Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is $O(\log n)$ since AVL trees are always well balanced.
2. The height balancing adds no more than a constant factor to the speed of insertion and deletion, thus both are $O(\log n)$.

Arguments against using AVL trees:

1. Must store and track height info.
2. Can get same amortized cost with less effort (e.g., splay trees)
3. Inefficient wrt. disk access (e.g., B-trees).

Splay Trees

- Blind adjusting version of AVL trees
  - Why worry about balances? Just rotate anyway!
- Worst case time for an operation is $O(n)$
- *Amortized time* per operation is $O(\log n)$
Find/Insert in Splay Trees

1. Find or insert a node $k$
2. Percolate $k$ to the root with rotations.

Why is this a good idea?
1. A form of caching
2. Fix up the tree without more complexity

Instead of “percolate $k$ to the root with rotations” we will often just say “splay $k$ to the root”.

The Splay Tree Idea

If you’re forced to make a really deep access:
Since you’re down there anyway, fix up a lot of deep nodes!

Do it all with AVL single rotations?

Consider the ordered “list tree” at left. Now do find(1) and splay it to the root with only AVL single rotations:

Do it all with AVL single rotations?

Cost of sequence: find(1), find(2), ... find($n$)?

Single rotations can help, but they are not enough...
Splay: Zig-Zag

Same as AVL double-rotation

Splay: Zig-Zig

Special Case for Root: Zig

Relationship to AVL rotations?

Single rotation
Let the magic begin

Insert 6, 5, 4, 3, 2, 1

Cost per insert? $O(1)$

Let the magic begin

Insert 1, 2, 3, 4, 5, 6

Cost per insert? $O(1)$

Example (continued)

Find(1) continued…

Find(1)
Find(1) finished

Another Splay: Find(2)

Find(2) finished

Another Splay: Find(3)
A bigger, badder example

Consider a “list tree” from 1-32. Doing find(1) gives:

Now do find(2):

How many comparisons?
How many rotations?
What is tree height now?

How many comparisons?
How many rotations?
What is tree height now?

A bigger, badder example

Now do find(3):

The height is cut in ~half again.
Why Splaying Helps

- If a node $x$ on the access path is at depth $d$ before the splay, it's typically at about depth $d/2$ after the splay

- Overall, nodes which are low on the access path tend to move closer to the root

- Splaying gets amortized $O(\log n)$ performance.

Practical Benefit of Splaying

- No heights to maintain, no imbalance to check for
  - Less storage per node, easier to code

- Data accessed once, is often soon accessed again
  - Splaying does implicit caching by bringing it to the root

Splay Operations: Find, Insert

Find($x$):
- Find the node in normal BST manner
- Splay the node to the root
  - if node not found, splay what would have been its parent

Insert($x$):
- Insert the node in normal BST manner
- Splay the node to the root

Splay Delete?
Splay Operations: Delete

Splay on the maximum element in L, then attach R

Now what?

Join(L, R):
given two trees such that (stuff in L) < (stuff in R), merge them:

Splay Tree Summary

All operations are in amortized $O(\log n)$ time

Splaying can be done top-down; this may be better because:
- only one pass
- no recursion or parent pointers necessary
- (we didn’t cover top-down in class)

Splay trees are very effective search trees
- Relatively simple
- No extra fields required
- Excellent \textit{locality} properties in the following sense: frequently accessed keys are cheap to find