AVL Trees Revisited

• Balance condition:
  Left and right subtrees of every node have heights differing by at most 1

  – Strong enough: Worst case depth is $O(\log n)$
  – Easy to maintain: one single or double rotation

• Running time for
  – Find ? $O(\log n)$
  – Insert ? $O(\log n)$
  – Delete ? $O(\log n)$
  – buildTree ? $O(n \log n)$
Single and Double Rotations
AVL Trees Revisited

• What **extra info** did we maintain in each node?
  – The height of each node

• **Where** were rotations performed?
  – At the bottom-most node where an imbalance is detected

• How did we **locate** this node?
  – Check balance on our way up out of the recursion

• Seems like a lot of work, doesn’t it?
  – Any wacky ideas?
Splay Trees

• Blind adjusting version of AVL trees
  – Why worry about balances? Just rotate like crazy!
  – Don’t track anything, store anything, just do it!

• *Amortized* time per operations is $O(\log n)$

• Worst case time per operation is $O(n)$
  – But guaranteed to happen rarely

• Splay Trees : AVL Trees :: Skew Heaps :: Leftist Heaps
Recall: Amortized Complexity

If a sequence of $M$ operations takes $O(M f(n))$ time, we say the amortized runtime is $O(f(n))$.

- Worst case time per operation can still be large, say $O(n)$
- Worst case time for any sequence of $M$ operations is $O(M f(n))$

Average time per operation for any sequence is $O(f(n))$

Amortized complexity is worst-case guarantee over sequences of operations.
Recall: Amortized Complexity

• Is amortized guarantee any weaker than worstcase?
  Yes, it is only for sequences

• Is amortized guarantee any stronger than averagecase?
  Yes, guarantees no bad sequences

• Is average case guarantee good enough in practice?
  No, adversarial input, bad day, …

• Is amortized guarantee good enough in practice?
  Yes, again, no bad sequences

• When is amortized maybe not good enough?
  If that very rare O(n) operation will kill somebody
If you’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!

All the way to the root!
Do it all with AVL single rotations?

Consider the ordered “list tree” at left. Now do \texttt{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…
Splaying node $k$ to the root: Need to be careful!

One option (that we won’t use) is to repeatedly use AVL single rotation until $k$ becomes the root: (see Section 4.5.1 for details)
Splaying node $k$ to the root: Need to be careful!

What’s bad about this process?

$r$ is pushed almost as low as $k$ was

Bad seq: find($k$), find($r$), find(...), ...

```
A  B  C
S  r  k
D  E  F
```

```
A  B  C
S  r  p
D  E  F
```
Find/Insert in Splay Trees

1. Find or insert a node $k$
2. Splay $k$ to the root using three operations: zig-zag rotation, zig-zig rotation, plain old zig rotation
   Depending on path from current location to the root

Why could this be good??
1. Helps the new root, $k$
   o Great if $k$ is accessed again
2. And helps many others!
   o Great if many others on the path are accessed
Splay: Zig-Zag

Just like an…

AVL double rotation

Which nodes improve depth?

\( k \) and its original children

Helps those in blue
Hurts those in red
Is this just two AVL single rotations in a row?

Not quite – we rotate g and p, then p and k

Why does this help?

Same number of nodes helped as hurt. k and its children benefit.
Special Case for Root: Zig

- Relative depth of $p$, $Y$, $Z$?
  - Down 1 level
- Relative depth of everyone else?
  - Nodes under $X$ have been repeatedly raised

Why not drop zig-zig and zig all the way?
- Zig only helps one child!
Splaying Example: Find(6)

Find(6)
Still Splaying 6

Zig-zig
Finally…

1

6

3

2

5

4

1

6

3

2

5

4

Zig
Another Splay: Find(4)
Example Splayed Out

Zig-zag
But Wait…

What happened here?

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

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

That still holds, though we must account for the previous steps used to create this tree.

What is the worst case?

Find keys in sorted (or reverse sorted) order
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.
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

- 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?

Amortized guarantee fails!
Bad sequence: find(leaf $k$), find($k$), find($k$), …
Splay Operations: Insert

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

What if we didn’t splay?

Amortized guarantee fails!
Bad sequence: insert($k$), find($k$), find($k$), …
Splay Operations: Remove

Everything else splayed, so we’d better do that for remove

Now what?
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

Similar to BST delete – find max = find element with no right child

Does this work to join any two trees? No, need L < R
Delete Example

Delete(4)

Find max

Find(4)
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  
    Like what? Skew heaps! (don’t need to wait)
  – 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
Splay E
Splay E
Splay E
Splay E
Other Possibilities?

• Could use different balance conditions, different ways to maintain balance, different guarantees on running time, ...

• Many other balanced BST data structures
  – Red-Black trees
  – AA trees
  – Splay Trees
  – 2-3 Trees
  – B-Trees
  – ...

Red-Black Trees

Structure property:

- Every node is “colored” either red or black.
- The root is black.
- If a node is red, its children are black. (A leaf can be red.)
- For each node, all paths down to null pointer must contain the same number of black nodes.
Red-Black Trees

Notes:
• Uses the standard rotations, plus some coloring operations, to maintain structure.
• **Worst case** find, insert, delete: $O(\log n)$
• Has nice top-down, non-recursive implementation.
• Java uses top-down red-black trees (TreeMap)
Treaps

Order property:
- Each node has a randomly assigned priority value, in addition to its key value.
- Tree has both BST and heap order!

Orange = low priority value, Yellow = high priority value.