Data Structure Review

COMP 221 — Suhas Arehalli

Now that we've developed a formal language around algorithmic analysis — proofs of correctness and Big-O analysis — we have the power to revisit some of the algorithmic material you've seen in Data Structures! You should see this as an opportunity to both review some exciting ideas in algorithmic analysis that you learned about in Data Structures and practice using the new tools we've developed in a setting where the tricky bits of the algorithm are (hopefully) still a bit familiar. Before reading through this document, which highlights proofs of interesting properties of the relevant data structures, you should make sure you remember how the relevant data structures work by reviewing your Data Structures readings and notes!

1 AVL Trees

Recall that a major goal for our tree unit of Data Structures was building **self-balancing binary search trees**. Let's first work through all of the terminology there briefly.

1.1 Background

A tree (for our purposes) is a data structure that organizes units called *nodes* into branching structure: each node has some number of child nodes that it is a parent of. Some number of nodes are *leaves*, which have no children, but only one node can have no parent — the *root*. Nodes can also only have at most one parent, which prevents situations like siblings being in a parent-child relationship or children being parents of their parents (or grandparents or so on).

This definition is a bit messy, but luckily we can begin to craft a more elegant definition by noting that trees are a *recursive* structure. Every node n in a tree T is the root of a *subtree* T_n that simply ignores every nodes that is not a descendant[1] of n in T. Therefore, every tree consists of a root node and some number of subtrees rooted at its children. We can formalize this into a recursive definition:

Def 1. *A **Tree** T is one of the following:*

1. *A Node n.*
2. *A pair (n, C) where n is a node (called the root of T) and C is a set of Trees (whose roots are referred to as n's children).*

We can see that this definition has the same structure as a recursive algorithm or inductive proof: A base case (Definition 1) that references a single node tree and a recursive case that defines

[1] For lack of space, I'll assume you're familiar with the geneological terminology surrounding trees from Data Structures: parent, child, and sibling, but also ancestor, descendant, and so on.

a tree in terms of other trees — those that are the subtrees of the root (Definition 2). However, if we look closely, we can see that we define a single node tree in two distinct ways: As n for some node n, and as $(n, \emptyset)$[2]. To make this less messy (but less obviously of the base/recursive case form), we can propose this alternative definition:

Def 2. *A **Tree** T is a pair (n, C) where n is a node (called the root of T) and C is a set of Trees (whose roots are referred to as n's children).*

Of course, we're interested here in *binary* trees: Those where we fix the number of children each node has to be at most 2. We also label the up-to-2 children as *left* or *right* children, imposing an ordering on them. We'll adopt a recursive definition for this special kind of tree that fits this new structure:

Def 3. *A **Binary Tree** T is either*

1. *An empty tree $NULL$*
2. *A 3-tuple (n, L, R) where n is a node and L, R as Binary Trees. We call n the root of T, L the left subtree (and its root the left child of n) and R the right subtree (and its root the right child of n).*

Here we have a clear base case (the empty tree $NULL$) and a recursive definition. Think of $NULL$ as representing a null pointer in an implementation of a binary tree like you saw in Data Structures. This lets us include the notion of a node having a right child without a left child ($(n, NULL, R)$) and allowing the empty tree to a be a tree (something our general tree definition did not give us).

To make sure we're on the same page, consider the visual representations of a BST you're used to, as in the one in Fig. 1. Under our definition, this is not a tree — it's a visual representation of the tree

$$T = (3, (1, NULL, (2, NULL, NULL)), (5, (4, NULL, NULL), (6, NULL, (7, NULL, NULL))))$$

. It's a mouthful, but this should seem not-too-dissimilar from the way you build a binary tree object in a high-level programming language — just treat $NULL$ as a null pointer and make each tuple a call to a constructor!

We can also define basic concepts like containment in the context of a binary tree:[3]

Def 4. *A node e is in a Binary Tree T (notated $e \in T$) if T is not $NULL$ and for $T = (n, L, R)$, either*

1. *$n = e$,*
2. *$e \in L$, or*
3. *$e \in R$.*

Which just so happens to align perfectly with a traversal-based tree search algorithm! We can turn this definition into an algorithm case-by-case and almost-accidentally rewrite a tree traversal algorithm (a *pre-order* traversal, in the case of Alg. 1).

[2] recall from Discrete Math that $\emptyset$ denotes the empty set — one with no elements!

[3] here, we implicitly assume that whatever a node is, equality is defined for it. in Java/COMP128 terms, this means the `equals` method is defined appropriately

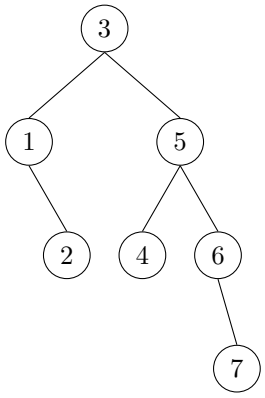

Figure 1: A Sample BST visualization. This will also happen to be a Fibonacci tree of height 4, but don't worry about that until this figure shows up again.

Algorithm 1 A Search for an element in a tree

```
function TreeContains(Binary Tree T, Node e)
    if T = NULL then
        return False
    end if
    Let T = (n, L, R)
    if n = e then
        return True
    end if
    inL ←TreeContains(L, e)
    inR ← TreeContains(R, e)
    return inL | inR
end function
```

This also lets us write a nice formal definition of a binary *search* tree:[4]

Def 5. *A **Binary Search Tree** is a Binary Tree $T = (n, L, R)$ such that*

1. *$\forall e \in L$, $e < n$,*
2. *$\forall e \in R$, $e \geq n$, and*
3. *L and R are BSTs.*

And here we can get to the topic at hand: AVL trees! These simply add an additional balance constraint to a binary search tree based on height.

Def 6. *Let $h(T)$ denote the **height** of a binary search tree T. Then,*

$$h(T) = \begin{cases} 0, & T = NULL, \\ 1 + \max(h(L), h(R)), & T = (n, L, R). \end{cases}$$

Note how we break our definition into the same cases laid out in our definition!

Def 7. *An **AVL tree** is a Binary Search Tree T such that either*

1. *$T = NULL$, or*
2. *$T = (n, L, R)$ for some node n and*
   (a) *L and R are AVL Trees*
   (b) *$|h(L) - h(R)| \leq 1$*

For convenience, I'll refer to the property in 2b as the **AVL Balance Property**, as this is the constraint that enforces balance. However, this is, as of yet, a claim that goes unproven — does this property actually get us to a tree that is suitably balanced (i.e., as you learned in Data Structures, does it make $h(T) \in O(n)$ where n is the number of nodes in the tree?

1.2 AVL Trees are Balanced

The answer is, as you might expect, yes. In fact, we'll show the following:

Statement 1. *For any AVL tree T with n nodes and height $h(T) \geq 1$, $h(T) \leq 2 \log n + 1$.*

A nice convenient bound on how tall our AVL tree can be in terms of the number of nodes! Your instinct should hopefully be to immediately translate this into our preferred Big-O terminology, getting the following corollary

Corr. 1. *For any AVL tree T with n nodes, $h(T) \in O(\log n)$*

[4] And here, we assume that nodes are totally ordered (i.e., notions of less than and greater than are also well defined. In Java/COMP128 terms, this means Node implements `Comparable`.

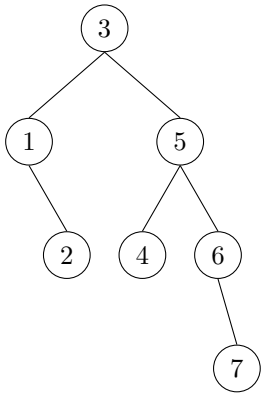

Figure 2: A Fibonacci Tree of height 4. Note how every level obeys the AVL property. We can also see it's left subtree rooted at 1 is a Fibonacci tree of height 2 and it's right subtree rooted at 5 is one of height 3.

You should have no trouble proving this follows from the statement and the definition of Big-O!

Now to the statement itself: we'll prove this via induction! The idea of the proof is not too tricky in concept, but is certainly daunting because there are a lot of moving parts! The key idea is identifying the worst possible case for our inequality: Since we fix $h(T)$ in each case, we'll be trying to show that even in the worst possible AVL tree of this height (the one with the smallest n, and thus the strictest upper bound) the inequality is still true. We'll prove a couple small cases ($h(T) \leq 2$) by enumerating through the possible tree structure, but for larger cases, we will appeal to a general argument that the AVL tree T_k of height $h(T) = k$ with the fewest nodes is what we call a *Fibonacci* Tree: That is, it has the structure $T_k = (n, T_{k-1}, T_{k-2})$ or $T(n, T_{k-2}, T_{k-1})$ for some node n, where T_{k-1} and T_{k-2} are Fibonacci Trees of heights $k-1$ and $k-2$ respectively. These create a scenario where at each node, we maintain the maximum possible AVL imbalance (a difference in subtree heights of exactly 1), so intuitively it makes sense that these are the worst you can get with the AVL balance property still holding. We then apply the IH to each subtree (which have smaller heights, and thus are subject to our IH) and do some algebra to show the inequality holds.

First, let's break out the proof of the idea that the minimum-node AVL trees have a Fibonacci tree structure:

Lemma 1. *The AVL tree T of height $h(T) = k \geq 1$ with the minimum number of nodes has the form $T = (n, L, R)$ where*

1. *one of L, R is an AVL tree of height $k-1$*
2. *one of L, R is an AVL tree of height $k-2$*

Proof. First, note that the fact that both L and R are AVL trees follows from the definition of an AVL tree.

Now, we'll show 1. Recall that by definition, $h(T) = k \geq 1$ iff $k = \max h(L), h(R) + 1$. This implies that either $h(L)$ or $h(R) = k - 1$, as desired.

Next we show 2. Assume without loss of generality that via 1, $h(L) = k - 1$ (for the parallel case, just swap L and R). Then note that since T is an AVL tree, we know that $|h(L) - h(R)| \leq 1$. This allows for 3 possible values of $h(R)$. We will rule out each one other than $h(R) = k - 2$.

Case 1: Suppose for contradiction that $h(R) = k$. However, if this were true, then $h(T) = \max k, k - 1 + 1 = k + 1$, but we were given that $h(T) = k$ — a contradiction. Thus $h(R) \neq k$.

Case 2: Suppose for contradiction that $h(R) = k - 1$. Via 1, we know that R has a subtree of height $k - 2$ — call that subtree R'. Let $n(T)$ denote the number of nodes in a tree T. Since R' doesn't contain the root and is a subtree of R, we know that $n(R') < n(R)$. Now let's construct $T' = (n, L, R')$ (i.e., we exchange R for R'). Now note that T' is an AVL tree of height k: T' obeys the AVL balance property ($|h(L) - h(R')| = 1$, R' is an AVL tree (since it's a subtree of R, another AVL tree), and the tree still has height k (since height is determined by L, which has not changed). However, it has strictly fewer nodes than T, since $n(R') < n(R)$, which contradicts that T is an AVL tree of height k with the minimum number of nodes! Thus, $h(R) \neq k - 1$.

This leaves $h(R) = k - 2$ as the only remaining option, allowing us to conclude 2. □

The interesting part here is Property 2, particularly the case showing why the other sub-tree can't also have height $k - 1$. Informally, we show that anytime we build an AVL tree with two subtrees of height $k - 1$, we can remove some nodes from one subtree without violating any AVL Tree properties by allowing height of one subtree to drop to $k - 2$. This might seem a bit fancy for what is intuitively a simple claim (a tree of height $k - 1$ can't have fewer nodes than one of height $k - 2$), but this is a nice example of a broad strategy for proving why an object with certain properties can't be the "smallest": Show that you can modify that object to make it smaller without violating any of those property. This will come up later!

It might also help to visualize the argument, as in Fig. 3.

Now, let's use the lemma to prove statement 1, reproduced here for convenience:

Statement 1. *For any AVL tree T with $n(T)$ nodes and height $h(T) \geq 1$, $h(T) \leq 2 \log n(T) + 1$.*

Proof. Proceed by induction over $h(T)$.

Base Case(s): Suppose $h(T) = 1$. There is only one tree of height 1 — the one containing a single node — which is trivially an AVL tree. Then observe that

$$\begin{aligned} 2 \log n(T) + 1 &= 2(0) + 1 \\ &= 1 \geq 1 \end{aligned}$$

Now suppose $h(T) = 2$. note that $N(T) \geq 2$, as we need at least two nodes in a tree of

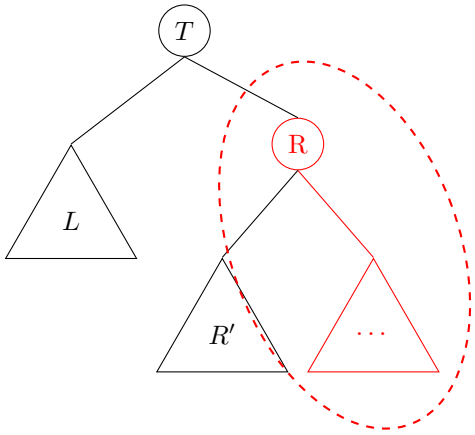

Figure 3: A schematic of the critical portion of the proof of Property 2 in our Lemma 1. If L and R have the same height, we can construct a tree T' with the same height and fewer nodes by remove all of the nodes highlighted in red and having R' become the right child of T.

height 2. Observe that for $h(T) = 2$ and $n(T) \geq 2$

$$\begin{aligned} 2\log n(t) + 1 &\geq 2\log 2 + 1 \\ &\geq 2(1) + 1 \\ &\geq 3 \qquad\qquad \geq h(T) \end{aligned}$$

as desired.

Inductive Step: Assume as our strong inductive hypothesis (IH) that for any AVL tree of height $1 \leq h(T) \leq k$, we know $h(T) \leq 2\log n(T) + 1$. We will show the inequality still holds for any AVL tree of height $k + 1$.

By lemma 1, the minimum-node AVL tree of height $k + 1$ will consist of a root node, one AVL subtree of height k and one AVL subtree of height $k - 1$.

Let n' be the number of nodes in our minimum-node AVL tree. The taller subtree of height k must obey our IH since it is also AVL, and thus we have that it contains n_1 nodes where

$$\begin{aligned} k &\leq 2\log n_1' + 1 \\ n_1' &\geq 2^{\frac{k-1}{2}} \end{aligned}$$

Similarly, for the shorter subtree, we have that it contains n_2 nodes where

$$n_2' \geq 2^{\frac{k-2}{2}}$$

Thus, our full tree contains all the nodes in both subtrees plus a root, giving us

$$\begin{aligned} n' &= 1 + n'_1 + n'_2 \\ n' &\geq 1 + 2^{\frac{k-1}{2}} + 2^{\frac{k-2}{2}} \\ n' &\geq 1 + (\sqrt{(}2) + 1)2^{\frac{k-2}{2}} \geq 2(2^{\frac{k-2}{2}}) \\ n' &\geq \ldots \qquad\qquad\qquad\quad \geq 2^{\frac{k}{2}} \end{aligned}$$

Now consider an arbitrary AVL Tree T of height $h(T) = k + 1$. Since n' is the number of nodes in the *minimum-node* AVL tree of height $k + 1$, we know $n(T) \geq n'$, and thus

$$\begin{aligned} 2n(T) &\geq 2^{\frac{k}{2}} \\ 2\log n(T) &\geq k \\ 2\log n(T) + 1 &\geq k + 1 \end{aligned}$$

Thus, for any AVL tree of height $h(T) = k + 1$, we have that $h(T) \leq 2\log n(T) + 1$.
Thus, by induction, all AVL trees of height ≥ 1 have that $h(t) \leq 2\log n(T) + 1$ □

Now we get to confirm our conclusion from Data Structures: AVL trees are height-balanced, and our $O(h)$ insert, delete, and remove algorithms are truly $O(\log n)$!

To be clear, we haven't proved anything about the correctness of our rebalancing algorithms (i.e., that our insert+rebalance algorithm actually guarantees that we end up with an AVL tree!). We *can* do that (we have all the tools!), but for the purpose of brevity I will only show a couple of fun facts formally in the next section.

1.3 Rotations preserve the BST property

Consider first our formal definition of a rightward rotation:

Def 8. *A right rotation of a binary tree T, $rot_r(T)$ is*

$$rot_r(T) = \begin{cases} NULL, & T = NULL \\ (n, NULL, R), & T = (n, NULL, R) \\ (p, L', (n, R', R)), & T = (n, (p, L', R'), R) \end{cases}$$

This should (mostly) align with what you remember from Data Structures (Draw out some trees, translate them to our tuple notation, and then confirm that rot_r does what you expect!). Note that we handle the *edge cases* for rotations as elegantly as we can: If the tree is empty, we do nothing! If there is no left subtree, we do nothing! This lets us make nice claims like the following:

Statement 2. *If T is a binary search tree, $rot_r(T)$ is a binary search tree.*

That is, a rightward rotation will preserve the BST property!

Proof. Proceed by cases:

Case 1: If $T = NULL$ or $T = (n, NULL, R)$ for some node n, and BST R, then $rot_r(T) = T$, and since we're given that T is a BST, we're done.

Case 2: Let $T = (n, (p, L', R'), R)$ for some nodes n, p and BSTs L', R', R. Since T (and, consequently, (p, L', R')) is a BST, we know that:

$$p < n \tag{1}$$
$$\forall e \in L', e < p \tag{2}$$
$$\forall e \in R', e \geq p \tag{3}$$
$$\forall e \cup L' \cup R', e < n \tag{4}$$
$$\forall e \in R, e \geq n \tag{5}$$

Now we must show that $rot_r(T) = (p, L', (n, R', R))$ is a BST. Let's start by showing (n, R', R) is a BST working from the definition. First, we must show that $\forall e \in R', e < n$ — this follows from (4). Then we must show that $\forall e \in R, e \geq n$ — this is exactly (5). Finally, we must show that both R' and R are BSTs — this follows from the fact that they are subtrees of the BST T (same goes for L'!)

Now we show that $rotr_r(T)$ is a BST. We have established that both children are BSTs, so we need only show that the left and right subtrees follow the ordering property. First, we must show that $\forall e \in L', e < p$ — this is simply (2). Now we must show that $\forall e \in (n, R', R), e \geq p$. We handle this by cases as well:

Case 2.1: If $e = n$, then this follows from (1).

Case 2.2: If $e \in R'$, this follows from (3).

Case 2.3: If $e \in R$, note that $e \geq n$ by (5) and $n > p$ by (1). Combining these gives us the desired inequality: $e \geq p$

Thus, if T is a BST, $rot_r(T)$ is a BST. □

Note that this proof is bulky, but not super interesting in terms of technique. This is a style of proof that is really just a matter of first unpacking the definitions of the objects in play (i.e., generating the 5 inequalities w.r.t elements of T using the definition of a BST) and then repackaging them into the desired properties (i.e., demonstrating that the 5 inequalities are equivalent to all of the inequalities needed to show that $rot_r(T)$ is also a BST). Make sure you can sit down and understand the process here. Test yourself by writing the appropriate definition and proof for the left-rotation equivalent!

Now that we've established that rotations preserve BST-ness, we can understand why all of our rebalancing operations used rotations when we needed to modify the structure of our BST — a rotation will move nodes around (and modify the height of nodes!) while never undoing the work that our BST insert did. Now we need only show that our use of rotations in each of the 4 cases of rebalancing (single rotation left/right, double rotation left/right) restore the AVL balance property. This is equally tedious, but also doable with our skills:

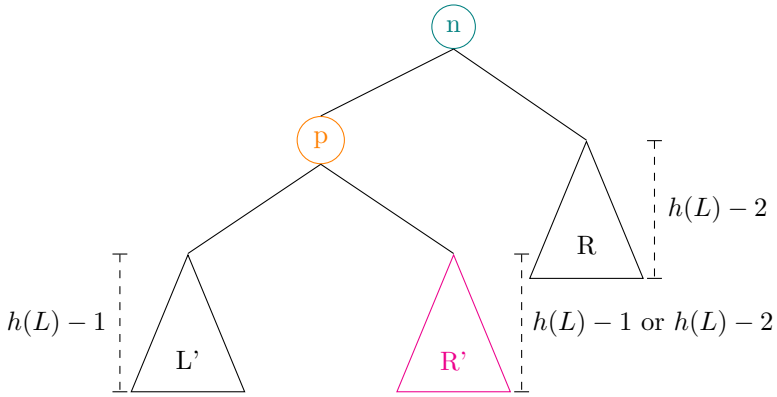

Figure 4: The situation under which a rightward rotation at n (i.e., a rotation on the full tree T) will restore balance.

1.4 AVL rotations restore the AVL property

Let's be quick about this one — following this should be more practice in the inequality manipulation we saw when working with Big-O definitions.

First, our claim — pay attention to the numerous assumptions! These specify both the pre-conditions of balancing (you added a single new element to what was previously an AVL tree) and the case we're in (we do a single rotation right because we added our new node within the left-left grandchild's subtree).

Statement 3. *Let $L = (p, L', R')$ be an AVL Tree with $h(L') \geq h(R')$. Let $T = (n, L, R)$ be a BST with $h(L) - h(R) = 2$ and R an AVL Tree. $rot_r(T)$ is an AVL Tree.*

Visually, this should look something like Fig. 4 pre-rotation (note the color-coding!). The first part of the proof is establishing all of the heights of relevant components in terms of $h(L)$ (indicated by the labels of the dashed lines), and then we conclude by using these facts to show that the height differences post-rotation are within AVL bounds.

Proof. First, note that $rot_r(T) = (p, L', (n, R', R))$.

By Statement 2, we know $rot_r(T)$ is a BST. Now we need only show that $|h(L') - h((n, R', R))| \leq 1$ and that $|h(R') - h(R)| \leq 1$.

Let's first attempt to compute all of the relevant heights (or the best bounds we can for them) in terms of $h(L)$.

Since L' is the left subtree of L and $h(L') \geq h(R')$, we know by the definition of the height function h that $h(L') = h(L) - 1$.

Now we compute $h((n, R', R))$. We are given that $h(R') \leq h(L') = h(L) - 1$, and from rearranging the given $h(L) - h(R) = 2$ we get that $h(R) = h(L) - 2$. From this we can conclude that

$$\begin{aligned}\max(h(R'), h(R)) &\leq \max(h(L) - 1, h(L) - 2) \\ &\leq h(L) - 1.\end{aligned}$$

Note that knowing the exact value of $h(R)$ w.r.t. $h(L)$ let's us compute a lower bound for the maximum as well:

$$\begin{aligned}\max(h(R'), h(R)) &\geq h(R) \\ &\geq h(L) - 2\end{aligned}$$

Thus we can bound $\max(h(R), h(R'))$ and subsequently $h((n, R', R))$:

$$\begin{aligned}h(L) - 2 \leq \max(h(R), h(R')) &\leq h(L) - 1 \\ h(L) - 1 \leq 1 + \max(h(R), h(R')) &\leq h(L) \\ h(L) - 1 \leq h((n, R', R)) &\leq h(L)\end{aligned}$$

Now, since we already know $h(L')$, can get us to the balance calculation for $rot_r(T)$ by subtracting $h(L')$ from all sides:

$$\begin{aligned}(h(L) - 1) - h(L') \leq h((n, R', R)) - h(L') &\leq h(L) - h(L') \\ (h(L) - 1) - (h(L) - 1) \leq h((n, R', R)) - h(L') &\leq h(L) - (h(L) - 1) \\ 0 \leq h((n, R', R)) - h(L') &\leq 1\end{aligned}$$

And thus it must be that $|h(L') - h((n, R', R))| \leq 1$ (the difference is either 0 or 1!).

The process for the right subtree's balance calculation is similar: We established above that $h(R) = h(L) - 2$ and that $h(R') \leq h(L) - 1$. Now note that since L is an AVL tree, $h(R') \geq h(L') - 1 = h(L) - 2$, giving us upper and lower bounds on $h(R')$: $h(L) - 2 \leq h(R') \leq h(L) - 1$. Thus, we can subtract $h(R)$ from all sides:

$$\begin{aligned}h(L) - 2 - h(R) \leq h(R') - h(R) &\leq h(L) - 1 - h(R) \\ h(L) - 2 - (h(L) - 2) \leq h(R') - h(R) &\leq h(L) - 1 - (h(L) - 2) \\ 0 \leq h(R') - h(R) &\leq 1\end{aligned}$$

And thus $|h(R') - h(R)| \leq 1$ as desired (the difference is either 0 or 1!).

Thus We've established that (n, R', R) is an AVL tree (a BST that satisfies the balance property and has AVL subtrees, since R is given to be AVL and R' is the right subtree of the given AVL tree L). We can then conclude that $rot_r(T)$ is an AVL tree, as it's left/right subtrees are both AVL (L' the left subtree of given AVL tree L, and (n, R', R) from the prior sentence) and it satisfies the balance property.

□

To show that rebalancing is truly correct, we need to establish a couple other facts — that the other 3 cases are correct, the correctness of BST insert and delete, that the only errors in balance after insert and delete will be due will be those that correspond to the four cases of AVL rebalancing, and the inductive argument that shows that rebalancing recursively upward from the new leaf will result in a full AVL tree. These are all within our grasp — and all would make good exercises to use your knowledge — but, for space, I'll keep things brief and end it here.

2 Amortized Analysis and ArrayStack

Now, let's move to another interesting bit from Data Structures we're now ready to see in it's full light: *Amortized* Time Complexity.

Typically, when we analyze the time complexity of a function, we consider it in isolation. Consider for instance, push in an ArrayStack, as in Alg. 2, which indicates that most of the time, we will simply need to do an assignment and increment — and $O(1)$ operation. However, occasionally, we will need to call ExpandCapacity, which is $O(n)$ where n is the size of the array, which accounts for having to copy over the existing array to a new memory allocation. Under a standard worst-case analysis, this algorithm must be $O(n)$, since a single call will sometimes take $O(n)$ time.

Algorithm 2

```
Let Array A[1...N] and Integer top represent the state of the stack.
function Push(Element e)
    if top > N then
        ExpandCapacity(A, 2N)
    end if
    A[top] ← e
    top ← top + 1
end function
```

This misses the point of the data structure's design though — these calls are not made in isolation! Data structures maintain state across multiple calls, and the data structure was designed such that these ExpandCapacity calls are vanishingly rare, as the size of the underlying array doubles each time we have to expand! Amortized analyses account for this.

Now there are a few kinds of Amortized analysis, and I'll show you two: One that's fairly straightforward but is sometimes tricky to apply in complex situations — the aggregate method — and another that is a bit trickier conceptual but feels more powerful – the accounting method. Use whichever makes things easier, as they both have their perks!

2.1 The Aggregate Method

Consider a sequence of operations $o_1, o_2, \ldots, o_n$ that we perform on our data structure, with each o_i have a corresponding cost, which we'll denote with c_i. This new notation, but the only difference is that we're moving from the language of growth functions for individual calls to a function to sequence of costs associated with a number of function calls in sequence. We then say the amortized cost of every operation is the *average* cost of each operation. That way, we distribute the high cost of a rare operation among the hopefully numerous cheap operations. What's critical, though, is

that we still try and consider the *worst-case* sequence — we can't simply avoid costly operations altogether. However, since those operations are assumed to occur in-sequence

Def 9. *Given a set of operations O, $\hat{C}$ is an amortized cost of each operation $o \in O$, if for any $n \in \mathbb{N}$ and sequence of operations $\{o_i\}_{1 \leq i \leq n} \subset O$ and their corresponding costs $\{c_i\}_{1 \leq i \leq n}$, we have that*

$$\frac{1}{n} \sum_{i=1}^{n} c_i \leq \hat{c}$$

That is, we use the term amortized cost to refer to an upper bound on the average cost of an operation in any sequence.

Now here's an overly formal proof of what should be a confidence-granting observation:

Statement 4. *Suppose each operation $o \in O$ has a fixed cost c_o that is independent of the data structure's state. Then each operation's minimum possible amortized cost is $\max_{o \in O} c_o$.*

Proof. Let $o' \in O$ be such that $c_{o'} = \max_{o \in O} c_o$. Then for any arbitrary n, consider the sequence of n o's, By our definition of amortized cost, we must have any any $\hat{c}$ satisfies

$$\begin{aligned} \hat{c} &\geq \frac{1}{n} \sum_{i=1}^{n} c_{o'} \\ &\geq c_{o'} \\ &\geq \max_{o \in O} c_o \end{aligned}$$

Now we will show that this is the sequence of n operations with the greatest possible cost. Assume for contradiction that it isn't, and there is some sequence of greater total cost that has some $\bar{o} \in O$, $o' \neq \bar{o}$ within it. Since sequence costs are fixed regardless of position, this means that there is some such that $c_{\bar{o}} > c_o$. But this contradicts that o' was defined to be the operation with the greatest fixed cost. Thus for any sequence $\{o_i\}_{1 \leq i \leq n}$,

$$\hat{c} = \max_{o \in O} c_o \geq \frac{1}{n} \sum_{i=1}^{n} c_{o_i}$$

and thus $\hat{c} = \max_{o \in O} c_o$ is a valid amortized cost. □

If none of our operations ever change in cost, our amortized cost assigns the cost of the worst operation of each. This means that unless we actually do something to make expensive operations infrequent, amortized cost doesn't let us cheat! What about something more interesting:

Statement 5. *Suppose we have a data structure with an $O(n)$ insert operation and an $O(1)$ remove operation, where n is the number of elements in the data structure. These operations have an amortized cost $\hat{c} \in O(n)$ where n is the length of the sequence.*

Proof. As any remove operation will reduce the number of elements sorted in the data structure, the upper bound on the time cost of a sequence will be a sequence of n inserts. The cost of this sequence with an initial size of k is thus

$$
\begin{aligned}
&= \frac{1}{n}\sum_{i=1}^{n} O(k+i) \\
&\in O(\frac{1}{n}\sum_{i=1}^{n}(k+i)) \\
&\in O(\frac{1}{n}(nk+\sum_{i=1}^{n} i) \\
&\in O(k+\frac{1}{n}(\frac{n(n+1)}{2})) \\
&\in O(k+\frac{1}{2}(n+1)) \\
&\in O(n)
\end{aligned}
$$

□

Note the change in use of n — now instead of being in terms of the size of our data structure, it's in terms on the number of operations! We acknowledge that the size of n is no longer an unknown, but is now limited by the fact that our operations only allow for constant growth in the size of the data structure! Additionally, it turns out that nothing funky happens when we account for this — this doesn't magically give us any speedups! This should convince us further that we've really earned our amortized time complexities when they do improve!

Now let's finally analyze ArrayStacks properly:

Statement 6. *For an ArrayStack with standard pop and peek operations and a push that doubles array size when capacity is reached, these operations have amortized time complexity* $O(1)$.

Proof. Observe that pop and peek are both always $O(1)$, and either don't affect or can only decrease array size. Thus the worst case sequence will always involve only push operations.

Let c be the initial array size, and k be in the initial number of elements stored in the array. Each push in a sequence of n inserts will then have a cost of $O(1)$ when array does not hit capacity and an additional cost of $O(i+k)$ when it does. Since we start at capacity c, we will double when we have c elements stored, $2c$ elements stored, and so on. Since we start at k reach a total size of $k+n$, this happens for every size of the form $2^i c$ for some i between k and

$k + n$. Analyzing this inequality further,

$$\begin{aligned} k \leq 2^i c &< k + n \\ \frac{k}{c} \leq 2^i &< \frac{k+n}{c} \\ \log k - \log c \leq i &< \log(k+n) - \log c \\ \log k - \log c \leq i &< \log(k+n) - \log c \end{aligned}$$

which gets us to a cost of

$$= \frac{1}{n}(\sum_{i=1}^{n} O(1) + \sum_{i=\lceil \log k - \log c \rceil}^{\lceil \log(k+n) - \log c \rceil} O(2^i))$$

Next, we move to upper bound the second summation rather than solve it explicitly. First, drop the lower bound on i to 1 — in big-O terms, this is just a constant since it doesn't depend on n. Then, we can recall the identity $\sum_{i=1}^{m} 2^i = 2^{m+1} - 1$, and thus our amortized cost, $\hat{c}$ is

$$\begin{aligned} &\leq \frac{1}{n}(\sum_{i=1}^{n} O(1) + O(2^{\lceil \log(k+n) - \log c \rceil})) \\ &\leq \frac{1}{n}(\sum_{i=1}^{n} O(1) + O(2^{\log(k+n)})) \\ &\leq \frac{1}{n}(O(n) + O(k+n)) \\ &\in O(1) \end{aligned}$$

Exactly as desired. □

2.2 The Accounting Method

Another more sophisticated approach is the **accounting method**. Like the name implies, we take inspiration from the world of accounting.

The idea is the same as standard amortization: We want to find a way to demonstrate that the number of cheap $O(1)$ pushes eventually subsidize our costly $O(n)$ resizing push so it is effectively $O(1)$. Note that we're doing this on an operation-specific basis here (unlike the aggregate method!).

How do we do this? Like you might do when building a budget where costs are uncertain. Take, for example, an electric bill. We may have cheap months (i.e., not the Minnesota winter with it's heating bills...) and expensive months, but your income doesn't change depending on the weather. What you can do to offset this is to set aside some extra money in the cheap months so that you have some savings to pay for some of the bill in later expensive months — this is the idea for the accounting method!

Formally, keep the same setup as before — sequences of operations $\{o_i\}$ and corresponding costs $\{c_i\}$, and we have the same intuition that the most expensive sequence will be the one with *only*

pushes. What we add is a separate set of fixed, amortized costs per operation $\hat{c}_o$ — one for each operation.

Def 10. *Under the accounting method, for a set of operations O, $\hat{c}_o$ is the amortized cost for operation $o \in O$ if for any $n \in \mathbb{N}$, sequence of operations $\{o_i\}_{1 \le i \le n}$ from the initial state, and corresponding sequence of costs $\{c_i\}_{1 \le i \le n}$, we have that*

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_{o_i}$$

Informally, all we're saying is that our accounting is good as long as for any sequence of operations, we never fall into debt — the sum of our estimated costs never exceeds the real cost! I.e., if we following our budget, we will never go over regardless of what happens.

The tricky part here is that we need to first come up with what the amortized costs should be — i.e., find $\hat{c}_o$ for each operation o — and only then prove the inequality for the worst case. This is similar to finding the appropriate constants to show Big-O, Ω, Θ — do a bit of scratchwork to find what works and then rewrite a polished version!

Statement 7. *For the ArrayStack as defined above, push, pop, and peek have an amortized cost of $O(1)$.*

Proof. Let k be the maximum cost of a simple push (no resizing), pop, peek, or copying a single element to a new array. Since each of these operations is $O(1)$, $k \in \mathbb{N}$.

Let $\hat{c}_{push} = 3k$, $\hat{c}_{peek} = k$, and $\hat{c}_{pop}$. Then see that, based on how we defined k, our amortized costs pay for the real cost of the pop and peek operations. Then note that peek does not affect the size of our ArrayStack and push reduces it (reducing the rate at which we reach a costly $O(n)$ push operation) so our worst case sequence in terms of cost must be n pushes. First, let's analyze the true cost after n inserts, noting that we'll pay a large cost at every power of 2 to copy over every element to a new array.[5]

$$\begin{aligned}
\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} k + \sum_{i=1}^{\lfloor \log n \rfloor} 2^i k &\le nk + k \sum_{i=1}^{\log n} 2^i \\
&\le nk + 2^{\log n + 1} k - 1 \\
&\le nk + 2nk - 1 \\
&\le 3nk - 1
\end{aligned}$$

Then we observe that the amortized cost for this sequence is simply $\sum_{i=1}^{n} 3k = 3nk$, which is gives us that

$$\sum_{i=1}^{n} c_i \le 3nk - 1 \le 3nk = \sum_{i=1}^{n} \hat{c}_{push}$$

And since we've established that the sequence of all pushes is the most costly, we can conclude that the amortized cost of push is $3k \in O(1)$, and pop/peek are $k \in O(1)$. □

Note that the math is actually not too different from what we saw above for the aggregate method — the key is that the $O(n)$ cost of resizing is shows up at a *logarithmic* rate — it only shows up at powers of two as we have our ArrayStack *double in size*! That way, we end up with our cheap push cost of k at each step and then a logarithmic number of linear costs[6] — the $nk + \sum_{i=1}^{\log n} 2^i k$ term that we see in the accounting version of things, and the messier version in the aggregate method version. We then get to apply our nice summation identity — $\sum_{i=1}^{k} 2^i = 2^{k+1} - 1$ — to arrive at the fact that a linear number of operations results in a linear total cost.

Now, to close out and double check that nothing funky is going on here, let's take a look at an *alternative* (bad) ArrayStack implementation — one that increases capacity by k rather than doubling in size. This means that we incur a linear cost every m steps, getting us a real total cost after n pushes of[7]

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} k + \sum_{i=1}^{\lfloor \frac{n}{m} \rfloor} imk \\
&= nk + \frac{mk}{2}(\lfloor \frac{n-1}{m} \rfloor (\lfloor \frac{n-1}{m} \rfloor + 1)) \\
&\geq nk + \frac{mk}{2}(\frac{n-1}{m})(\frac{n-1}{m} + 1) \\
&\geq nk + \frac{k}{2m}((n-1)^2 + m(n-1)) \\
&\in \Omega(n^2)
\end{aligned}$$

Oh no! Over n operations, we get $\Omega(n^2)$ costs, which means that each operation is on average linear in cost in this worst-case sequence. From this, we can conclude that it is, in fact, our choice to double array size that makes the amortized analysis work. Props to good data structure design choices!

[6] careful with the indexing of the summation — it *looks* exponential, but that's with respect to i. $2^i k$ represents the cost when $n = 2^i$.

[7] Assuming an initial size of m