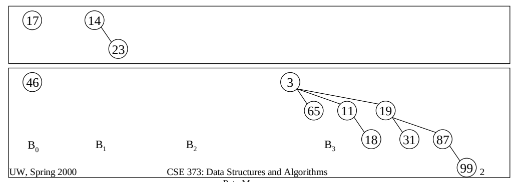
CSE 373: Amortized Analysis & Splay Trees

Pete Morcos
University of Washington
5/15/00

http://www.cs.washington.edu/education/courses/cse373/00sp

Back to Binomial Queues

- Recall BQ operations
  - Merge
  - Insert
  - DeleteMin



BQ Analysis

- How long does it take to build a BQ with N Inserts?
- A BQ can contain up to log N trees, so worst-case time for Insert is O(log N)
  - Leading us to believe O(N log N) for total cost
- Actually, it's only O(N)
  - In contrast, N Inserts on a heap are O(N log N)
  - But, there was an O(N) method called BuildHeap
- We need a better analysis technique than just multiplying the worst case for an individual operation by the number of ops

A Pattern

- Count the work done for each insertion
  - Cost varies each time
  - But, we *can* write an expression that is the same for each step
  - Let T_i = # of trees at step *i*
  - $(T_i - T_{i-1}) + C_i = 2$
  - Each time we add a tree, the step is cheap
  - When we remove trees, the step is more expensive
- Key observation:
  - Worst case (log N) can't happen every time

before insert	after insert	cost
	B_0	1
B_0	B_1	2
B_1	B_0 B_1	1
B_0 B_1	B_2	3
B_2	B_0 B_2	1
B_0 B_2	B_1 B_2	2
B_1 B_2	B_0 B_1 B_2	1
B_0 B_1 B_2	B_3	4
B_3	B_0 B_3	1

The Potential

- The number of trees is called the *potential*; a bookkeeping device
  - Think of it as a bank account
- We give each Insert a "budget" of 2 work units
  - i.e., Insert has an "amortized cost" of O(1), even though its worst-case cost is O(log N)
- The potential tracks how much "credit" we've accumulated by doing under-budget operations
- The potential is non-negative (# of trees), so we know we never go into "debt"

BQ Potential

- As we observed before,
  - cost + ΔP = 2
- For any sequence of N steps
  - Σ (cost + ΔP) = 2N
  - Σ cost = 2N – Σ ΔP
  - But, Σ ΔP = P, and P $\geq$ 0
  - So, Σ cost $\leq$ 2N = **O(N)**
- Key points for amortization:
  - P begins at its minimum value
  - cost + ΔP is a simple function (the "budget")

Previous tree	old P	Insert cost	ΔP
	0	**1**	**+1**
B_0	1	**2**	**0**
B_1	1	**1**	**+1**
B_0 B_1	2	**3**	**-1**
B_2	1	**1**	**+1**
B_0 B_2	2	**2**	**0**
B_1 B_2	2	**1**	**+1**
B_0 B_1 B_2	3	**4**	**-2**
B_3	1		

Why Amortize?

- We could have just figured out the total cost by hand
  - Costs form an interesting regular pattern; remember the draw_ruler homework?
- But that would only be valid for sequences of nothing but Inserts
  - Throwing in a DeleteMin would violate the calculation
- However, using the *same* potential, DeleteMin can be shown to have amortized cost of O(log N)
  - Thus, *any* sequence of M DeleteMins and N Inserts costs O(N + M log N)

Splay Trees—a new ADT

- A splay tree is a binary search tree
- We know that an unbalanced tree has O(N) worst case behavior
  - A sequence of M operations is, then, O(MN)
- AVL trees used rotations to keep tree balanced, giving worst case time of O(M log N)
- Splay trees can be unbalanced, but each time a node is accessed, we move it to the root via *splaying*

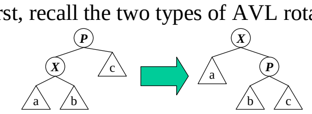
Splaying

- After accessing a node, we have to move it
  - If we didn't, repeated operations would cost the same. If the node was deep [O(N)], the total cost will be too high.
  - We choose to move it all the way to the root
- We have to maintain the binary search tree property
- AVL rotations were a way to move a node within a tree without destroying the property
- Repeated use of rotations can move a node all the way to the root

Splay Rotations

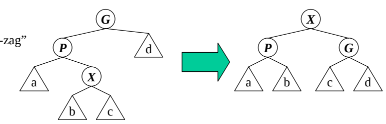
- First, recall the two types of AVL rotation

"zig"



X moves up one level

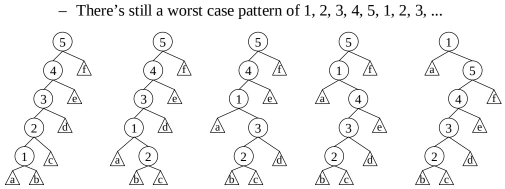
"zig-zag"



X moves up two levels

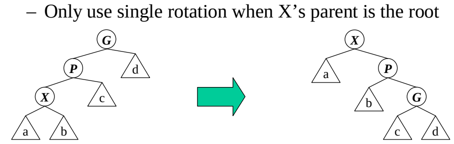
Not Good Enough

- Consider using single rotations to fix a typical worst-case tree
  - Worst [O(MN)] access pattern is 1, 1, 1, 1, ...
- Yes, 1 is at root, but other nodes all got worse
  - There's still a worst case pattern of 1, 2, 3, 4, 5, 1, 2, 3, ...



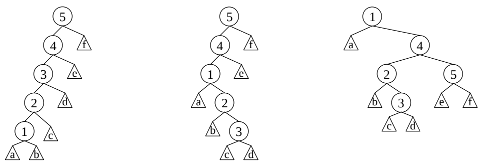
zig-zig

- One more rotation type needed, to replace "zig" (single-rotation)
  - This new rotation helps to balance the tree
  - Only use single rotation when X's parent is the root



Using zig-zig

- Result is a wider, shallower tree
  - 5 is still fairly shallow, unlike previous single rotation example
  - No node is at depth 4 any more



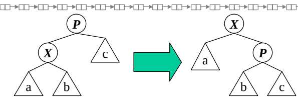
Splay Analysis Difficult

- Splaying can cause nodes to move down as well as up
  - Even all the way down to N-1 depth
  - Consider accessing 1, 2, 3, 4, 5 in previous example
- So any *single* operation could always cost O(N), even after we've done several splays
- The way we've done analysis so far, we'd be forced to say worst case for M operations is O(MN)
- Turns out not to be as bad as we think
  - Splays do improve the tree; some operations will be better than O(N)
- We need a more sophisticated analysis, using amortization

Splay Potential

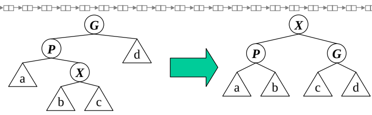
- The potential function is more complex this time
- At each step i, let $S_i(X)$ = size of the subtree rooted at X (including X itself)
- Let $R_i(X)$= log $S_i(X)$, known as the *rank* of X
- Potential P = Σ R_i, over entire tree
- We want to compute an amortized bound on the total cost of a splay, which is an unknown sequence of zigs, zig-zags, and zig-zigs

zig Amortized Time Cost



- actual cost is 1, and ΔPot = ΔR(X) + ΔR(P)
  - R(P) obviously decreases, so ΔR(P) is negative
  - Thus, ΔPot <= ΔR(X)
  - and amortized time budget is: AT <= 1 + ΔR(X)
    - remember, budget = actual cost plus Δpotential

zig-zag Amortized Cost

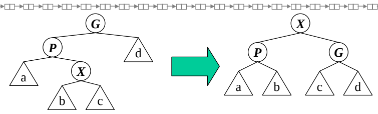


- actual cost is 2, ΔPot = ΔR(X) + ΔR(P) + ΔR(G)
  - AT = 2 + [$R_f(X)$ - $R_i(X)$] + [$R_f(P)$ - $R_i(P)$] + [$R_f(G)$ - $R_i(G)$]
  - but $R_f(X)$ = $R_i(G)$, so
  - AT = 2 - $R_i(X)$ + [$R_f(P)$ - $R_i(P)$] + $R_f(G)$
  - and, since $R_i(X)$ <= $R_i(P)$,
  - AT <= 2 – 2 * $R_i(X)$ + $R_f(P)$ + $R_f(G)$

a sidetrack

- It turns out that, for positive a, b, c
  - If a + b <= c
  - Then log a + log b <= 2 log c – 2
  - (see book for proof)
- In terms of this problem, using sizes and ranks,
  - If S(a) + S(b) <= S(c)
  - Then R(a) + R(b) <= 2 R(c) - 2

zig-zag continued



- We ended with AT <= $2 - 2 * R_i(X) + R_f(P) + R_f(G)$
- We see that $S_f(P) + S_f(G) <= S_f(X)$
  - So, $R_f(P) + R_f(G) <= 2 R_f(X) - 2$, using last slide
  - Thus, AT(zig-zag) <= $2 * R_f(X) - 2 * R_i(X) = 2 \Delta R(X)$
- A similar analysis holds for zig-zig:
  - AT(zig-zig) <= $3 \Delta R(X)$

Putting it together

- We have:
  - AT(zig) <= $1 + \Delta R(X)$ <= $1 + 3 [R_f(X) - R_i(X)]$
  - AT(zig-zag) <= $2 \Delta R(X)$ <= $3 [R_f(X) - R_i(X)]$
  - AT(zig-zig) <= $3 \Delta R(X)$ <= $3 [R_f(X) - R_i(X)]$
- We repeat the steps until X replaces the root R
  - zig only happens once, so the 1 is only added once
  - Each time, the last $R_f(X)$ is cancelled by the next $-R_i(X)$
- The only terms left are: AT(total) <= $1 + 3*[R_{root}(X) - R_{initial}(X)]$
- $R_{initial}(X)$ could be as low as 0, $R_{root}(X)$ as high as log N
- Thus, total budget for whole sequence is **O(log N)**

Whew!

- You don't have to remember all this analysis for splay trees
- The important points are:
  - What a splay tree is
  - How a splay tree works
    - Rotation details of splaying
    - Find(X)—search normally, then splay X, or last node seen if X not found
    - Insert(X)—insert normally, then splay X
  - Max cost of a single splay is O(N)
    - But all those rotations make future accesses faster
  - Amortized cost of a single splay is O(log N)
    - *Any* sequence of M operations costs O(M log N)

Amortization Summary

- If worst case cost can't happen every time, amortization may give a tighter bound
  - Worst case often makes many future steps cheaper
  - Actual cost usually complex, and varies each step—hard to use
- Trick is to simplify a complex cost function by adding a potential
  - actual cost + Δpotential = simpler function (the amortized budget)
- Potential starts at its minimum (usually zero)
  - If it could later drop below start value, we'd be over budget!
- Amortization useful when thinking about arbitrary sequences of mixed operations ("N Inserts, M Deletes, etc.")
  - Must use same potential function to analyze each one

Amortization is not Average-case!

- Amortized analysis says
  - There are N operations
  - Show that together, all N ops cost < C no matter what
  - Then amortized cost is the average cost C / N
- We take the average of several steps used to process an input, true for *any* input!

- Average-case analysis says
  - There are Z possible inputs
  - Show that total cost of all inputs is X
  - Then the average cost of running the program is X / Z
- We take the average of several *inputs*, but some inputs may be worse than average!