanged, so it still has the order property
- Z is a BST because t3 is still to its right, and t2 as to its left before
- Y is a BST because X, t0, and t1 were to its left and still are, and t2, Z, and t3 were to its right and still are
AVL Rebalancing

(proof continued)

case 2: val(Y) < val(X) < val(Z) [val(Y) > val(X) > val(Z) is symmetric]

Before the rotation: X and Y were balanced but Z was not
• Z was balanced before the insert/remove: h(X) = h(t3)+1
• X is Y’s tallest child, so h(t0) = \{h(t1), h(t1)+1\}
• Similarly, h(t2) = \{h(t3), h(t3)-1\} <= h(t3)

After the rotation:
• t0, t1, t2, and t3 have the same children before, so all still balanced
• Y has children of height h(t0) and h(t1), which are within one, so balanced
• Z has children of height h(t2) and h(t3), which are within one, so balanced
• X has children Y, Z of heights 1+max\{h(t0), h(t1)\} and 1+max\{h(t2), h(t3)\}
• But max\{h(t0),h(t1)\} = \{h(X), h(X)-1\} and max\{h(t2),h(t3)\} = h(t3) = h(X)-1!

Finally, why is this rebalancing maintain the binary search-order property?
• we can quickly check that anything left of Y, X, and Z was left before
• and anything to the right was to the right before