CSE 332: Data Abstractions Адельсóн-Вéльский Лáндис дерево (Part II)

Richard Anderson
Spring 2016

Announcements

- Project 1
- Project 2

The AVL Tree Data Structure

Structural properties

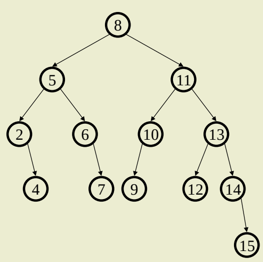
1. Binary tree property
2. Balance:
   left.height – right.height
3. Balance property:
   balance of every node is between -1 and 1

Result:

Worst-case depth is O(log *n*)

Ordering property

– Same as for BST

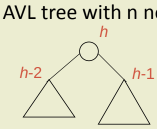


Bounding the height of an AVL tree

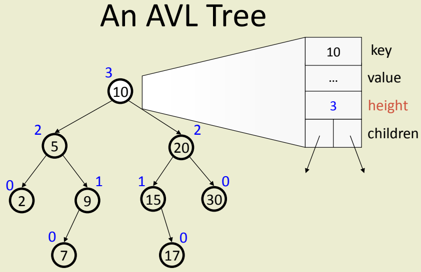
Let *S*(*h*) = the minimum number of nodes in an AVL tree of height *h*

Can prove for all *h*, $S(h) > \phi^h - 1$ where ϕ is the golden ratio, $(1+\sqrt{5})/2$

This shows that an AVL tree with n nodes has height at most $\log_\phi n$



An AVL Tree



Track height at all times!

AVL tree operations

- **AVL `find`**:
  – Same as BST `find`
- **AVL `insert`**:
  – First BST `insert`, *then* check balance and potentially "fix" the AVL tree
  – Four different imbalance cases
- **AVL `delete`**:
  – The "easy way" is lazy deletion
  – Otherwise, do the deletion and then have several imbalance cases

Insert: detect potential imbalance

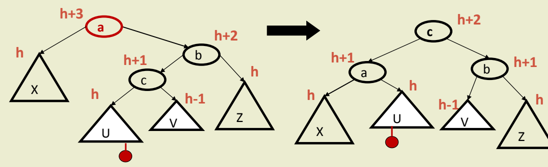
1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height
3. So after recursive insertion in a subtree, detect height imbalance and perform a *rotation* to restore balance at that node. Four types of rotations
   1. Left-left
   2. Right-right
   3. Left-right
   4. Right-left

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

Right-Left rebalancing

- 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

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

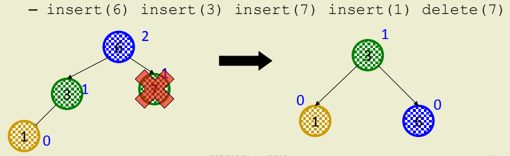
- Worst-case complexity of **`find`**: $O($**`log`** $n)$
  - Tree is balanced
- Worst-case complexity of **`insert`**: $O($**`log`** $n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there's an $O($`log` $n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of **`buildTree`**: $O(n$ **`log`** $n)$

Will take some more rotation action to handle **`delete`**…

Key points for AVL Delete – same type of rotations to restore balance as during insert, but multiple rotations may be needed. Details less important.

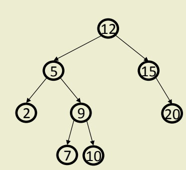
AVL Tree Deletion

- Similar to insertion: do the delete and then rebalance
  - Rotations and double rotations
  - Imbalance may propagate upward so rotations at multiple nodes along path to root may be needed (unlike with insert)
- Simple example: a deletion on the right causes the left-left grandchild to be too tall
  - Call this the *left-left case*, despite deletion on the *right*
  - `insert(6) insert(3) insert(7) insert(1) delete(7)`



Properties of BST delete

We first do the normal BST deletion:
- 0 children: just delete it
- 1 child: delete it, connect child to parent
- 2 children: put successor in your place, delete successor leaf

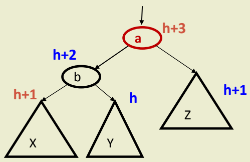


Which nodes' heights may have changed:
- 0 children: path from deleted node to root
- 1 child: path from deleted node to root
- 2 children: path from *deleted successor leaf* to root

Will rebalance as we return along the "path in question" to the root

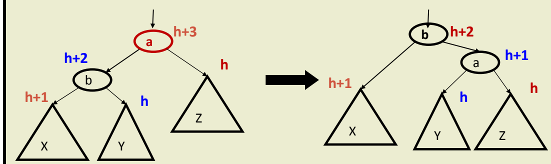
Case #1 Left-left due to right deletion

- Start with some subtree where if right child becomes shorter we are unbalanced due to height of left-left grandchild



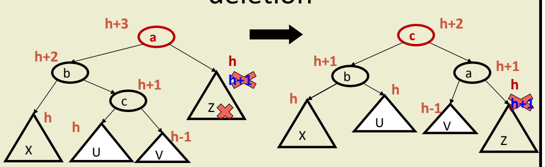
- A delete in the right child could cause this right-side shortening

Case #1: Left-left due to right deletion



- Same single rotation as when an insert in the left-left grandchild caused imbalance due to X becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

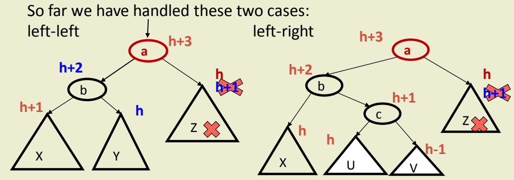
Case #2: Left-right due to right deletion



- Same double rotation when an insert in the left-right grandchild caused imbalance due to c becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

No third right-deletion case needed

So far we have handled these two cases:
left-left left-right



But what if the two left grandchildren are now *both* too tall (h+1)?
- Then it turns out left-left solution still works
- The children of the "new top node" will have heights differing by 1 instead of 0, but that's fine

And the other half

- Naturally two more mirror-image cases (not shown here)
  - Deletion in left causes right-right grandchild to be too tall
  - Deletion in left causes right-left grandchild to be too tall
  - (Deletion in left causes both right grandchildren to be too tall, in which case the right-right solution still works)
- And, remember, "lazy deletion" is a lot simpler and might suffice for your needs

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
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, our next data structure)
5. If *amortized* logarithmic time is enough, use splay trees (skipping, see text)

Now what?

- Have a data structure for the dictionary ADT that has worst-case *O*(`log` *n*) behavior
  - One of several interesting/fantastic balanced-tree approaches
- About to learn another balanced-tree approach: B Trees
- First, to motivate why B trees are better for really large dictionaries (say, over 1GB = 2^{30} bytes), need to understand some ***memory-hierarchy basics***
  - Don't always assume "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE351/333/471, focus here on relevance to data structures and efficiency