Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 24
Linked Stacks

L24_Linked_Stacks

– FIT 1008

Introduction to Computer Science

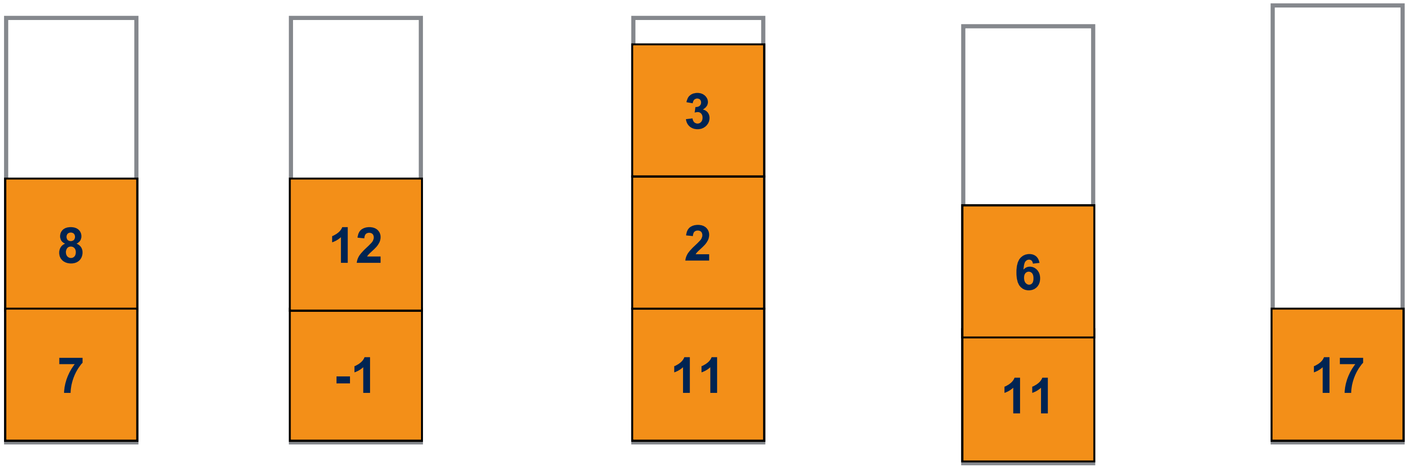
Objectives for these this lecture

- To understand:
  - The concept of **linked data structures**
  - Their use in **implementing stacks**

- To be able to:
  - Implement, use and modify linked stacks
  - Decide when it is appropriate to use them (rather than arrays)

7 8 - 12 + 2 3 * +

↑



Reverse Polish Notation

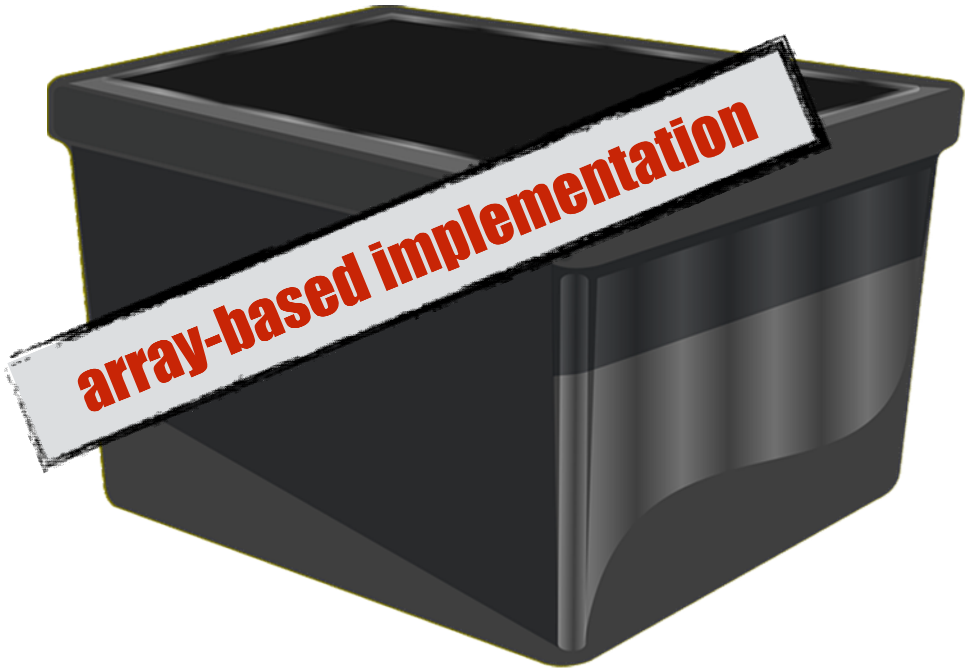
17

Where are we at?

- Implemented container ADT using arrays
- Know about Linked Structures
- Have implemented Nodes

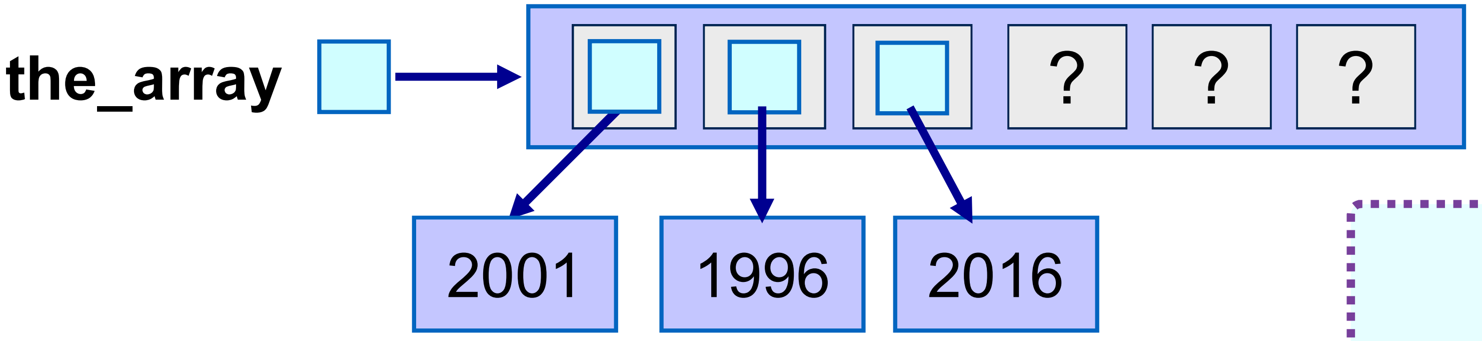
Container ADTs

- **Stores** and **removes** items **independent of contents.**
- **Examples** include:
  - List ADT ☑
  - Stack ADT ☑
  - Queue ADT. ☑
- Core **operations**:
  - ○ add item
  - ○ remove item



Array implementation

- Array **characteristics**:
  - Fixed size
  - Data items are stored sequentially
  - Each item occupies exactly the same amount of space

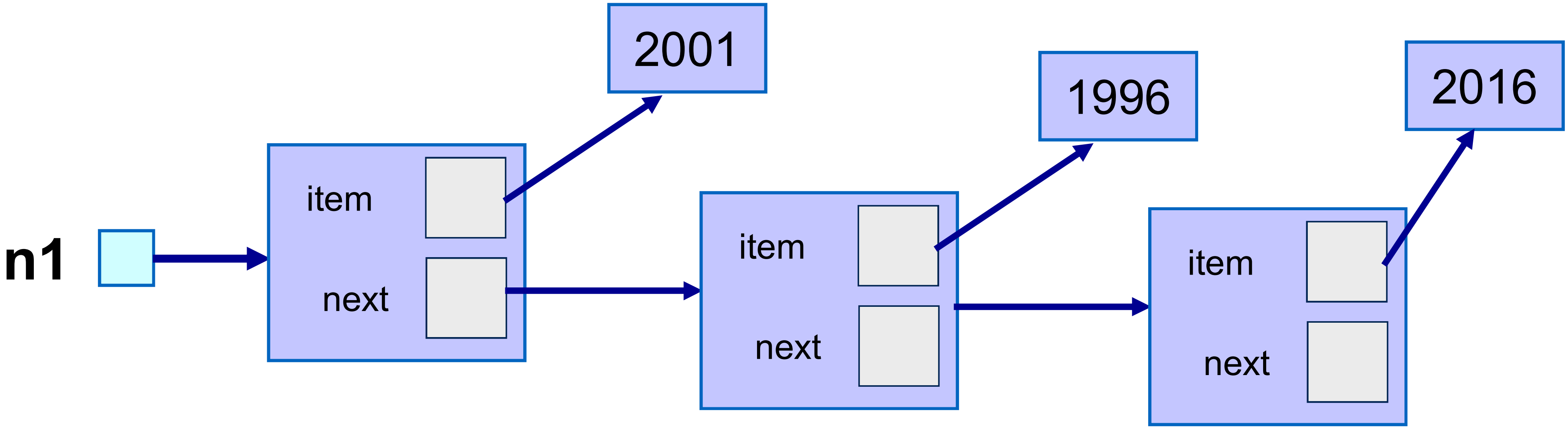


- Main **advantages**:
  - Very **fast** access O(1)
  - Very **compact** representation if the array is full

Python lists: array growth pattern is 0, 4, 8, 16, 25, 35, 46, 58, 72, 88,…

- Main **disadvantages**:
  - Non-resizable: maximum size specified on creation
  - Changing size is costly: **create a new array + copy all items**
  - Slow operations if shuffling elements is required

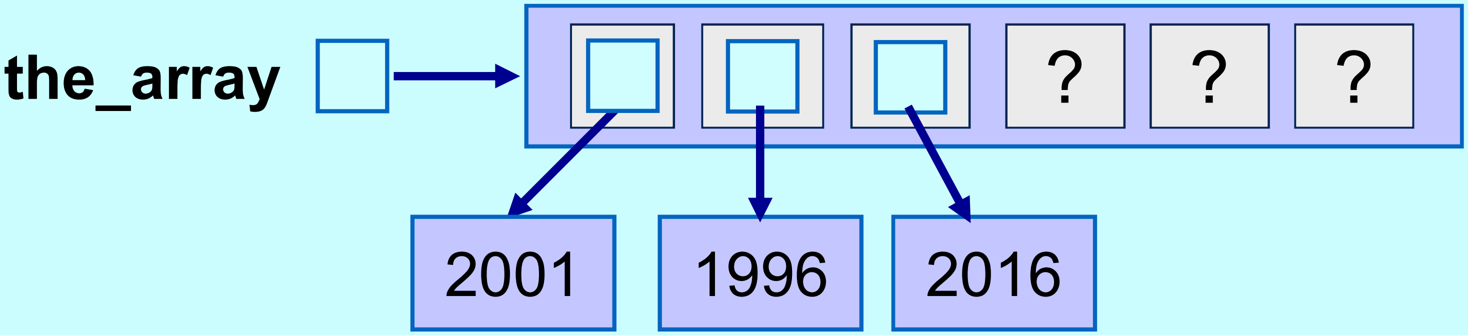
Linked Data Structures



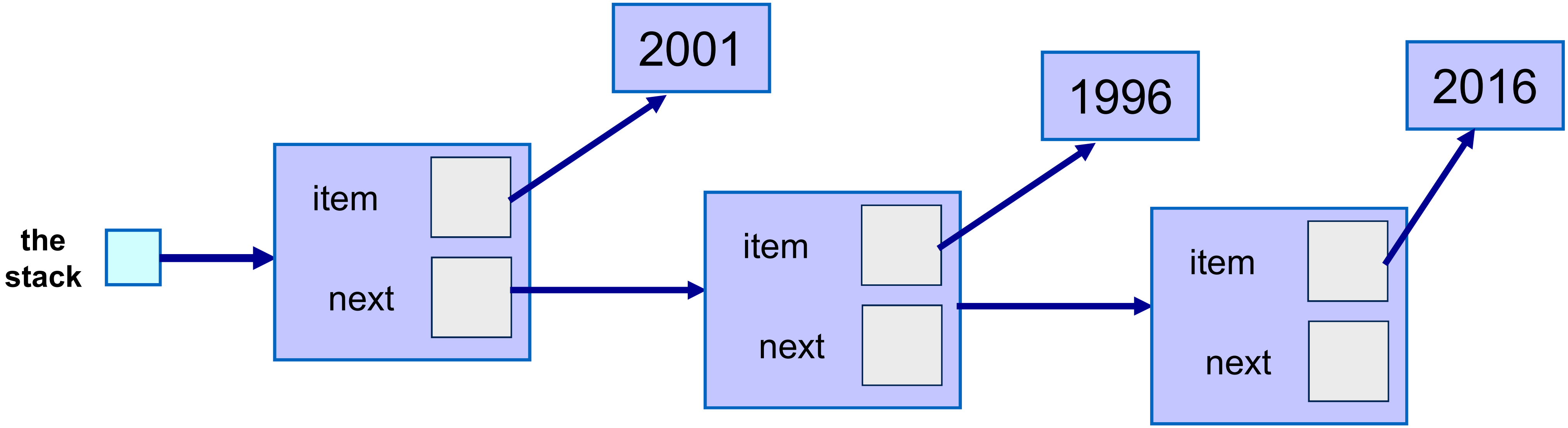
- Collection of nodes

- Each node contains:
  - One or more **data items**
  - One or more **links to other nodes**

Array-based Data Structures:



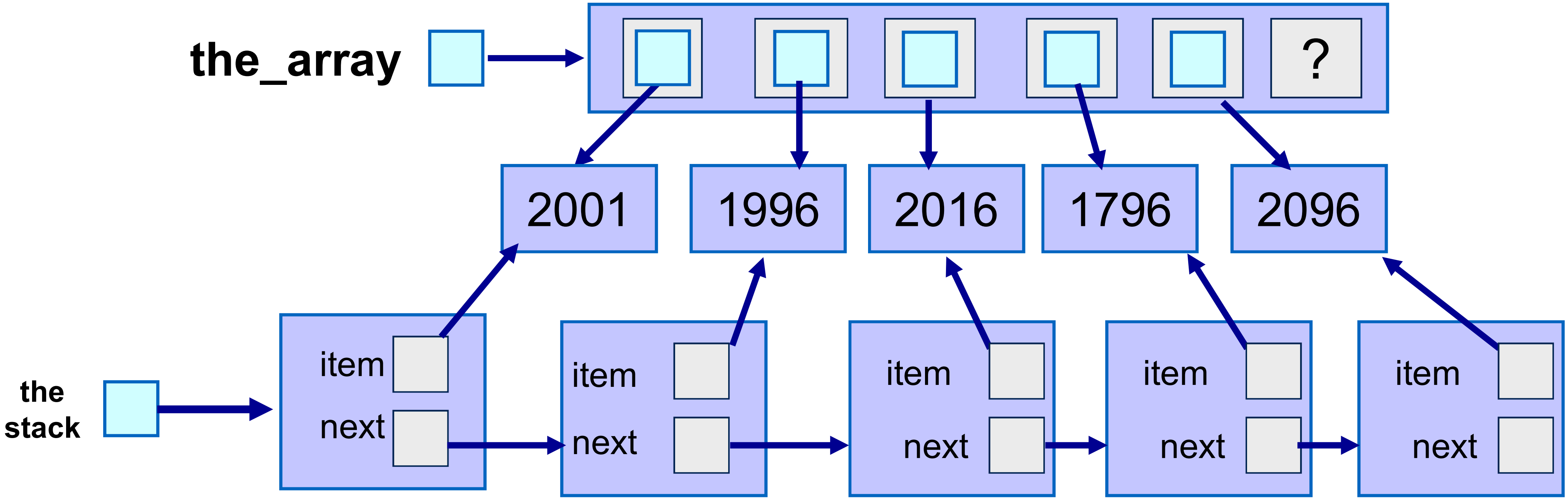
Linked Data Structures:



Linked Data Structures: **Advantages**

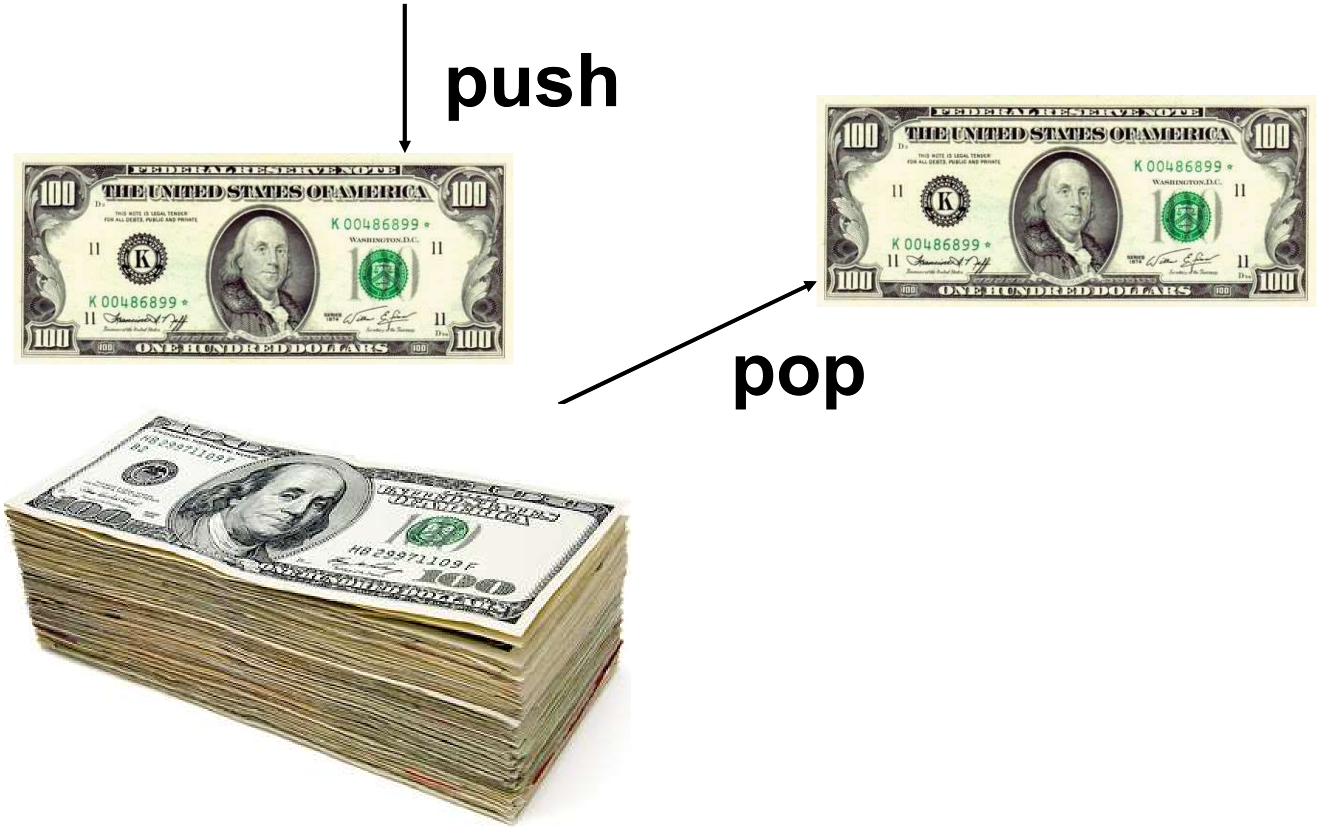
- **Fast** insertions and deletions of items
  (no need for reshuffling)
- Easily **resizable**: just create/delete node
- Never full (only if no more memory left)
- Less memory used than an array if the array-based implementation is relatively empty

Linked Data Structures: Disadvantages



- More **memory** used than an array if the array is relatively full (Reason: every data item has an associated link)

- For some data types certain operations are more time consuming (e.g., no random access)



push

pop

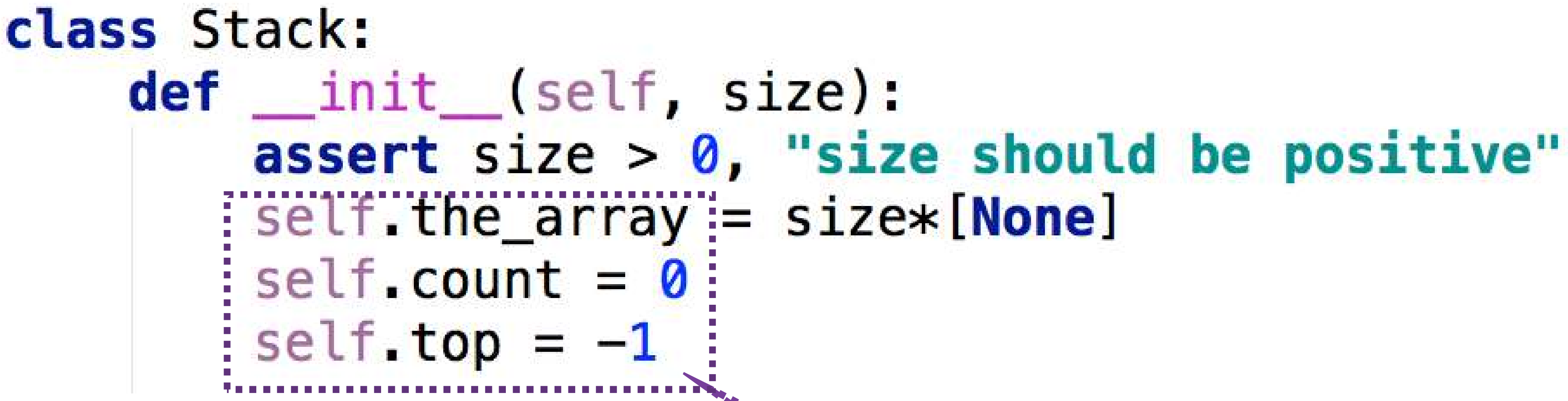
Stack Data Type

- Follows a **LIFO model**

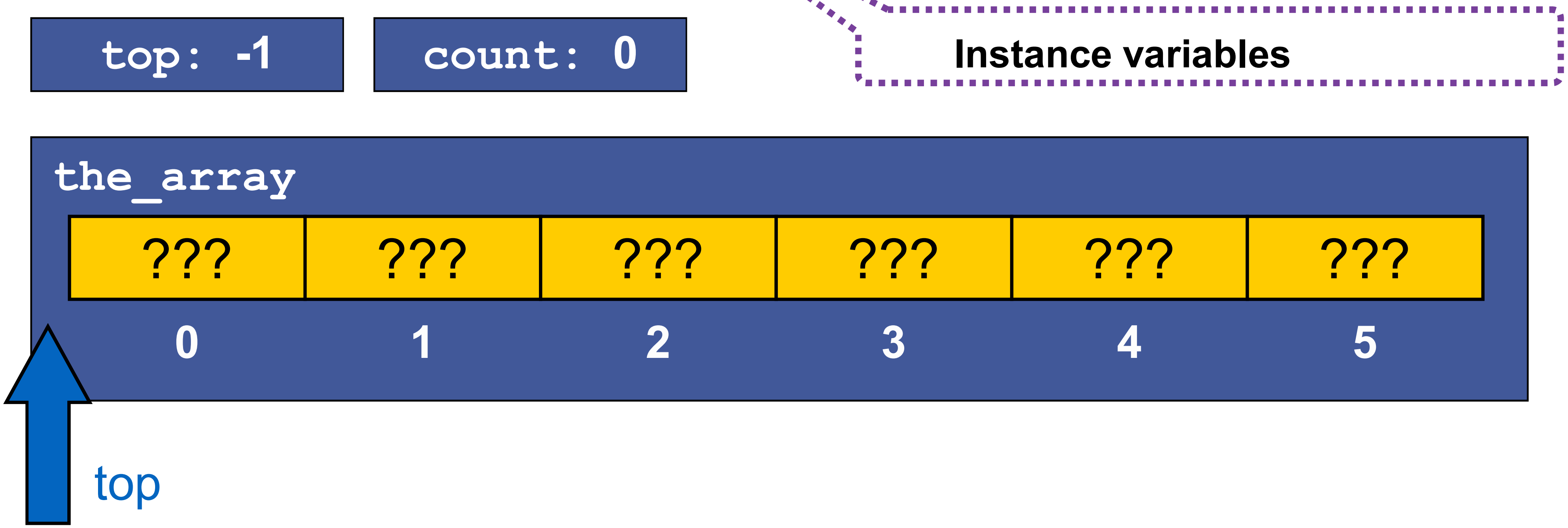
- Its **operations** (interface) are :
  - **Create** a stack (Stack)
  - Add an item to the top (**push**)
  - Take an item off the top (**pop**)
  - Look at the item on top, don’t alter the stack (top/**peek**)
  - Is the stack **empty**?
  - Is the stack **full**?
  - Empty the stack (**reset**)

Remember: it only provides access to the element at the top of the stack (last element added)

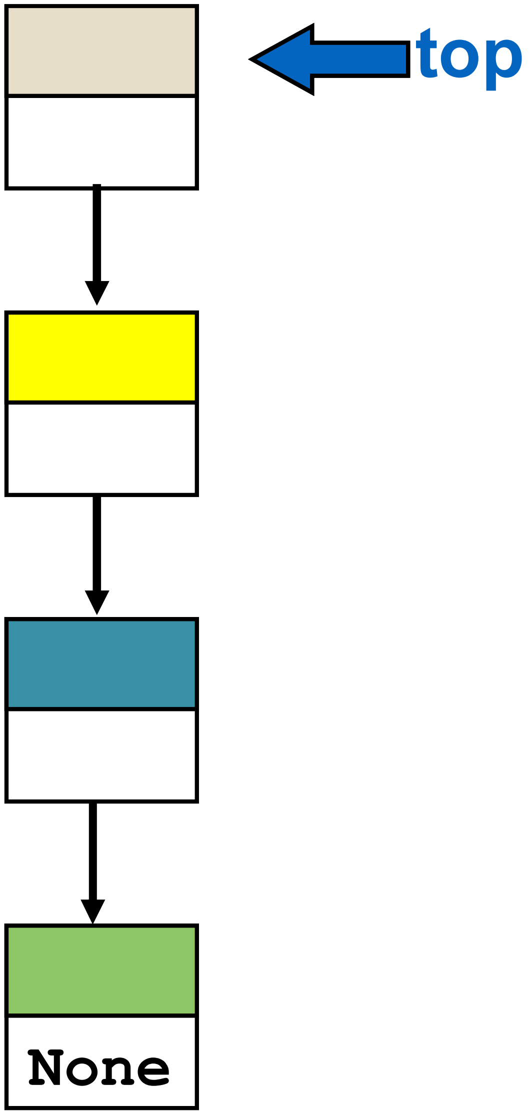
Stack Data Type



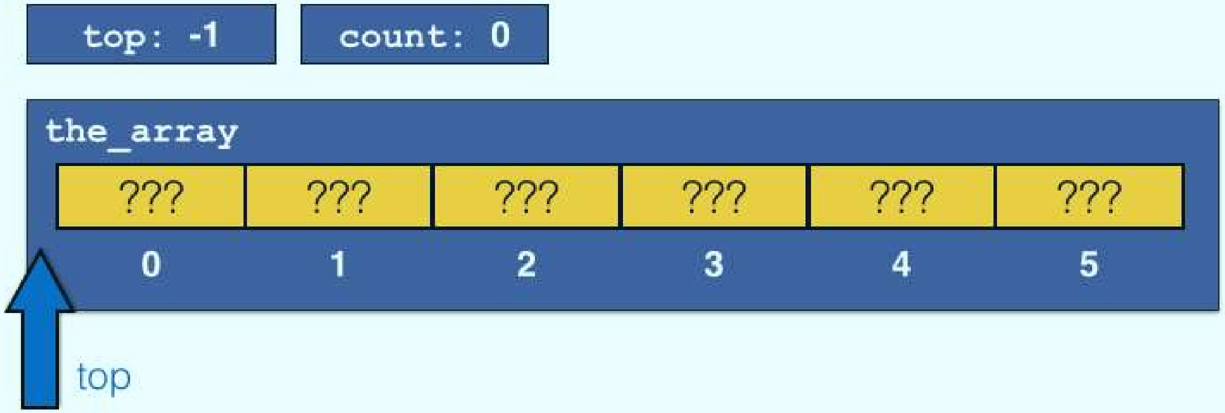
```python
class Stack:
    def __init__(self, size):
        assert size > 0, "size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.top = -1
```



Linked Stack Implementation



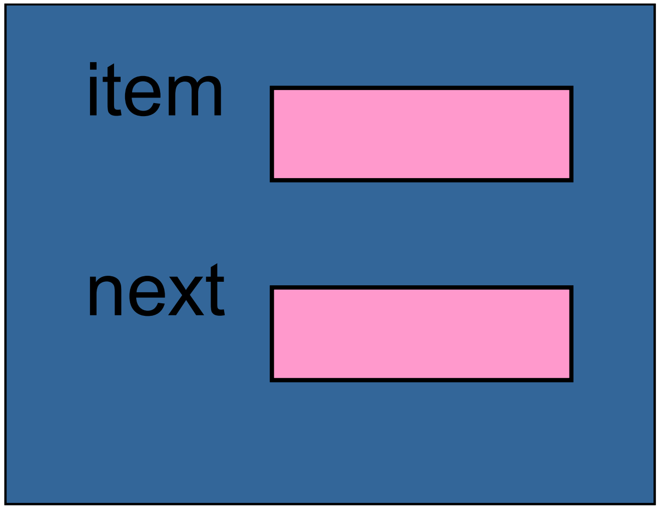
Array Stack Implementation

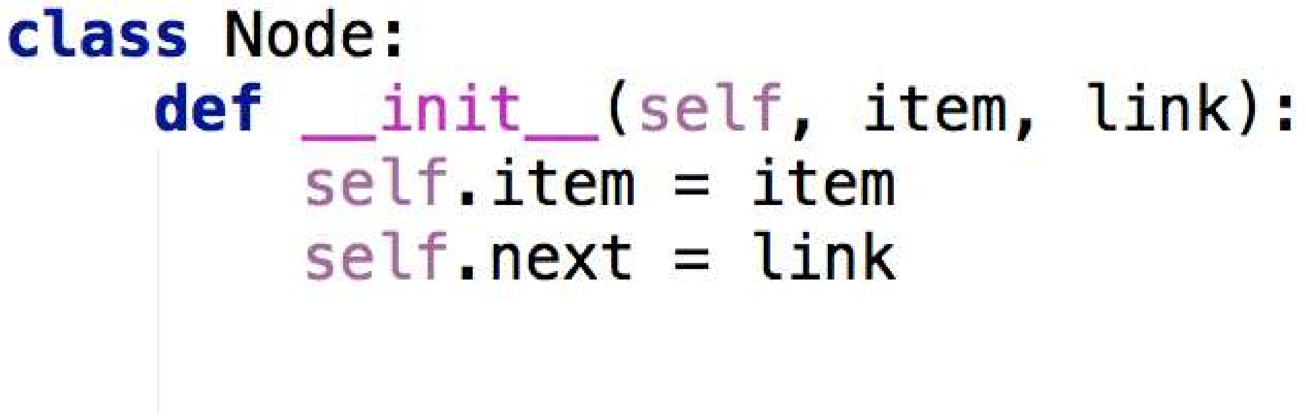


What do we need for a linked implementation?

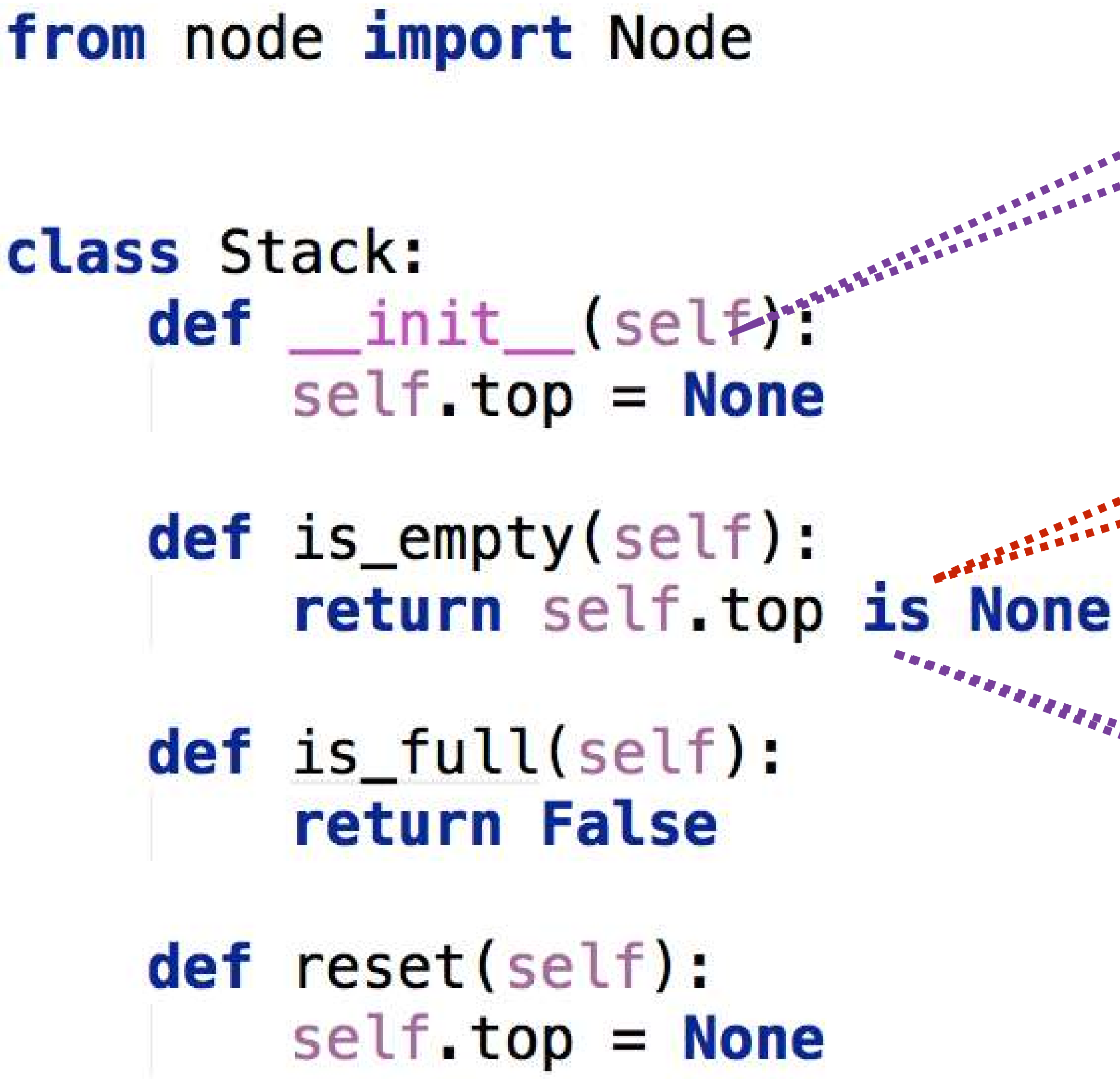
Nodes!

Node





```
class Node:
    def __init__(self, item, link):
        self.item = item
        self.next = link
```



```python
from node import Node


class Stack:
    def __init__(self):
        self.top = None

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

    def is_full(self):
        return False

    def reset(self):
        self.top = None
```

No need for size when initialising the object

self.top == **None** ?
== can be overloaded
implementing
__eq__(self, rhs)

True if pointing to the same object.

Push: algorithm

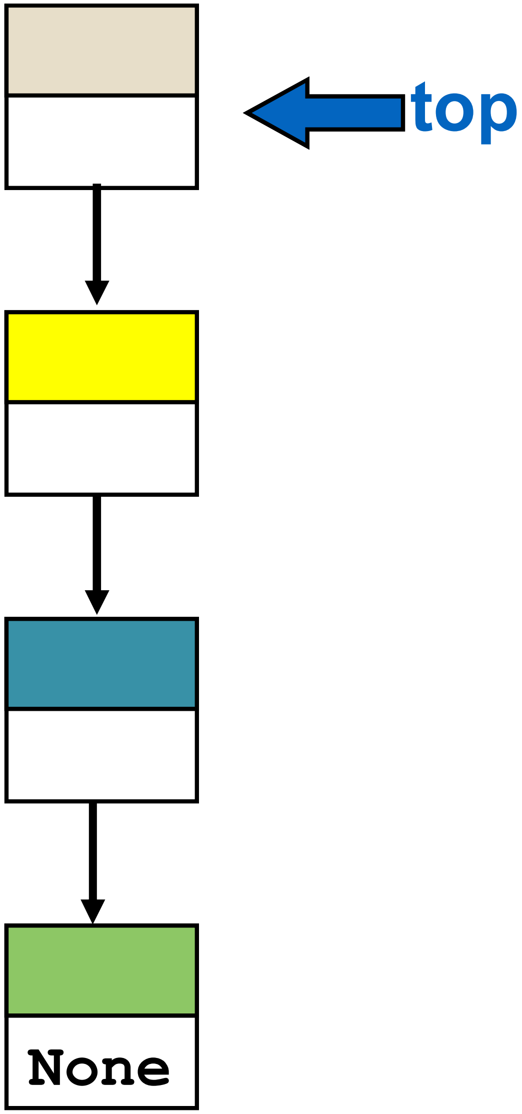
Array implementation:

- If the array is full raise exception
- Else
  - Add item in the position marked by top
  - Increase top

No need for is_full check.
If no more memory can be allocated the system will raise an exception.

Linked implementation:

- Create a **new node** that contains the new item and is linked to the current top
- Make the **new node** the **new top**



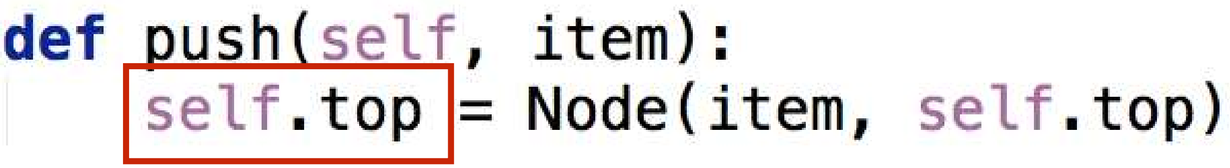
Create a new node with the new item.

linked to the current top 🟫

Make the new node the new **top**

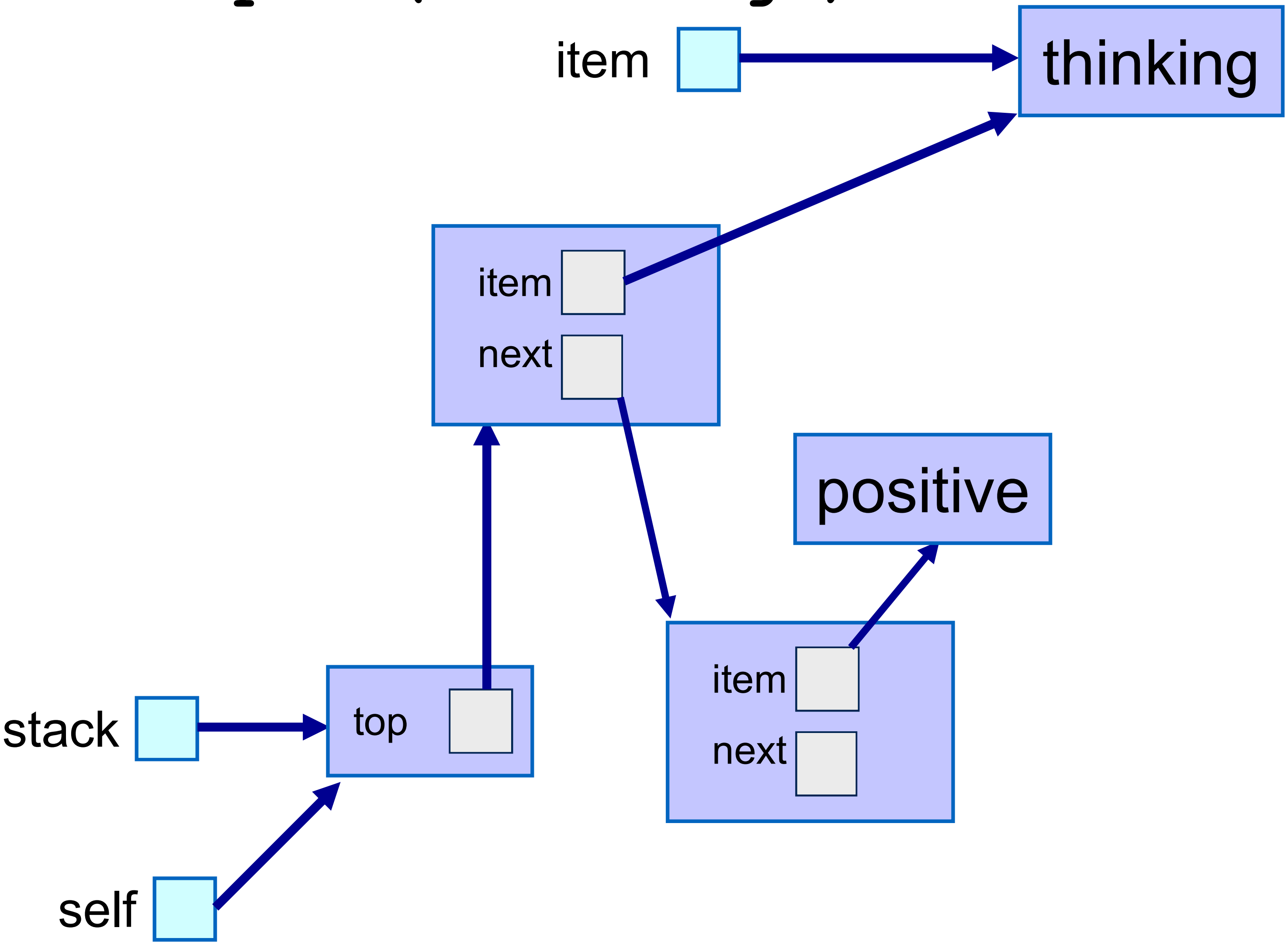
```
def push(self, item):
    self.top = Node(item, self.top)
```

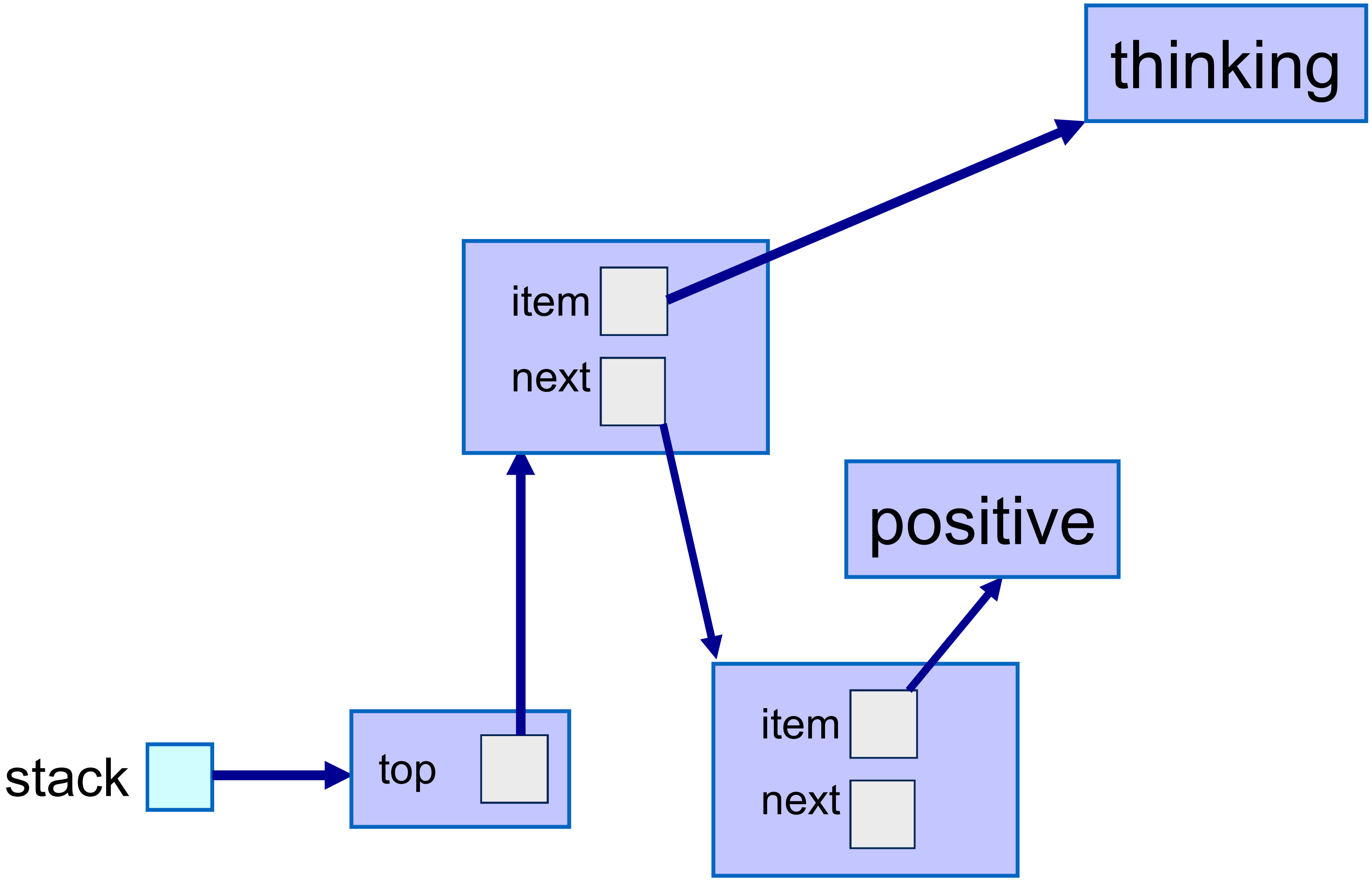
Consider a stack with "**positive**" on top

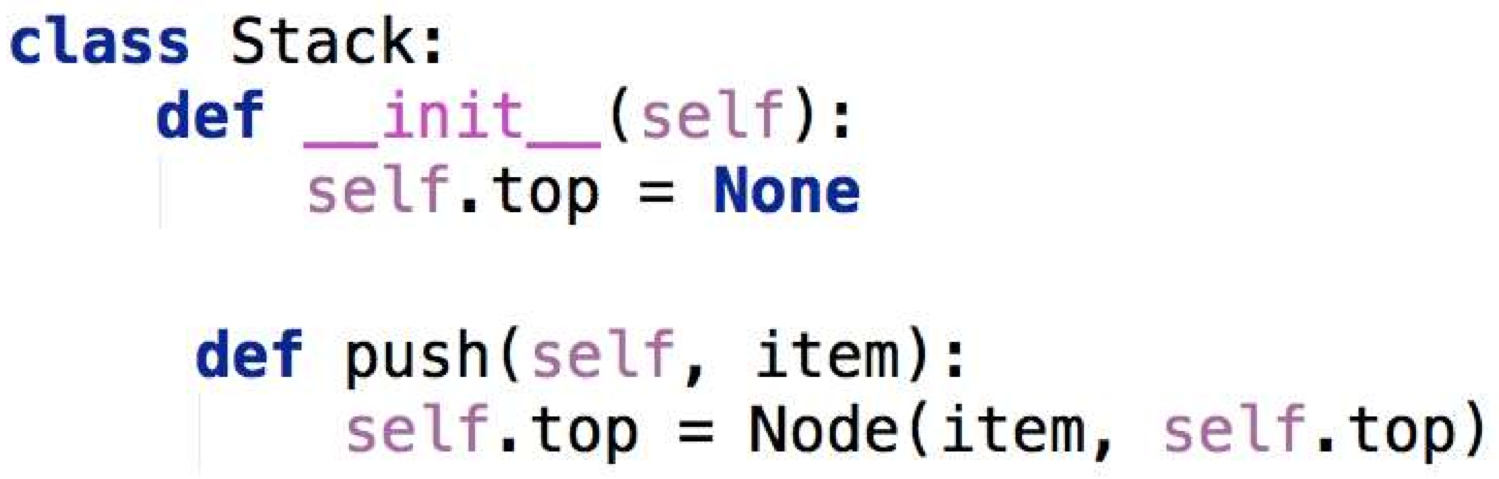


```
def push(self, item):
    self.top = Node(item, self.top)
```

`stack.push("thinking")`



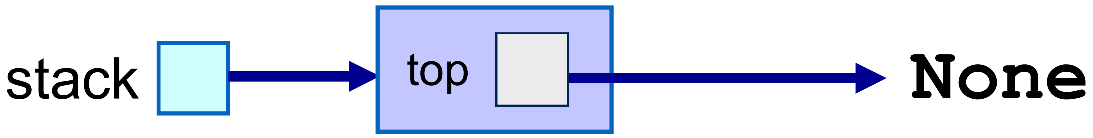

thinking
item
next
positive
item
next
stack
top

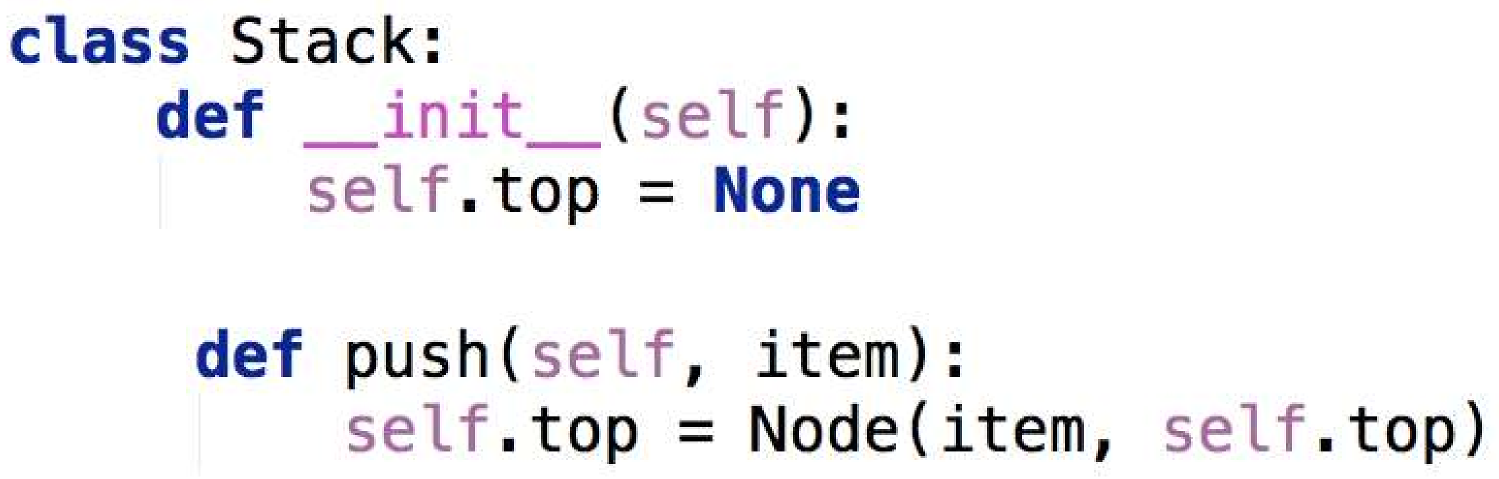


```
class Stack:
    def __init__(self):
        self.top = None

    def push(self, item):
        self.top = Node(item, self.top)
```

stack = Stack()



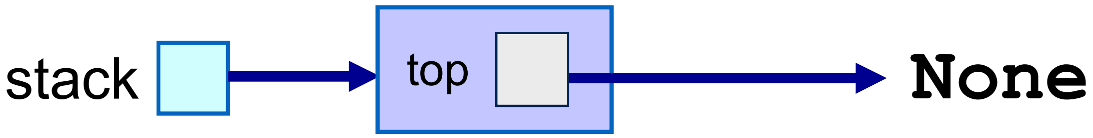


```
class Stack:
    def __init__(self):
        self.top = None

    def push(self, item):
        self.top = Node(item, self.top)
```

stack = Stack()

stack.push(“telemarketers”)

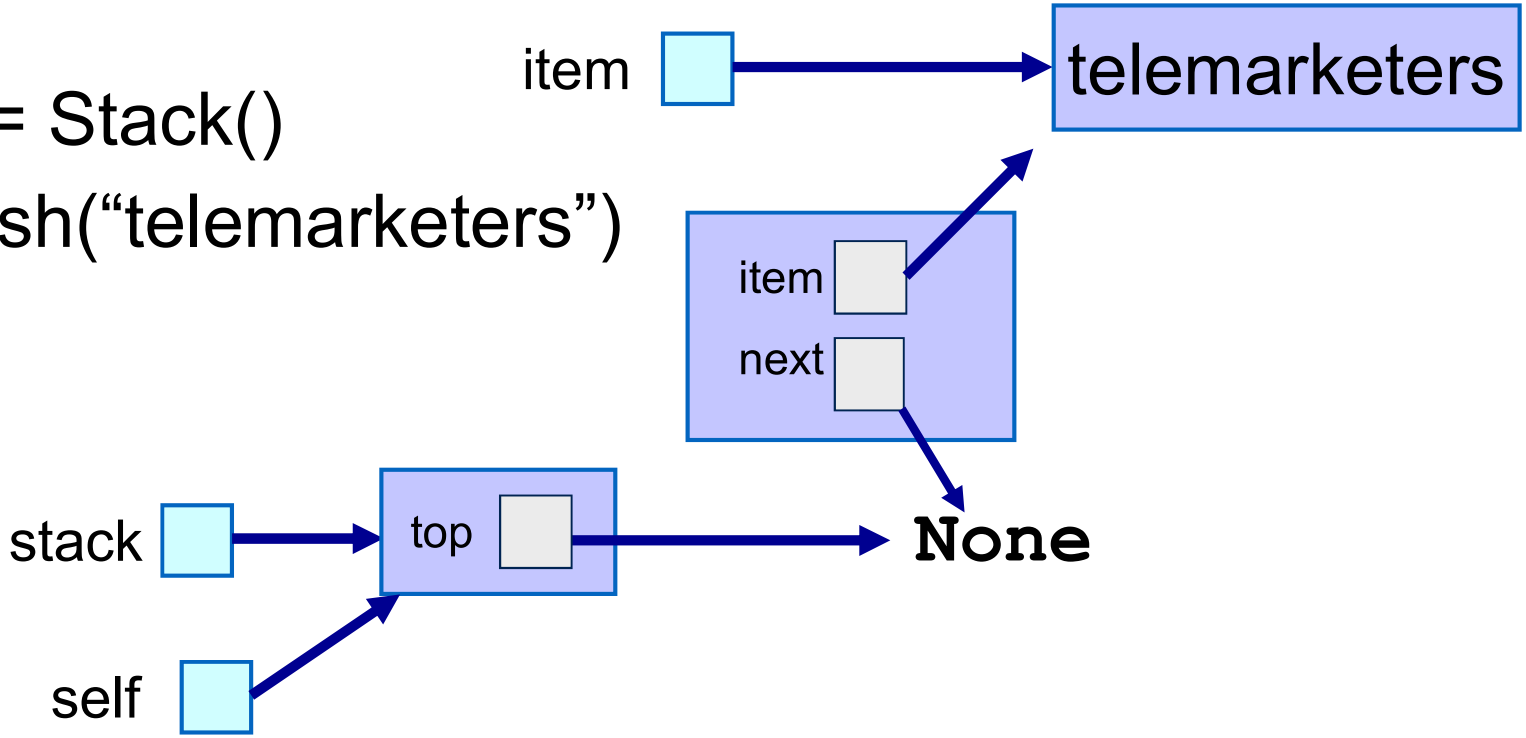


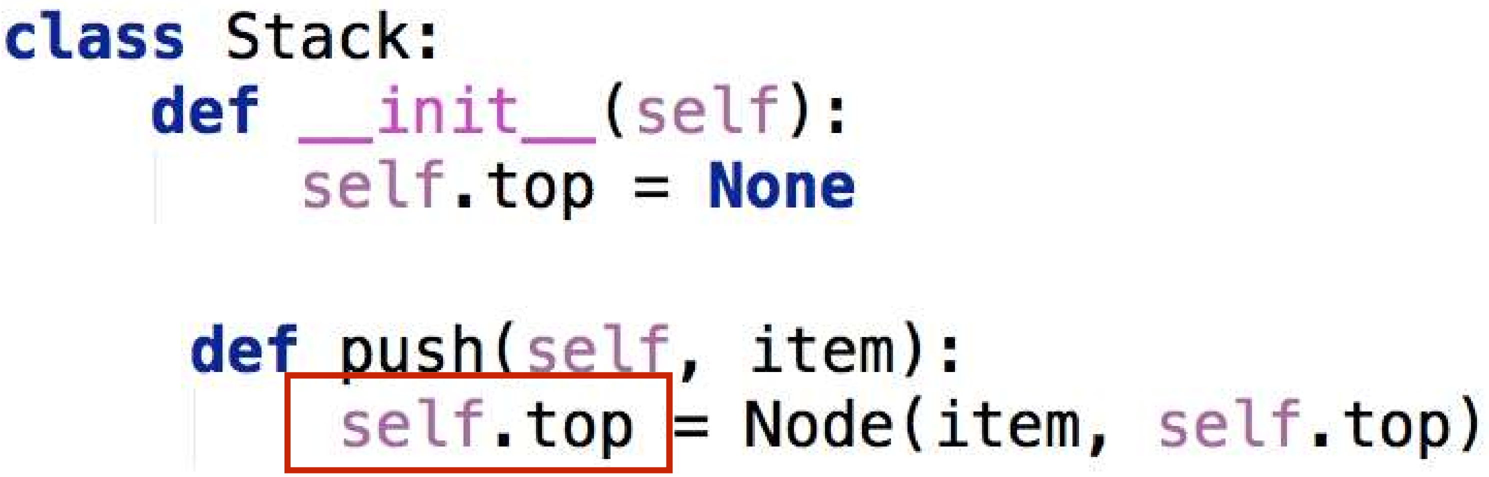
```
class Stack:
    def __init__(self):
        self.top = None

    def push(self, item):
        self.top = Node(item, self.top)
```

stack = Stack()

stack.push(“telemarketers”)



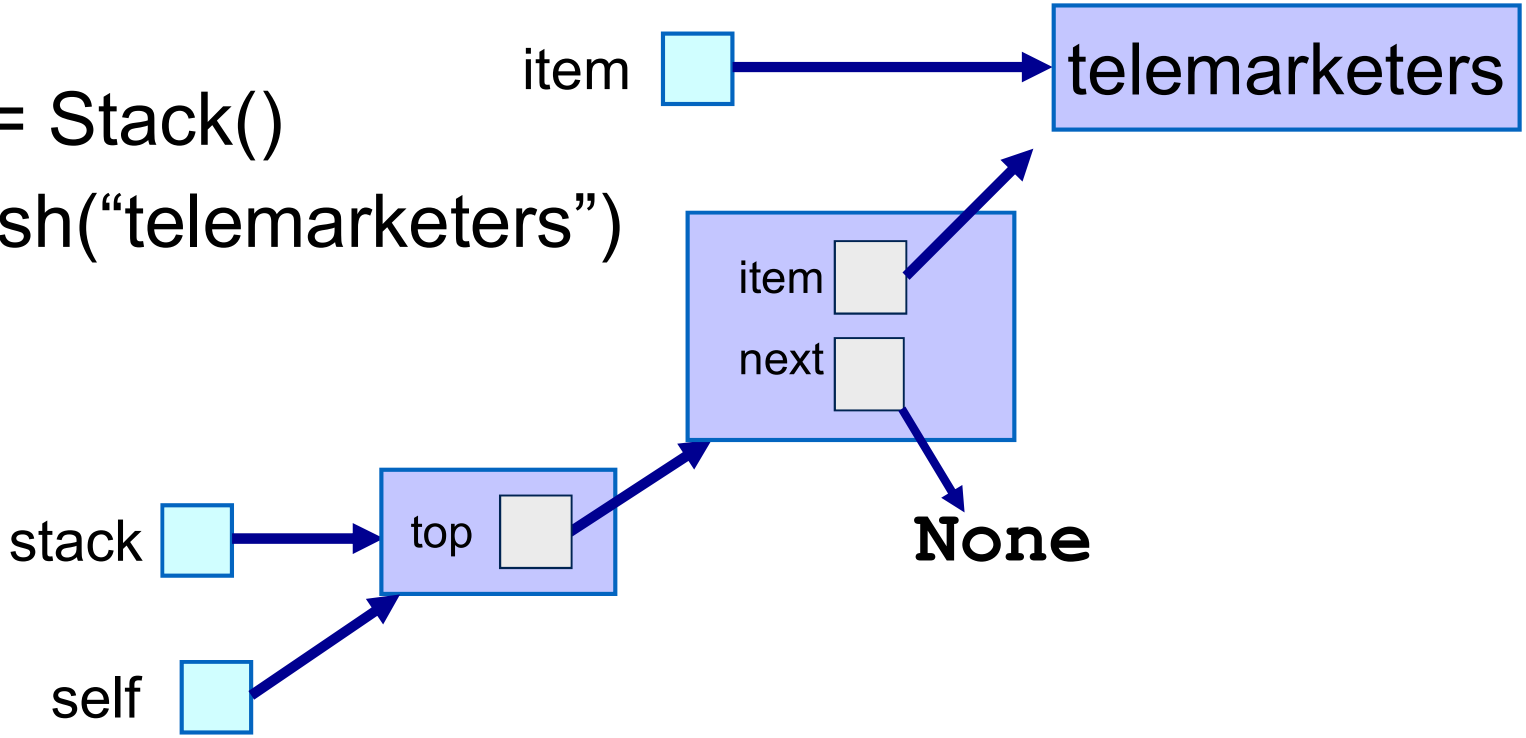


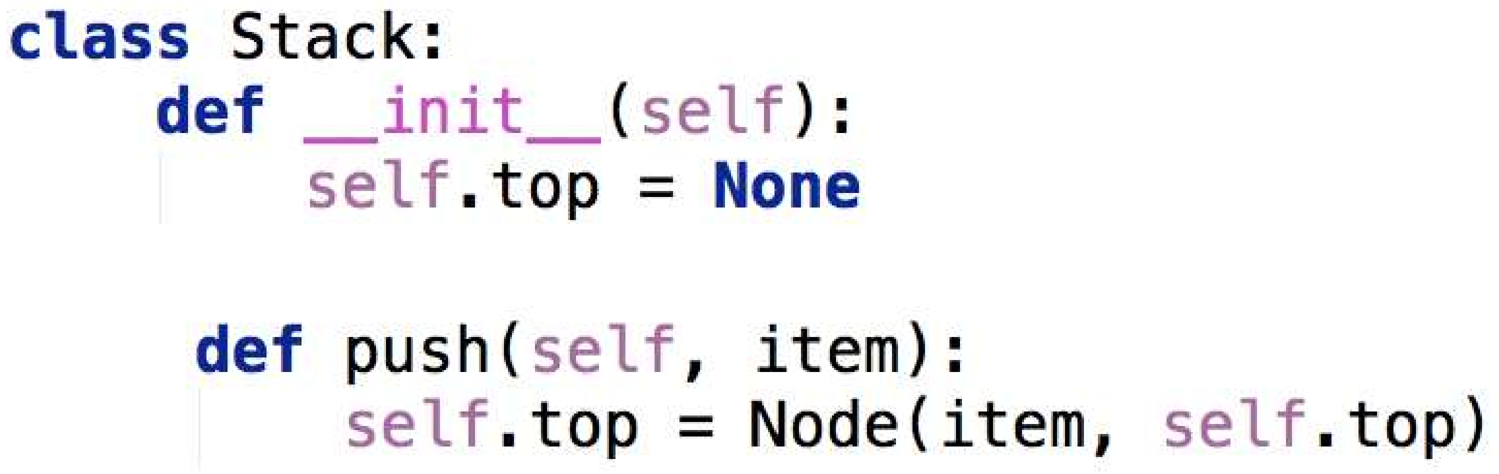
```
class Stack:
    def __init__(self):
        self.top = None

    def push(self, item):
        self.top = Node(item, self.top)
```

stack = Stack()

stack.push(“telemarketers”)



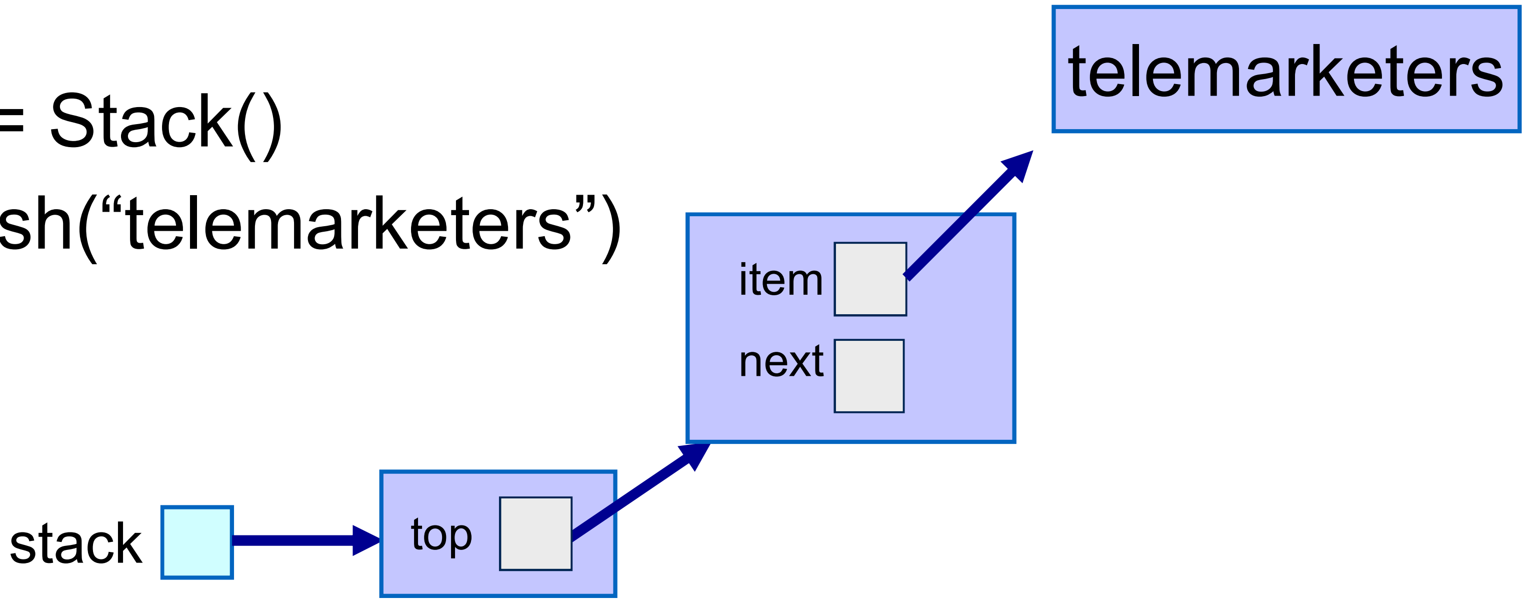


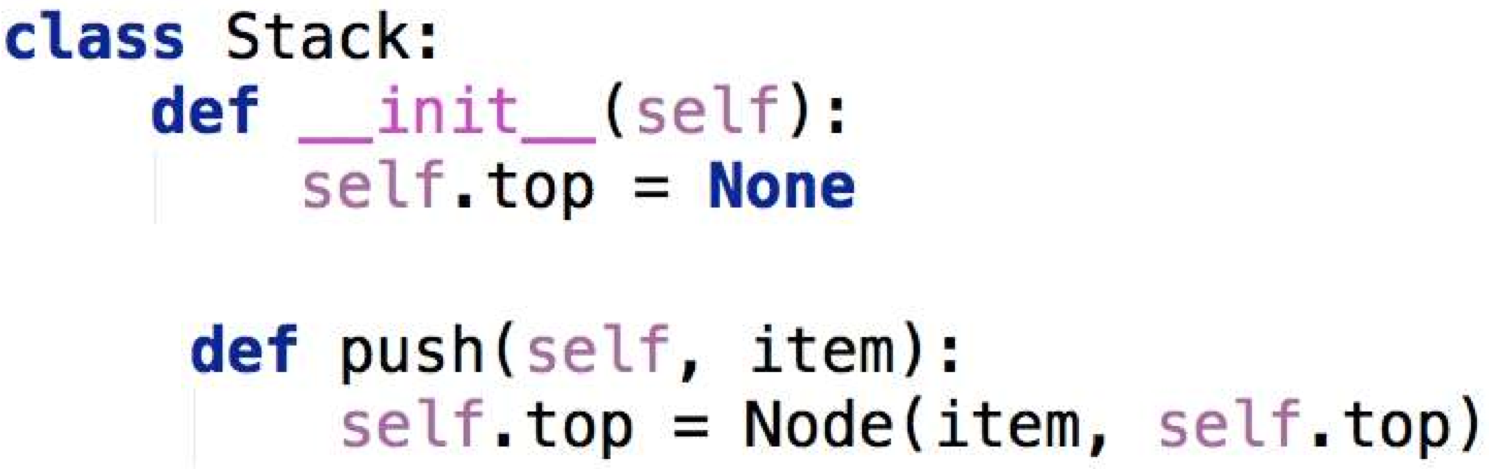
```
class Stack:
    def __init__(self):
        self.top = None

    def push(self, item):
        self.top = Node(item, self.top)
```

stack = Stack()

stack.push(“telemarketers”)



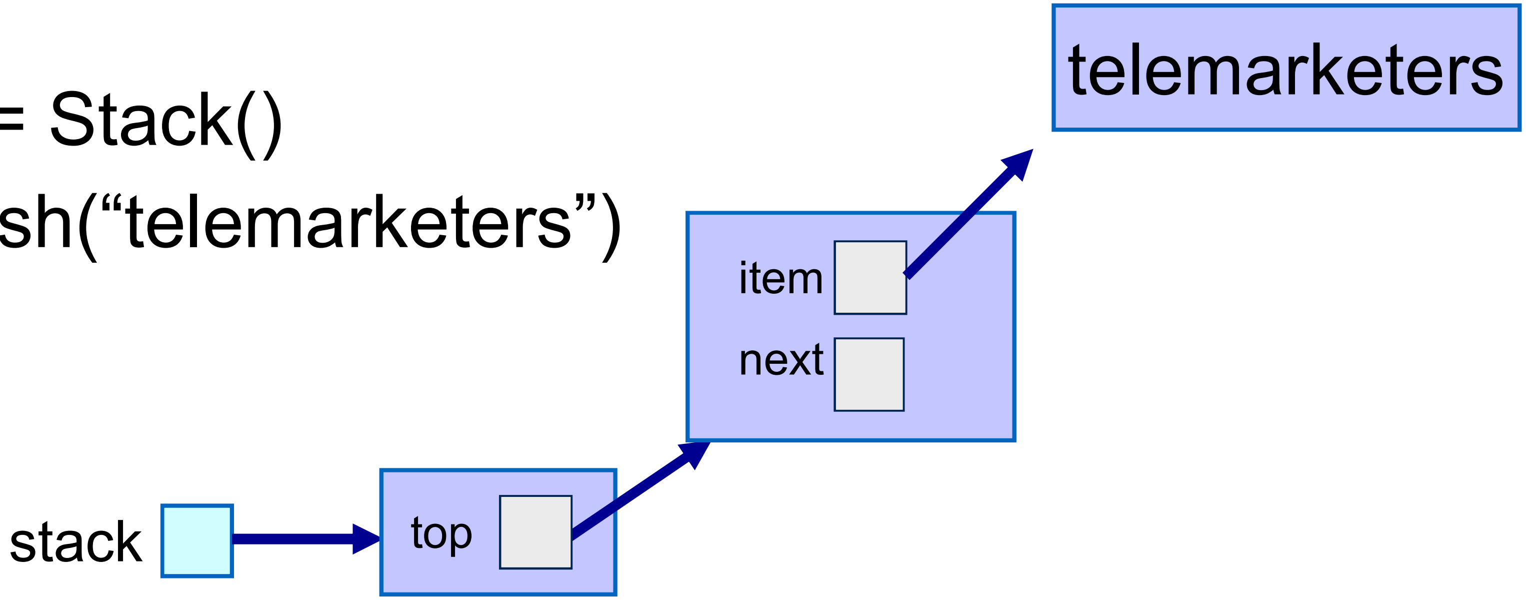


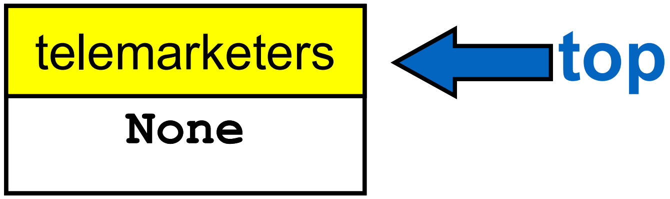
```
class Stack:
    def __init__(self):
        self.top = None

     def push(self, item):
         self.top = Node(item, self.top)
```

stack = Stack()

stack.push(“telemarketers”)





Pop: algorithm

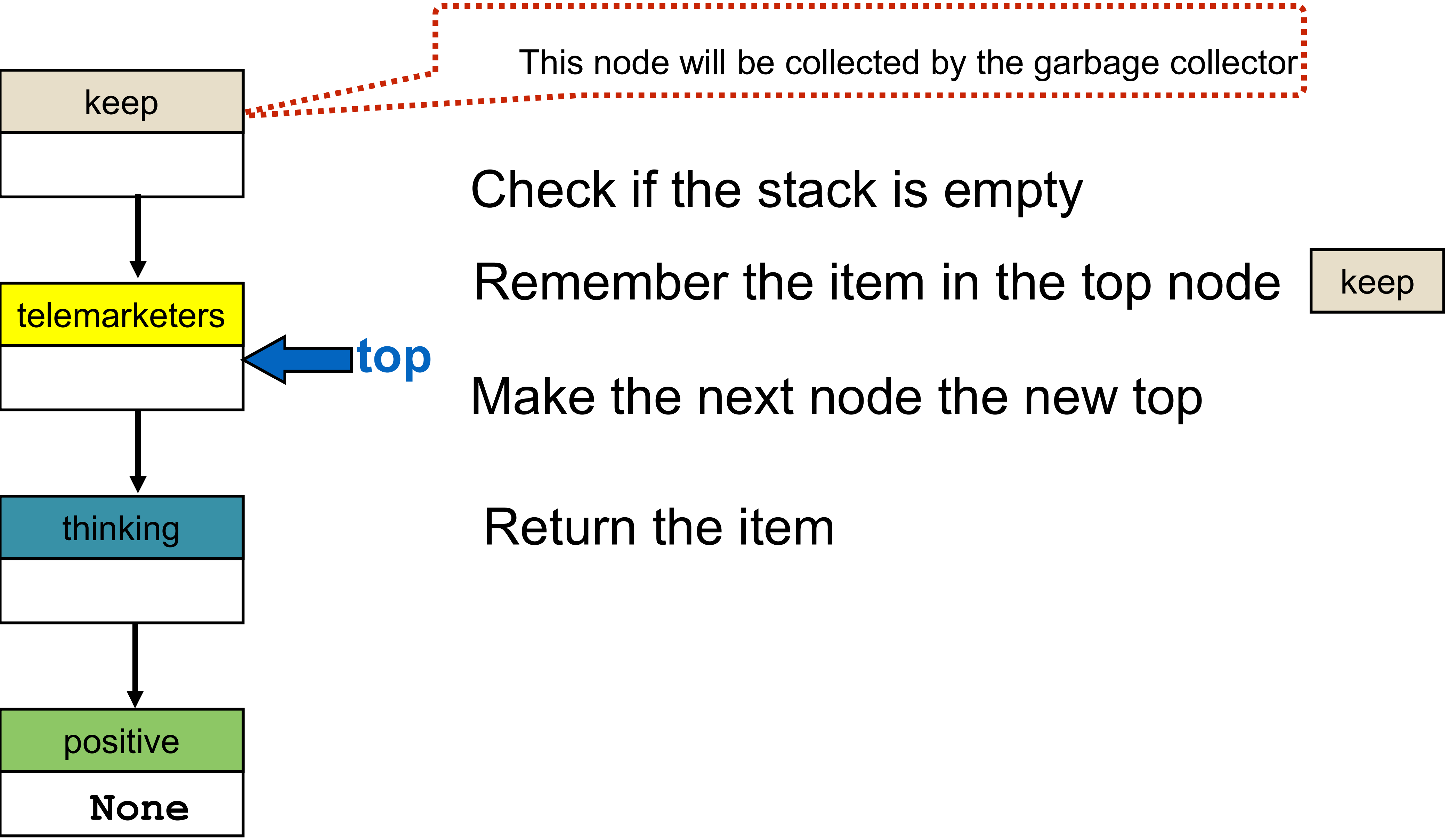
Array implementation:

- If the array is empty raise exception
- Else
  - Remember the top item
  - Decrease top
  - Return the item

Linked implementation:

- If the stack is empty raise exception
- Else
  - Remember the top item
  - **Change top to point to the next node**
  - Return the item

Pop: algorithm



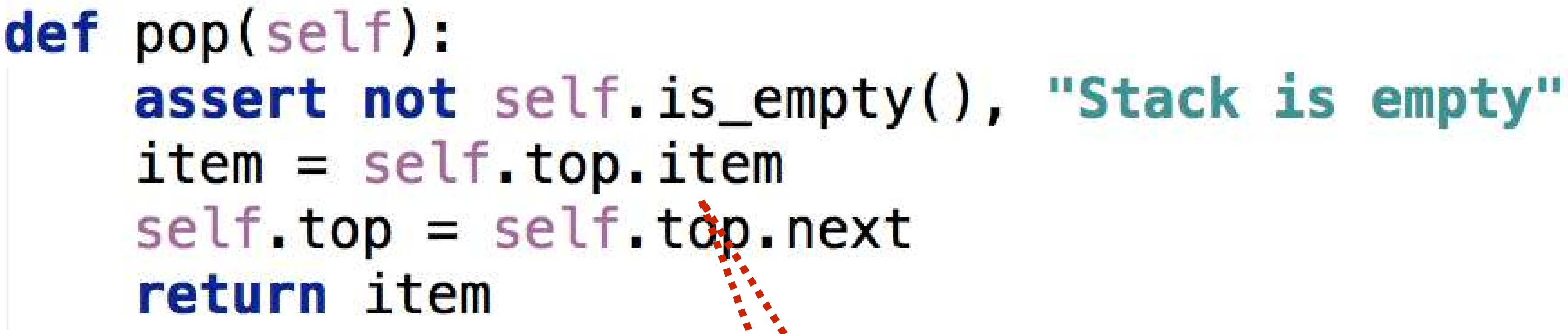
This node will be collected by the garbage collector

Check if the stack is empty

Remember the item in the top node

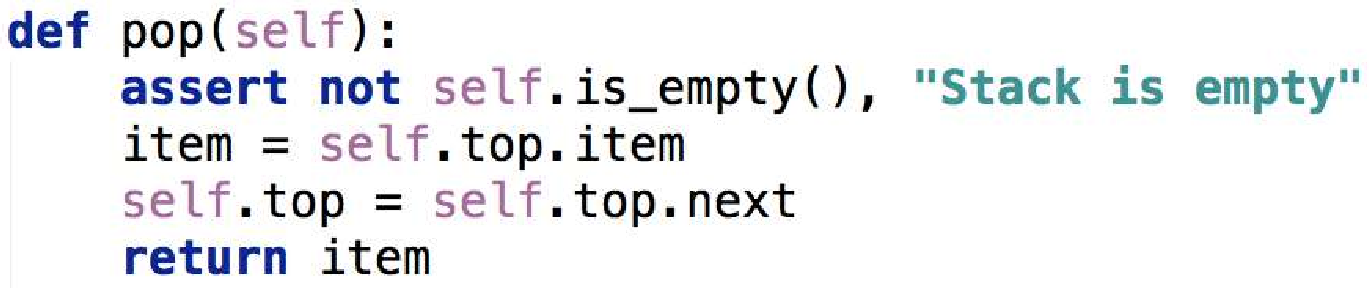
Make the next node the new top

Return the item



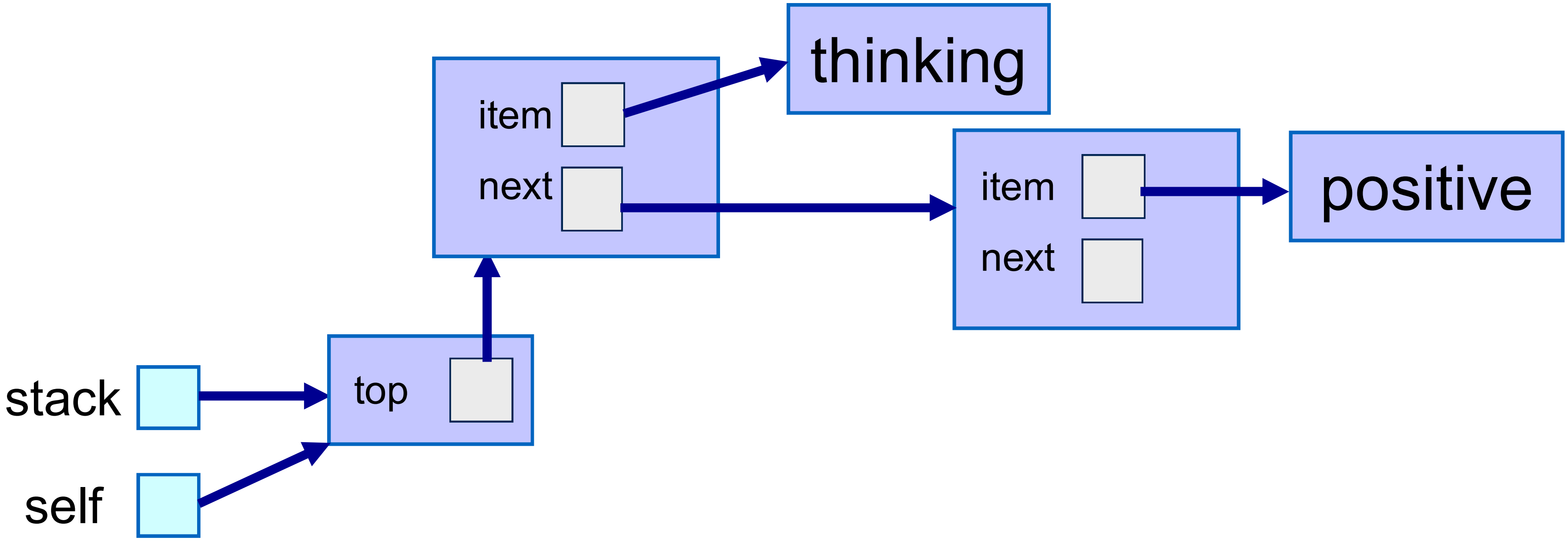
```
def pop(self):
    assert not self.is_empty(), "Stack is empty"
    item = self.top.item
    self.top = self.top.next
    return item
```

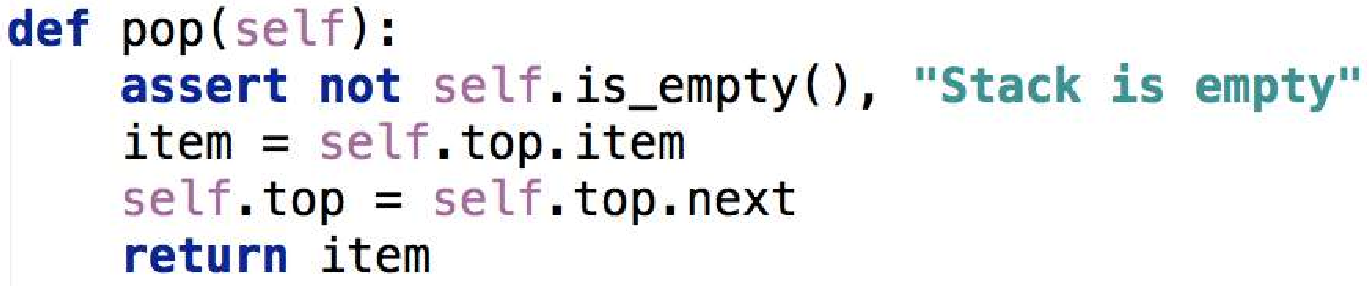
Note: it is **self.top.item** not **self.top**



```python
def pop(self):
    assert not self.is_empty(), "Stack is empty"
    item = self.top.item
    self.top = self.top.next
    return item
```

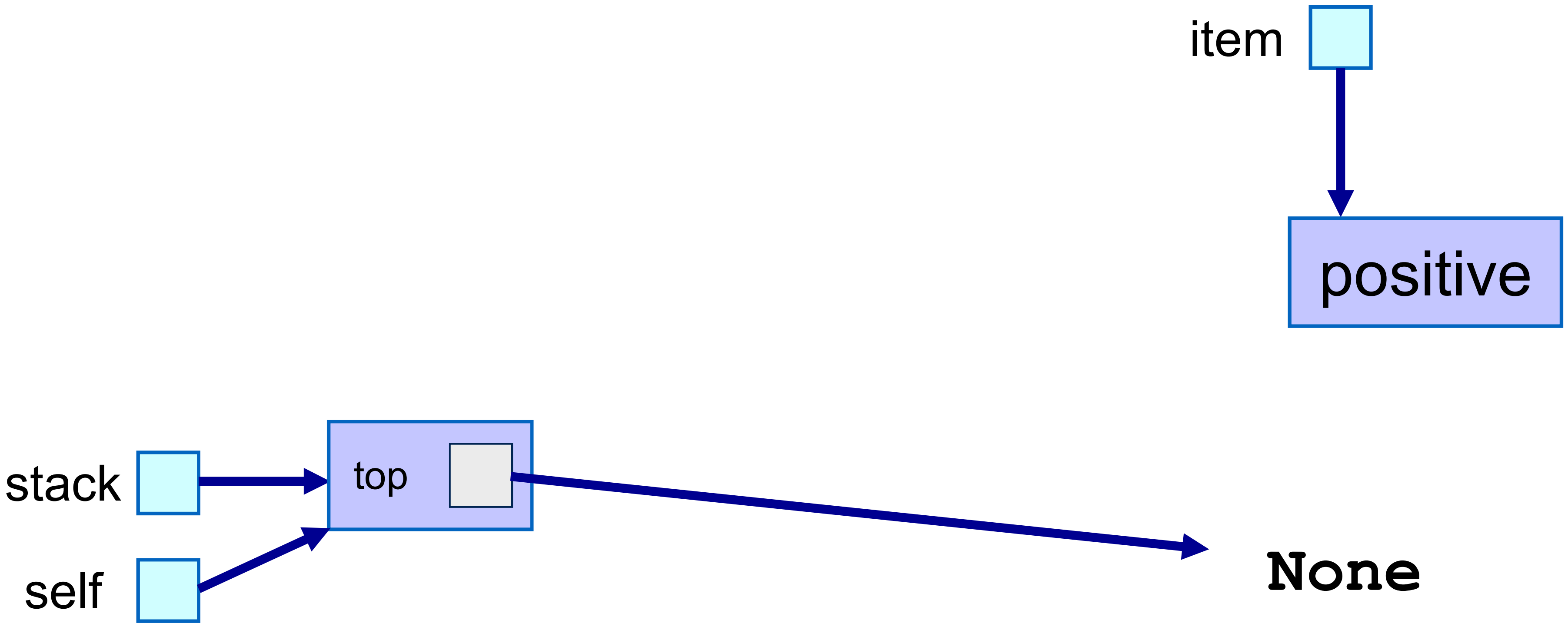
stack.pop( )

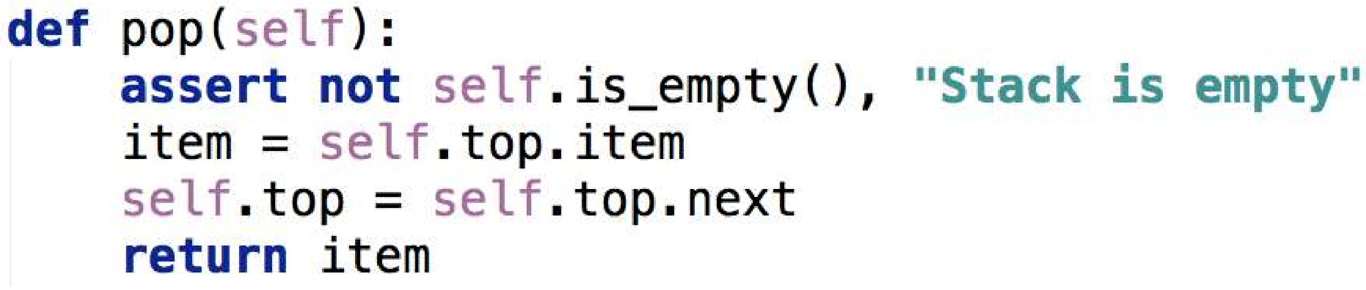




```
def pop(self):
    assert not self.is_empty(), "Stack is empty"
    item = self.top.item
    self.top = self.top.next
    return item
```

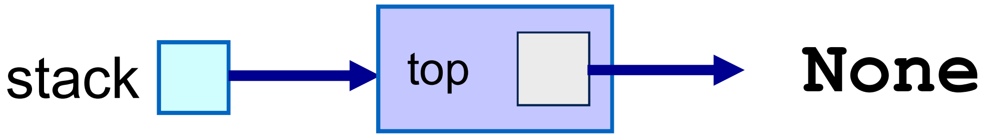
stack.pop( )

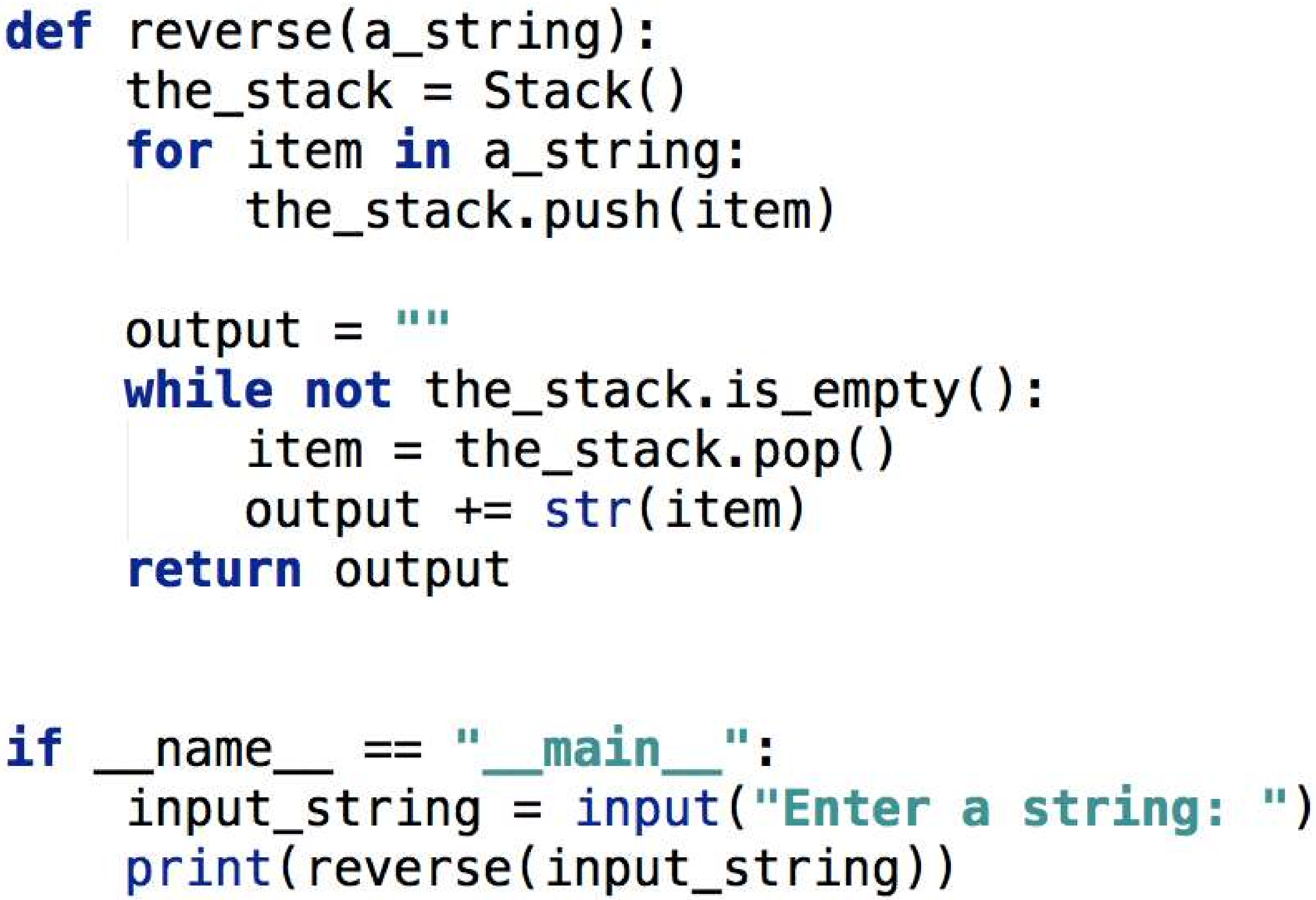




```
def pop(self):
    assert not self.is_empty(), "Stack is empty"
    item = self.top.item
    self.top = self.top.next
    return item
```

stack.pop( )





```python
def reverse(a_string):
    the_stack = Stack()
    for item in a_string:
        the_stack.push(item)

    output = ""
    while not the_stack.is_empty():
        item = the_stack.pop()
        output += str(item)
    return output


if __name__ == "__main__":
    input_string = input("Enter a string: ")
    print(reverse(input_string))
```

Summary

- Advantages and disadvantages of linked data structures
- Stacks implemented with linked data structures