CSE 332

Data Abstractions

Amortized Analysis

Outline

Stack ADT & ArrayStack Analysis

Stack ADT

`push(`**val**`)`	Adds **val** to the stack.
`pop()`	Returns the **most-recent** item not already returned by a pop. (Errors if empty.)
`peek()`	Returns the **most-recent** item not already returned by a pop. (Errors if empty.)
`isEmpty()`	Returns true if all inserted elements have been returned by a pop.

Let's analyze the time complexity for these various methods. (You know how they work, because you just implemented them!)

Method	Time Complexity
`isEmpty()`	$\Theta(1)$
`peek()`	$\Theta(1)$
`pop()`	$\Theta(1)$
`push(`**val**`)`	??

push is actually slightly more interesting.

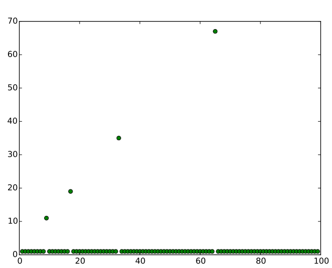
Analyzing push for an ArrayStack

Best Case

There's more space in the underlying array! Then, it's $\Omega(1)$.

Worst Case

If there's no more space, we double the size of the array, and copy all the elements. So, it's $\mathcal{O}(n)$.



Insight: Our analysis seems wrong. Saying linear time feels wrong.

Analyzing push for an ArrayStack

This is where "amortized analysis" comes in. Sometimes, we have a **very rare** expensive operation that we can "charge" to other operations.

Intuition: Rent, Tuition

You pay one big sum for a long period of time, but you can afford it because it happens very rarely.

Back to ArrayStack

Say we have a full Stack of size n. Then, consider the next n pushes:

- The next push will take $\mathcal{O}(n)$ (to resize the array to size $2n$)
- The $n-1$ operations after that will all be $\mathcal{O}(1)$, because we know we have enough space

Considering these operations in aggregate, we have n operations that take $(c_0 + c_1 n) + (n-1) \times c_2$ time.
So, how long does **each** operation take:

$$\frac{(c_0 + c_1 n) + (n-1) \times c_2}{n} \le \frac{n \max(c_0, c_2) + c_1 n}{n} = \max(c_0, c_2) + c_1 = \mathcal{O}(1)$$

Analyzing `push` for an `ArrayStack` — 4

What happens if we change our resize rule to each of the following:

- $n \to n+1$
  This is really bad! We can only amortize over the single operation which gives us:
  $$\frac{n}{1} = \mathcal{O}(n)$$
- $n \to \frac{3n}{2}$
  This still works. Now, we go over the next $\frac{3n}{2} - n$ operations:
  $$\frac{n + (n/2 - 1) \times 1}{\frac{n}{2}} = \mathcal{O}(1)$$
- $n \to 5n$
  This is good too:
  $\frac{n + (4n-1) \times 1}{4n} = \mathcal{O}(1)$

Which is better $2n$, $\frac{3n}{2}$, or $5n$?

Java uses $\frac{3n}{2}$ to minimized wasted space.

Binary Counter — 5

Time Complexity of A Binary Counter

We would like to analyze an n-bit binary counter with the single method `increment()`. For example,

$$0000 \xrightarrow{1} 0001 \xrightarrow{2} 0010 \xrightarrow{1} 0011 \xrightarrow{3} 0100 \xrightarrow{1} 0101 \xrightarrow{2} 0110 \xrightarrow{1} 0111 \xrightarrow{4}$$
$$1000 \xrightarrow{1} 1001 \xrightarrow{2} 1010 \xrightarrow{1} 1011 \xrightarrow{3} 1100 \xrightarrow{1} 1101 \xrightarrow{2} 1110 \xrightarrow{1} 1111$$

Asymptotic Time Complexity of `increment`

The best case is that we change a single bit: $\mathcal{O}(1)$
The worst case is that we change all the previous bits: $\mathcal{O}(n)$

Binary Counter — 6

Time Complexity of A Binary Counter

We would like to analyze an n-bit binary counter with the single method `increment()`. For example,

$$0000 \xrightarrow{1} 0001 \xrightarrow{2} 0010 \xrightarrow{1} 0011 \xrightarrow{3} 0100 \xrightarrow{1} 0101 \xrightarrow{2} 0110 \xrightarrow{1} 0111 \xrightarrow{4}$$
$$1000 \xrightarrow{1} 1001 \xrightarrow{2} 1010 \xrightarrow{1} 1011 \xrightarrow{3} 1100 \xrightarrow{1} 1101 \xrightarrow{2} 1110 \xrightarrow{1} 1111$$

Amortized Time Complexity of `increment`

As always, the first step is to split the operations into "chunks". Where's a good splitting point?

$$\underbrace{11\cdots1}_{n} \to \underbrace{11\cdots1}_{n+1}$$

Looking at the ones we've already calculated, we get:

n	$n=0$	$n=1$	$n=2$	$n=3$
$T(n)$	1	3	5	7

Great. So, it looks like each range takes $T(n) = 2^{n+1} - 1$ bit changes. Let's prove it.

Binary Counter — 7

Time Complexity of A Binary Counter

We would like to analyze an n-bit binary counter with the single method `increment()`.

$$0000 \xrightarrow{1} 0001 \xrightarrow{2} 0010 \xrightarrow{1} 0011 \xrightarrow{3} 0100 \xrightarrow{1} 0101 \xrightarrow{2} 0110 \xrightarrow{1} 0111 \xrightarrow{4}$$
$$1000 \xrightarrow{1} 1001 \xrightarrow{2} 1010 \xrightarrow{1} 1011 \xrightarrow{3} 1100 \xrightarrow{1} 1101 \xrightarrow{2} 1110 \xrightarrow{1} 1111$$

Amortized Time Complexity of `increment`

n	$n=0$	$n=1$	$n=2$	$n=3$
$T(n)$	1	2	4	8

We go by induction on n. Let $P(n)$ be the statement

"incrementing the counter from $\underbrace{11\cdots1}_{n}$ to $\underbrace{11\cdots1}_{n+1}$ changes $2^{n+1} - 1$ bits"

for all $n \in \mathbb{N}$.

Base Case ($n = 0$).
This changes 1 bit. Note that $2^1 - 1 = 2 - 1 = 1$. So, the base case holds.

Induction Hypothesis.

Suppose $P(k)$ is true for all $0 \le k \le \ell$ for some $l \in \mathbb{N}$.

Binary Counter — 8

Amortized Time Complexity of `increment`

$P(n) =$ "incrementing from $\underbrace{11\cdots1}_{n}$ to $\underbrace{11\cdots1}_{n+1}$ changes $2^{n+1} - 1$ bits"

Induction Step. We are interested in the range $\underbrace{1\cdots1}_{\ell+1} \to \underbrace{11\cdots1}_{\ell+2}$.

We split this range into pieces:

- $0111\cdots11 \to 1000\cdots00$
- $1000\cdots00 \to 1000\cdots01$
- $1000\cdots01 \to 1000\cdots11$
- ...
- $1001\cdots11 \to 1011\cdots11$
- $1011\cdots11 \to 1111\cdots11$

Luckily for us, the bits that change in these ranges are **identical** to the previous cases! In particular, in the $(i+1)$st range, we do $2^{i+1} - 1$ changes. Note that the first step takes $\ell + 2$ changes.
Thus, the entire range changes:

$$\left(\sum_{k=0}^{\ell+1} 2^k - 1\right) + (\ell + 2) = \frac{2^{\ell+2} - 1}{2 - 1} - \ell - 1 + \ell + 1 = 2^{\ell+2} - 1$$

Binary Counter — 9

Time Complexity of A Binary Counter

We would like to analyze an n-bit binary counter with the single method `increment()`.

$$0000 \xrightarrow{1} 0001 \xrightarrow{2} 0010 \xrightarrow{1} 0011 \xrightarrow{3} 0100 \xrightarrow{1} 0101 \xrightarrow{2} 0110 \xrightarrow{1} 0111 \xrightarrow{4} 1000$$

Amortized Complexity of `increment`

So, now we know incrementing from $2^k - 1$ to $2^{k+1} - 1$ changes $2^{k+1} - 1$ bits.

Note that $2^{k+1} - 1 - (2^k + 1) = 2^k$. So, the amortized cost of incrementing the counter is $\frac{2^{k+1} - 1}{2^k} \le 2$.

A New Data Structure!

Amortized Sorted Array Dictionary

Consider the following data structure:

- We have an array of **sorted** arrays of `ints`. The ith array has size 2^i. So, for example:

  `a[0]`: 5
  `a[1]`: `null`
  `a[2]`: 6, 8, 9, 11
  `a[3]`: 1, 10, 12, 14, 16, 18, 20, 22

- The single method `add(`**val**`)` which works as follows:
  - Make a new array `temp`: **val**
  - Until we find a `null` array:
    - If `a[i]` is `null`, set `a[i] = temp`.
    - Otherwise, `temp = merge(a[i], temp); a[i] = null;` and loop.

Asymptotic Complexity of `add`

First, let's define our variables. There are n arrays and $m = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ elements in the array.
In the worst case, we need to go through all n arrays, merging at each step. In this case, our runtime is

$$\sum_{i=0}^{n} 2(2^i) = 2(2^{n+1} - 1) = 2m = \mathcal{O}(m)$$

A New Data Structure!

Amortized Sorted Array Dictionary

`a[0]`: 5
`a[1]`: `null`
`a[2]`: 6, 8, 9, 11
`a[3]`: 1, 10, 12, 14, 16, 18, 20, 22

- The single method `add(`**val**`)` which works as follows:
  - Make a new array `temp`: **val**
  - Until we find a `null` array:
    - If `a[i]` is `null`, set `a[i] = temp`.
    - Otherwise, `temp = merge(a[i], temp); a[i] = null;` and loop.

Amortized Complexity of `add`

A natural split-up would be going from a data structure with only the largest array filled to a data structure with only the next largest array filled. Aha! **This is the same problem as the binary counter!** We showed previously that there are 2^n edits in the block from 2^{n-1} to 2^n, but this time, each of those edits is $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$. So, the total cost is going to be $\sum_{i=0}^{n-1} i(2^i - 1) + n(n) \approx 2^n(n-2)$. Since we're amortizing over 2^{n-1} operations, this gives us $\mathcal{O}(n) = \mathcal{O}(\log m)$. Not bad!

And one more thing...

How does `search` look in this data structure?

We have to binary search through each array. In the worst case, we have to binary search through **all of them**:

Asymptotic Complexity of `search`

$$T(n) = \sum_{i=0}^{n} \lg(2^i) = \frac{n(n+1)}{2}$$

Notice that n here is the **number of arrays**. The number of elements is **logrithmic in the number of arrays**. That is if there are n arrays, then there are $m = 2^1 + 2^2 + \cdots + 2^n = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ elements. That is, $n = \lg(m+1)$. So, the runtime is $\frac{\lg(m+1)\lg(m+1)+1}{2} = \mathcal{O}(\lg^2(m))$