CSC263 Week 7

Thursday

http://goo.gl/forms/S9yie3597B
Announcement

Pre-test office hour today at BA5287
11am~1pm, 2pm~4pm

PS5 out, due next Tuesday
Recap: Amortized analysis

• We do amortized analysis when we are interested in the total complexity of a sequence of operations.
  • Unlike in average-case analysis where we are interested in a single operation.

• The amortized sequence complexity is the “average” cost per operation over the sequence.
  • But unlike average-case analysis, there is NO probability or expectation involved.
For a sequence of $m$ operations:

Amortized sequence complexity

$$\text{worst-case sequence complexity} = \frac{\text{The MAXIMUM possible total cost of among all possible sequences of } m \text{ operations}}{m}$$
Methods for amortized analysis

• Aggregate method

• Accounting method

• Potential method (skipped, read Chapter 17 if interested)
Recap: Amortized analysis

• Real-life intuition: Monthly cost of living, a sequence of 12 operations
Aggregate method

What is the amortized cost per month (operation)?

Just **sum up** the costs of all months (operations) and **divide** by the number of months (operations).
Aggregate method: sum of all months’ spending is $126,000, divided by 12 months – the amortized cost is $1,050 per month.
Accounting method

Instead of calculating the average spending, we think about the cost from a different angle, i.e.,

How much money do I need to earn each month in order to keep living? That is, be able to pay for the spending every month and never become broke.
Accounting method: if I earn $1,000 per month from Jan to Nov and earn $1,600 in December, I will never become broke (assuming earnings are paid at the beginning of month).

So the amortized cost: $1,000 from Jan to Nov and $1,600 in Dec.
Aggregate vs Accounting

• Aggregate method is easy to do when the cost of each operation in the sequence is concretely defined.

• Accounting method is more interesting
  • It works even when the sequence of operation is not concretely defined
  • It can obtain more refined amortized cost than aggregate method (different operations can have different amortized cost)
Amortized Analysis on Dynamic Arrays
Problem description

• Think of an **array** initialized with a **fixed** number of slots, and supports **APPEND** and **DELETE** operations.

• When we **APPEND** too many elements, the array would be **full** and we need to **expand** the array (make the size larger).

• When we **DELETE** too many elements, we want to **shrink** to the array (make the size smaller).

• Requirement: the array must be using one **contiguous block** of memory all the time.

How do we do the **expanding** and **shrinking**?
One way to **expand**

- If the array is full when **APPEND** is called
  - Create a new array of **twice** the size
  - Copy all the elements from the old array to the new array
  - Append the element

```
3  7  2  1
|   |   |   |
```
Amortized analysis of `expand`

Now consider a dynamic array initialized with size 1 and a sequence of $m$ APPEND operations on it.

Analyze the amortized cost per operation

Assumption: only count array assignments, i.e., `append` an element and `copy` an element
Use the **aggregate** method

The cost sequence would be like:

\[
c_i = \begin{cases} 
  i + 1 & \text{if } i \text{ is power of 2} \\
  1 & \text{otherwise}
\end{cases}
\]

1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, ...

Cost sequence concretely defined, sum-and-divide can be done, but we want to do something more interesting...
Use the **accounting** method!

How much money do we need to **earn** at each operation, so that all future costs can be paid for?

How much money to earn for **each APPEND’ed element** ?

$1 \, ? \quad $2 \, ? \quad $3 \, ? \quad \$\log m \, ? \quad \$m \, ?$
Earn $1 for each appended element

This $1 (the “append-dollar”) is spent when appending the element.

But, when we need to copy this element to a new array (when expanding the array), we don’t any money to pay for it --

BROKE!
Earn $2 for each appended element

$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

What if the element is copied for a second time (when expanding the array for a second time)?

BROKE!
Earn $3 for each appended element

$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

$1 (the “recharge-dollar”) is used to recharge the old elements that have spent their “copy-dollars”.

NEVER BROKE!
$1 (the “recharge-dollar”) is used to recharge the old elements that have used their “copy-dollar”.

Old elements who have used their “copy-dollars”

New elements each of whom spares $1 for recharging one old element’s “copy-dollar”.

There will be enough new elements who will spare enough money for all the old elements, because the way we expand – TWICE the size.
So, in summary

If we earn $3 upon each APPEND it is enough money to pay for all costs in the sequence of APPEND operations.

In other words, for a sequence of $m$ APPEND operations, the amortized cost per operations is 3, which is in $O(1)$.

In a regular worst-case analysis (non-amortized), what is the worst-case runtime of an APPEND operation on an array with $m$ elements?
By performing the amortized analysis, we showed that “double the size when full” is a good strategy for expanding a dynamic array, since it’s amortized cost per operation is in $O(1)$.

In contrast, “increase size by 100 when full” would not be a good strategy. Why?
Takeaway

Amortized analysis provides us valuable insights into what is the proper strategy of expanding dynamic arrays.
Shrinking dynamic arrays

A bit trickier...
First that comes to mind...

When the array is $\frac{1}{2}$ full after DELETE, create a new array of half of the size, and copy all the elements.

Consider the following sequence of operations performed on a **full** array with $n$ element...

**APPEND, DELETE, APPEND, DELETE, APPEND, ...**

$\Theta(n)$ amortized cost per operation since every APPEND or DELETE causes allocation of new array.

NO GOOD!
The right way of shrinking

When the array is $\frac{1}{4}$ full after DELETE, create a new array of $\frac{1}{2}$ of the size, and copy all the elements.

Earning $3$ per APPEND and $3$ per DELETE would be enough for paying all the cost.

• 1 append/delete-dollar
• 1 copy-dollar
• 1 recharge-dollar
The array, after shrinking...

Elements who just spent their copy-dollars

Array is half-empty

Before the **next expansion**, we need to **fill** the empty half, which will spare enough money for copying the **green** part.

Before the **next shrinking**, we need to **empty** half of the **green** part, which will spare enough money for copying what’s left.
So, overall

In a dynamic array, if we expand and shrink the array as discussed (double on full, halve on ¼ full)...

For any sequence of APPEND or DELETE operations, earning $3 per operation is enough money to pay for all costs in the sequence,...

Therefore the amortized cost per operation of any sequence is upper-bounded by 3, i.e., O(1).
Next week

Graphs!

http://goo.gl/forms/S9yie3597B