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!

Find/Insert in Splay Trees

1. Find or insert a node $k$

2. Splay $k$ to the root using:
    - zig-zag, zig-zig, or plain old zig rotation

Why could this be good??

1. Helps the new root, $k$
   - Great if $k$ is accessed again
2. And helps many others!
   - Great if many others on the path are accessed

Splay: Zig-Zag

*Just like an… Which nodes improve depth?

Splay: Zig-Zig

*Is this just two AVL single rotations in a row?

Special Case for Root: Zig

Relative depth of $p$, $Y$, $Z$?

Relative depth of everyone else?

Why not drop zig-zig and just zig all the way?

Splaying Example: Find(6)

Find(6)
Finally…

Another Splay: Find(4)

Example Splayed Out

But Wait…

What happened here?

Didn’t two find operations take linear time instead of logarithmic?

What about the amortized $O(\log n)$ guarantee?
Why Splaying Helps

- If a node \( n \) on the access path is at depth \( d \) before the splay, it’s 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. (Maybe not now, but soon, and for the rest of the operations.)

Practical Benefit of Splaying

- No heights to maintain, no imbalance to check for
  - Less storage per node, easier to code
- Often data that is accessed once, is soon accessed again!
  - Splaying does implicit caching by bringing it to the root

Splay Operations: Find

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

What if we didn’t splay?

Splay Operations: Insert

- Insert the node in normal BST manner
- Splay the node to the root

What if we didn’t splay?

Splay Operations: Remove

Join

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

- Splay on the maximum element in L, then attach R
Delete Example

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 locality properties: frequently accessed keys are cheap to find

The Memory Hierarchy & Locality

Why do we need to know about the memory hierarchy/locality?

• One of the assumptions that Big-Oh makes is that all operations take the same amount of time.
• Is that really true?

Definitions

Cycle – (for our purposes) the time it takes to execute a single simple instruction. (ex. Add 2 registers together)

Memory Latency – time it takes to access memory
Moore’s Law

Processor-Memory Performance Gap

• x86 CPU speed (100x over 10 years)

What can be done?

• Goal: Attempt to reduce the number of accesses to the slower levels.
• How?

Locality

Temporal Locality (locality in time) – If an item is referenced, it will tend to be referenced again soon.

Spatial Locality (locality in space) – If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Caches

• Each level is a sub-set of the level below.

Cache Hit – address requested is in cache
Cache Miss – address requested is NOT in cache
Cache line size (chunk size) – the number of contiguous bytes that are moved into the cache at one time

Examples

\[
\begin{align*}
  x &= a + 6; &  x &= a[0] + 6; \\
  y &= a + 5; &  y &= a[1] + 5; \\
  z &= 8 \times a; &  z &= 8 \times a[2]; 
\end{align*}
\]