CS 4349 Lecture—November 29, 2017

Main topics are #amortized_analysis, #dynamic_arrays, and #splay_trees.

Prelude

- Homework 11 due November 6th. It has more questions, but they should be easier to answer than the usual ones.
- YOU MUST TURN THIS ONE IN AT THE BEGINNING OF LECTURE SO WE CAN TALK ABOUT IT (or any time Monday).

Dynamic Arrays

- Today, we're going to apply some of the amortized analysis tools from Monday to analyze two data structures.
- We'll start by discussing a variant of the standard array which we'll call a "dynamic array".
- You've probably seen these before in different languages' standard libraries under different names. C++ calls them vectors. Java calls them ArrayLists. Python just calls them lists.
- Say we have a dynamic array A. Let A.size denote the number of elements we have stored in A.
- To keep things simple, we'll just concentrate on lookups and insertions for now.
  - Lookup(A, i): Return the element at position i where 1 ≤ i ≤ A.size.
  - Insert(A, x): Add element x to position A.size + 1 and increment A.size.
- If we implement A with a standard array, we'll need to allocate a brand new copy of A every time we do an insertion!
  - Lookup takes Theta(1) time.
  - Insert takes Theta(A.size) time.
- Well, that's a pretty bad insertion time!
- One easy trick to save on insertion time is to store the elements of A in a larger standard array that leaves room for expansion.
- Call the standard array A.storage and let A.capacity be its length. We'll initially start with A.storage being a 0 length standard array.
- Lookups still take Theta(1) time in the worst case.
- Lookup(A, i):
  - return A.storage[i]
- We would like the capacity to be larger than the size so we can cheaply insert elements, but we don't want the capacity to be too large, because we'd like to use only Theta(A.size) space.

- Call alpha(A) = A.size / A.capacity the *load factor* of A.
- We only need make a copy of the underlying array whenever we do an insert while alpha(A) = 1.
- To balance space vs. time to make copies, let's double the size of the capacity when we fill up A.storage so that alpha(A) ≥ 1/2 at all times.
- Here's the code for an insert:
- Insert(A, x):
  - if A.size = 0
    - A.storage ← new array of length 1
    - A.capacity ← 1
  - if A.size = A.capacity
    - A.temp ← new array of length 2 * A.size
    - for i ← 1 to A.size
      - A.temp[i] ← A.storage[i]
    - delete A.storage
    - A.storage ← A.temp
    - A.capacity ← 2 * A.size
  - A.storage[A.size + 1] ← x
  - A.size ← A.size + 1
- So how long do insertions take?
- In the worst case, they still take Theta(A.size) time, because we have to do all that copying.
- But we can use a simple taxation scheme to show the amortized cost is much better.
- The running time of Insert is proportional to the number of array assignments, so we'll only count those.
- Every Insert, tax each element of the standard array *A.storage* 2 /A.capacity dollars. Also, tax the new element x $1.
- That $1 pays for actually inserting x into the array during the current insert.
- Now, suppose we need to double the capacity of A. Well, last time we doubled the capacity, A.size = A.capacity / 2. So we have done A.capacity / 2 insertions, meaning we have a credit of A.capacity / 2 * 2 / A.capacity = 1 dollar for every element in A.storage. The credit is enough to pay for copying that element to A.temp!
- The amortized cost of an insert is the total amount of tax we collected, which is 1 + A.capacity * 2 / A.capacity = 3.
- Intuitively, we wait quite a while between moments when we double the capacity, meaning the average insertion time is still constant.

- OK, but what if we want to do deletions too?
  - Delete(A): Remove the element at position A.size and decrement A.size.
- Well, we could just do the decrement and ignore the element we're supposed to be

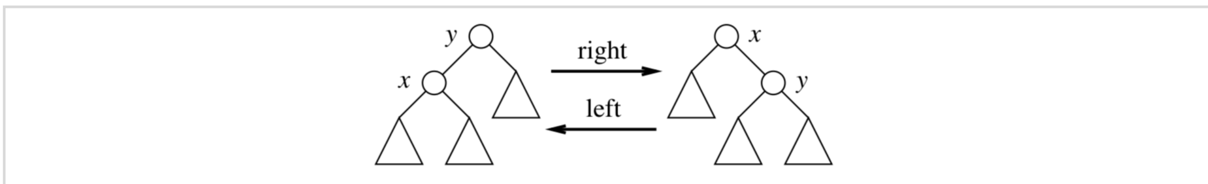
deleting. But then we could end up with an arbitrarily small load balance, and that is pretty wasteful of space.

- Another strategy would be to do essentially the opposite of Insert. If your load balance ever falls below 1/2, then make a new underlying standard array of length A.size and copy all the elements over. But then we have this issue where we could insert an element that doubles the capacity, delete an element to halve the capacity, insert an element to double the capacity, and so on. The cost of *all* these operations would be Theta(A.size).
- So what we can do instead is relax our requirement on alpha(A) being at least 1/2 to being at least 1/4. If the load balance ever drops to below 1/4, then we'll *halve* the capacity.
- Delete(A, x):
  - A.size ← A.size - 1
  - if A.size = 0
    - delete A.storage
    - A.capacity ← 0
  - if A.size < A.capacity / 4
    - A.temp ← new array of length 2 * A.size
    - for i ← 1 to A.size
      - A.temp[i] ← A.storage[i]
    - delete A.storage
    - A.storage ← A.temp
    - A.capacity ← 2 * A.size
- For the analysis, we can use that exact same taxation scheme but charge the tax during both Insert and Delete. We'll have enough credit to pay for copies during Inserts, because we still have A.capacity = 2.size any time A.storage is replaced.
- But now we need to have done at least A.capacity / 2 deletions before we can halve the capacity of A. So any element that needs copied to the new storage will have A.capacity / 2 * 2 / A.capacity = 1 tax credit available to do the copying.
- The tax scheme still charges exactly $3 per Insert or Delete, so they both have amortized cost O(1).
- This general scheme of storing an underlying data structure of double capacity works with other kinds of data structures as well such as hash tables.

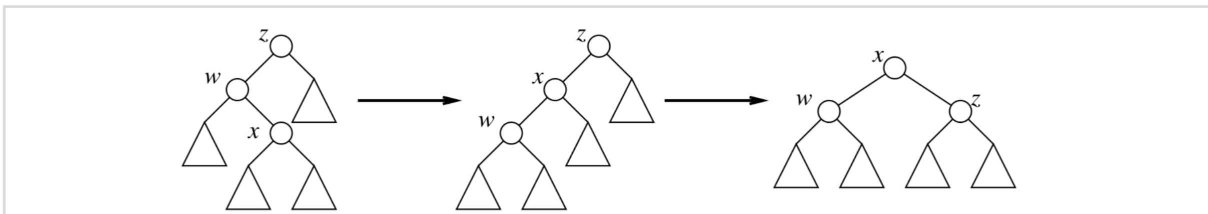
Splay Trees

- Let's finish by discussing a data structure for balanced binary search trees.
- So, let's say we have a binary search tree. Nodes are labeled with *keys* and each node has two children. The left child has a smaller key and the right child has a larger key.
- Ideally, the tree is balanced so every root to leaf path has length O(log n), meaning Search, Insertion, and Deletion all take O(log n) time.

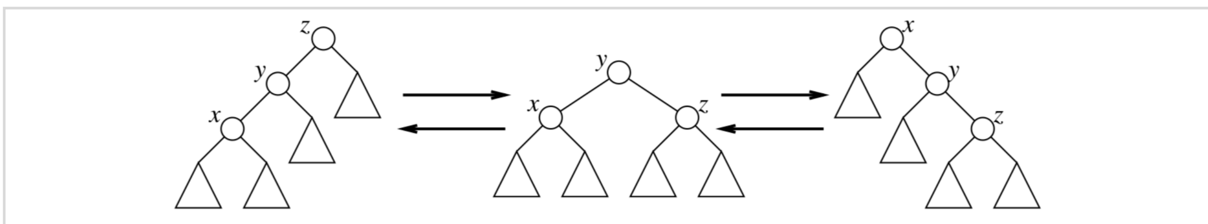
- But you can get a very unbalanced tree where some searches can take O(n) time.
- One way to deal with long search times is to *guarantee* the binary search tree is balanced at all times. This usually involves storing some extra information about the tree like colors on the nodes and imposing some rules involving that extra information. Algorithms for insertion and deletion can get pretty gross.
- Today, I'll show you another way to keep search times down, at least on average, using a binary search tree called a splay tree. I find this method much easier to digest.
- Like the data structures you've probably seen already, we'll need to use an operation called a rotation. A rotation at node x decreases its depth by 1 and increases its parent's depth by 1. They can be performed in constant time.



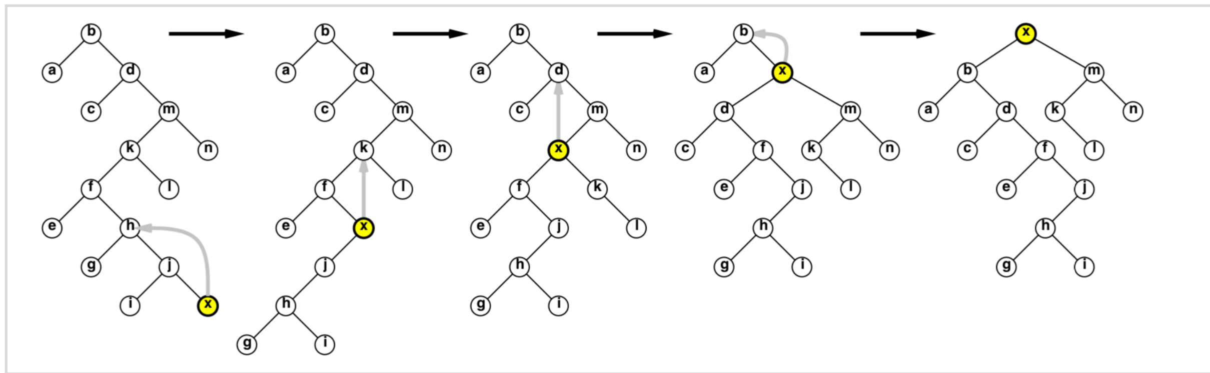
- For splay trees, we actually need to work with pairs of rotations called double rotations. Double rotations at x leave the depth of x's parent the same as it started, but they increase the depths of x's grandparent by 1 or 2.
- If x's parent and grandparent pointers go opposite directions, then the double rotation we'll use is called a zig-zag. Rotate at x twice:



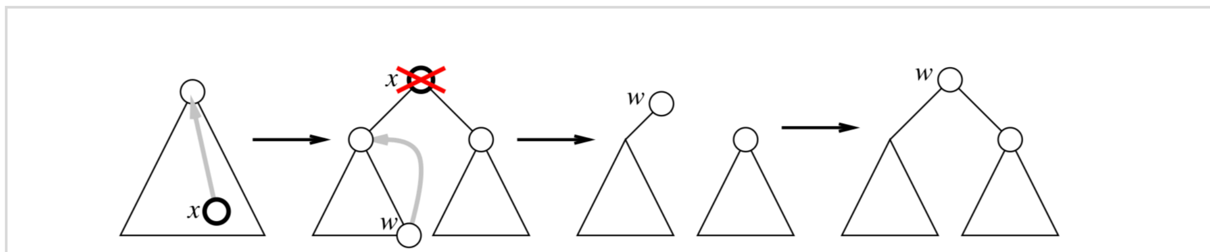
- If x's parent and grandparent pointers go in the same direction, we'll use a double rotation we can call a roller-coaster. Rotate at x's parent and then at x:



- Finally, there's one more operation called a splay. To splay x, we perform as many double rotations as we can, and then maybe do a single rotation at the end so that x becomes the root of the tree. Splay takes Theta(depth(x)) time.



- In a *splay tree*, you do pretty much every operation by finding a node and then splaying it. Intuitively, we're keeping popular nodes near the root so we can find them faster.
    - Search: Find node x or its predecessor if x is not in the tree. Splay the node found.
    - Insert: Insert x as a leaf as usual. Splay it.
    - Delete: Find x, splay it, and delete it. Find its predecessor in the left subtree, splay it, join it to the right subtree.



- In every case, we do a constant number of searches and splays. So the number of rotations during splays gives a bound on the running time.
- But how do we analyze the number of rotations? The easiest way is with the potential method.
- size(x) := # nodes in x's subtree
- ranks(x) := floor{lg size(x)}
- phi := sum_x rank(x)
- If the tree is perfectly balanced, phi = Theta(n). If it's a path, phi = Theta(n log n), so it does get bigger as things get "worse".
- Suppose we do a rotation. Let rank(x) refers to the rank **before** the rotation, and rank'(x) refer to the rank after.
- Access Lemma: The amortized cost of a single rotation at x is at most 1 + 3rank'(x) - 3rank(x). The amortized cost of a double rotation is at most 3rank'(x) - 3rank(x).
- Time permitting, I'll prove it before we leave. But look at what happens during a splay.
- The amortized costs of those double rotations have a nice telescoping sum, so the total amortized cost of a splay is 1 + 3rank'(x) - 3rank(x) where rank'(x) is the rank after the *whole* splay. After the splay, x is the root, so rank'(x) = floor(lg n).
- The amortized cost of the splay is at most 3lg n - 1 = O(log n).
- I should emphasize that the tree can still become unbalanced, and some splays may be very expensive, even Theta(n). But the *average* splay uses only O(log n) rotations.

Other Bounds

- Now, the cool thing about splay trees is that they perform better if the sequence of searches is "nice" in some way.
- Say we give each node x a non-negative real weight w(x).
- s(x) := total weight of x's subtree
- redefine r(x): floor(lg s(x))
- phi: = sum_x rank(x) still
- The access lemma is still true, and depending on what weights we use, we get different amortized bounds.
- For example, we set all weights to 1 to get O(log n).
- But we get other bounds as well.
- Static Optimality: Suppose each node x is accessed t(x) times, and let T = sum_x t(x). The amortized cost of accessing x is O(log T - log t(x)).
  - In other words, frequently accessed nodes tend to stay near the root.
  - Proof: Set w(x) = t(x) for each node x so that rank(x) ≥ floor(lg t(x)).
  - The bound implies searches on a splay tree take only a constant factor longer than searches on the best static / unchanging search tree for those access frequencies.
  - And somehow this is possible without the splay tree knowing in advance how often the nodes will be accessed.
- dist(x, z): number of nodes between x and z if arranged in order (dist(x, x) = 0).
- Static Fingers: For any fixed node f, the amortized cost of accessing x is O(lg dist(f, x)).
  - In other words, don't stray far from f, and searches go faster.
  - Proof: Set w(x) := 1/dist(x, f)^2 for each node x. s(root) ≤ sum_{i = 1}^infinity 2/i^2 = pi^2 / 3 = O(1), and rank(x) ≥ lg w(x) = -2 lg dist(f, x).
- Here's some other properties whose proof require more than just setting some weights.
- Working Set: The amortized cost of accessing x is O(log D), where D is the number of *distinct* nodes accessed since we last accessed x.
- Scanning: Splaying all nodes in order takes Theta(n) rotations. This is a special case of the…
- Dynamic Finger Theorem: Immediately after accessing y, the amortized cost of accessing x is O(log dist(x, y)).
- All of these theorems are special cases of a popular conjecture that says the following: Given any search sequence, the total time spent doing searches in a splay tree is at most a constant times the total time spent doing searches and rotations in *any* dynamic binary search tree; even one specifically designed to behave well for that particular search sequence. This is even more powerful than static optimality.