

Indian Institute of Technology Madras
ZANZIBAR CAMPUS

Z5007 · PROGRAMMING AND DATA STRUCTURES

Dynamic Arrays and Amortized Analysis

Lecture 6 · Thursday 15 October 2026 · Canning, Broder & Lafore Ch 2; CLRS Ch 16 (Amortized Analysis)

Dr. Innocent Nyalala | MTech Semester 1 | IIT Madras Zanzibar

Today's Plan

IIT Madras Zanzibar

1. (0–6 min) The problem: an array has a fixed size, but data keeps arriving
2. (6–15 min) The dynamic array: spare room, and a bigger block when full
3. (15–19 min) Why we need a proof: two answers that disagree
4. (19–26 min) Proof 1: add up every step (aggregate method)
5. (26–33 min) Proof 2: save in advance (accounting method)
6. (33–39 min) Growing and shrinking well: why ×2 and not +1
7. (39–50 min) Code, costs of list operations, exercise, summary

Learning Outcomes

IIT Madras Zanzibar

By the end of this session, you will be able to:

Explain	how a dynamic array grows: size, capacity, resize and copy
Count	the exact cost of any sequence of appends
Define	amortized cost, and say how it differs from worst case per operation
Prove	that append is O(1) amortized, by adding up and by saving in advance
Justify	why arrays grow by a factor, and why they shrink at one quarter

Key Terms for Today

IIT Madras Zanzibar

Term	Meaning	Example
Array	items stored in boxes side by side in memory	[12, 7, 33, 5]
Index	the position of a box, counted from 0	a[2] is 33
Size	how many items are actually stored	4
Capacity	how many boxes are reserved	8
Dynamic array	an array that grows by moving to a bigger block	Python list
Resize	reserve a new block and copy every item into it	capacity 4 → 8
O(1) / O(n)	cost does not grow with n / cost grows in proportion to n	a[i] / copy n items
Amortized cost	total cost of n operations ÷ n, guaranteed for every sequence	append: < 3 steps

IIT Madras Zanzibar

→ **The Problem: Arrays Have a Fixed Size**

The Dynamic Array

Why We Need a Proof

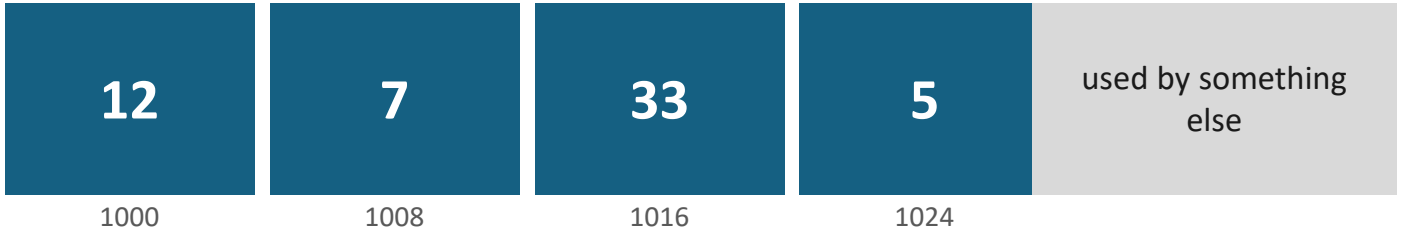
Proof 1: Add Up Everything

Proof 2: Save in Advance

Growing and Shrinking Well

Practice

Recap: An Array Is a Row of Boxes in Memory



Two facts to remember

1. Address of box i = start address + i × box size. Example: box 2 is at 1000 + 2 × 8 = 1016.
2. So reading or writing a[i] takes ONE step, no matter how long the array is: O(1).
3. The number of boxes is fixed when the block is created. The memory after it may already be in use, so the block cannot grow in place.

Definition: O(1), constant time: the number of steps does not depend on how many items there are. Reading a[i] is O(1).

The Problem: We Rarely Know How Much Data Will Come

IIT Madras Zanzibar

1 **Mobile money**

transactions are logged all day; nobody knows tonight's count this morning

2 **Chat app**

messages arrive one by one, forever

3 **Sensors**

a weather station adds a reading every minute

Two bad choices, and what we want

1. Reserve too little: the array fills up and the program fails.
2. Reserve too much: memory is wasted, and we may still be wrong.
3. What we want: an array that GROWS when needed and still reads a[i] in one step. This is a dynamic array (Python list, Java ArrayList, C++ vector).

IIT Madras Zanzibar

The Problem: Arrays Have a Fixed Size

→ **The Dynamic Array**

Why We Need a Proof

Proof 1: Add Up Everything

Proof 2: Save in Advance

Growing and Shrinking Well

Practice

Idea 1: Keep Spare Room (Size and Capacity)



Two numbers describe the array

1. size = number of items actually stored (here 5).
2. capacity = number of boxes reserved (here 8).
3. Append when size < capacity: write into box[size], then size = size + 1. Cost: 1 step.

```
# append with room
block[size] = x
size = size + 1

# cost: 1 step
```

Definition: **Dynamic array:** an array that keeps size ≤ capacity, and moves to a bigger block when size reaches capacity.

Idea 2: When Full, Move to a Block Twice as Big

IIT Madras Zanzibar

Before: size 4, capacity 4 (full)



After: size 5, capacity 8



Append when full

1. Reserve a new block, 2 × capacity.
2. Copy each old item: 4 copies.
3. Write the new item: 1 step.

Cost of this append: 4 copies + 1 write = 5 steps. Expensive, but it buys 3 free boxes, so the next 3 appends cost 1 step each.

Definition: **Resize:** reserve a new block (here twice as big) and copy every item into it. With n items it costs n copies: O(n).

Worked Example: 9 Appends, Every Step Counted

IIT Madras Zanzibar

Append #	Size before	Capacity before	Full?	Copies	Total steps
1	0	1	no	0	1
2	1	1	yes → capacity 2	1	2
3	2	2	yes → capacity 4	2	3
4	3	4	no	0	1
5	4	4	yes → capacity 8	4	5
6, 7, 8	5, 6, 7	8	no	0	1 each
9	8	8	yes → capacity 16	8	9
All 9				15	24

Observation: expensive appends happen at 2, 3, 5, 9, 17, …: further and further apart. 24 steps / 9 appends < 3.

IIT Madras Zanzibar

The Problem: Arrays Have a Fixed Size

The Dynamic Array

→ **Why We Need a Proof**

Proof 1: Add Up Everything

Proof 2: Save in Advance

Growing and Shrinking Well

Practice

Two Honest Answers That Disagree

IIT Madras Zanzibar

1

Engineer A: 'it's slow'

one append can cost O(n), so n appends cost n × O(n) = $O(n^2)$

2

Engineer B: 'it's fast'

I measured 1,000,000 appends: a fraction of a second

Who is right, and why we need a proof

1. A is too pessimistic: she assumes EVERY append is the expensive one. In reality the expensive ones are rare.
2. B is right only for what she tested: one machine, a few sizes. It is evidence, not a guarantee.
3. A proof gives the true total for EVERY n and EVERY sequence of appends. That is the guarantee engineers build on: a log that must hold a billion records cannot rely on 'it seemed fast'.

The Definition: Amortized Cost

IIT Madras Zanzibar

amortized cost per operation = (worst-case total cost of n operations) ÷ n

1. **Means 'spread out'**

   school fees: a big payment spread over 12 months

2. **A guarantee, not luck**

   holds for EVERY sequence, not just random ones

3. **Not per operation**

   a single append can still cost O(n)

Our goal: show that n appends cost less than 3n steps in total. Then the amortized cost is less than 3: O(1).

? **Think:** your phone app freezes for a moment when its list resizes. Does 'O(1) amortized' promise that never happens?

Dynamic Arrays and Amortized Analysis

The Problem: Arrays Have a Fixed Size

The Dynamic Array

Why We Need a Proof

→ **Proof 1: Add Up Everything**

Proof 2: Save in Advance

Growing and Shrinking Well

Practice

Proof 1 (Aggregate): Step by Step

IIT Madras Zanzibar

1 Count writes

each append writes one item:
n writes

2 Count copies

resizes copy 1, 2, 4, 8, … up to at most n items

3 Add the copies

1 + 2 + 4 + … + n < 2n
(next slide)

4 Divide by n

total < n + 2n = 3n
per append < 3: O(1)

Check it against real numbers

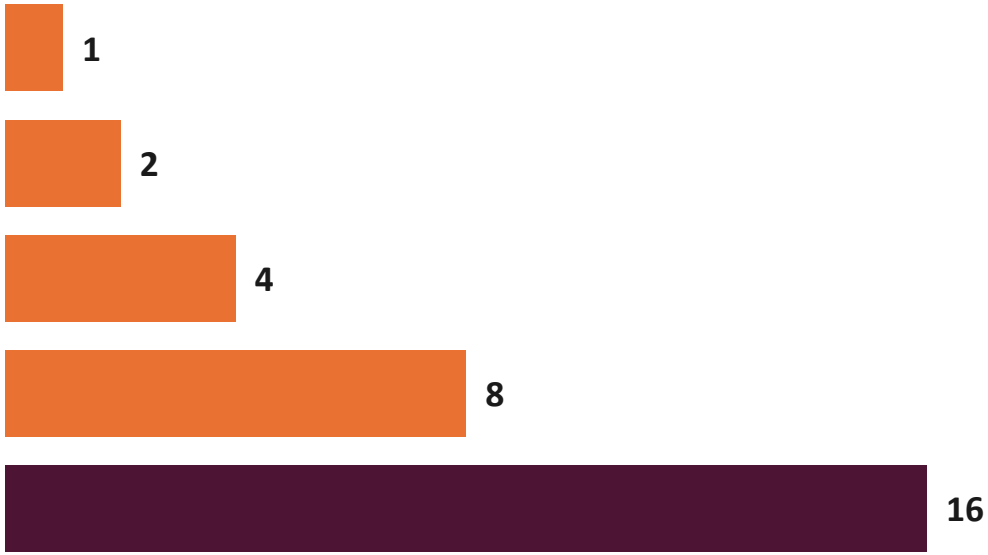
1. Check with our table, n = 9: writes = 9, copies = 1 + 2 + 4 + 8 = 15 (less than 2 × 9 = 18), total 24 (less than 3 × 9 = 27).
2. Check n = 16: writes 16, copies 1 + 2 + 4 + 8 = 15, total 31, under 48.

Definition: **Aggregate method:** find the total cost T(n) of ANY n operations, then amortized cost = T(n) ÷ n.

Why 1 + 2 + 4 + … + n Is Less Than 2n

Terms	Sum	Next power of 2
1	1	2
1 + 2	3	4
1 + 2 + 4	7	8
1 + 2 + 4 + 8	15	16
1 + 2 + 4 + 8 + 16	31	32

Rule: $1 + 2 + 4 + \ldots + n = 2n - 1 < 2n$ (when n is a power of 2)



1. Each term is larger than all earlier terms added together.
2. So the sum up to n is less than n + n = 2n.

IIT Madras Zanzibar

The Problem: Arrays Have a Fixed Size

The Dynamic Array

Why We Need a Proof

Proof 1: Add Up Everything

→ **Proof 2: Save in Advance**

Growing and Shrinking Well

Practice

Proof 2 (Accounting): Charge a Fixed Fee and Save the Change

IIT Madras Zanzibar

The rule

1. Charge every append exactly 3 shillings.
2. 1 shilling pays for writing the new item.
3. 2 shillings are saved: put them on the item just written.
4. At a resize, each copy costs 1 shilling, paid from the savings.
5. If the savings never go below zero, the real total is at most the total charged: 3n.

1 **Why 3?**

1 for the write, 2 saved for future copying

2 **Why a second proof?**

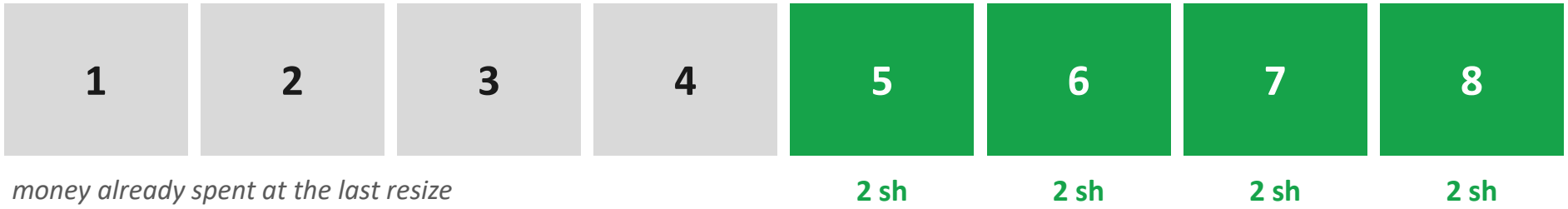
it checks one operation at a time, so it extends to append AND pop

3 **Everyday version**

a savings plan: pay a little extra each month, and the big bill is already covered

Definition: **Accounting method:** charge each operation a fixed fee; show the saved surplus always covers the expensive steps. Then real total ≤ total fees.

Why the Savings Never Run Out: One Resize, Up Close



The count at a resize

1. Capacity 8 is full. Items 5–8 arrived after the last resize (4 → 8); each carries 2 saved shillings.
2. Savings: 4 items × 2 = 8 shillings. The resize must copy 8 items at 1 shilling each = 8 shillings. Exactly enough.
3. In general: at capacity k, the k/2 new items carry k shillings, enough for k copies. The savings never go negative, so the total real cost ≤ 3n.

Check: The Savings Account for 9 Appends

IIT Madras Zanzibar

Append #	Fee	Real cost	Savings after
1	3	1	2
2	3	2 (copy 1)	3
3	3	3 (copy 2)	3
4	3	1	5
5	3	5 (copy 4)	3
6	3	1	5
7	3	1	7
8	3	1	9
9	3	9 (copy 8)	3

Read the last column

1. Savings rise between resizes and drop at each resize.
2. They never go below zero.
3. So the real total (24) is at most the fees (27): under 3 per append.
4. Think: would a fee of 2 work? Try append 5.

Definition: **Invariant:** a statement that stays true after every step. Here: savings ≥ 0.

The Problem: Arrays Have a Fixed Size

The Dynamic Array

Why We Need a Proof

Proof 1: Add Up Everything

Proof 2: Save in Advance

→ **Growing and Shrinking Well**

Practice

Grow by a Factor, Never by a Fixed Amount

IIT Madras Zanzibar

n appends	Grow ×2 when full: copies	Grow +1 when full: copies	Grow +100 when full: copies
1,000	1,023	499,500	about 4,500
100,000	about 131,000	about 5 billion	about 50 million

1

+1 or +100

copies = $c + 2c + 3c + \ldots \approx n^2/(2c)$: $O(n^2)$ total

2

×2 or ×1.5

copies form a doubling series: O(n) total, O(1) amortized

Definition: **Growth factor:** the number the capacity is multiplied by at each resize (2 for doubling). Use a factor > 1, never a fixed amount.

Shrinking: Why at One Quarter, Not One Half

IIT Madras Zanzibar

Shrink at 1/2: thrashing

1. Capacity 8, size 8. Append → full → grow to 16, copy 8.
2. Pop → 8 of 16 = half → shrink to 8, copy 8.
3. Append → full again → grow, copy 8.
4. Pop → shrink, copy 8 … every operation copies: O(n) each.

Shrink at 1/4: safe

1. Grow when full (to 2 × capacity).
2. Shrink when only 1/4 full (to capacity / 2).
3. After any resize the array is half full, far from both limits.
4. Many cheap operations must happen before the next resize: O(1) amortized again.

Definition: **Thrashing:** repeated grow, shrink, grow, shrink, where almost every operation triggers an expensive resize.

Dynamic Arrays and Amortized Analysis

The Problem: Arrays Have a Fixed Size

The Dynamic Array

Why We Need a Proof

Proof 1: Add Up Everything

Proof 2: Save in Advance

Growing and Shrinking Well

→ **Practice**

Code, Part 1: Setting Up and append()

```
 1  class DynamicArray:
 2      def __init__(self):
 3          self.size, self.cap = 0, 1
 4          self.copies = 0
 5          self.block = [None] * self.cap
 6
 7      def append(self, x):
 8          if self.size == self.cap:
 9              self._resize(2 * self.cap)
10          self.block[self.size] = x
11          self.size += 1
```

Line	What it does
1	define a new type called DynamicArray
2–3	start empty: size 0, capacity 1
4	copies counter, to check our proofs
5	reserve a block of cap empty boxes
7	append(x) adds x at the end
8	full? size has reached capacity
9	if full: move to a block twice as big
10	write x into the first free box
11	one more item stored

Code, Part 2: _resize(), the Expensive Step

```
1  def _resize(self, new_cap):
2      new = [None] * new_cap
3      for i in range(self.size):
4          new[i] = self.block[i]
5          self.copies += 1
6      self.block, self.cap = new, new_cap
```

Line	What it does
1	_ means internal: called only by append and pop
2	reserve a new block of new_cap boxes
3	visit every stored item, i = 0 … size − 1
4	copy item i into the new block
5	count the copy (to check our proofs)
6	use the new block from now on
Cost	the loop runs size times: O(n)

Code, Part 3: pop() and Shrinking at One Quarter

IIT Madras Zanzibar

```
1  def pop(self):
2      if self.size == 0: raise IndexError
3      self.size -= 1
4      x = self.block[self.size]
5      if 0 < self.size <= self.cap // 4:
6          self._resize(self.cap // 2)
7      return x
```

Line	What it does
1	pop() removes the LAST item
2	nothing to remove: report an error
3	one fewer item stored
4	read the removed item
5	only a quarter full? (// = whole-number division)
6	shrink to half the capacity
7	return the removed item

What Each Python List Operation Really Costs

IIT Madras Zanzibar

Operation	Cost	Reason
a[i], a[i] = x	O(1)	address = start + i × size
a.append(x)	O(1) amortized	resizes are rare and paid for (today's proofs)
a.pop()	O(1) amortized	remove last; no shifting
a.insert(0, x)	O(n)	every item shifts right
a.pop(0)	O(n)	every item shifts left
x in a	O(n)	may check every item

? **Think:** a program calls a.pop(0) 100,000 times on a list of 100,000 items. Roughly how many steps in total?

Common Mistakes

IIT Madras Zanzibar

1 **'Amortized = average over random inputs'**

No: it is a guarantee for EVERY sequence of operations.

2 **'Amortized O(1) = every append is fast'**

No: one append can still copy everything.

3 **Multiplying worst case by n**

$n \times O(n) = O(n^2)$ is true but far too pessimistic.

4 **Growing by +1 or +100**

gives $O(n^2)$ total; grow by a factor.

In-Class Exercise (6 minutes, in pairs)

IIT Madras Zanzibar

IN-CLASS TASK: show your working: count steps, or keep a savings table

1. Start with capacity 1 and double when full. For 8 appends: total copies? total steps? steps per append?

2. Same array, but charge only 2 shillings per append (1 for the write, 1 saved). Keep the savings table for 5 appends. What goes wrong?

3. Capacity 4, size 4, with the shrink-at-HALF rule. Give 4 operations that cause 4 resizes, and count the copies.

Solution

IIT Madras Zanzibar

1

1: 7 copies, 15 steps, under 2 per append

resizes at appends 2, 3, 5 copy 1 + 2 + 4 = 7; plus 8 writes = 15; 15 / 8 ≈ 1.9.

2

2: the savings go negative at append 5

savings after appends 1–4: 1, 1, 0, 1. Append 5 costs 5: 1 + 2 − 5 = −2. A fee of 2 is not enough: each new item must pay for copying itself AND one older item.

3

3: append, pop, append, pop: 16 copies

append → grow to 8, copy 4; pop → 4 of 8 = half → shrink, copy 4; repeat. Every operation copies 4.

Quiz 1: expect a counting question, a short savings table, and 'amortized' explained in your own words.

Summary and Exam Checkpoints

IIT Madras Zanzibar

Size and capacity

spare room makes most appends 1 step

Double when full

copy everything, rarely; copies < 2n in total

Amortized = spread out

worst-case total ÷ n; a guarantee, not luck

Two proofs, one answer

add up (copies < 2n); save in advance (fee 3)

Next

Today 2:00 pm: Lab 1: build the array and the dynamic array, and test today's claims

Fri 16 Oct: Simple sorting: bubble sort and selection sort

Read: Canning, Broder & Lafore, Chapter 2

References and Further Reading

[1] Canning, J., Broder, A. & Lafore, R. (2022). Data Structures & Algorithms in Python. Addison-Wesley. Chapter 2.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). Introduction to Algorithms, 4th ed. MIT Press. Chapter 16 (Amortized Analysis): Sections 16.1–16.2 and 16.4 (dynamic tables).

[3] Tarjan, R. E. (1985). Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods, 6(2), 306–318.