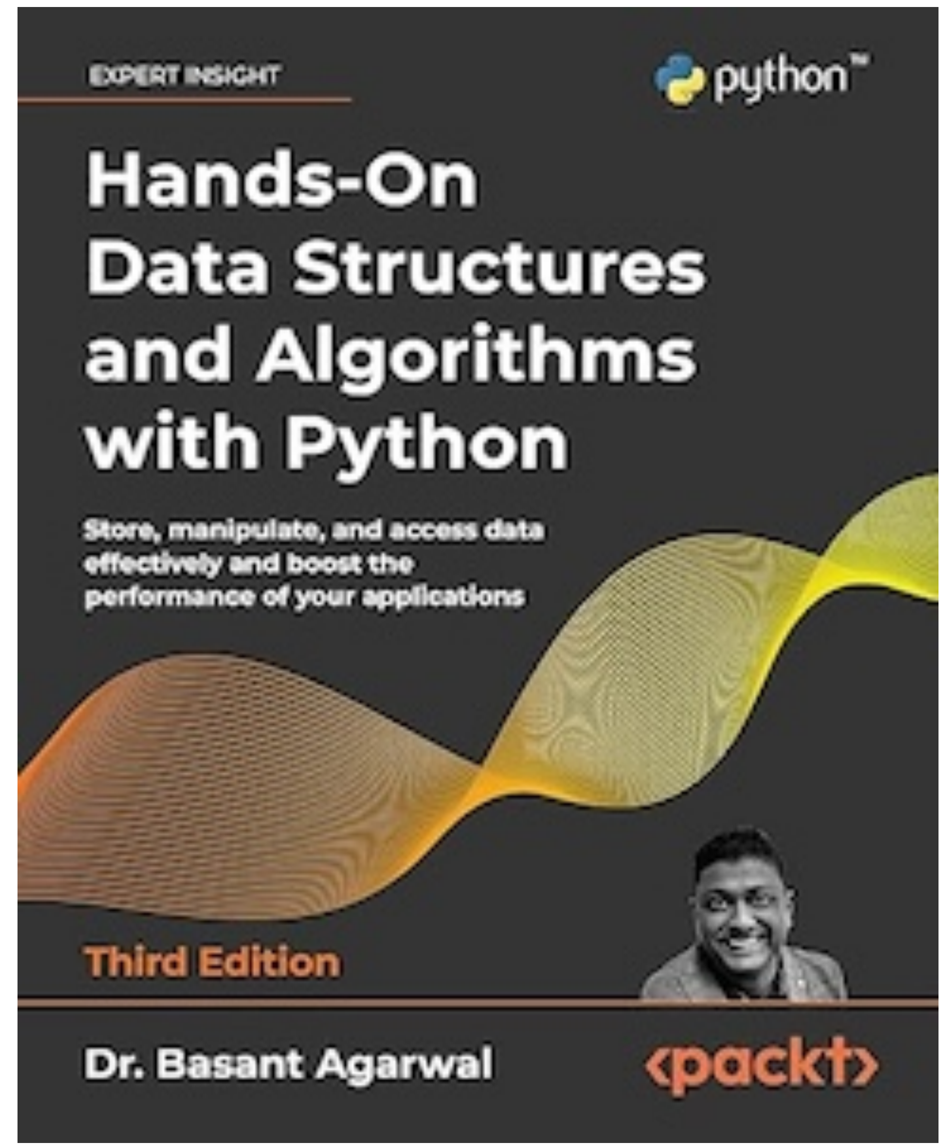
5 Stacks and Queues



For COMSC 132

Sam Bowne

Sep 26, 2024

Topics

- Stacks
- Queues

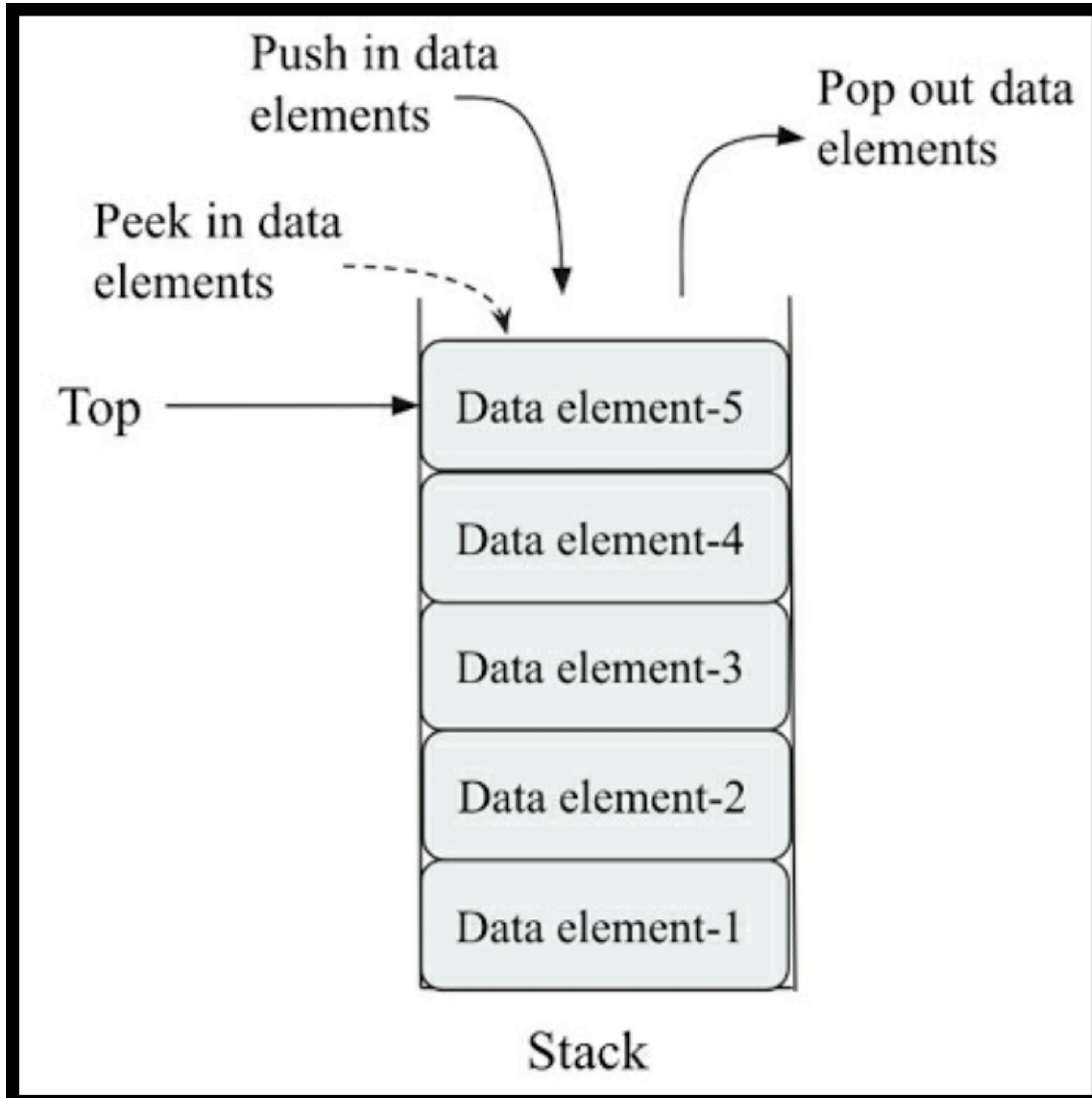
Stacks

Last In, First Out (LIFO)

- Data elements can only be inserted at the end
  - **push**
- Can only be deleted from the end
  - **pop**
- Can only be read from the end
  - **peek**

Stack

- **top** pointer marks the top of the stack
- Called the Stack Pointer in modern processors
- **rsp** in 64-bit Intel processors



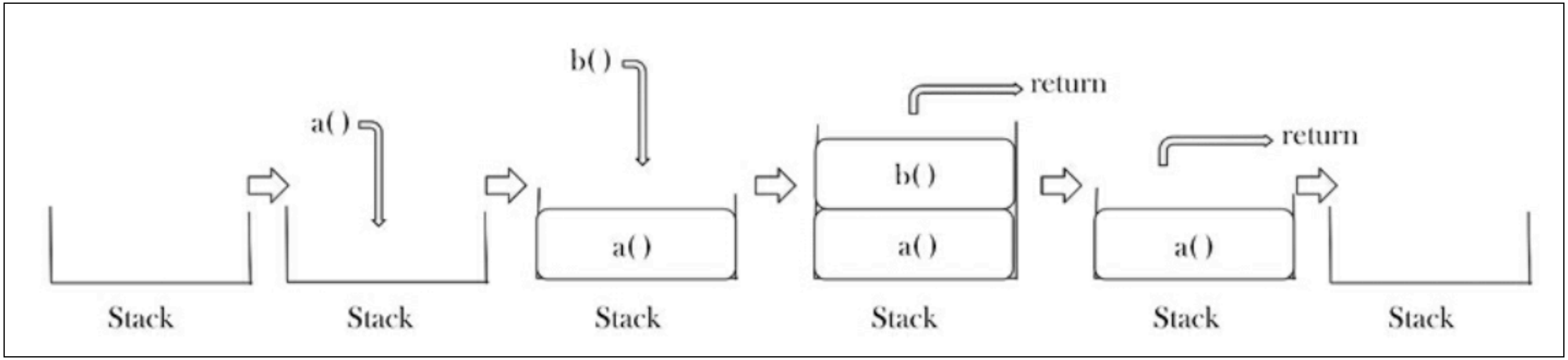
Stack Operations

Stack operation	Size	Contents	Operation results
`stack()`	0	`[]`	Stack object created, which is empty.
`push "egg"`	1	`['egg']`	One item `egg` is added to the stack.
`push "ham"`	2	`['egg', 'ham']`	One more item, `ham`, is added to the stack.
`peek()`	2	`['egg', 'ham']`	The top element, `ham`, is returned.
`pop()`	1	`['egg']`	The `ham` item is popped off and returned. (This item was added last, so it is removed first.)
`pop()`	0	`[]`	The `egg` item is popped off and returned. (This is the first item added, so it is returned last.)

Stack Frames and Return Pointers

- Each function call pushes a new **stack frame** onto the stack
- Stores local variables and the **return pointer**

```python
def b():
    print('b')
def a():
    b()
a()
print("done")
```

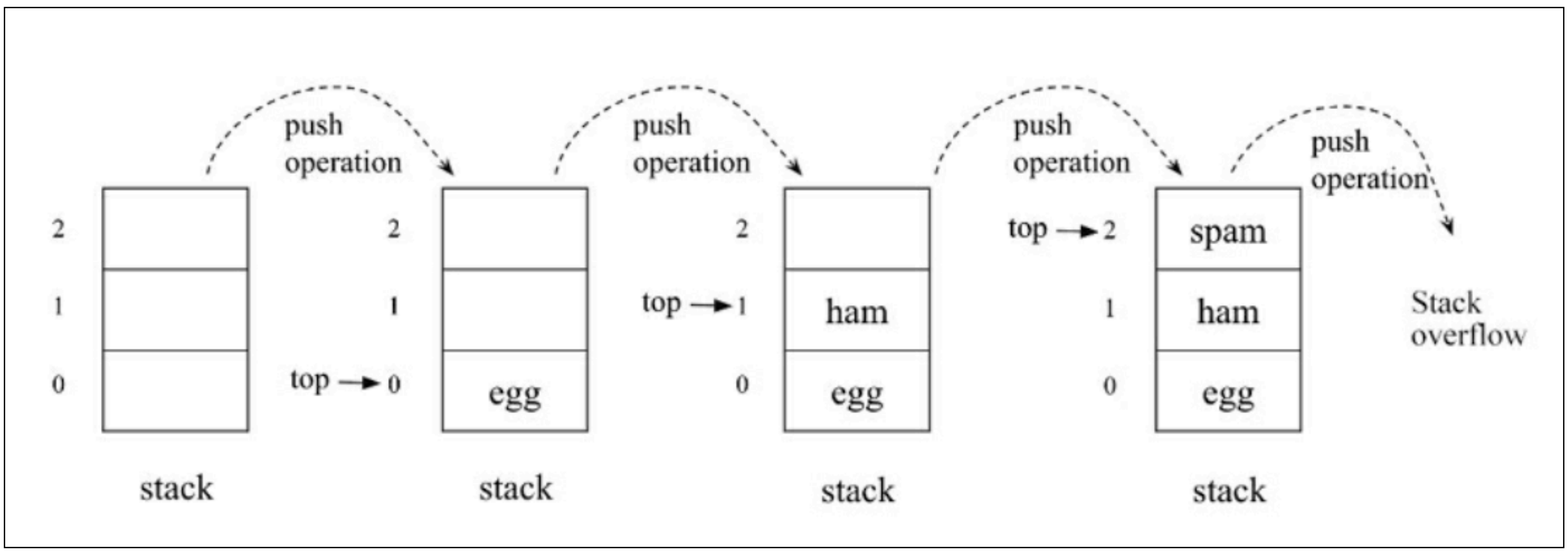


Stack implementation

- With an **array**
  - Fixed length
- With a **list**
  - Variable length

Stack implemented with array

- **push** may fail because the stack is full
- **pop** causes an underflow error when the stack is empty



Python code for push

- Notice how the array is initialized

```
size = 3
data = [0]*(size)   #Initialize the stack
top = -1
def push(x):
     global top
     if top >= size - 1:
           print("Stack Overflow")
     else:
           top = top + 1
           data[top] = x
```

pushing onto the stack

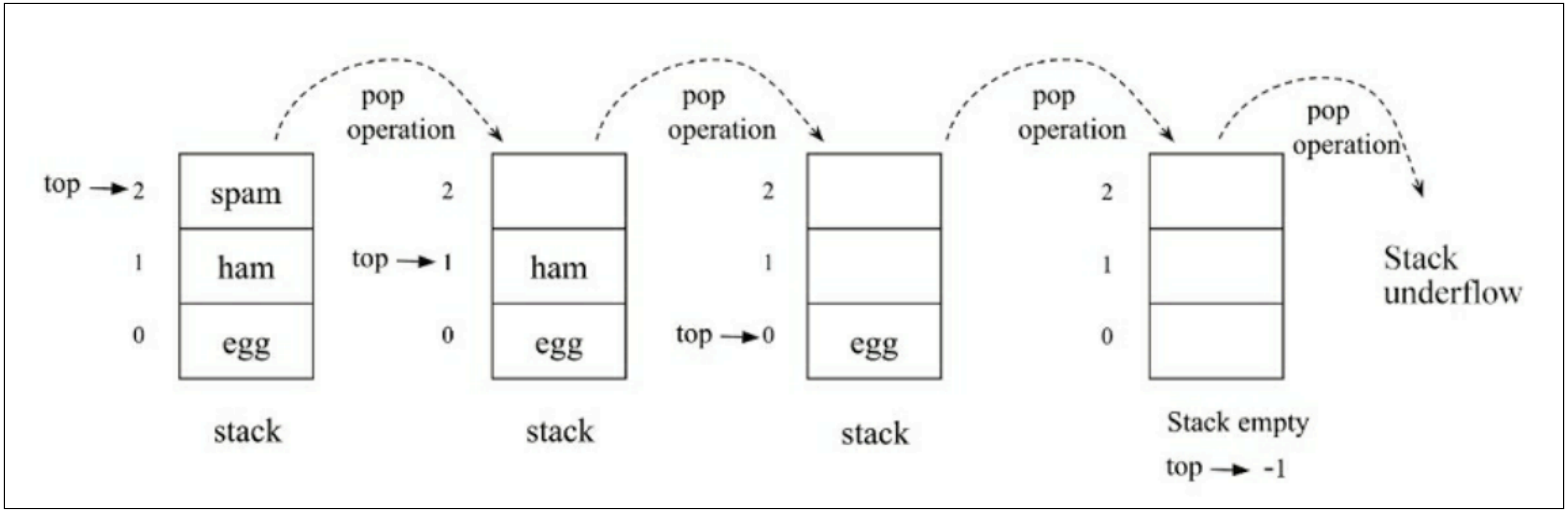
- Notice that the print statement breaks into the internal structure of the stack
- It isn't possible with normal stack operations
  - **push**
  - **pop**
  - **peek**

```
push('egg')
push('ham')
push('spam')
print(data[0 : top + 1] )
push('new')
push('new2')
```

```
['egg', 'ham', 'spam']
Stack Overflow
Stack Overflow
```

pop code

```
def pop():
    global top
    if top == -1:
        print("Stack Underflow")
    else:
        top = top - 1
        data[top] = 0
        return data[top+1]
```



pop underflow

```
print(data[0 : top + 1])
pop()
pop()
pop()
pop()
print(data[0 : top + 1])
```

```
['egg', 'ham', 'spam']
Stack Underflow
[]
```

peek

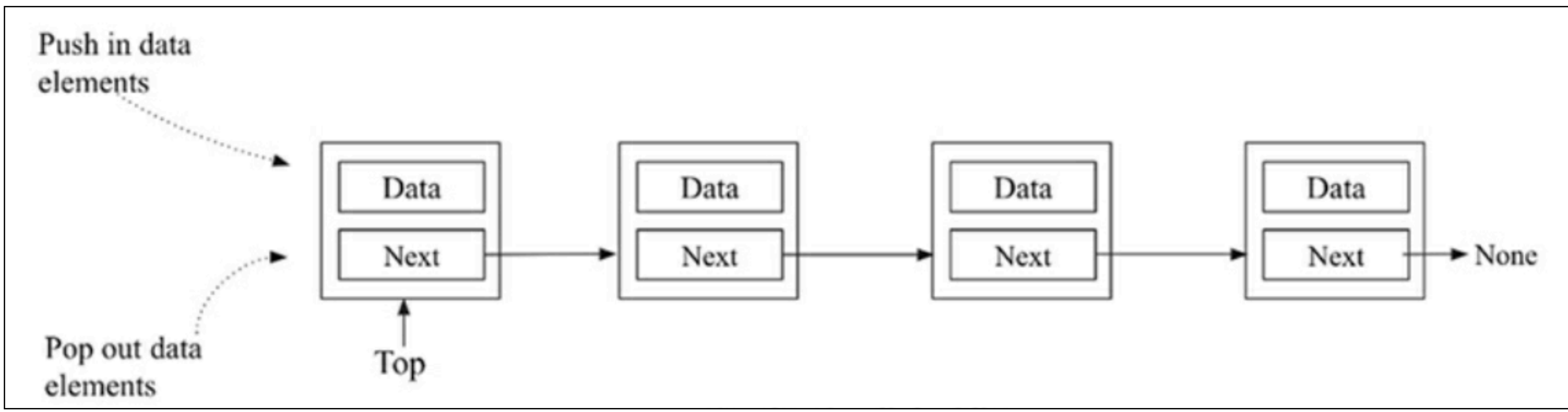
- Fails on empty stack

```
def peek():
    global top
    if top == -1:
        print("Stack is empty")
    else:
        print(data[top])
```

Stack implementation with linked list

```python
class Stack:
    def __init__(self):
        self.top = None
        self.size = 0
```

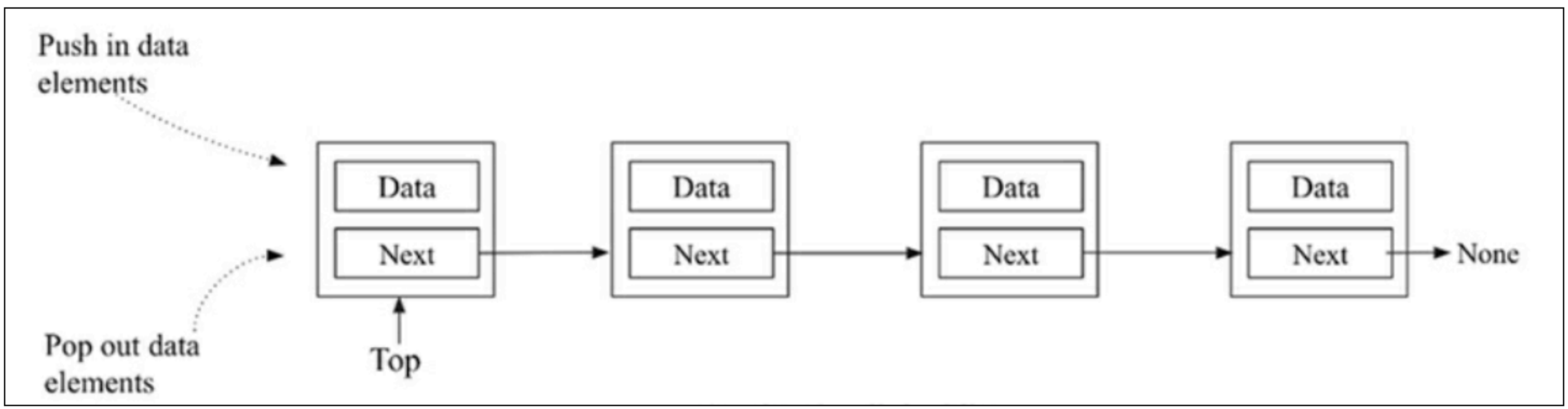
```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```



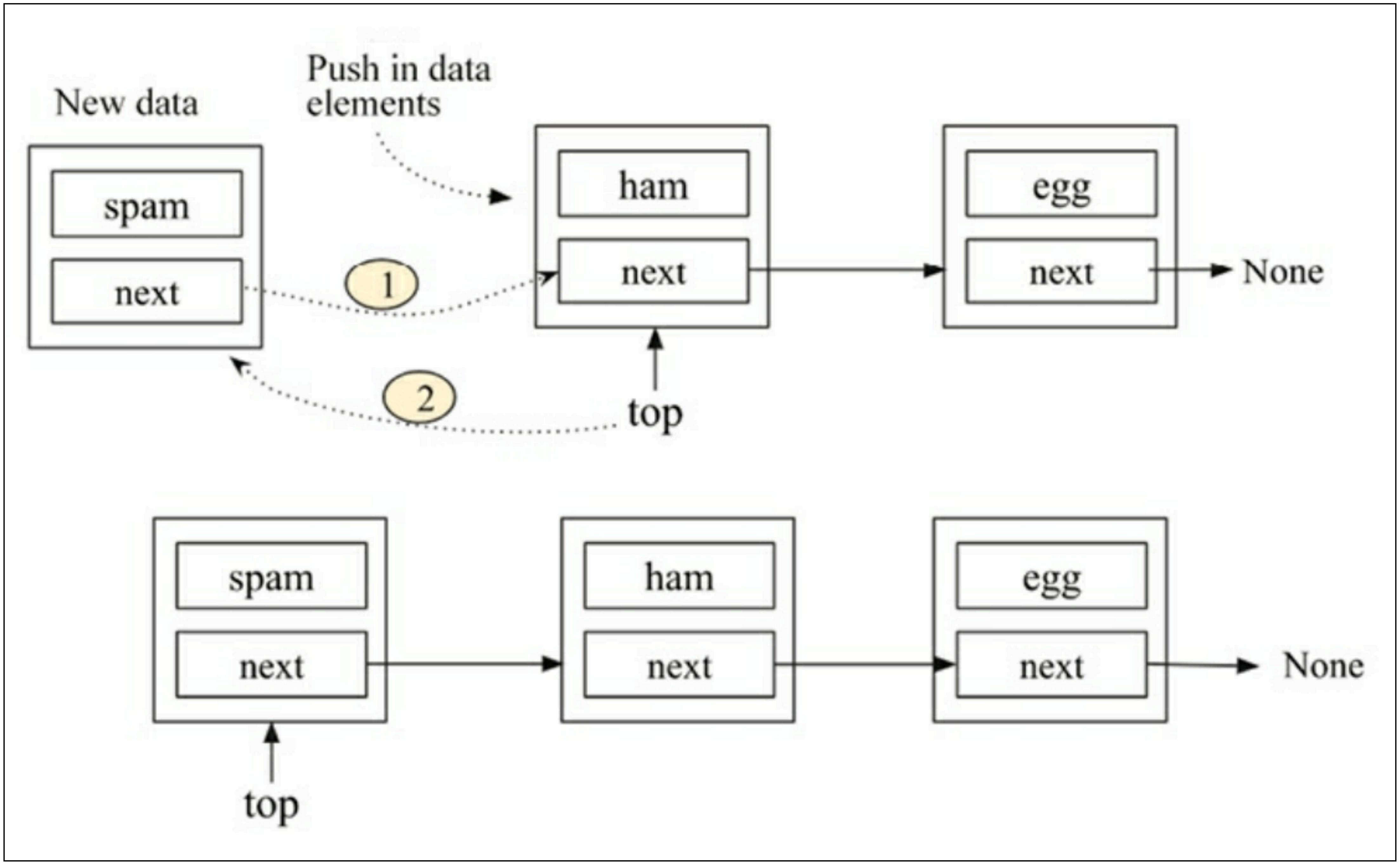
Stack implementation with linked list

- The **node** class isn't quite right for a stack

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

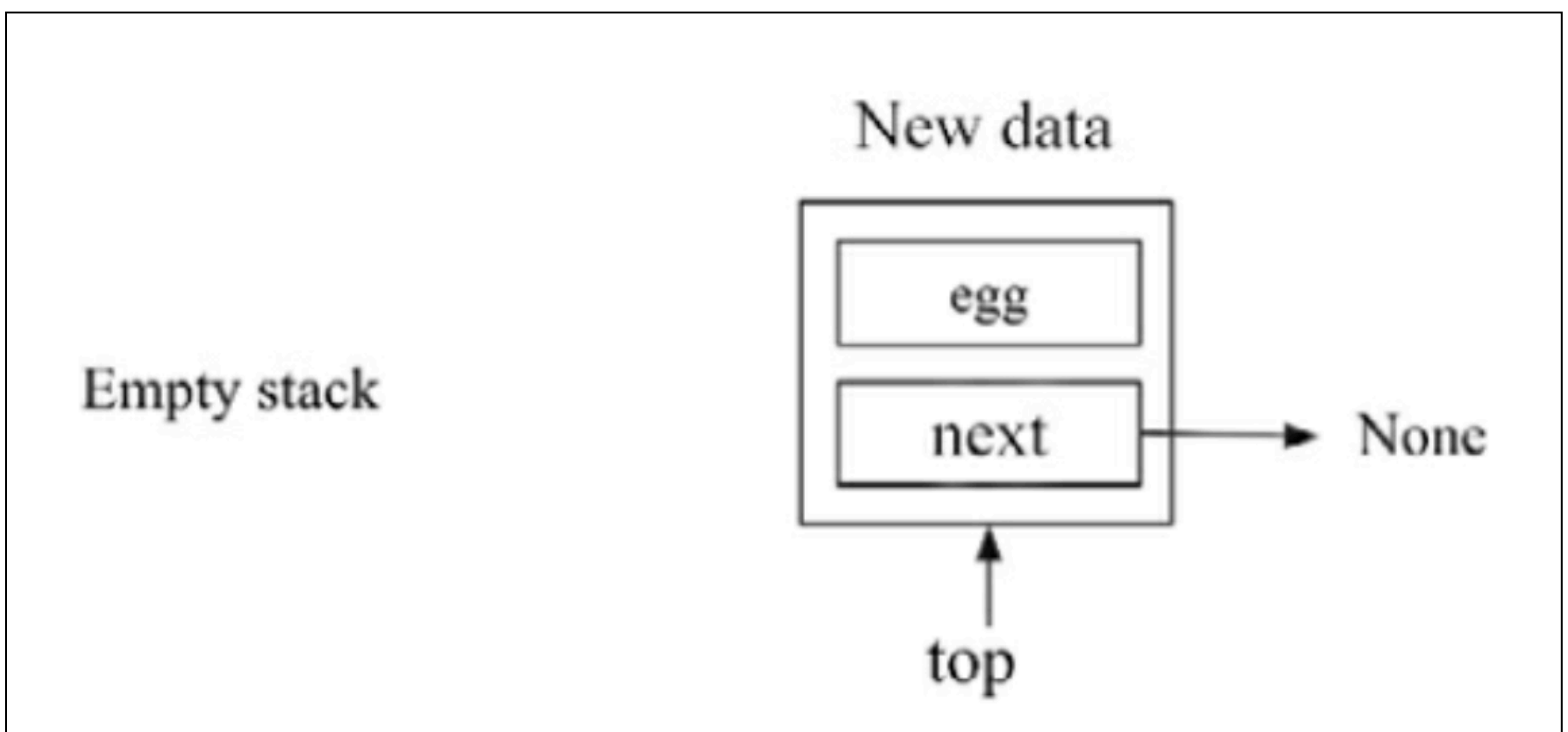


Push operation



Inserting an item into an empty stack

```
def push(self, data):
        # create a new node
        node = Node(data)
        if self.top:
            node.next = self.top
            self.top = node
        else:
            self.top = node
        self.size += 1
```



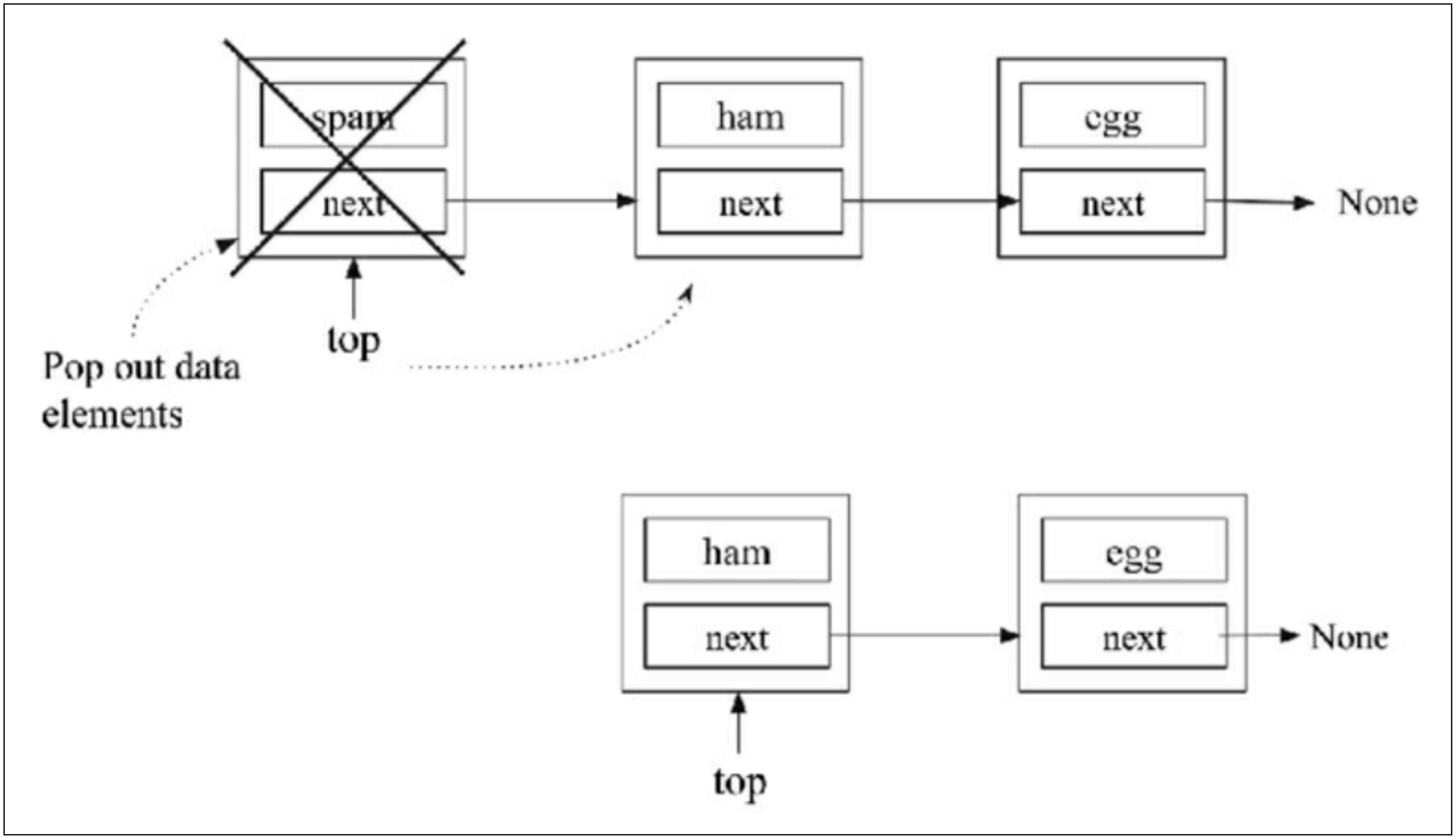
Creating a stack of three elements

- **while** loop again reaches into the Node class, not using stack operations

```
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')
#print the stack elements.
current = words.top
while current:
      print(current.data)
      current = current.next
```

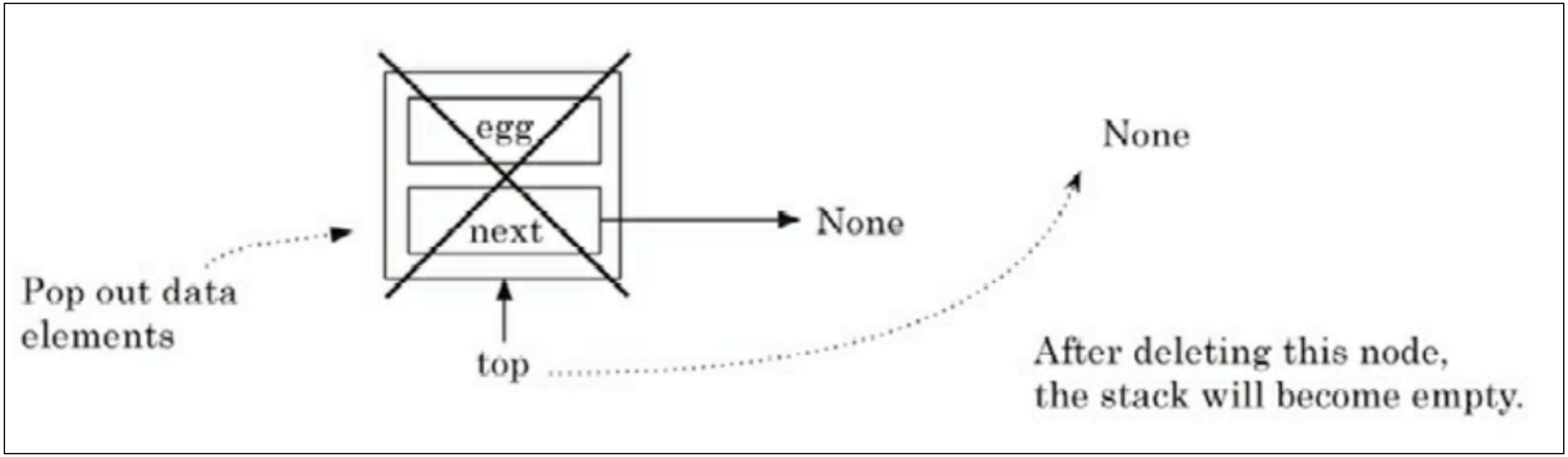
```
spam
ham
egg
```

pop operation



pop on a stack with one element

```
def pop(self):
    if self.top:
        data = self.top.data
        self.size -= 1
        if self.top.next:  #check if there is more than one node.
            self.top = self.top.next
        else:
            self.top = None
        return data
    else:
        print("Stack is empty")
```



Code for Demonstration

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

class Stack:
    def __init__(self):
        self.top = None
        self.size = 0
    def push(self, data):
        # create a new node
        node = Node(data)
        if self.top:
            node.next = self.top
            self.top = node
        else:
            self.top = node
        self.size += 1
    def pop(self):
        if self.top:
            data = self.top.data
            self.size -= 1
            if self.top.next:  #check if there is more than one node.
                self.top = self.top.next
            else:
                self.top = None
            return data
        else:
            print("Stack is empty")
```

```python
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')

print(words.pop())
print(words.pop())
print(words.pop())
```

Demonstration

```python
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')

print(words.pop())
print(words.pop())
print(words.pop())
```

```
spam
ham
egg
```

peek implementation

```
def peek(self):
     if self.top:
          return self.top.data
     else:
         print("Stack is empty")
```

Applications of stacks

- Bracket-matching

```
def check_brackets(expression):
    brackets_stack = Stack()      #The stack class, we defined in previous sec-
tion.
    last = ' '
    for ch in expression:
        if ch in ('{', '[', '('):
            brackets_stack.push(ch)
        if ch in ('}', ']', ')'):
            last = brackets_stack.pop()
            if last == '{' and ch == '}':
                continue
            elif last == '[' and ch == ']':
                continue
            elif last == '(' and ch == ')':
                continue
            else:
                return False
    if brackets_stack.size > 0:
        return False
    else:
        return True
```

Bracket-matching example

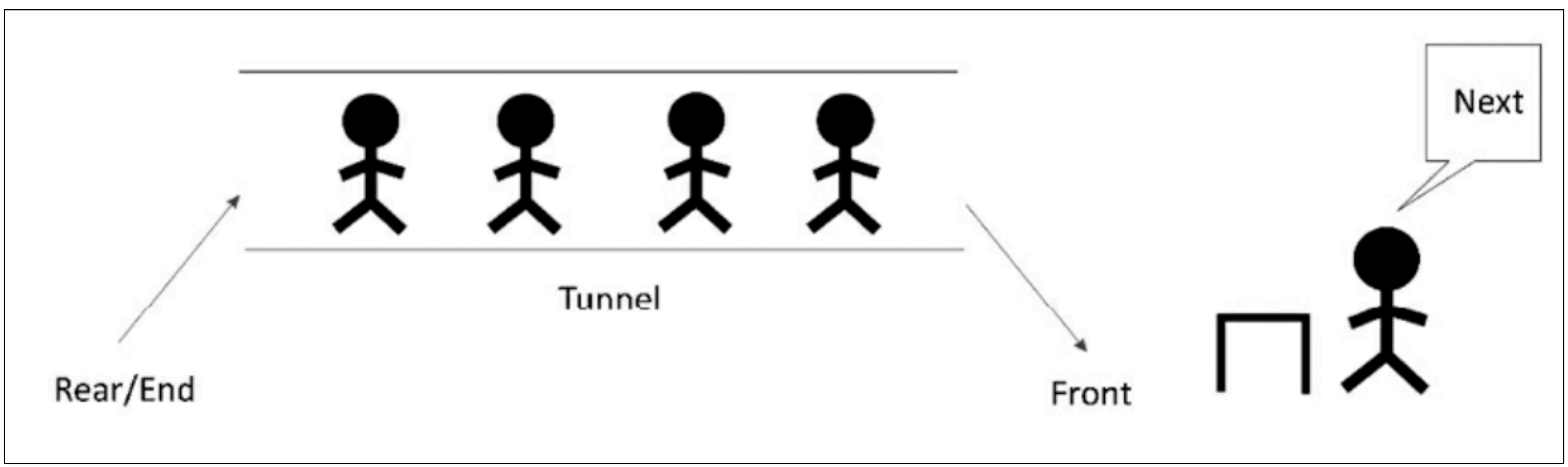
```
sl = (
    "{(foo)(bar)}[hello](((this)is)a)test",
    "{(foo)(bar)}[hello](((this)is)atest",
    "{(foo)(bar)}[hello](((this)is)a)test))"
)
for s in sl:
    m = check_brackets(s)
    print("{}: {}".format(s, m))
```

```
{(foo)(bar)}[hello](((this)is)a)test: True
{(foo)(bar)}[hello](((this)is)atest: False
{(foo)(bar)}[hello](((this)is)a)test)): False
```

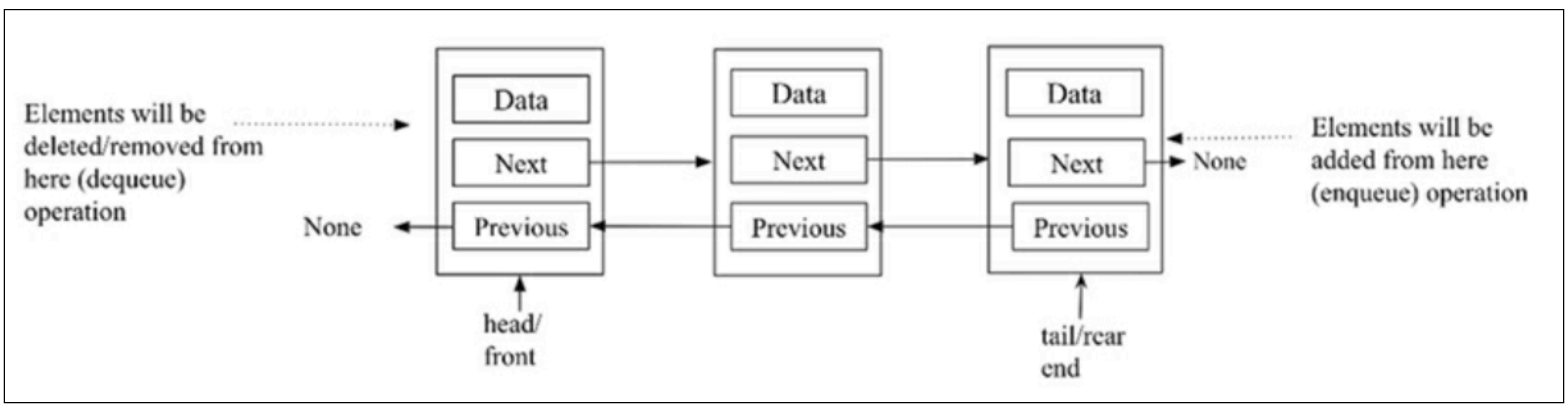
Queues

Queue: First In, First Out (FIFO)

- Data elements can only be inserted at the rear
- Data elements can only be deleted from the front
- Only data elements at the front can be read



enqueue and dequeue operations



Queue operations

Queue operation	**Size**	**Contents**	**Operation results**
`queue()`	**0**	`[]`	**Queue object created, which is empty.**
`enqueue- "packt"`	**1**	`['packt']`	**One item, `packt`, is added to the queue.**
`enqueue "publishing"`	**2**	`[ 'packt', 'publishing']`	**One more item, `publishing`, is added to the queue.**
`Size()`	**2**	`[ 'packt', 'publishing']`	**Return the number of items in the queue, which is 2 in this example.**
`dequeue()`	**1**	`['publishing']`	**The `packt` item is dequeued and returned. (This item was added first, so it is removed first.)**
`dequeue()`	**0**	`[]`	**The `publishing` item is dequeued and returned. (This is the last item added, so it is returned last.)**

Three ways to implement queues

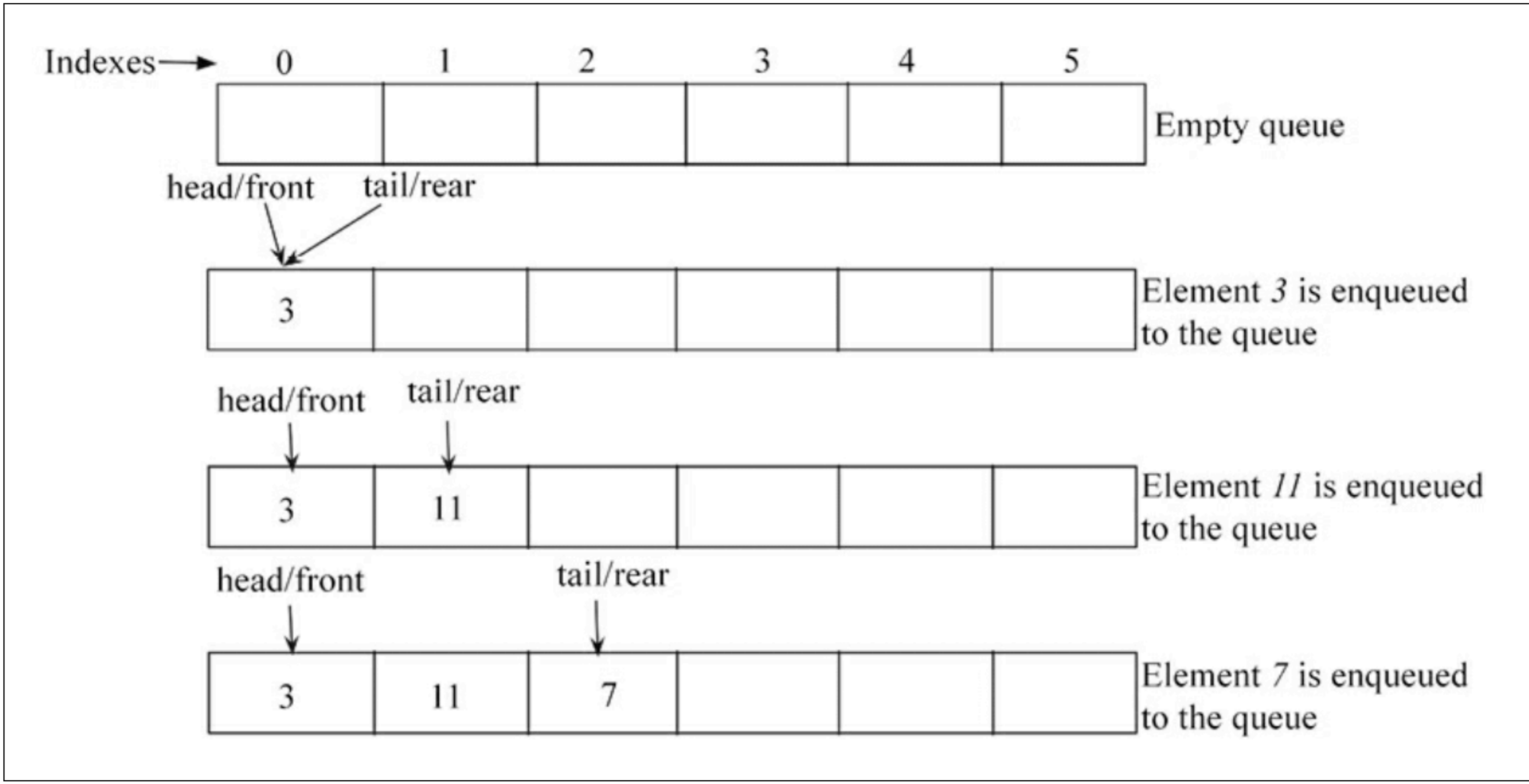
- Python's built-in list
- Stacks
- Node-based linked lists

Python's list-based queues

- Data is stored in a list
  - See **items**

```
class ListQueue:
    def __init__(self):
        self.items = []
        self.front = self.rear = 0
        self.size = 3     # maximum capacity of the queue
```

Enqueue operation



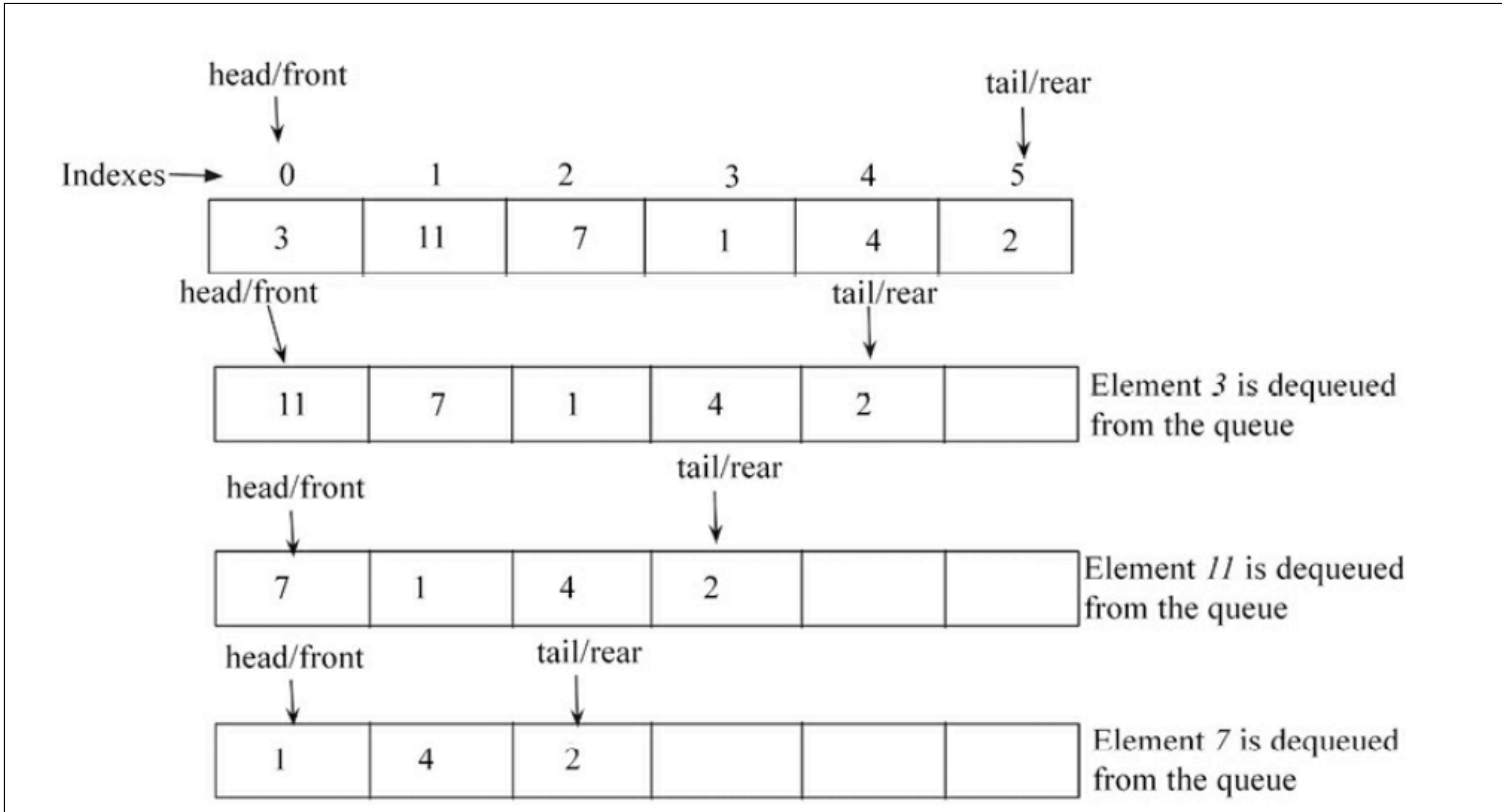
Enqueue code

```python
def enqueue(self, data):
    if self.size == self.rear:
        print("\n Queue is full")
    else:
        self.items.append(data)
        self.rear += 1
```

```python
q= ListQueue()
q.enqueue(20)
q.enqueue(30)
q.enqueue(40)
q.enqueue(50)
print(q.items)
```

```
Queue is full
[20, 30, 40]
```

Dequeue operation



Dequeue code

```
def dequeue(self):
    if self.front == self.rear:
        print("Queue is empty")
    else:
        data = self.items.pop(0)    # delete the item from front end of the queue
        self.rear -= 1
        return data
```

- Python's List class has a **pop** method, which does these two things:
  - Delete last item from the list
  - Returns the deleted item

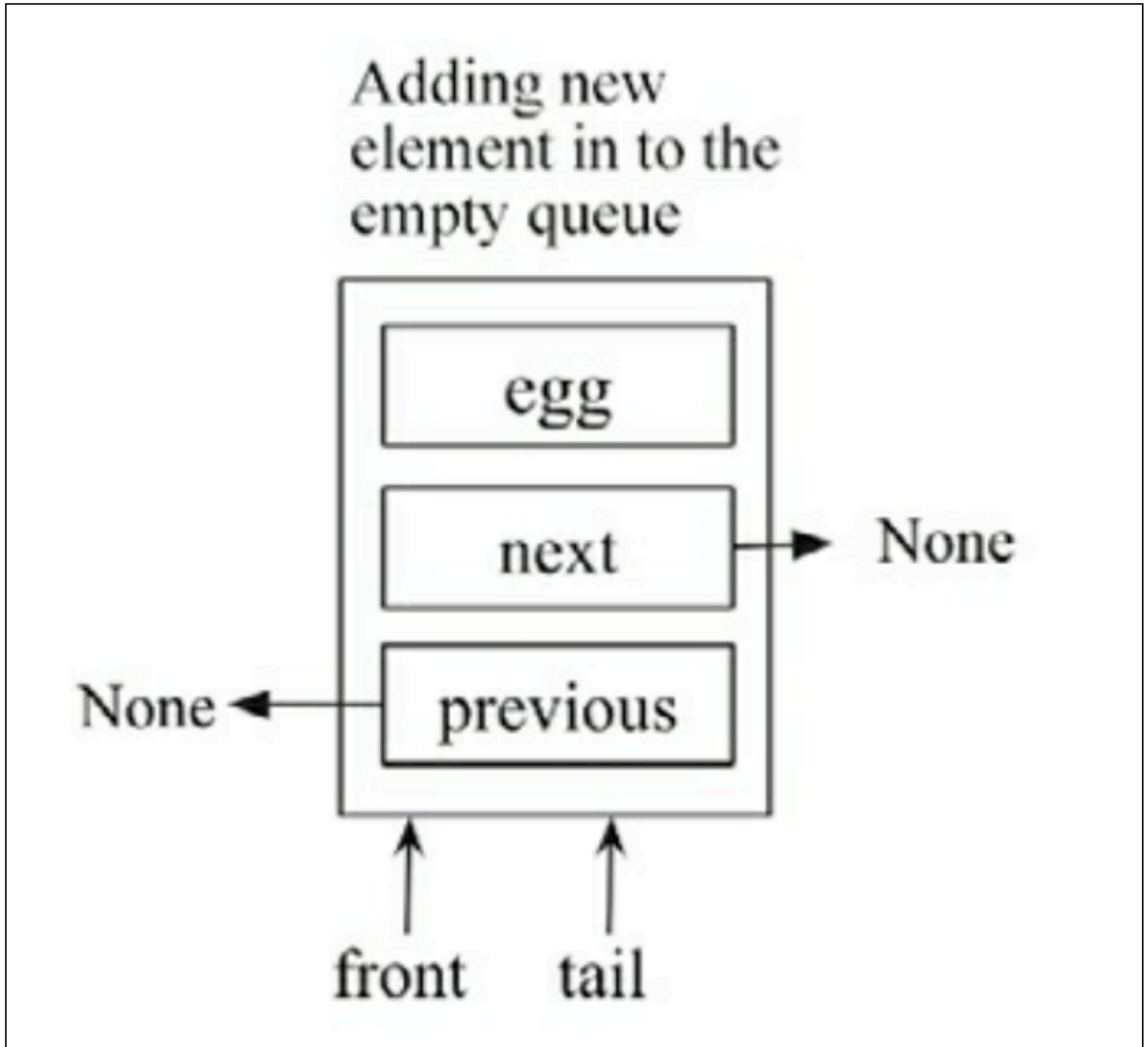
```
data = q.dequeue()
print(data)
print(q.items)
```

```
20
[30, 40]
```

Linked list based queues

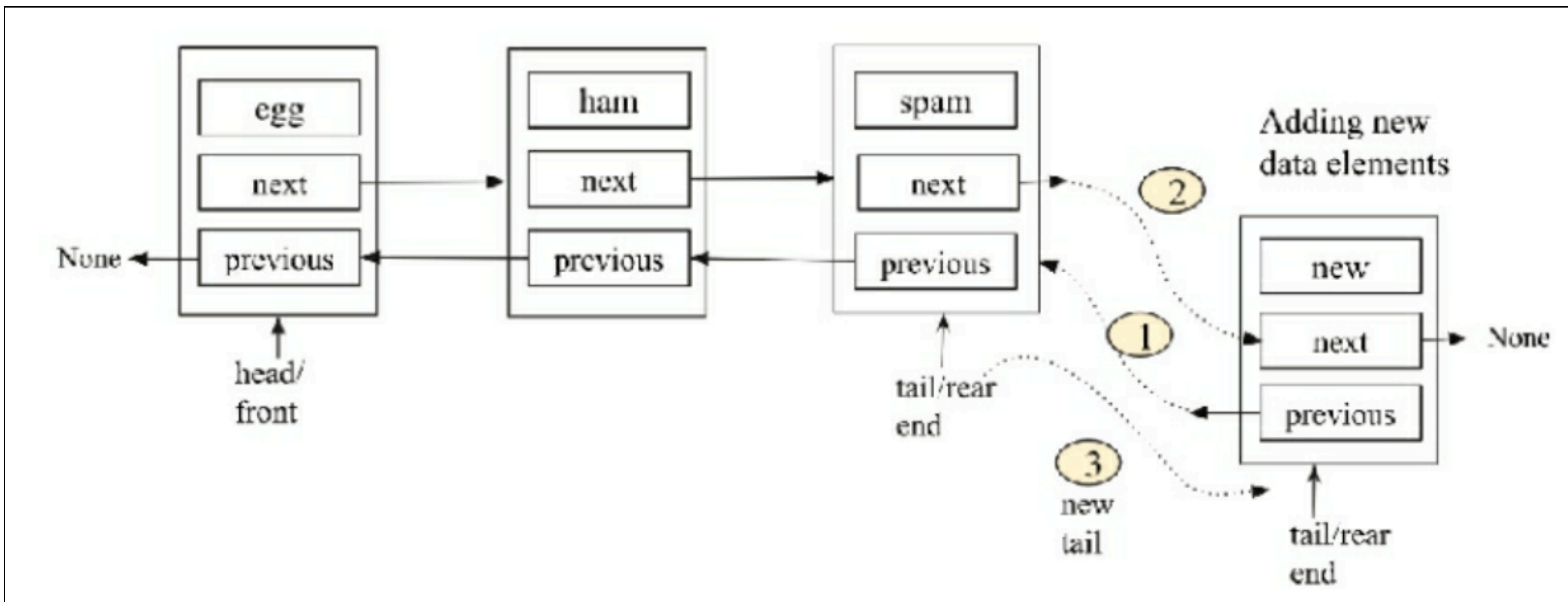
```
class Node(object):
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
```

Enqueue operation



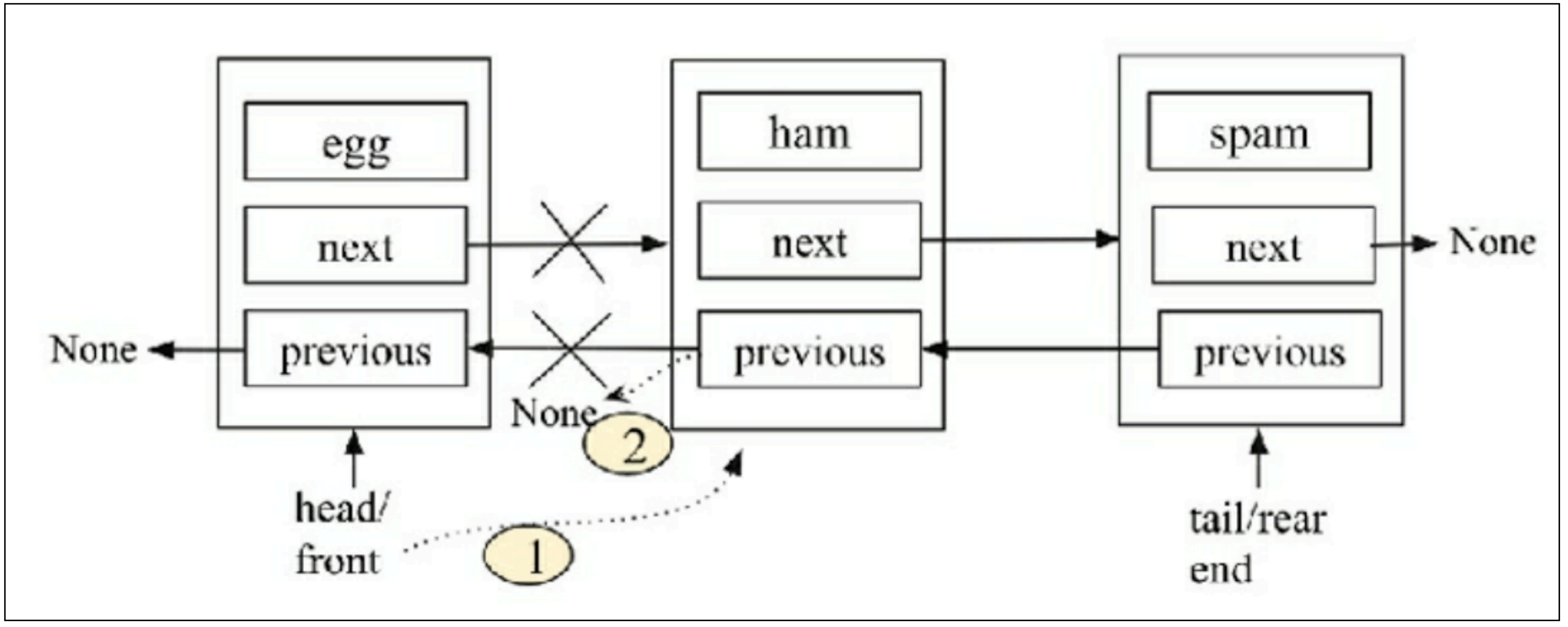
Enqueue operation

```
def enqueue(self, data):
    new_node = Node(data, None, None)
    if self.head == None:
        self.head = new_node
        self.tail = self.head
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.count += 1
```



Dequeue operation

```
def dequeue(self):
    if self.count == 1:
        self.count -= 1
        self.head = None
        self.tail = None
    elif self.count > 1:
        self.head = self.head.next
        self.head.prev = None
    elif self.count <1:
        print("Queue is empty")
    self.count -= 1
```

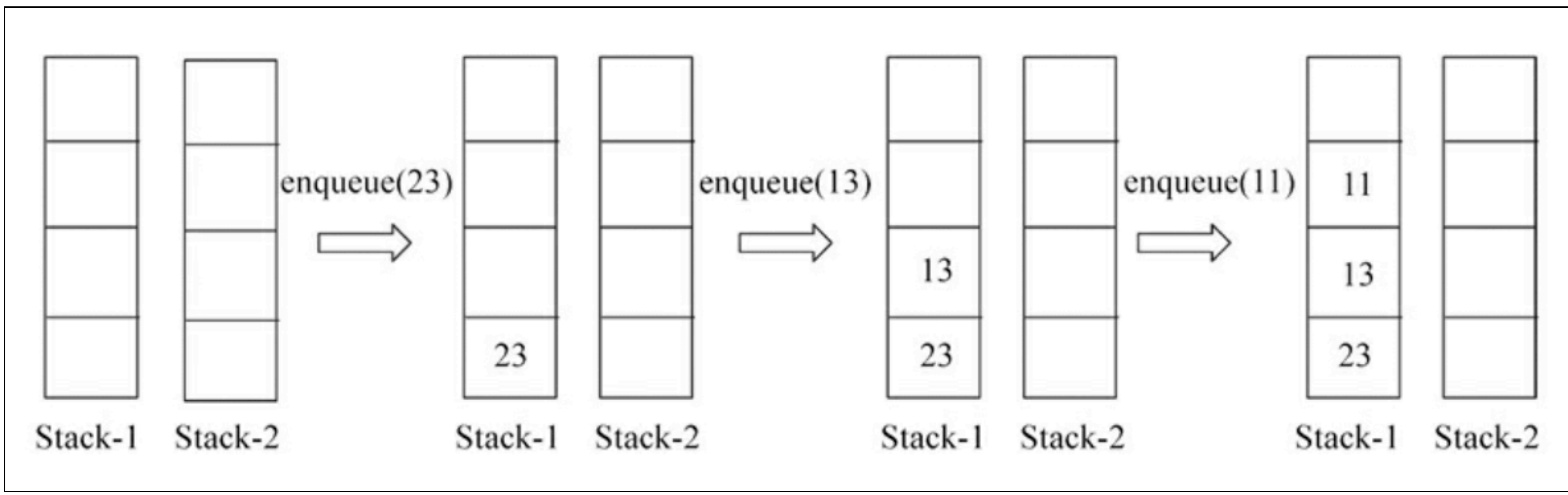


Stack-based queues

- Two approaches
  - When the dequeue operation is costly
  - When the enqueue operation is costly

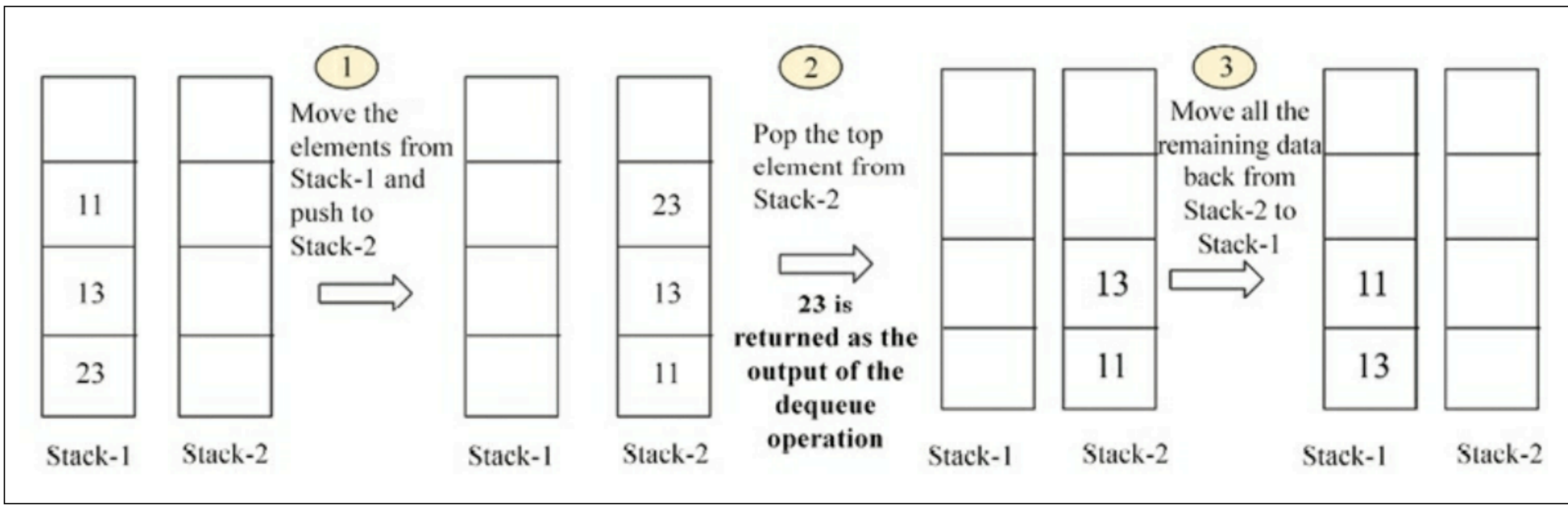
Approach 1:
When the dequeue operation is costly

- Use two stacks
- **enqueue** uses **push** to add items to the first stack



Dequeue operation

- **pop** elements off stack-1 and **push** them onto stack-2
- **pop** top element off stack-2, return this value
- **pop** remaining elements off stack-2 and **push** them onto stack-1

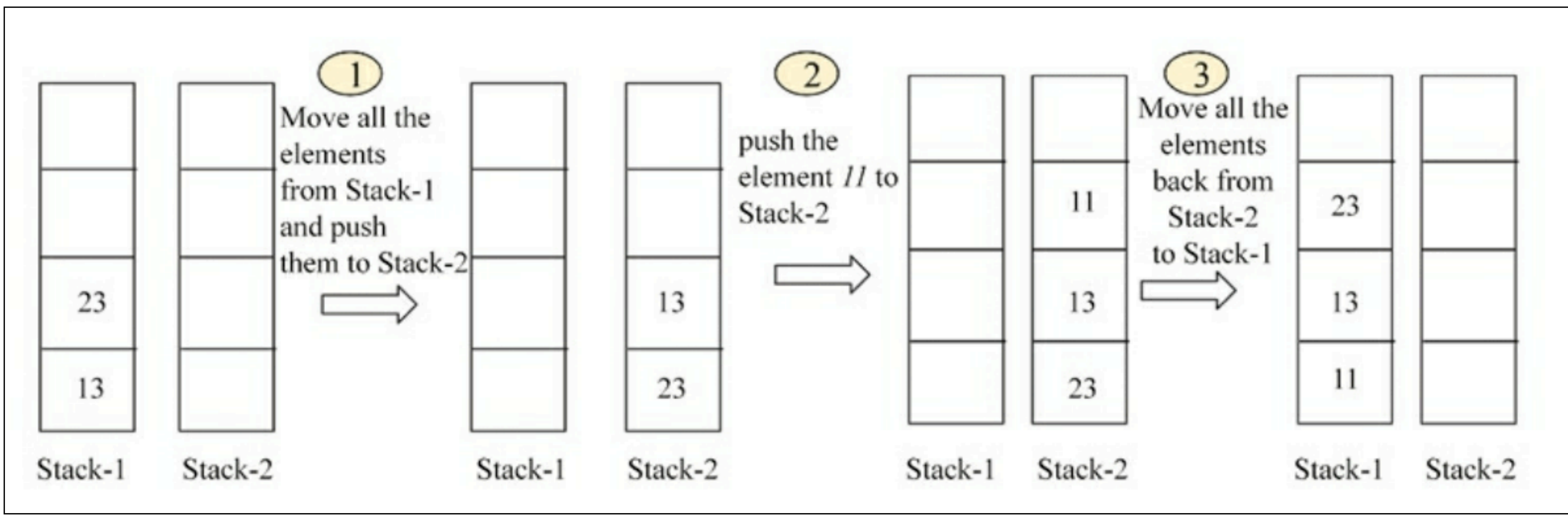


Approach 1:
When the dequeue operation is costly

- Time complexity:
  - **enqueue** is O(***1***)
    - Because any element can be added directly to the first stack
  - **dequeue** is O(***n***)
    - Because all items were transferred from stack-1 to stack-2, and back

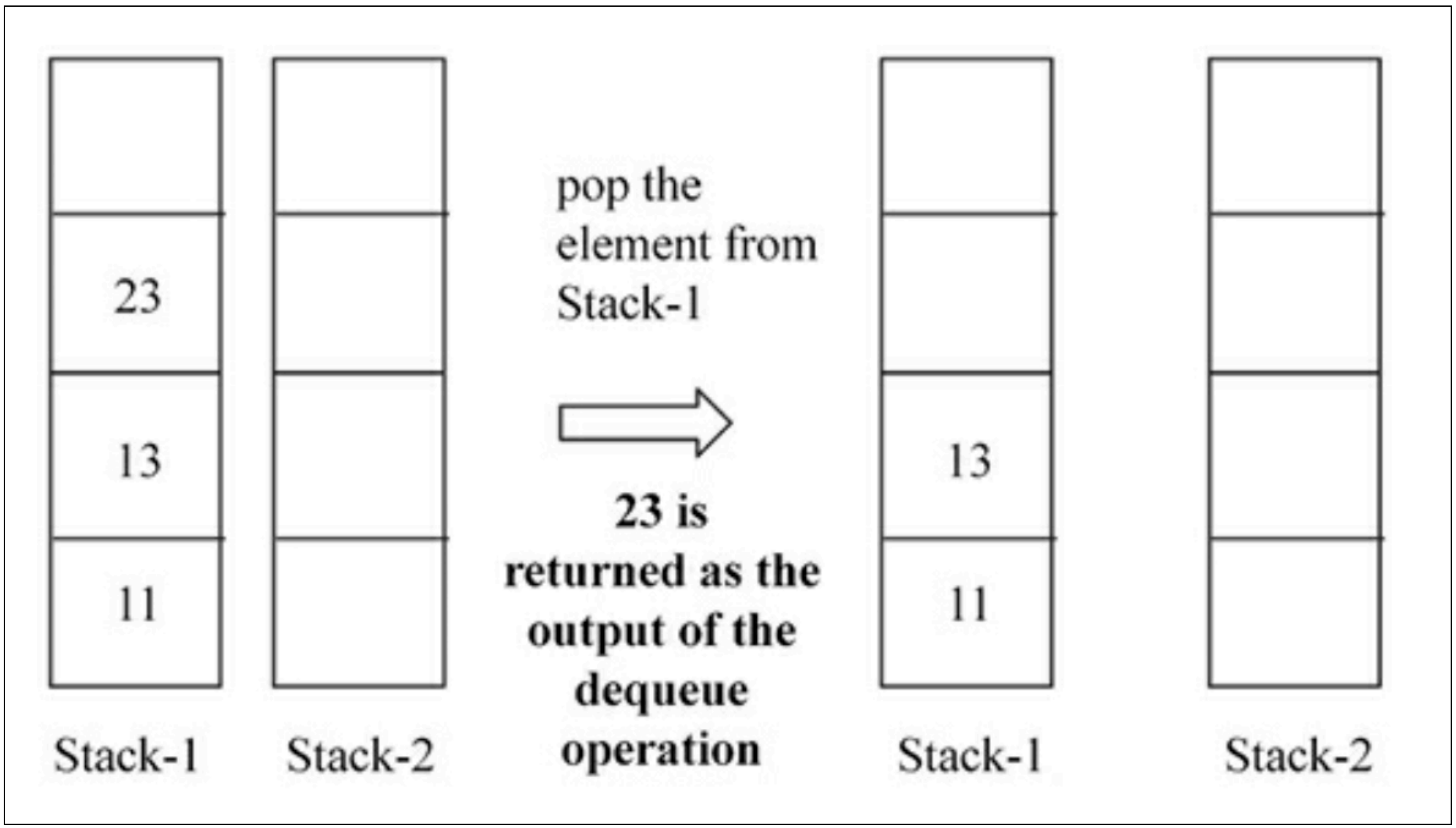
Approach 2:
When the enqueue operation is costly

- Use two stacks
- **enqueue** moves data from stack-1 to stack-2 and back
- Complexity O(***n***)



Dequeue operation

- **pop** from stack-1, return that value
- Complexity O(*1*)



Applications of queues

- Printer queue
- Music playlist

Kahoot!

Ch 5