CS 261: Graduate Data Structures

Week 1: Introduction, amortization, arrays, stacks, queues, and deques

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Course overview

Who is running the course?

Instructor:
David Eppstein
eppstein@uci.edu

Teaching assistant:
Hadi Khodabande
khodabah@uci.ed

Online resources

Course web site:
`https://www.ics.uci.edu/~eppstein/261/`

Online course discussions: Piazza

Turn in homework and return graded homeworks: Gradescope

Confidential questions about your performance: Email us!

Course material

Online lectures

Provided as video (voice-over)
and slides through course web site

One set of slides per week,
broken into multiple lecture videos
(not necessarily of uniform length)

Weekly homeworks

Do them on your own

(Will include a final set of homeworks due in finals week!)

Expectations

What do I expect you to know already?

- Undergraduate-level data structures: how to implement standard and basic structures including stacks, queues, lists, heaps, and binary search trees. Some basic analysis of these structures.
- Undergraduate-level algorithms: how to describe algorithms in pseudocode, what O-notation means, divide-and-conquer algorithms and their analysis using recurrence equations, and some other standard algorithms such as depth-first search and Dijkstra's algorithm

What is a data structure?

Example

Let's look at a standard algorithm: depth-first search, to find the subset of vertices in a graph G that can be reached from a given starting vertex s. (In Python; later we'll use pseudocode.)

```
def DFS(s,G):
    visited = set()      # already-processed vertices

    def recurse(v):      # call for each vertex we find
        visited.add(v)   # remember we've found it
        for w in G[v]:   # look for more in neighbors
            if w not in visited:
                recurse(w)

    recurse(s)
    return visited
```

What data structures does this algorithm use?

Even this very simple algorithm uses multiple structures:

- A set of visited vertices
- The input graph, organized as a dictionary whose keys are vertices and whose values are collections of neighbors
- The collections of neighbors for each vertex
- (Implicitly) a stack, keeping track of the sequence of subroutine calls and their local variables

So what is a data structure?

The information a program or algorithm needs
to access and update as it runs

The layout of that information
into words of memory on a computer

A catalog of methods by which
the information is accessed and modified

Algorithms for performing those methods efficiently

Analysis of how much memory the structure uses
and how much time its methods use

Levels of abstraction

API

What operations does the structure provide?

Implementation

How do we organize the data, and how do we perform its operations?

Analysis

How much space and time per operation does it use?

Levels of abstraction: API

What operations does the structure provide?

Example: Stack

- push(x): add item x to the stack
- pop(): remove and return most-recently-added item
- top(): return the item that pop would return, but without removing it
- isempty(): test whether stack has any remaining items
- new(): create a new empty stack object

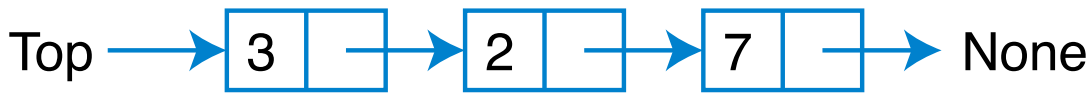
Levels of abstraction: Implementation

How do we organize the data, and how do we perform its operations?

Stacks using singly-linked lists:

- Stack is a collection of nodes with value and pointer to next node; stack itself points to top node
- Empty stack = null pointer
- To push, make a new node and point to it
- To pop, find top value and change stack pointer to next item

For example after push(7), push(2), and push(3):



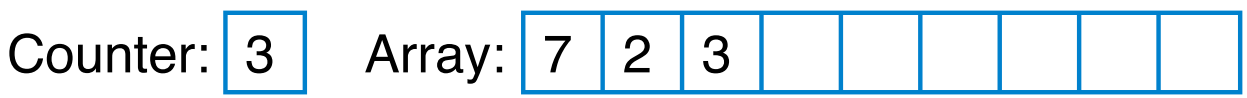
Levels of abstraction: Implementation

How do we organize the data, and how do we perform its operations?

Stacks using arrays:

- Stack is an array of cells holding values and a length counter
- Empty stack = array with counter value zero
- To push, increase counter and add value to array
- To pop, find top value and decrease counter

For example after push(7), push(2), and push(3):



Levels of abstraction: Analysis

How much space and time per operation does it use?

Stack example:

- Both implementations use $O(1)$ time per operation
- List implementation uses $O(1)$ space per element
- Array implementation uses space = max # of elements
- Array is probably faster in practice (no overhead for node allocation/release; more predictable memory access pattern) but showing this involves experimentation, not theory

Styles of analysis

Worst-case analysis

- Consider each operation X (and its implementation) separately from each other
- Find a sequence P of previous operations that causes X to be as slow as possible
- Equivalently, find a possible state S of the data structure that causes X to be as slow as possible
- Worst case time for operation X
  $=$ time for X after sequence P
  $=$ time for X on state S

Amortized analysis

Intuitions:

- We are usually going to be doing these operations multiple times
- Not all calls to an operation will take the worst-case time
- We usually care only about the total time of the overall algorithm, not how much time each individual operation took
- So we should average the time per operation over a whole sequence of operations, while still looking for the worst-case sequence, the one that makes the average as large as possible

Example of averaging: Binary counter

When you add one to a binary number:

- The least significant bit that has value 0 changes to 1
- All lower-order bits change from 1 to 0

Example:

- $167 = 10100111_2$
- $168 = 10101000_2$

Example of averaging: 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
```

Averaged analysis of binary counter

Total time over a sequence of n increments $= O(\#$ bits flipped$)$

Cell $C[i]$ gets flipped $\lfloor n/2^i \rfloor$ times.

So total $\#$ flips $\leq \sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq 2n - 1$ (geometric series).

Average time per increment is $O\left(\frac{2n-1}{n}\right) = O(1)$.

In comparison, worst case is $\Theta(\log n)$
(e.g. when incrementing to the largest power of two $\leq n$)

Naive averaging is problematic!

- What sequences of operations are the slow ones?

  (It's easier to analyze single operations than to find worst-case sequences.)

- If you have more than one type of operation in a sequence, which are the ones that cause it to be slow (on average)?

Amortization is a method for averaging that lets us handle both of these problems.

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

Given a function Φ from states to numbers, required to satisfy:

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

Choose an arbitrary constant C in units of time (the same constant for all operations, large enough to make the analysis work)

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

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

Effect of Φ on amortized time

If Φ increases: 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: amortized time $<$ actual time

We are spending some of our saved-up time

Amortized analysis of the binary counter

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

When we increment from $n-1$ to n, we change some of 1's to 0's, and a single 0 to a 1; let k_n count the number of 1's changed to 0.

The actual time to increment is $O(k_n + 1)$.

$\Delta\Phi = 1 - k_n$ (one new 1-bit, k_n previous 1-bits turned to 0)

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

We can choose C to be at least as large as the constant (in units of time) in the O-notation. For this choice, the k's cancel but the 1's don't, leaving amortized time $O(1)$

But what does it mean?

Consider 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 $\Phi_0, \Phi_1, \Phi_2, \Phi_3, \ldots, \Phi_n$
(starting before we have done any operations).

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

But most of the Φ's cancel, leaving only
$$T_\Phi = T + C(\Phi_n - \Phi_0) = T + C(\Phi_n - 0) \geq T$$

That is, total amortized time is always greater than 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.

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 a small number of slower 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.

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:

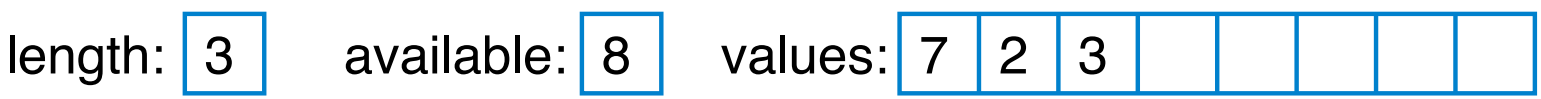
- $O(1)$ amortized time per operation
- Total space $= O(\text{max length})$

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 $0, 2, \ldots, \text{A.available} - 1$)

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 $(2 \times \text{available})$ cells
  - Copy all values from available to B
  - Set values to point to B
  - Set available to $(2 \times \text{available})$
- 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 $= O(1)$
- $\Delta\Phi \leq 1$ (equals 1 for create, some decrease-length ops)
- Amortized time $= O(1) + C\Delta\Phi = O(1) + C = O(1)$ (regardless of which constant value we choose for C)

Dynamic array analysis, II

Increase-length that reallocates the array:

- Actual time $= O(1 + \text{length})$
- $\Delta\Phi = -\text{length}$ (see previous slide for before/after)
- Amortized time $= O(1 + \text{length}) + C\Delta\Phi =$
  $O(1 + \text{length}) - C \cdot \text{length} = O(1)$
  (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 $1/\varepsilon$

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.

What are dequeues and queues?

Queue: 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)

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: make local changes at one of the two ends of the contiguous block

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