CS 261: Data Structures

Week 1: Introduction

Lecture 1c: Arrays, stacks, and queues

David Eppstein
University of California, Irvine

Spring Quarter, 2022

This work is licensed under a Creative Commons Attribution 4.0 International License
Amortized dynamic arrays
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, ..., A.available − 1)

Example:

length: 3  available: 8  values: 7 2 3
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 $0 \leq i < \text{length}$
  Return $\text{values}[i]$

- **Set** $i, x$:
  Check that $0 \leq i < \text{length}$
  Set $\text{values}[i]$ to $x$

- **Decrease length**:
  Check that $\text{length} > 0$
  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
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 $\Phi$ 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 $\Phi$.

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 $C$)
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
Can get space $O(\text{current length})$ rather than $O(\text{max length})$ 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 $2 \times \text{max length}$ or $3 \times \text{current length}$ to $(1 + \varepsilon) \times \text{current length}$ for your favorite $\varepsilon > 0$ (e.g. $\varepsilon = 1/4$)

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

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

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

Amortized time $= \text{larger constant, proportional to } 1/\varepsilon$
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 ⇒ stack
- Add and remove from opposite ends ⇒ 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
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.