

Indian Institute of Technology Madras
ZANZIBAR CAMPUS

Z5007 · PROGRAMMING AND DATA STRUCTURES

Implementing a Stack: Arrays and Linked Lists

Lecture 10 · Thursday 22 October 2026 · Canning, Broder & Lafore, Ch 4 (Stacks) and Ch 5 (Linked Lists)

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

Today's Plan

IIT Madras Zanzibar

1. (0–4 min) The Stack ADT: five operations, two ways to store them
2. (4–15 min) Part 1: the array stack, push and pop step by step
3. (15–20 min) Resizing: why push stays O(1) when the array is full
4. (20–32 min) Part 2: the linked stack, nodes, head and tail
5. (32–35 min) Array vs linked list: which one, and when
6. (35–41 min) Live demo: one test, two implementations
7. (41–50 min) Exercise, solution and summary

Learning Outcomes

IIT Madras Zanzibar

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

Implement	an array stack and a linked-list stack in Python, from memory
Trace	the array, _top and head after any sequence of push and pop
Explain	why doubling the array keeps push amortized O(1)
Choose	between the two designs and justify the trade-off

The Stack ADT: Five Operations, All O(1)

IIT Madras Zanzibar

The interface: what every stack promises

`push(x)`	put x on the top	O(1)
`pop()`	remove and return the top item	O(1)
`peek()`	return the top item without removing it	O(1)
`is_empty()`	True if there is nothing on the stack	O(1)
`size()`	number of items currently stored	O(1)

Where you already use one

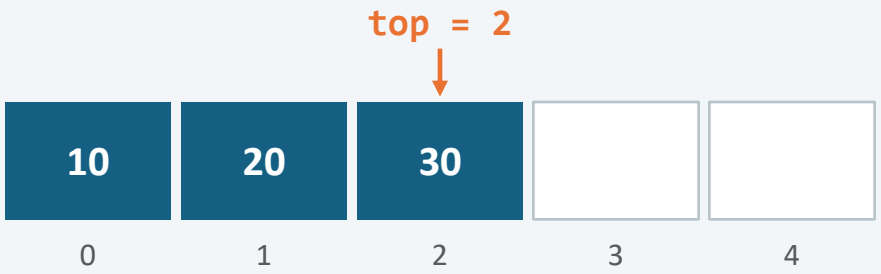
1. **Undo (Ctrl+Z)**
   each edit is pushed; undo pops the latest
2. **Browser Back**
   each page visited is pushed; Back pops it
3. **Function calls**
   Python pushes a frame per call; return pops it

? **Think:** push and pop both happen at the top. If you store the stack in a Python list, which end should be the top, and why?

Two Ways to Store a Stack: One Block or a Chain of Nodes

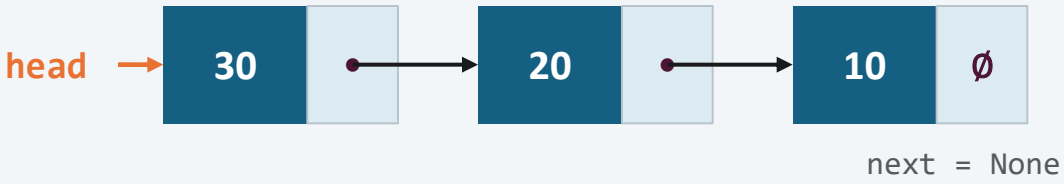
IIT Madras Zanzibar

A Array (contiguous block)



- Items sit **side by side** in one block of memory
- top is just an **integer index**
- Fast and cache-friendly; when full it must **grow by copying**

B Linked list (chain of nodes)



- Each item lives in its **own node object**
- top is a **reference to the head** node
- Grows one node at a time; **extra memory** for every next pointer

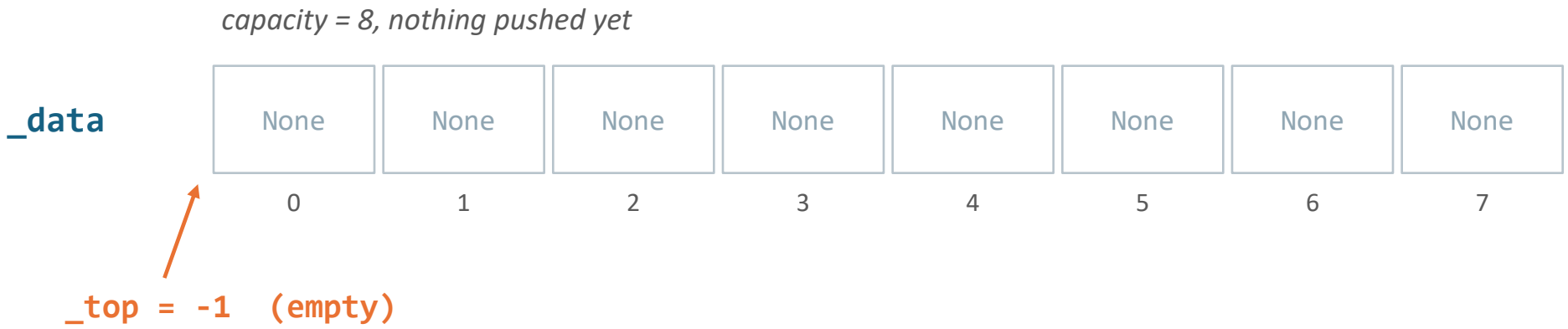
? **Think:** which of these can run out of room while memory is still free? What would you do about it?

Implementing a Stack: Where We Are

IIT Madras Zanzibar

1. The Stack ADT and two ways to store it
2. **Part 1: Stack on an array**
3. Part 2: Stack on a linked list
4. Array vs linked list
5. Live demo, exercise and summary

An Array Stack Is a Fixed Array Plus One Integer



State: just two fields

```
self._data = [None] * capacity
self._top  = -1   # index of the top item
```

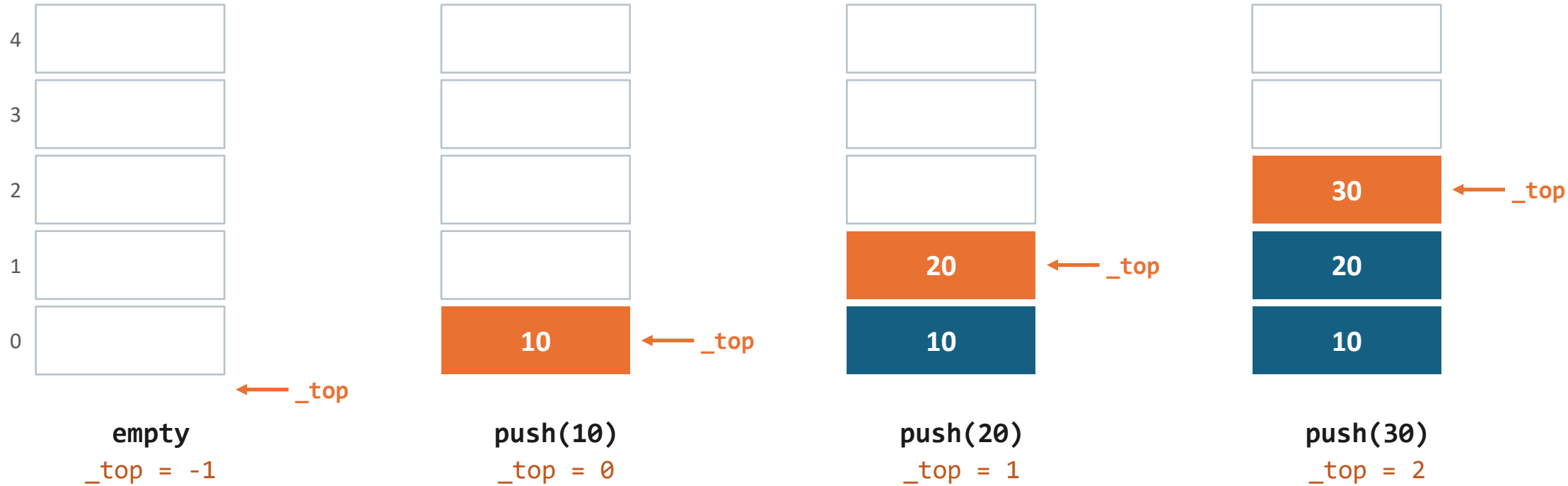
Everything else follows from _top

top item	`_data[_top]`
empty?	`_top == -1`
full?	`_top == capacity - 1`
size	`_top + 1`

? **Think:** why does _top start at -1 and not at 0? What would go wrong with 0?

push(x): Move _top Up One Slot, Then Write

Capacity 5, drawn upright: slot 0 is the bottom. Orange = the item just pushed. Click to push.



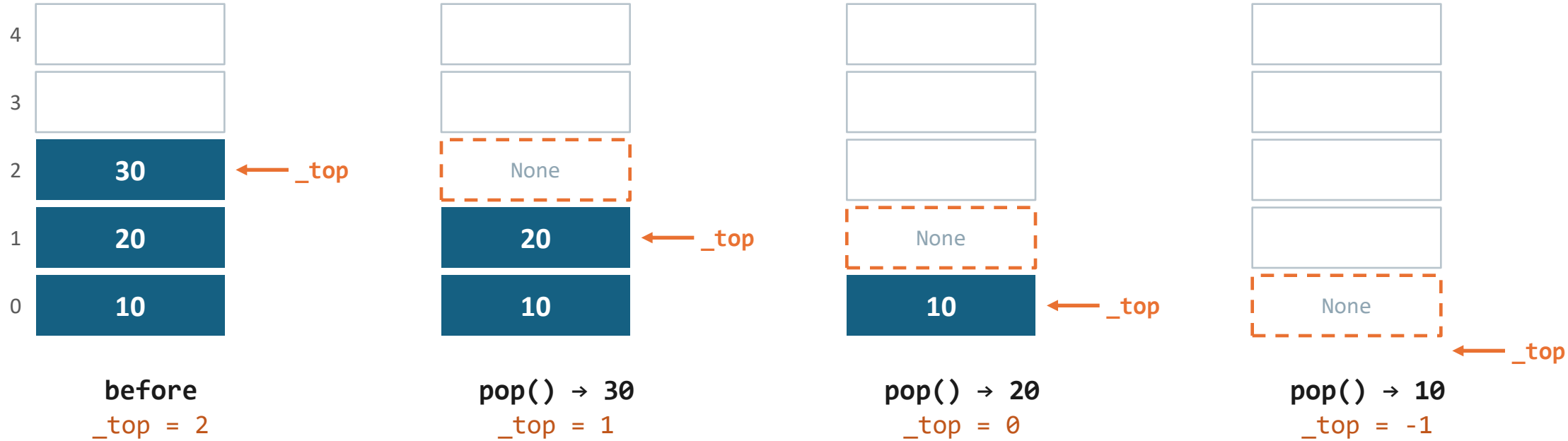

```
def push(self, x):
    if self._top == len(self._data) - 1:
        raise OverflowError('stack is full')
    self._top += 1                  # 1. move up
    self._data[self._top] = x       # 2. write
```

Two O(1) steps. Check for room, move _top up, write. Nothing else in the array moves.

pop(): Read the Top, Clear It, Move _top Down

IIT Madras Zanzibar

Start from the stack we just built. Dashed orange = the slot just emptied. Click to pop.




```
def pop(self):
    if self._top == -1:
        raise IndexError('pop from empty stack')
    x = self._data[self._top]        # 1. read
    self._data[self._top] = None     # 2. clear
    self._top -= 1                   # 3. move down
    return x
```

Underflow: pop() on an empty stack raises `IndexError` instead of returning garbage.

`peek()` is step 1 only: read `_data[_top]`, change nothing.

The Complete ArrayStack Class

IIT Madras Zanzibar

```
class ArrayStack:
    """LIFO stack on a fixed-size array."""

    def __init__(self, capacity=8):
        self._data = [None] * capacity
        self._top = -1

    def is_empty(self):
        return self._top == -1

    def __len__(self):
        return self._top + 1

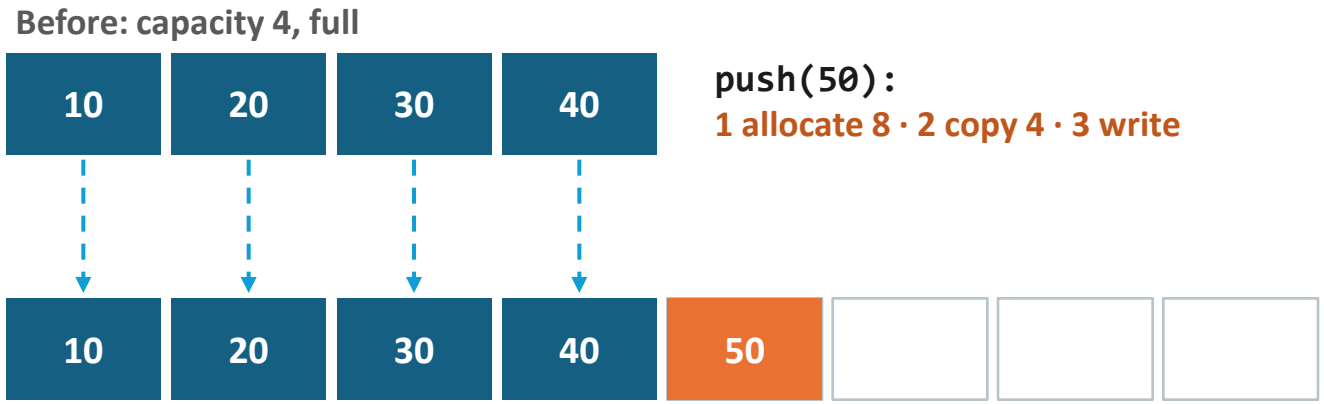
    def peek(self):
        if self.is_empty():
            raise IndexError('peek at empty stack')
        return self._data[self._top]
```

```
    def push(self, x):
        if self._top == len(self._data) - 1:
            raise OverflowError('stack is full')
        self._top += 1
        self._data[self._top] = x

    def pop(self):
        if self.is_empty():
            raise IndexError('pop from empty stack')
        x = self._data[self._top]
        self._data[self._top] = None
        self._top -= 1
        return x
```

You have used this class all along. A Python list is an array stack with resizing built in: **`append(x)`** is push, **`pop()`** is pop, **`lst[-1]`** is peek.

When the Array Is Full, Double It: Amortized O(1) Push




```
def push(self, x):
    if self._top == len(self._data) - 1:
        self._resize(2 * len(self._data))
    self._top += 1
    self._data[self._top] = x
def _resize(self, cap):          # O(n), but rare
    new = [None] * cap
    new[:len(self)] = self._data[:len(self)]
    self._data = new
```

Why push is still O(1)

Push n items, starting at capacity 1:

- **writes:** n
- **copies:** 1 + 2 + 4 + … + n/2 < n
- **total:** < 2n operations

So the average cost per push is a constant: **amortized O(1)**. One unlucky push costs O(n); the other pushes pay for it.

? **Think:** why double the capacity? What happens to the total cost if each resize adds just 10 slots instead?

Implementing a Stack: Where We Are

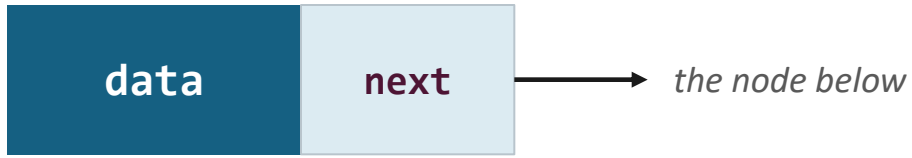
IIT Madras Zanzibar

1. The Stack ADT and two ways to store it
2. Part 1: Stack on an array
3. **Part 2: Stack on a linked list**
4. Array vs linked list
5. Live demo, exercise and summary

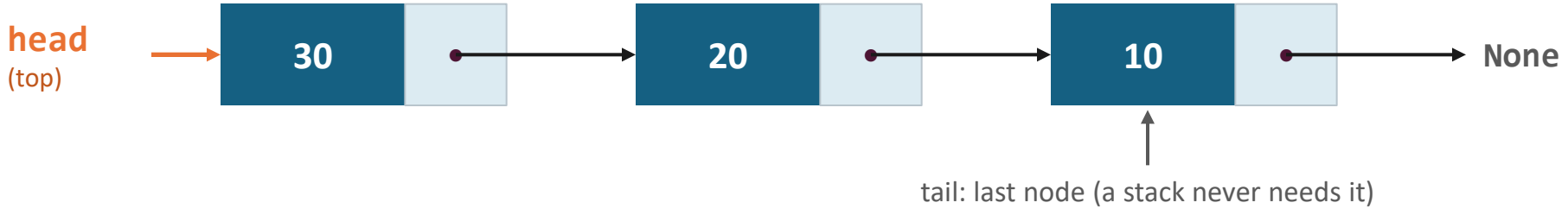
A Linked Stack Is a Chain of Nodes; the Head Is the Top

IIT Madras Zanzibar

```
class Node:
    __slots__ = ('data', 'next')
    def __init__(self, data, next=None):
        self.data = data   # the value
        self.next = next   # the node below
```



One node = one value + one reference



Top at the head ✓

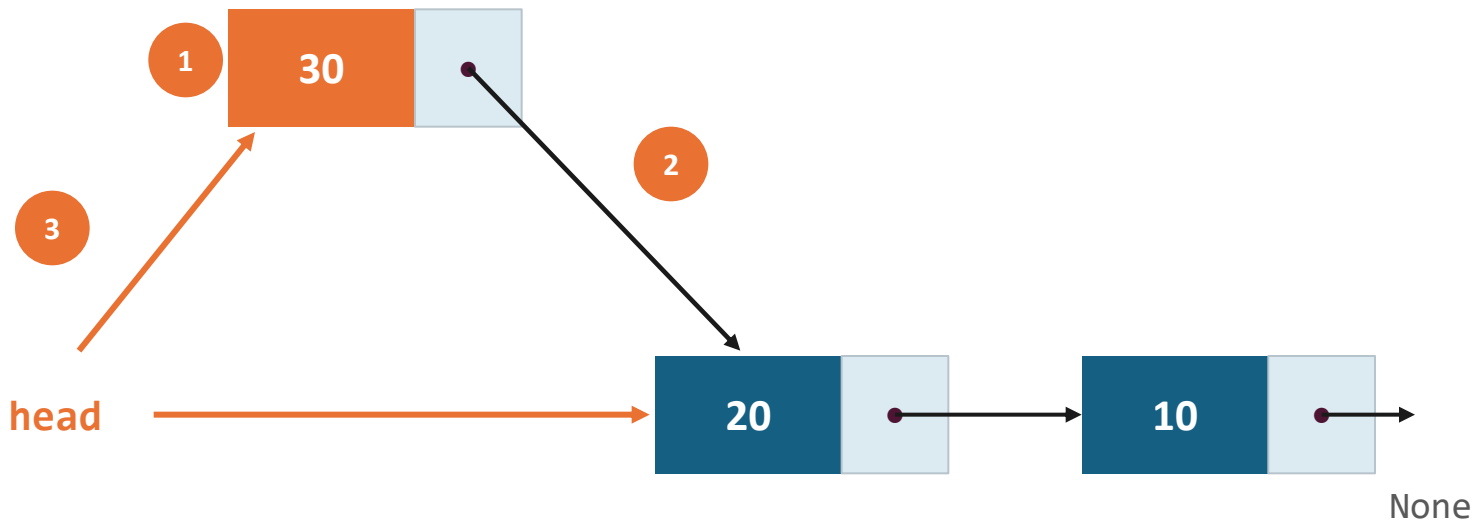
- push: link new node in front, O(1)
- pop: head = head.next, O(1)

Top at the tail ✗

- push: tail.next = new node, O(1)
- pop: must walk from head to find the new tail, O(n)

push(x) on a Linked List: Create, Link, Move head

Start: the stack holds 20 (top) and 10. Click to run push(30) one line at a time.




```
def push(self, x):
    node = Node(x)          # 1
    node.next = self._head  # 2
    self._head = node       # 3
    self._size += 1
```

1 create a node for 30

2 point it at the old top

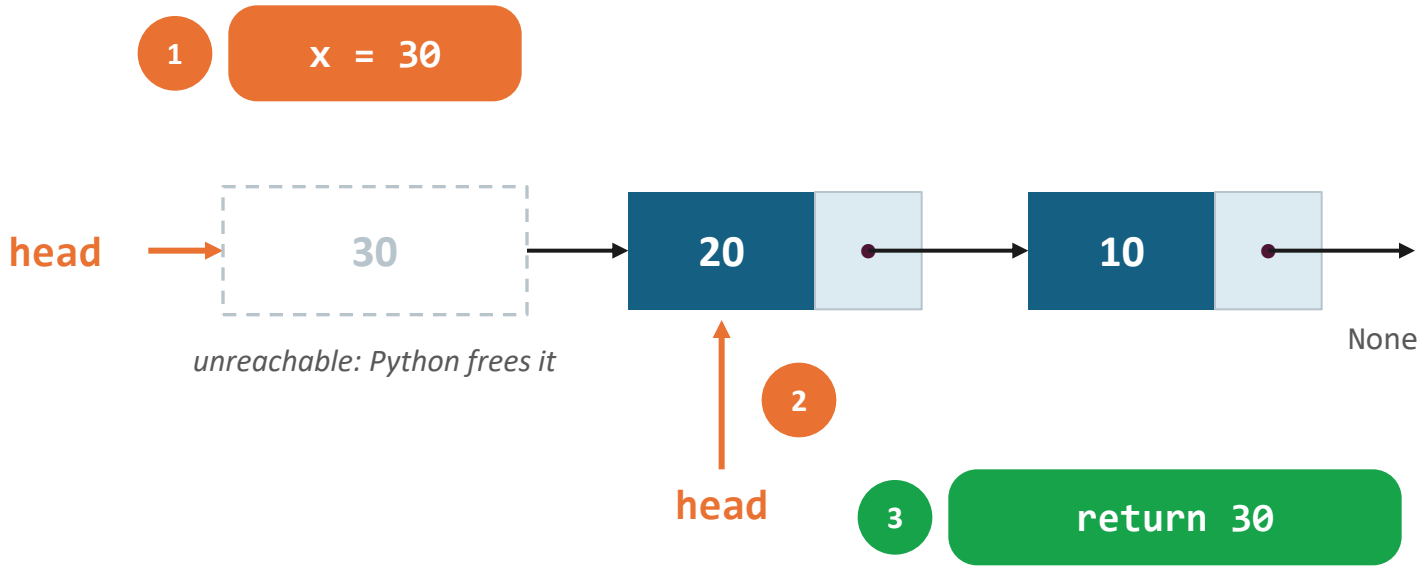
3 make it the new head

Order matters. Swap steps 2 and 3 and head already points at the new node, so **`node.next = self._head`** makes the node point at itself. 20 and 10 are lost. No array, no copying, no overflow: push is O(1) every time, not just on average.

pop() on a Linked List: Read, Unlink, Return

IIT Madras Zanzibar

Start: 30 (top), 20, 10. Click to run pop() one line at a time.




```
def pop(self):
    if self._head is None:
        raise IndexError('empty')
    x = self._head.data      # 1
    self._head = self._head.next  # 2
    self._size -= 1
    return x                 # 3
```

peek() = step 1 only: return `self._head.data`

Empty stack ⇔ `self._head is None`

One assignment does the work. **`self._head = self._head.next`** skips the old top. Nothing points at it any more, so Python's garbage collector reclaims the node. In C you would have to call free() yourself.

The Complete LinkedStack Class

IIT Madras Zanzibar

```
class Node:
    __slots__ = ('data', 'next')
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

class LinkedStack:
    """LIFO stack on a singly linked list."""
    def __init__(self):
        self._head = None    # top of stack
        self._size = 0

    def is_empty(self):
        return self._head is None

    def __len__(self):
        return self._size
```

```
    def push(self, x):
        self._head = Node(x, self._head)
        self._size += 1

    def pop(self):
        if self.is_empty():
            raise IndexError('pop from empty stack')
        x = self._head.data
        self._head = self._head.next
        self._size -= 1
        return x

    def peek(self):
        if self.is_empty():
            raise IndexError('peek at empty stack')
        return self._head.data
```

Same five methods, same names, same exceptions. Any code written for **`ArrayStack`** runs unchanged on **`LinkedStack`**. That is what separating the interface from the implementation buys you.

Array vs Linked List: Same Interface, Different Trade-offs

IIT Madras Zanzibar

	Array stack (dynamic)	Linked stack
push	O(1) amortized; O(n) on a resize	O(1) worst case
pop, peek	O(1)	O(1)
Memory per item	one reference, plus spare slots (up to 2×)	a whole node object: data + next
Speed in practice	contiguous memory: cache-friendly, fast	nodes scattered in memory: slower
When full	grow by copying to a 2× block	never full until RAM runs out
Use it when	almost always (it is Python's list)	every push must be fast, e.g. real-time code

Rule of thumb: use a Python list as your stack. Write the linked version when one slow push is unacceptable, or when the language gives you no dynamic array (C, embedded code).

Six Mistakes Everyone Makes Implementing a Stack

IIT Madras Zanzibar

1. **Wrong order in push**

   array: write, then move _top. linked: move head before linking

2. **Off-by-one on _top**

   full is _top == capacity − 1, not == capacity

3. **Returning None when empty**

   raise IndexError; None may be a real value

4. **Top at the tail**

   singly linked pop becomes O(n): keep the top at the head

5. **list.insert(0, x) as push**

   shifts every element: O(n). Use append and pop()

6. **Forgetting _size or the clear**

   len() goes wrong; old slots keep objects alive

? **Think:** which of these would a simple push/pop test catch, and which would pass every test and still be wrong?

Live Demo: One Test, Two Implementations

IIT Madras Zanzibar

```
from stacks import ArrayStack, LinkedStack

def check(S):
    s = S()
    for x in [10, 20, 30]:
        s.push(x)
    print(S.__name__, 'peek:', s.peek(), 'len:', len(s))
    print('popped:', [s.pop() for _ in range(3)])
    try:
        s.pop()
    except IndexError as e:
        print('empty pop ->', e)

check(ArrayStack)
check(LinkedStack)

def reverse(text, S=LinkedStack):
    s = S()
    for ch in text:
        s.push(ch)
    return ''.join(s.pop() for _ in range(len(s)))

print(reverse('stack'))
```

Expected output

```
ArrayStack peek: 30 len: 3
popped: [30, 20, 10]
empty pop -> pop from empty stack
LinkedStack peek: 30 len: 3
popped: [30, 20, 10]
empty pop -> pop from empty stack
kcats
```

Change one thing, run again

- **`reverse('stack', ArrayStack)`**: same answer
- Push 9 items into a fixed **`ArrayStack(8)`**: OverflowError
- Time 10^6 pushes on each with **`timeit`**

In-Class Exercise (8 minutes, in pairs)

IIT Madras Zanzibar

TRACE IT, DRAW IT, EXTEND IT

1. ArrayStack(capacity=4) with doubling: push 5, push 8, pop, push 3, push 7, push 1, push 9. Give **`_data`**, **`_top`** and the capacity at the end. How many items were copied?

2. LinkedStack: push 'a', push 'b', pop, push 'c'. Draw the nodes and head. What do **`peek()`** and **`len()`** return?

3. Extend LinkedStack with **`get_min()`** that returns the smallest item in O(1). Hint: what could each node remember?

4. Someone puts the top of a singly linked stack at the tail. Which operation becomes O(n), and why does a tail pointer not save it?

Solution 1: Tracing the Array Stack Step by Step

IIT Madras Zanzibar

Operation	_data (index 0 on the left)	_top	capacity
`push(5)`	5	0	4
`push(8)`	5 8	1	4
`pop() → 8`	5 None	0	4
`push(3)`	5 3	1	4
`push(7)`	5 3 7	2	4
`push(1)`	5 3 7 1	3	4 full
`push(9)`	5 3 7 1 9	4	8

full, so resize to 8 first

4 items copied

Orange cell = slot just written. Triangle = where **_top** points. Dashed = slot cleared by pop.

Final answer

_data = [5, 3, 7, 1, 9, None, None, None]

_top = 4 (index; size is 5)

capacity = 8 after one doubling

4 items copied, once, on the 7th operation

Common slip: writing _top = 5. That is the size, not the index.

Solution 2: Drawing the Linked Stack Step by Step

IIT Madras Zanzibar

Operation	Nodes after the operation (top on the left)	_size
`push('a')`	head → 'a' → None	1
`push('b')`	head → 'b' → 'a' → None	2
`pop() → 'b'`	head → 'a' → None; 'b' *unlinked, returns 'b'*	1
`push('c')`	head → 'c' → 'a' → None	2

Final answer

head → **'c'** → **'a'** → None

`peek()` returns 'c'

`len(s)` returns 2

Node 'b' is gone: nothing points to it, so Python frees it.

Common slip: keeping 'b' in the drawing after the pop.

Orange node = node just pushed. Dashed = node removed by pop.

Solution 3: get_min() in O(1) by Storing a Min in Every Node

IIT Madras Zanzibar

Operation	Nodes after the operation (top on the left)	get_min()
`push(5)`	head → [5 | min 5] → None	5
`push(3)`	head → [3 | min 3] → [5 | min 5] → None	3
`push(7)`	head → [7 | min 3] → [3 | min 3] → [5 | min 5] → None	3
`pop() → 7`	head → [3 | min 3] → [5 | min 5] → None; [7 | min 3] *unlinked, returns 7*	3
`pop() → 3`	head → [5 | min 5] → None; [3 | min 3] *unlinked, returns 3*	5

Orange node = node just pushed. Green cell = min stored in that node. Dashed = node removed by pop.

Final answer

Store a running minimum in every node.

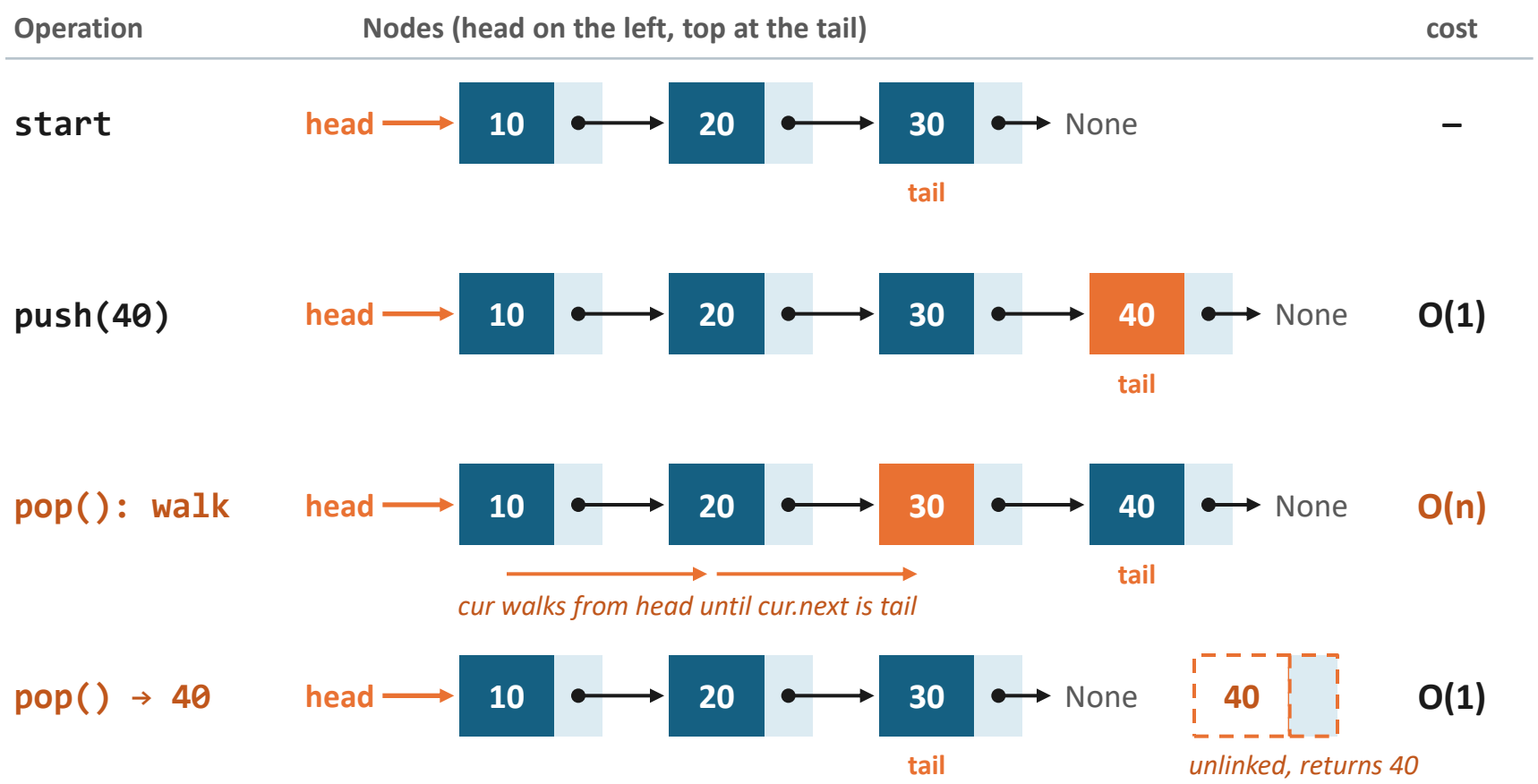
`push`: the new node's min is min(x, head.min), or x if the stack is empty

`get_min()` returns self._head.min, O(1)

`pop()` needs no change: the node below already holds the old minimum

Common slip: one self._min field. After pop it cannot recover the old minimum without an O(n) scan.

Solution 4: With the Top at the Tail, pop() Becomes O(n)



Final answer

pop() becomes O(n); push stays O(1).

pop must move tail back to the second-to-last node.

next only points forward, so finding that node takes n − 1 hops from head.

Fix: keep the top at the head, as LinkedStack does.

Common slip: thinking a tail pointer makes pop O(1). It finds the last node, not the one before it.

Orange node = node the step lands on. Orange arrows = cur moving forward. Dashed = node removed by pop.

Summary and Exam Checkpoints

IIT Madras Zanzibar

- ✓ **ADT vs implementation:** push, pop, peek, is_empty, len: one interface, many ways to build it
- ✓ **Array stack:** fixed block + _top index; push moves up then writes; pop reads, clears, moves down
- ✓ **Resizing:** double when full: one O(n) copy, amortized O(1) per push
- ✓ **Linked stack:** top = head; push links a new node in front; pop sets head = head.next
- ✓ **Trade-off:** array: compact and fast in practice; linked: worst-case O(1), never full

Next

Today 2:00 pm: Lab 2: sorts, Stack, bracket checker, postfix calculator

Fri 23 Oct: Queues, deques and priority queues

Practice: LeetCode 155 'Min Stack' and 20 'Valid Parentheses'

? **Think:** tomorrow's queue removes from the FRONT. Which of today's two designs adapts more easily, and what extra pointer would it need?

References and Further Reading

1. Canning, J., Broder, A. & Lafore, R. (2022). Data Structures & Algorithms in Python. Addison-Wesley. Chapter 4 (Stacks and Queues) and Chapter 5 (Linked Lists).
2. Goodrich, M. T., Tamassia, R. & Goldwasser, M. H. (2013). Data Structures and Algorithms in Python. Wiley. Sections 5.3 (dynamic arrays and amortization), 6.1 (stacks) and 7.1 (linked stacks).
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). Introduction to Algorithms, 4th ed. MIT Press. Chapter 16 (Amortized Analysis, dynamic tables).
4. Python Software Foundation. Using lists as stacks. docs.python.org/3/tutorial/datastructures.html
5. LeetCode. Problem 155 (Min Stack) and Problem 20 (Valid Parentheses). leetcode.com/problems