Lecture 6: AVL Trees

CSE 373
Edition 2016
**Insert: detect potential imbalance**

1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node’s height
3. So after recursive insertion in a subtree, detect height imbalance and perform a *rotation* to restore balance at that node

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

Fact that an implementation can ignore:

- There must be a deepest element 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
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?
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 here

Intuition: 3 must become root
new-parent-height = old-parent-height-before-insert
The example generalized

- Node imbalanced due to insertion somewhere in left-left grandchild increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make a imbalanced
The general left-left case

- Node imbalanced due to insertion *somewhere* in **left-left grandchild**
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at \(a\), using BST facts: \(X < b < Y < a < Z\)

- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced
Another example: \texttt{insert(16)}
Another example: insert(16)
The general right-right case

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code
Two cases to go

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
Two cases to go

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)

- Second wrong idea: single rotation on the child of the unbalanced node
Sometimes two wrongs make a right 😊

- First idea violated the BST property
- Second idea didn’t fix balance
- But if we do both single rotations, starting with the second, it works! (And not just for this example.)
- Double rotation:
  1. Rotate problematic child and grandchild
  2. Then rotate between self and new child

Intuition: 3 must become root
The general right-left case
**Comments**

- 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:

Easier to remember than you may think:
Move c to grandparent’s position
Put a, b, X, U, V, and Z in the only legal positions for a BST
The last case: left-right

- Mirror image of right-left
  - Again, no new concepts, only new code to write
Insert, summarized

- 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
Now efficiency

- Worst-case complexity of \textbf{find}: $O(\log n)$
  - Tree is balanced

- Worst-case complexity of \textbf{insert}: $O(\log n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there’s an $O(\log n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced

- Worst-case complexity of \textbf{buildTree}: $O(n \log n)$

Takes some more rotation action to handle \textbf{delete}...
Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are always balanced
2. Height balancing adds no more than a constant factor to the speed of insert and delete

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. Most large searches are done in database-like systems on disk and use other structures (e.g., B-trees, a data structure in the text)
5. If amortized (later, I promise) logarithmic time is enough, use splay trees (also in the text)