

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen



SCHOOL OF
DATA SCIENCE
數據科學學院

Introduction to Computer Science: Programming Methodology

Lecture 9
Recursion, Stack and Queue

Tongxin Li
School of Data Science

Linear Recursion

- If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is known as linear recursion

- Finding the smallest number and binary search are both linear recursive algorithms

Practice: Sum of a List

- Use recursion to calculate the sum of a list of numbers.

```
[1, 2, 3, 4, 5, 9, 100, 46, 7]
```

Which smaller problem should the function solve?

Sum of a List: Implementation

```
def linear_sum(values, n):
    if n == 0:
        return 0
    return linear_sum(values, n - 1) + values[n - 1]

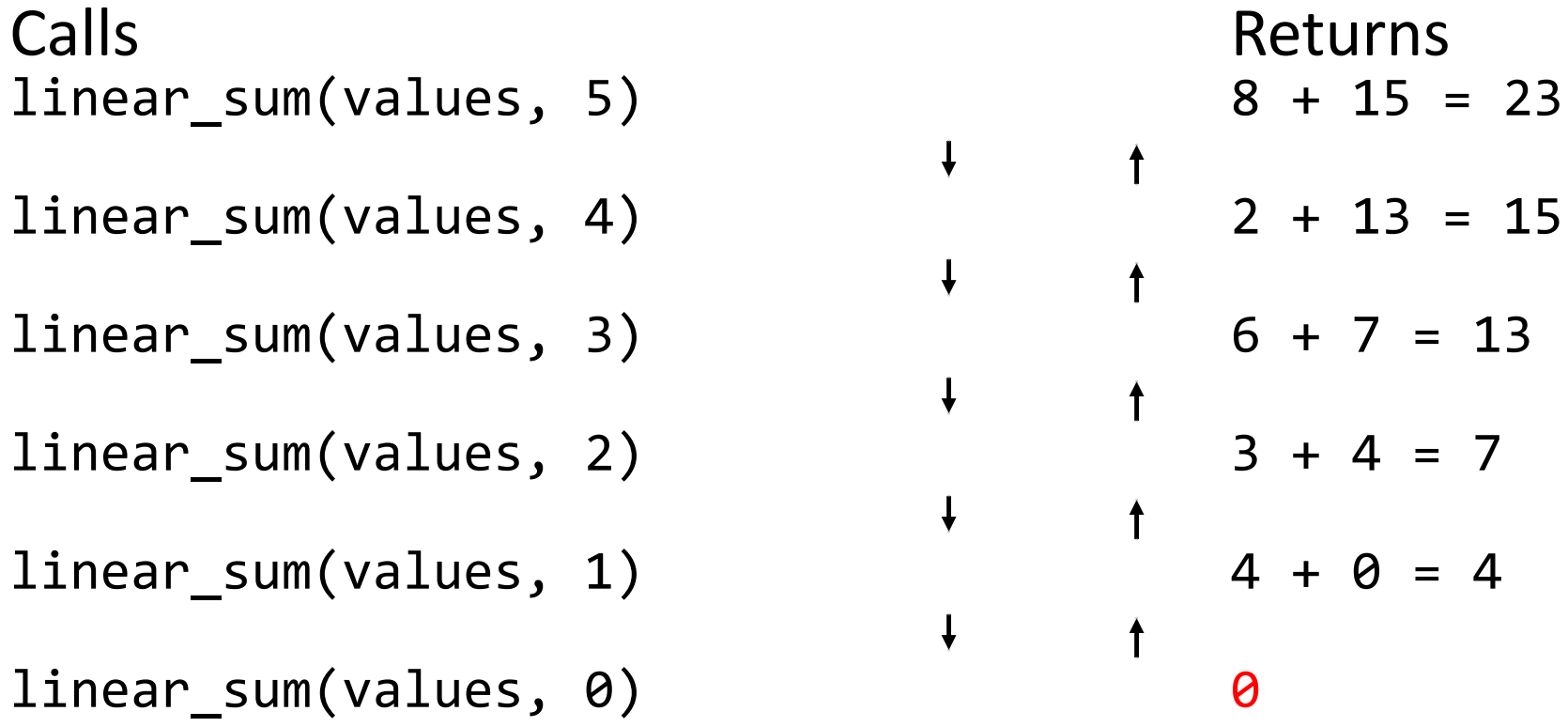


values = [1, 2, 3, 4, 5, 9, 100, 46, 7]
print(linear_sum(values, len(values)))


Output: 177
```

Recursive Sum: Call Trace

Example list: [4, 3, 6, 2, 8]



Practice: Power Function

- Compute x raised to a nonnegative integer power n using recursion.

- Aim for O(log n) time.

Example: $2^5 = 32$

Power Function: Recursive Definition

Compute the half-power once: h = power(x, n // 2)

n = 0	`return 1`
n is even	`return h * h`
n is odd	`return x * h * h`

Power Function: Implementation

```
def power(x, n):
    if n == 0:
        return 1
    half = power(x, n // 2)
    result = half * half
    if n % 2 == 1:
        result = result * x
    return result
```

```
print(power(2, 5))
print(power(2, 0))
```

Output: 32, then 1

Multiple Recursion

- When a function makes two or more recursive calls, we say that it uses multiple recursion

- Drawing the English ruler is a multiple recursion program

Practice: Binary Sum

- Write binary_sum() to calculate the sum of a list.

- Use two recursive calls: one for each half.

```
[1, 2, 3, 4, 5, 6, 7]
```

Binary Sum: Implementation

```
def binary_sum(values, start, stop):
    if start >= stop:
        return 0
    if stop - start == 1:
        return values[start]
    mid = (start + stop) // 2
    left = binary_sum(values, start, mid)
    right = binary_sum(values, mid, stop)
    return left + right

values = [1, 2, 3, 4, 5, 6, 7]
print(binary_sum(values, 0, len(values)))
```

Output: 28

Reverse a List

- Print the elements of a list in reverse order using recursion.

Input



Output

3 2 1

- What should the stopping condition be?

Reverse a List: Implementation

```
def print_reverse(arr, size):
    if size == 0:
        return
    print(arr[size - 1])
    print_reverse(arr, size - 1)

values = [1, 2, 3]
print_reverse(values, len(values))
```

Output: 3, 2, 1 (one number per line)

Merge Sort (Optional)

- Can we sort a list using recursion?

Before sorting

6	5	12	10	9	1

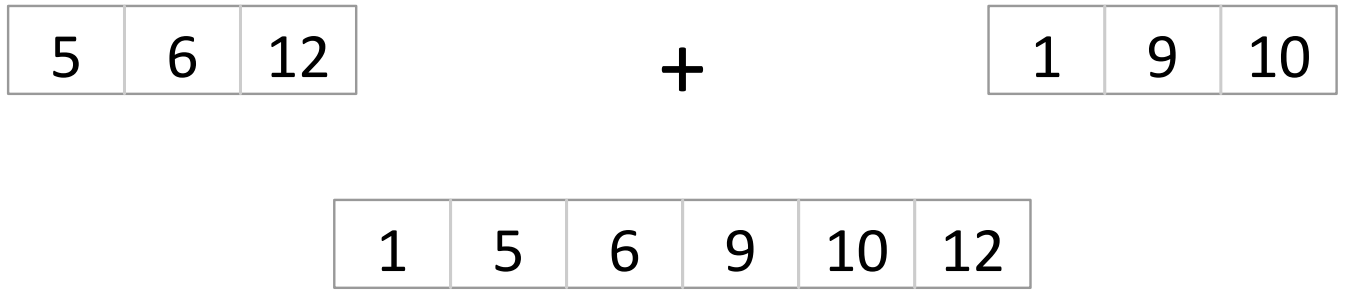
After sorting

1	5	6	9	10	12

- How can smaller sorted lists help?

Merge Sort: Split and Merge

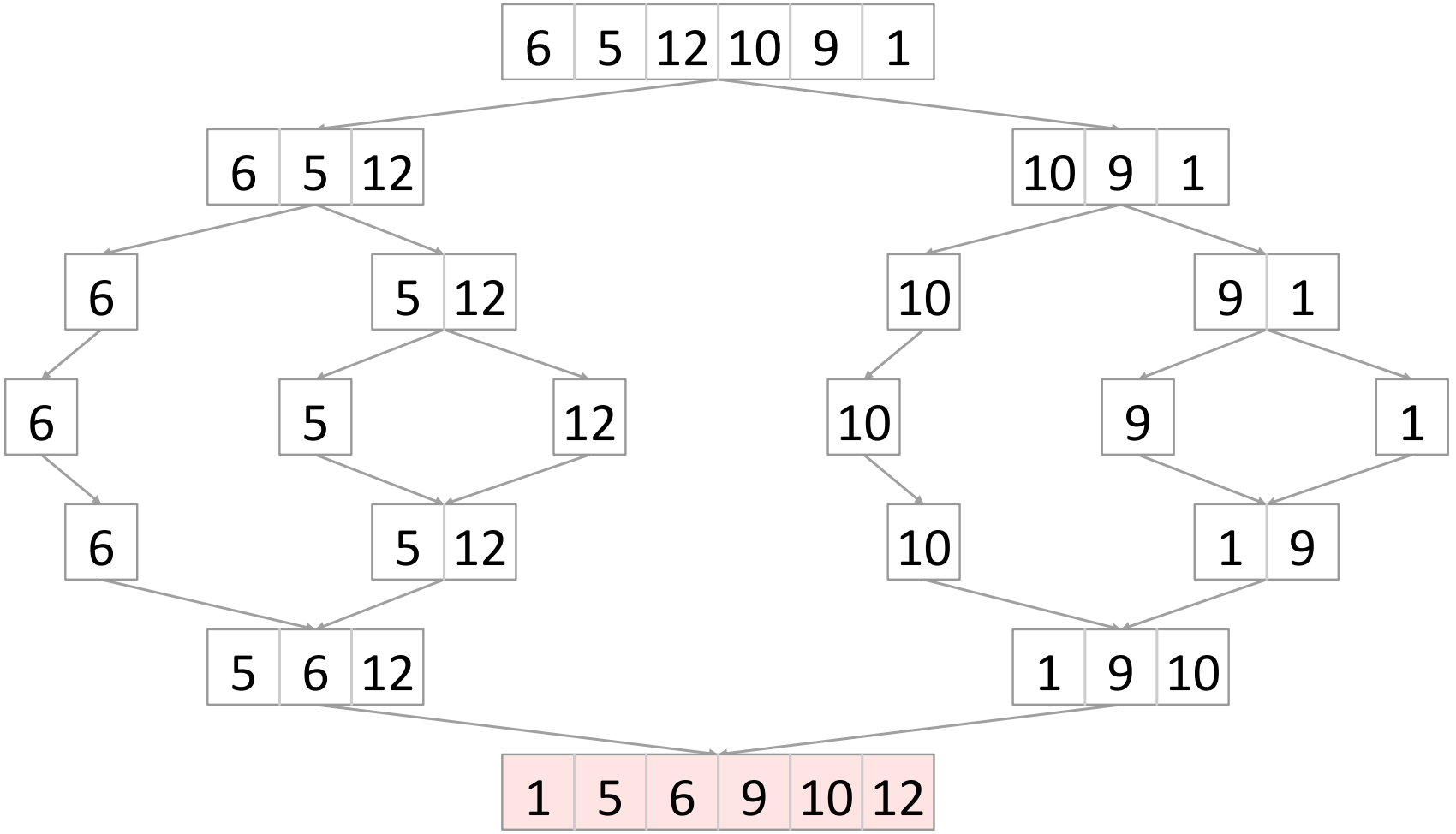
- Split the list into two halves.
- Sort each half using the same function.
- Merge the two sorted halves.



Merge Sort: Recursive Trace

Split

Merge



Merge Sort: Recursive Function

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

values = [6, 5, 12, 10, 9, 1]
print(merge_sort(values))

Output: [1, 5, 6, 9, 10, 12]
```

Merge Sort: Merging Two Lists

```
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Merge Sort: Complexity

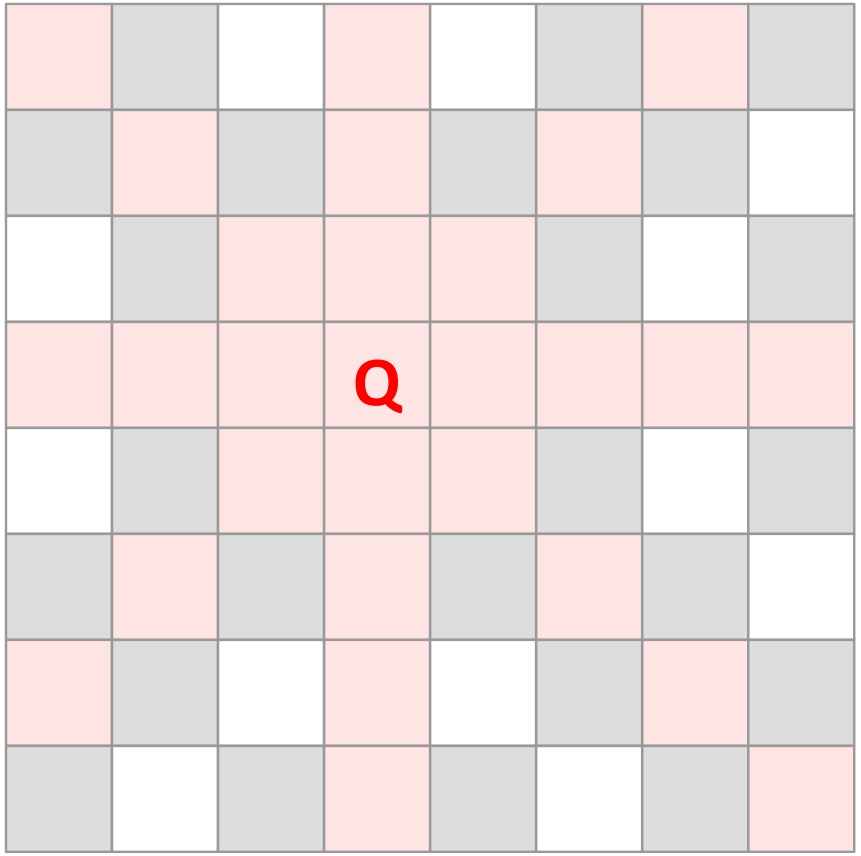
Worst-case time: O(n log n)

- There are O(log n) levels of recursion.
- Merging processes O(n) elements per level.

Peak auxiliary space: O(n)

Temporary lists use O(n) space.
The recursive call stack uses O(log n).

Eight Queens (Optional)



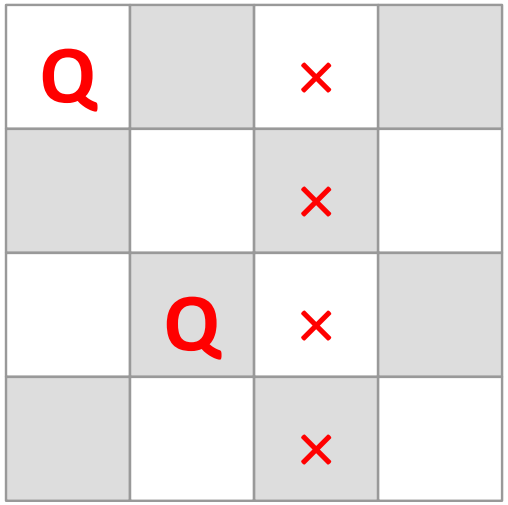
Place eight queens on
an 8 × 8 chessboard.

No two queens may share
a row, column, or diagonal.

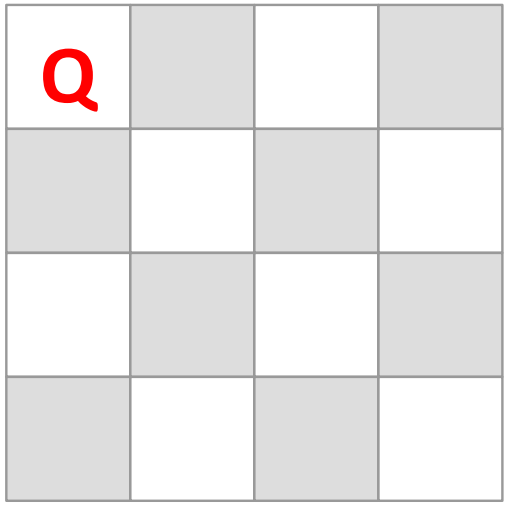
Pink squares show
where this queen attacks.

Backtracking: A Four-Queen Example

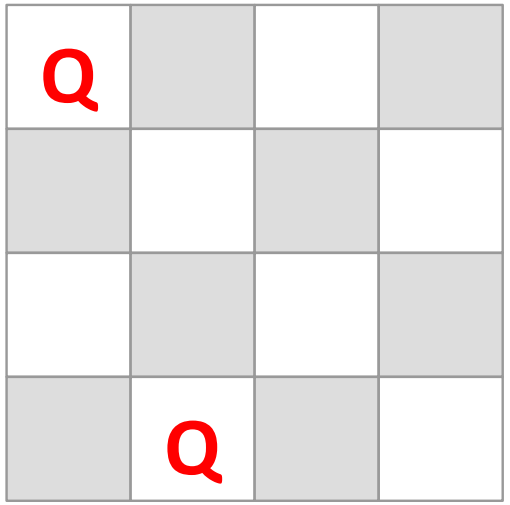
No safe row



Remove the queen



Try the next row



If the next column has no safe square, go back and change the previous choice.

Eight Queens: Recursive Search

```
def solve(board, col):
    if col == len(board):
        return True
    for row in range(len(board)):
        if is_safe(board, row, col):
            board[col] = row
            if solve(board, col + 1):
                return True
            board[col] = -1
    return False
```

board[col] stores the row of the queen in that column.

Eight Queens: Checking a Square

```
def is_safe(board, row, col):
    for c in range(col):
        r = board[c]
        if r == row:
            return False
        if abs(r - row) == abs(c - col):
            return False
    return True
```

Same row: r == row

Same diagonal: equal row and column distances

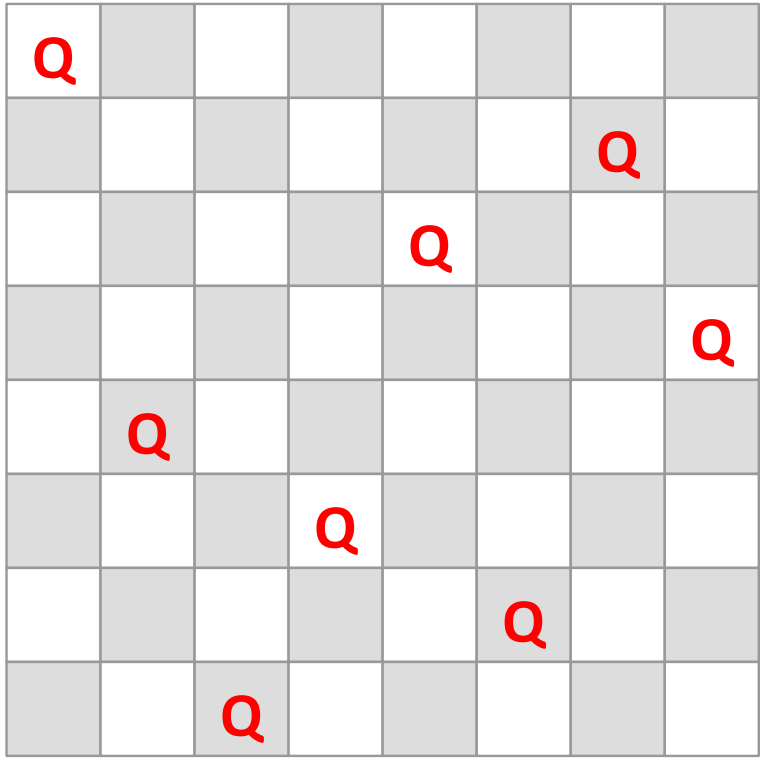
Eight Queens: One Solution

```
board = [-1] * 8
if solve(board, 0):
    print(board)
```

Rows, indexed by column:

```
[0, 4, 7, 5, 2, 6, 1, 3]
```

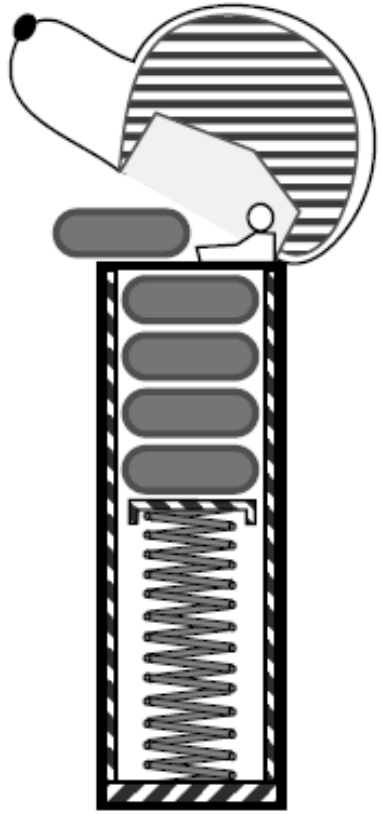
Rows and columns
are numbered 0 to 7.



Stack

- A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle

- A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack)







Stacks: Browser History

- Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.

Stacks: Undo

- Text editors usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

Stack Operations

`stack.push(item)`	Add an item at the top.
`stack.pop()`	Remove and return the top item.
`stack.top()`	Return the top item without removing it.
`stack.is_empty()`	Check whether the stack is empty.
`len(stack)`	Return the number of items.

On an empty stack, top() and pop() print a message
and return None.

Implementing a Stack

```
class ListStack:
    def __init__(self):
        self._data = []

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        return len(self) == 0

    def push(self, item):
        self._data.append(item)
```

```
def top(self):
    if self.is_empty():
        print("Empty stack")
        return None
    return self._data[-1]

def pop(self):
    if self.is_empty():
        print("Empty stack")
        return None
    return self._data.pop()
```

Using a Stack

```
stack = ListStack()
print(stack.is_empty())
stack.push(100)
stack.push(200)
stack.push(300)
print(stack.top())
print(stack.pop())
print(stack.top())
```

Output

```
True
300
300
200
```

Practice: Reversing a List with a Stack

- Use ListStack to reverse the order of a list.

```
Input:  [1, 2, 3, 4, 5]

Output: [5, 4, 3, 2, 1]
```

Reversing a List with a Stack

```
def reverse_with_stack(values):
    stack = ListStack()
    result = []
    for item in values:
        stack.push(item)
    while not stack.is_empty():
        result.append(stack.pop())
    return result

values = [1, 2, 3, 4, 5]
print(reverse_with_stack(values))


Output: [5, 4, 3, 2, 1]
```

Practice: Matching Brackets

- Check whether every opening bracket has a matching closing bracket in the correct order.

( ) [ ] { }

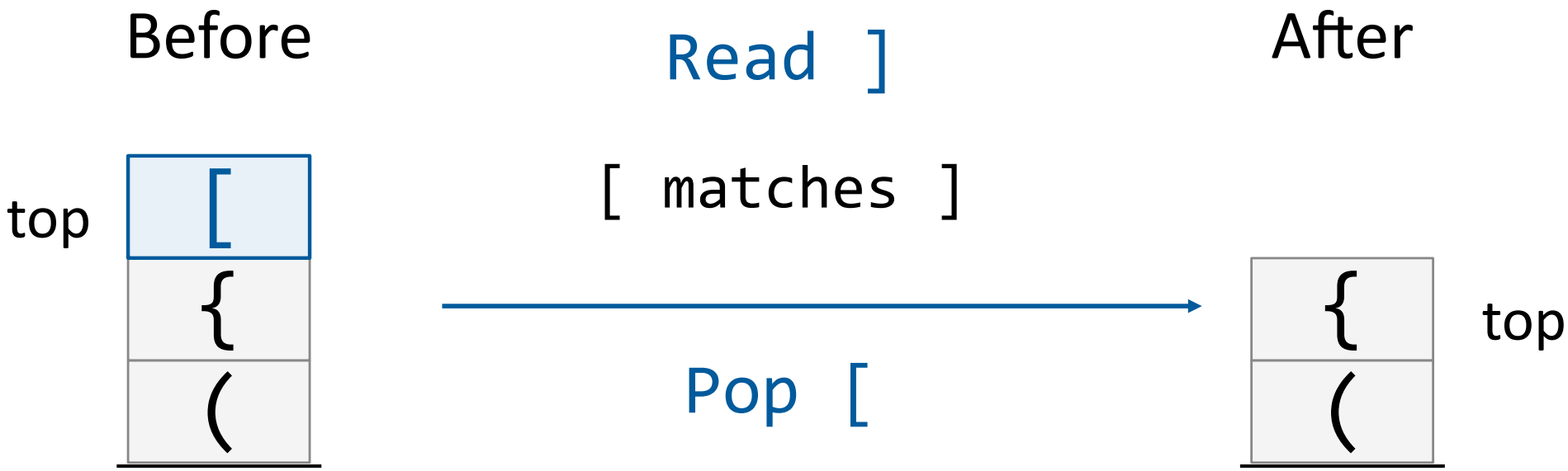
Can a stack remember which bracket must close next?

Balanced and Unbalanced Brackets

Balanced	Unbalanced
✓ ()\{[]\}	× (]
✓ (\{\{\}\})	× ()\{([])\}]
✓ (\{[]\})	× ()[[]\{\}

Matching Brackets: The Stack Rule

A closing bracket must match the most recent unmatched opening bracket.



Matching Brackets: Algorithm

- Start with an empty stack.
- Opening bracket: push it onto the stack.
- Closing bracket: pop and compare.
  An empty stack or a mismatch means False.
- At the end, return True only if the stack is empty.

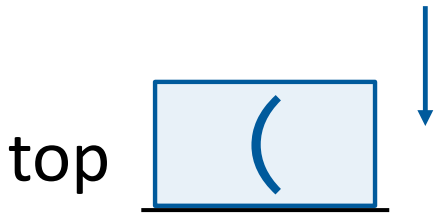
Matching Brackets: Opening Brackets

Input: ({[]})

Stack after each step

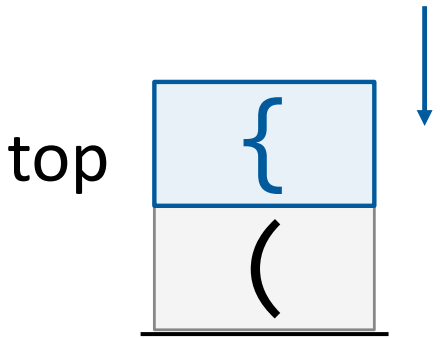
1. Read (

Push



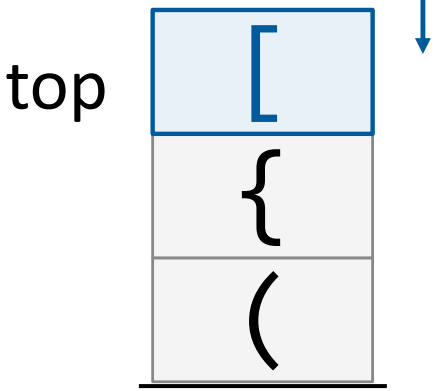
2. Read {

Push



3. Read [

Push

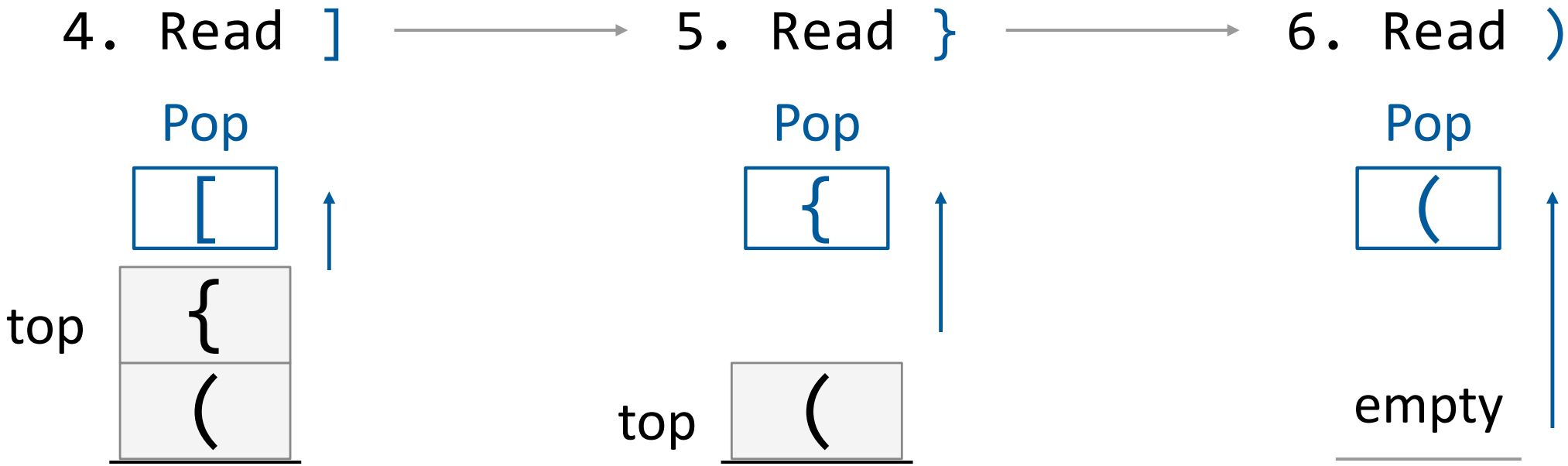


Each opening bracket becomes the new top.

Matching Brackets: Closing Brackets

Input: ({[]})

Stack after each step



Every pair matches. The stack is empty: True.

Practice: Tracing Bracket Matching

Trace the stack after reading each character.

A: {(][])}

B: ()[[]{}

Which condition makes each input fail?

Bracket Matching: A Mismatch

Input: {(][])}

Stack after each step

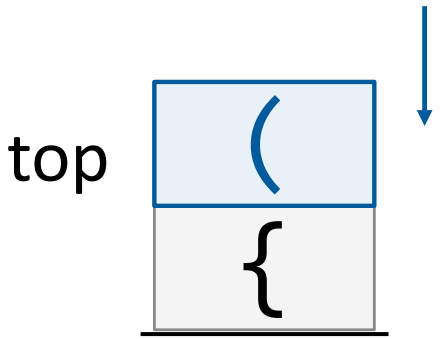
1. Read {

Push



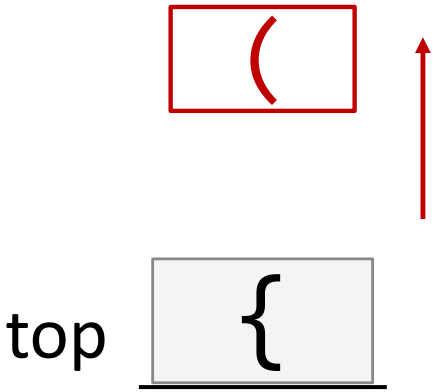
2. Read (

Push



3. Read]

Pop: mismatch

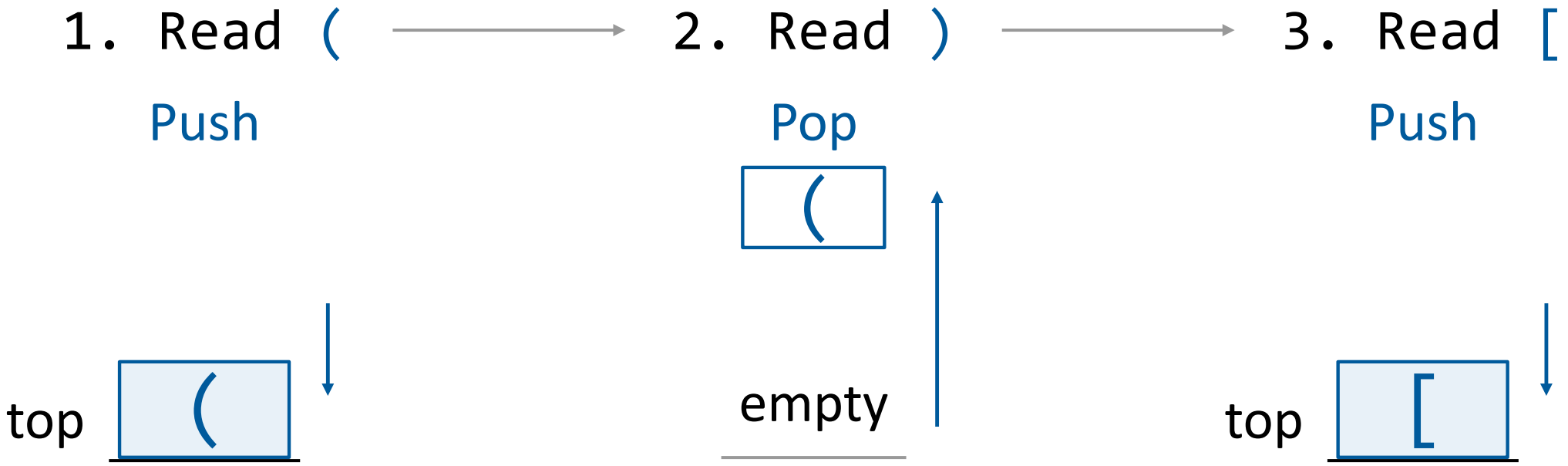


] needs [, but the top was (. Stop: False.

Unclosed Brackets: Trace (1)

Input: ()[]{}

Stack after each step



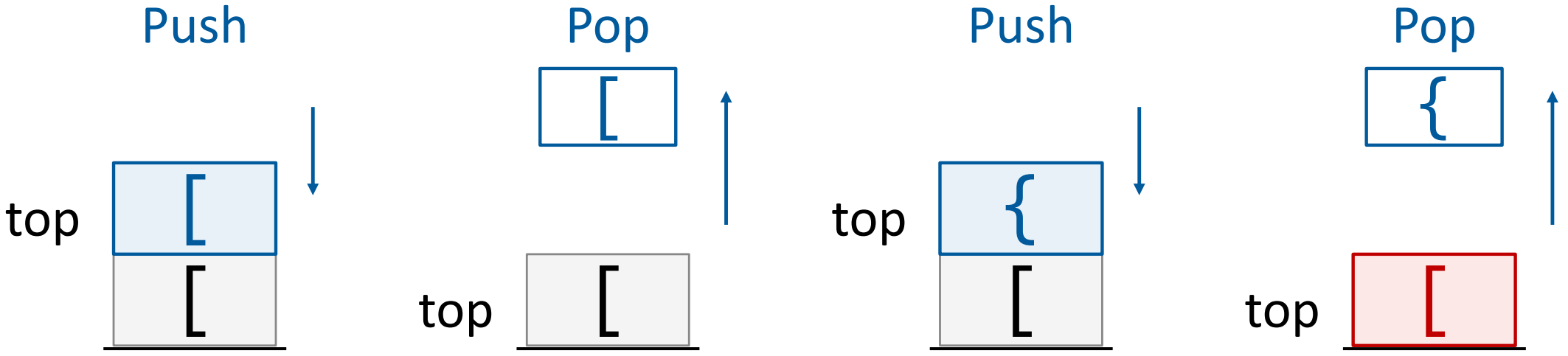
The first pair matches. Continue reading.

Unclosed Brackets: Trace (2)

`Input: ()[[]{}`

Stack after each step

`4. Read [` → `5. Read ]` → `6. Read {` → `7. Read }`



Input finished, but [is still unclosed: False.

Matching Brackets: Implementation

```
def is_matched(expr):
    pairs = {')': '(', ']': '[', '}': '{'}
    stack = ListStack()
    for symbol in expr:
        if symbol in "([{":
            stack.push(symbol)
        elif symbol in ")]}":
            if stack.is_empty():
                return False
            if stack.pop() != pairs[symbol]:
                return False
    return stack.is_empty()
```

({[]}) : True {(][])} : False

Practice: Matching HTML Tags

- An opening tag <name> must match a closing tag </name>.

- Check whether the tags are properly nested.

For this exercise, use simple paired tags without attributes.

```
<p><b>Hello</b></p>
```

Common HTML Tags

`<body>`	Document body
`<h1>`	Main heading
`<center>`	Centered content
`<p>`	Paragraph
`<ol>`	Ordered list
`<li>`	List item

HTML Source and Rendered Page

```
<body>
    <h1>Study Plan</h1>
    <p>Today:</p>
    <ol>
        <li>Read</li>
        <li>Practice</li>
    </ol>
</body>
```

Study Plan

Today:

1. Read
2. Practice

Finding Tags in a String

```
def get_tags(raw):
    tags = []
    start = raw.find("<")
    while start != -1:
        stop = raw.find(">", start + 1)
        if stop == -1 or stop == start + 1:
            return None
        tags.append(raw[start + 1:stop])
        start = raw.find("<", stop + 1)
    return tags

print(get_tags("<p>Hello</p>"))
```

```
Output: ['p', '/p']
```

Matching HTML Tags: Implementation

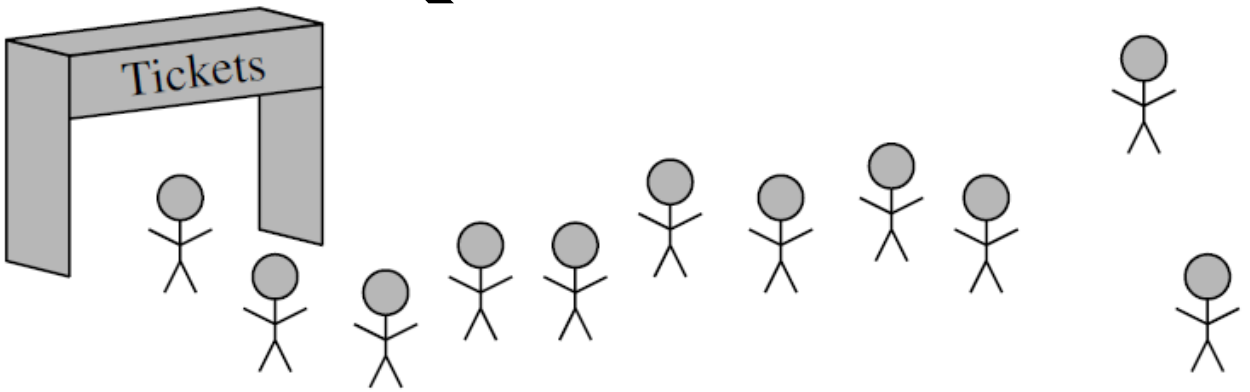
```
def is_matched_html(raw):
    tags = get_tags(raw)
    if tags is None:
        return False
    s = ListStack()
    for tag in tags:
        if tag.startswith("/"):
            if s.is_empty() or s.pop() != tag[1:]:
                return False
        else:
            s.push(tag)
    return s.is_empty()
```

Hi : True

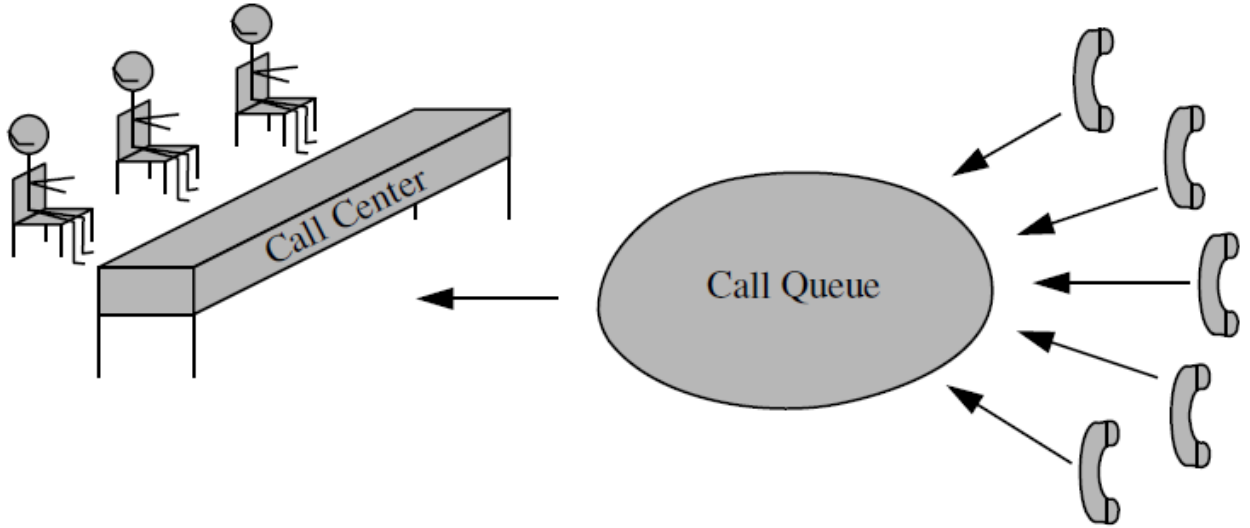
Queue

- Queue is another fundamental data structure
- A queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle
- Elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed

Applications of Queues



(a)



(b)

Queues in Everyday Life

People wait for service in arrival order.



Queue Class: State and Access

```
class ListQueue:
    def __init__(self, cap=5):
        self._data = [None] * cap
        self._size = 0
        self._front = 0
        self._end = 0

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

```
def is_empty(self):
    return len(self) == 0

def first(self):
    if self.is_empty():
        print("Empty queue")
        return None
    i = self._front
    return self._data[i]
```

The array has a fixed capacity. The front and end indices wrap around when they reach its length.

Queue Class: Enqueue and Dequeue

```
def enqueue(self, item):
    n = len(self._data)
    if len(self) == n:
        print("Full queue")
        return None
    i = self._end
    self._data[i] = item
    self._end = (i + 1) % n
    self._size += 1
```

```
def dequeue(self):
    if self.is_empty():
        print("Empty queue")
        return None
    n = len(self._data)
    i = self._front
    item = self._data[i]
    self._data[i] = None
    self._front = (i + 1) % n
    self._size -= 1
    return item
```

enqueue() adds at the back. dequeue() removes and returns the item at the front.

Practice: Simulating a Web Service

- A website stores arriving requests in a queue.
- New requests join the back of the queue.
- The server processes requests in arrival order.
- Simulate arrivals and processing. Give each request a randomly chosen processing time.

Web Service: Implementation

```
class WebService:
    def __init__(self, cap=5):
        self.cap = cap
        self.names = ListQueue(cap)
        self.times = ListQueue(cap)

    def add(self, name, secs):
        if len(self.names) == self.cap:
            print("Full service")
            return None
        self.names.enqueue(name)
        self.times.enqueue(secs)
```

```
def process(self):
    if len(self.names) == 0:
        return None
    nq = self.names
    tq = self.times
    name = nq.dequeue()
    secs = tq.dequeue()
    print(name, secs)
    return name, secs
```

Output gives the request name and its processing time.

Web Service: Simulation Loop

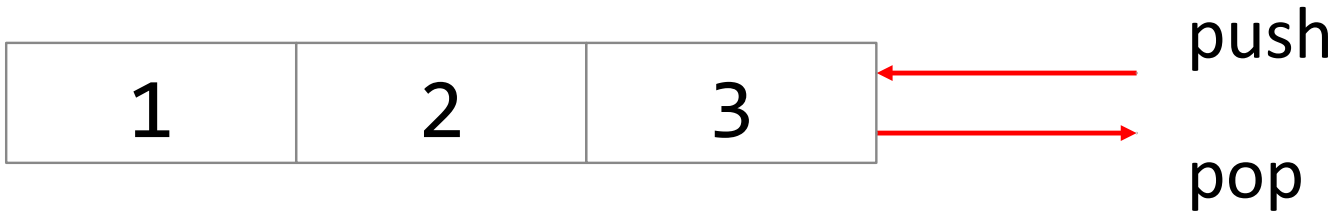
```
from random import choice, randint, random

service = WebService()
names = ["Video A", "Video B", "Video C"]
for _ in range(30):
    if random() < 0.6:
        name = choice(names)
        secs = randint(1, 10)
        service.add(name, secs)
    else:
        service.process()
```

Each iteration creates or processes a request.

Stacks and Queues

Stack: last in, first out



Queue: first in, first out



Practice: A Stack with Two Queues

- Implement a stack using two queues.
- Keep push() simple: enqueue the new item.



How can pop() remove 3 using queue operations?

Two-Queue Stack: Push and Pop

```
class TwoQueueStack:
    def __init__(self, cap=5):
        self.q1 = ListQueue(cap)
        self.q2 = ListQueue(cap)

    def push(self, item):
        self.q1.enqueue(item)

    def __len__(self):
        return len(self.q1)

    def is_empty(self):
        return self.q1.is_empty()
```

```
def pop(self):
    if self.is_empty():
        print("Empty stack")
        return None
    a, b = self.q1, self.q2
    while len(a) > 1:
        item = a.dequeue()
        b.enqueue(item)
    item = a.dequeue()
    self.q1, self.q2 = b, a
    return item
```

Two-Queue Stack: Top and Size

```
def top(self):
    if self.is_empty():
        print("Empty stack")
        return None
    a, b = self.q1, self.q2
    while len(a) > 1:
        item = a.dequeue()
        b.enqueue(item)
    item = a.dequeue()
    b.enqueue(item)
    self.q1, self.q2 = b, a
    return item
```

```
stack = TwoQueueStack()
stack.push(100)
stack.push(200)
print(stack.top())
print(len(stack))
print(stack.pop())
print(stack.pop())
```

```
Output:
200
2
200
100
```

Practice: An Alternative Two-Queue Stack

- Can you make pop() take O(1) time?
- Keep the newest item at the front of the active queue.
- How should push() rearrange the two queues?