CS 261: Data Structures

Week 1: Introduction

Lecture 1b: Amortized analysis and dynamic arrays

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Amortized analysis

Potential-function amortization: Overview

Applicable to data structures defined by

- How the data is organized
- Algorithms for performing its operations
- A "potential function" Φ from states to numbers, used only in the analysis (not by the algorithms)

Intuitively: Φ can be thought of as meaning:

- How far the data structure is from its ideal state
- How much extra time we've saved up by doing fast operations and can spend on a later slow operation without hurting the average too much

Potential-function amortization: Details

We are given (or choose for ourselves) a "potential function" Φ, from states to numbers, obeying two properties:

- $\Phi(X) \geq 0$ for all states X
- $\Phi(\iota) = 0$ for the initial state ι

We choose an arbitrary constant C in units of time (a single C for all of our analysis, large enough to make the analysis work)

Define the "amortized time" for each operation to be its actual time plus $C\big(\Phi(\text{after}) - \Phi(\text{before})\big)$.

Abbreviate this expression: actual time $+ C\Delta\Phi$

Then we do worst-case analysis on individual operations (not sequences), but using amortized time instead of actual time

Effect of Φ on amortized time

If Φ increases ($\Delta\Phi > 0$): amortized time $>$ actual time

We are charging more time to this operation than it actually takes, and saving the extra time to use later)

If Φ decreases ($\Delta\Phi < 0$): amortized time $<$ actual time

We are spending saved-up time to make this operation look faster

Example of amortization: Binary counter

Represent a binary number as an array C where $C[i]$ is the ith bit in the binary representation of the number (starting from $i = 0$ for the ones bit), so the number itself is $\sum_i 2^i C[i]$.

Initially all cells are zero.

Increment by flipping bits until finding the lowest-order 0 and flipping it to 1:

```
def increment():
    i = 0
    while C[i] == 1:
        C[i] = 0
        i += 1
    C[i] = 1
```

Potential function for the binary counter

Define $\Phi = \sum_i C[i]$, the number of nonzero cells in the counter

Does it have the correct properties for a potential function?

Initial state: all cells are zero, so $\Phi = 0$

Any state: number of nonzero cells is ≥ 0

Amortized analysis of the binary counter

Only one operation: increment

It changes some number k of 1s to 0s, and one more 0 to 1

- Actual time $= O(k+1)$
- $\Delta\Phi = 1 - k$
- Amortized time = actual + $C\Delta\Phi = O(k+1) + C(1-k)$

O(something) really means some unknown constant
(in units of time) multiplied by something

We can choose $C =$ the same constant

The k's cancel but the 1's don't, leaving amortized time $O(1)$

What does it mean?

Only the final potential counts in the total

For any sequence of operations $X_1, X_2, X_3, \ldots X_n$
with actual time = $T = t(X_1) + t(X_2) + t(X_3) + \cdots + t(X_n)$
and potential function values $\Phi_0, \Phi_1, \Phi_2, \Phi_3, \ldots, \Phi_n$

Total amortized time: $T_\Phi = t(X_1) + C(\Phi_1 - \Phi_0) + t(X_2) + C(\Phi_2 - \Phi_1) + \cdots$

Φ_0: defined to be zero!
Most other Φ_i: added once, subtracted once $\Rightarrow$ total = zero!

$$T_\Phi = T + C\Phi_n$$

This sort of cancellation of terms is called a "telescoping sum"

So what does it mean?

$$T_\Phi = T + C\Phi_n \geq T \qquad \text{(because } \Phi_n \geq 0\text{)}$$

That is, total amortized time is always
greater than or equal to total actual time

So we may use amortized time as a valid upper bound on the average time per operation over any sequence of operations.

If our analysis says an operation is fast, it is fast

If it says it is slow, maybe slow or maybe analysis is sloppy

Amortized analysis intuition

Φ is like a bank account for time

Most operations increase Φ by a small amount

- Because it's small, doesn't hurt amortized time per operation very much
- Saves up time in the account that you can use later

If you need to do infrequent slow operations, you can pay for them by taking money out of the account (decreasing Φ)

Amortized analysis is not magic

Different potential functions may lead to different analysis

but...

If one potential function gives small amortized time per operation, all sequences of operations will be fast in actual time,

or equivalently:

If you can find a sequence of operations whose actual time is slow, its amortized time will also be slow for all choices of Φ.

Example where amortization doesn't help

Let's modify the same binary counter to have both increment ($+1$) and decrement (-1) operations

To decrement: Almost the same as increment

Flip bits, starting from the rightmost bit and moving left, until finding the lowest-order 1 and flipping it to 0

Starting from 0, the sequence of operations

- Repeat $2^k - 1$ times: increment
- Repeat 2^k times: increment then decrement

takes actual time proportional to $n \log n$, where n is the total number of operations

No amount of amortization can give a smaller time bound

Amortized dynamic arrays

The problem

Arrays versus node-link based structures:

- Good: More compact layout
- Good: Random access to arbitrary positions
- Bad: Must know size at initialization time
- Bad: Wasted space when initial size is too big

Can we get the compact layout and random access without choosing an initial size and without wasting space?

Yes! This is what Java ArrayList and Python list both do.

Dynamic array API

Operations:

- Create new array of length 0
- Increase the length by 1
- Decrease the length by 1
- Get the item at position i
- Set the item at position i to value x

Goals for analysis:

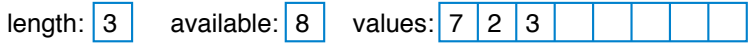
- $O(1)$ amortized time per operation
- Total space $= O(\text{max length})$

Dynamic array representation

Store:

- A.length: Current length of the array
  (how many cells we are actually using)
- A.available: Total length currently available
  (maximum length possible without resizing)
- A.values: block of memory cells of length A.available
  (indexed as $0, 2, \ldots,$ A.available $- 1$)

Example:



Dynamic array implementation

Easy operations:

- Create:
  Allocate values as a block of one memory cell
  Set length $= 0$ and available $= 1$
- Get i:
  Check that $0 \leq i <$ length
  Return values[i]
- Set i, x:
  Check that $0 \leq i <$ length
  Set values[i] to x
- Decrease length:
  Check that length > 0
  Decrease length by one

Dynamic array implementation

Implementing increase-length operation:

If length == available:

Get a new block B of ($2 \times$ available) cells
Copy all values from available to B
Set values to point to B
Set available to ($2 \times$ available)

Increase length by one

Dynamic array potential function

The only operation whose actual time can be slow is an increase-length that has to reallocate a new larger block of cells

Goals: Choose Φ so that before this operation it is large, and afterwards it is small, so we can pay for the slow operation

Other operations should make only small changes to Φ

Solution: $\Phi = |2 \times \text{length} - \text{available}|$

Before a reallocation step: $\Phi = \text{length}$

After a reallocation step: $\Phi = 0$

Dynamic array analysis, I

All operations other than an increase-length that reallocates the array:

- Actual time $= O(1)$
- $\Delta\Phi \leq 2$ (equals 1 for create, 2 for some decrease-length ops)
- Amortized time $= O(1) + C\Delta\Phi \leq O(1) + 2C = O(1)$
  (regardless of which constant value we choose for C)

Dynamic array analysis, II

Increase-length that reallocates the array:

- Actual time $= O(1 + \text{length})$
- $\Delta\Phi = 2 - \text{length}$ (see previous slide for before/after)
- Amortized
  time $= O(1 + \text{length}) + C\Delta\Phi = O(1 + \text{length}) + C \cdot (2 - \text{length}) = O(1)$
  (choosing C to be large enough to cancel the O-notation)

Conclusion: All operations take constant amortized time
Space = at most twice the maximum value of length ever seen

Improvements in space complexity, I

Can get space O(current length) rather than O(max length) by reallocating the block of memory when length/available becomes too small

- If we increase length by factors of two, "too small" needs to be a fraction strictly less than $1/2$, e,.g. $1/3$, to prevent sequences of operations that alternate between increase-length and decrease-length from causing many reallocations
- Reallocated block size should be chosen to make the potential function become close to zero again

Improvements in space complexity, II

Reduce space from $2 \times$ max length or $3 \times$ current length to $(1 + \varepsilon) \times$ current length for your favorite $\varepsilon > 0$ (e.g. $\varepsilon = 1/4$)

Main idea: when increasing or decreasing array size, aim for $(1 + \frac{1}{2}\epsilon) \times$ current length

Reallocate whenever array size (available) becomes more than $(1 + \epsilon)$ factor away from current length in either direction

Modified potential function $\Phi = |(1 + \frac{1}{2}\varepsilon)\text{length} - \text{available}|$

Amortized time = larger constant, proportional to $1/\varepsilon$

What do Java and Python actually do?

Java (Sun JDK6):

```
newCapacity = oldCapacity + (oldCapacity >> 1);
```

Grows by a factor of 3/2; never shrinks

Python (CPython; JPython is same as Java):

```
new_allocated = ((size_t)newsize + (newsize >> 3) + 6)
                & ~(size_t)3;
```

Grows by a factor of 9/8; never shrinks

Stacks, deques, and queues, revisited

Stacks using dynamic arrays

Each stack is associated with its own dynamic array

Push x:

increase length of array

Store x in array[length $-$ 1]

Pop:

top $=$ array[length $-$ 1]

decrease length of array

return top

etc.

Queues

Two operations to add and remove items, like push and pop for stacks

Add is called "enqueue", remove is called "dequeue"

Dequeue removes oldest not-yet-removed item (vs stack: newest)

Sergei Eisenstein, The Queue of 150 Persons, 1916, Russian State Archive of Literature and Art

Just like the line to order at a restaurant or coffee shop: you enter the back of the line ("enqueue") and the cashier takes the next customer from the front of the line ("dequeue")

Stacks + Queues = Deques

Deque: Maintain a contiguous sequence of items, allowing addition and removal at both ends of the sequence

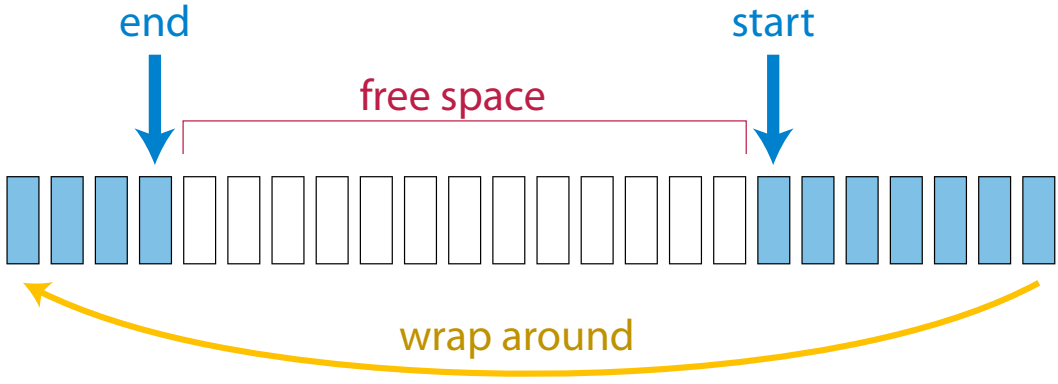
Special cases:

- Always add and remove from same end $\Rightarrow$ stack
- Add and remove from opposite ends $\Rightarrow$ queue

Pronounced "deck", stands for "double-ended queue"

Dequeues and queues: Layout and Operations

Store as contiguous block of cells in a dynamic array that "wraps around" from the end of the array back to the beginning



Most operations: local changes at one end of the block

When the array becomes full and two ends bump into each other: Double length and move old values to contiguous block of new array

Dequeues and queues: Analysis

For stacks, every stack operation is a single dynamic array operation, so we get $O(1)$ amortized time per operation immediately (no new analysis)

For dequeues and queues, doubling the array when full also involves moving data to different locations, not quite in the same way that a standard dynamic array would.

So the analysis needs to be redone, but the same potential function works in exactly the same way.

$O(1)$ amortized time per operation, again

Week 1: Summary

Summary

- Most algorithms use multiple data structures, both explicitly and implicitly
- We usually care about performance over sequences of operations, rather than for isolated operations
- When operations are usually fast but occasionally slow, amortized analysis allows us to prove that the average time is fast, while still only analyzing a single operation at a time
- Dynamic arrays that double in size when full have constant time for all operations, even though doubling steps are slow
- A single dynamic array can be used to implement a dynamic stack, queue, or deque

Image credits

- CC-BY-SA image “Hand Putting Deposit Into Piggy Bank”, https://commons.wikimedia.org/wiki/File:Hand_Putting_Deposit_Into_Piggy_Bank_(5737295175).jpg, by Ken Teegardin
- CC-BY-SA images “Luneta, Instituto Histórico e Geográfico de Vila Velha”, https://commons.wikimedia.org/wiki/File:Luneta,_Instituto_HistÃşrico_e_GeogrÃąfico_de_Vila_Velha_(0026_A_1).png and https://commons.wikimedia.org/wiki/File:Luneta,_Instituto_HistÃşrico_e_GeogrÃąfico_de_Vila_Velha_(0026_A).png, from the Midiateca Capixaba collection