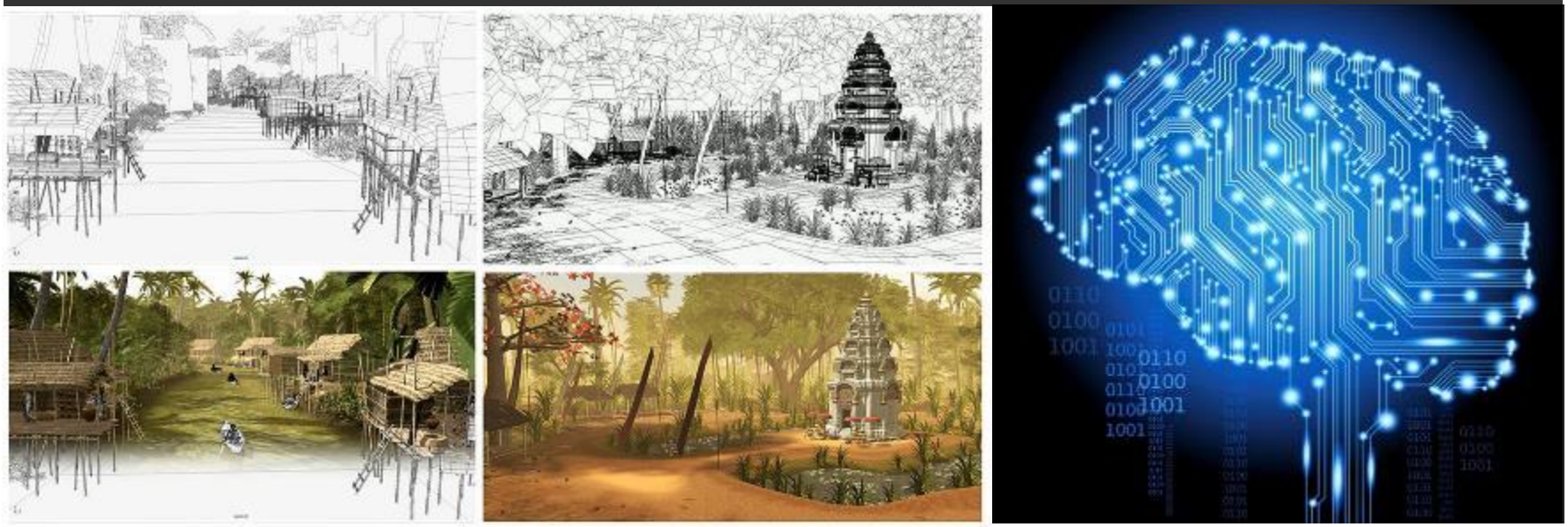
MONASH University

Information Technology

FIT1008&2085 Lecture 17

Prepared by: M. Garcia de la Banda

Stacks with Arrays



Where we were at?

- **We are now familiar with:**
  - Developing simple algorithms in Python
  - Computing Big O for them
  - The concept of Abstract Data Type
  - Implementing a sorted/unsorted list ADT using arrays
  - (Simplified) internal variable and object representation in Python
  - Mutable/immutable types
  - Basic exception handling, assertions and unit testing

Objectives for this lecture

- **To understand the Stack abstract data types:**
  - Main operations
  - Their complexity
- **To be able to**
  - Implement it with arrays
  - Use it
  - Modify its operations and
  - Reason about their complexity

Revision: LIFO

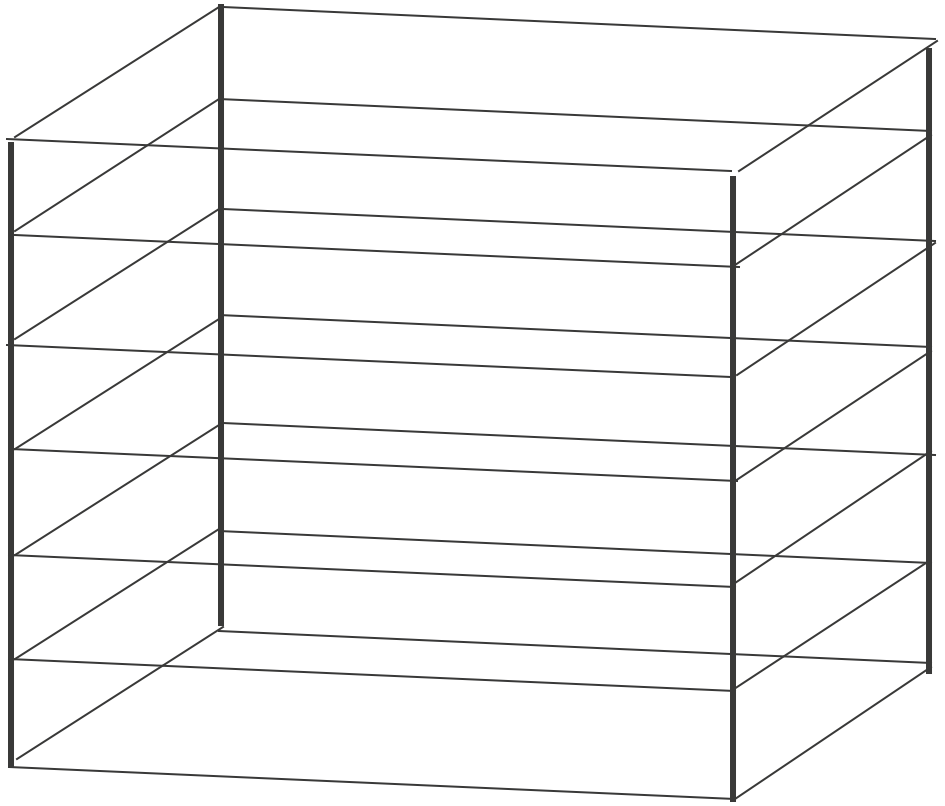
- **Last In First Out (LIFO) processing:**
  - The last element to arrive, is the first to be processed
- **In terms of data storage:**
  - The last element to be added, is the first to be deleted
- **Access to any other element is unnecessary (and thus not allowed)**

The Stack Data Type

- **Follows a LIFO model**
- **Its operations are defined in its interface:**
  - 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: basic idea

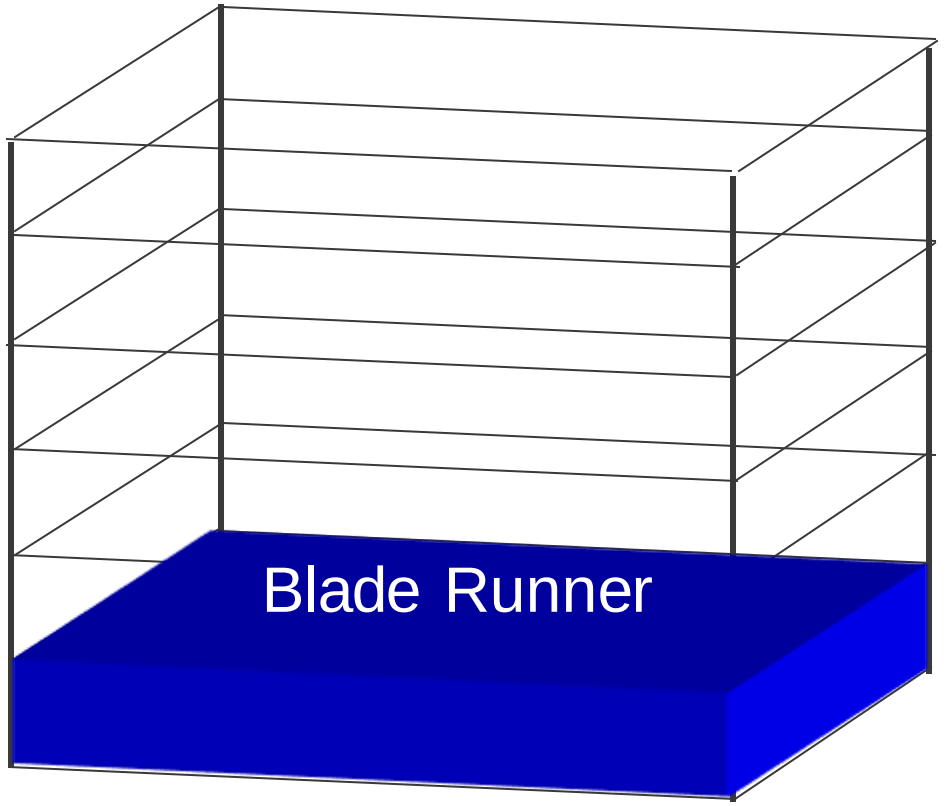
- Create a new stack: initially no elements



- Max size = 6
- Used = 0

Stack: basic idea

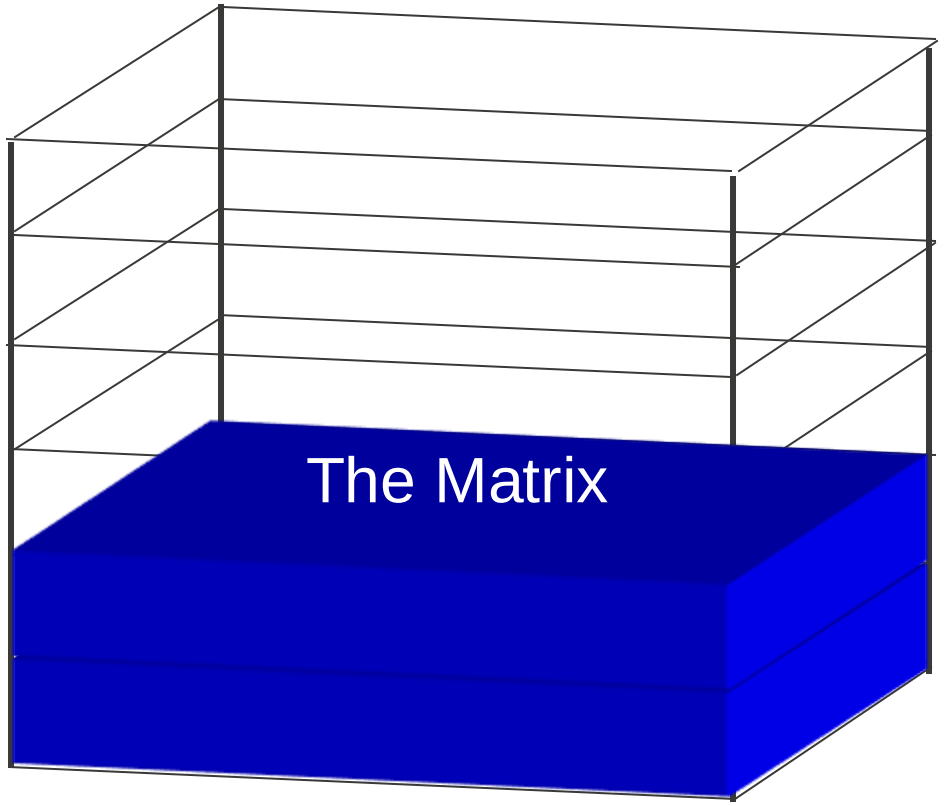
- Add an item to the top (push)



- Max size = 6
- Used = 1

Stack: basic idea

- **Add another item to the top (push)**



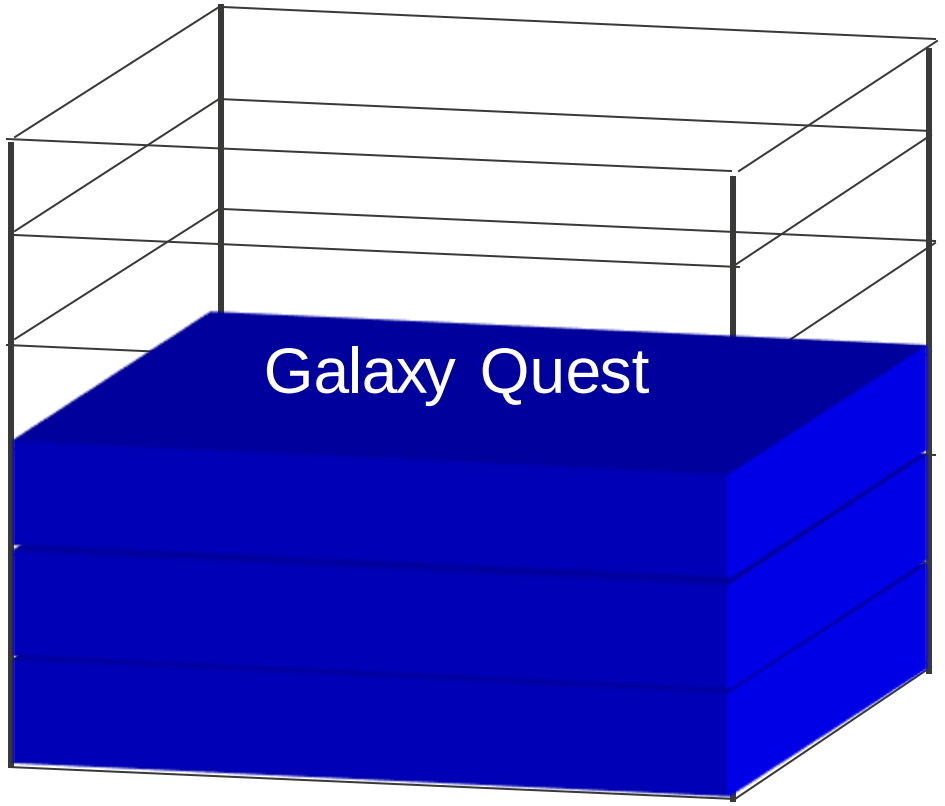
- **Max size = 6**
- **Used = 2**

IMPORTANT: only the top element is accessible. You do not know what is underneath (you know there is something, but not what)

Stack: basic idea

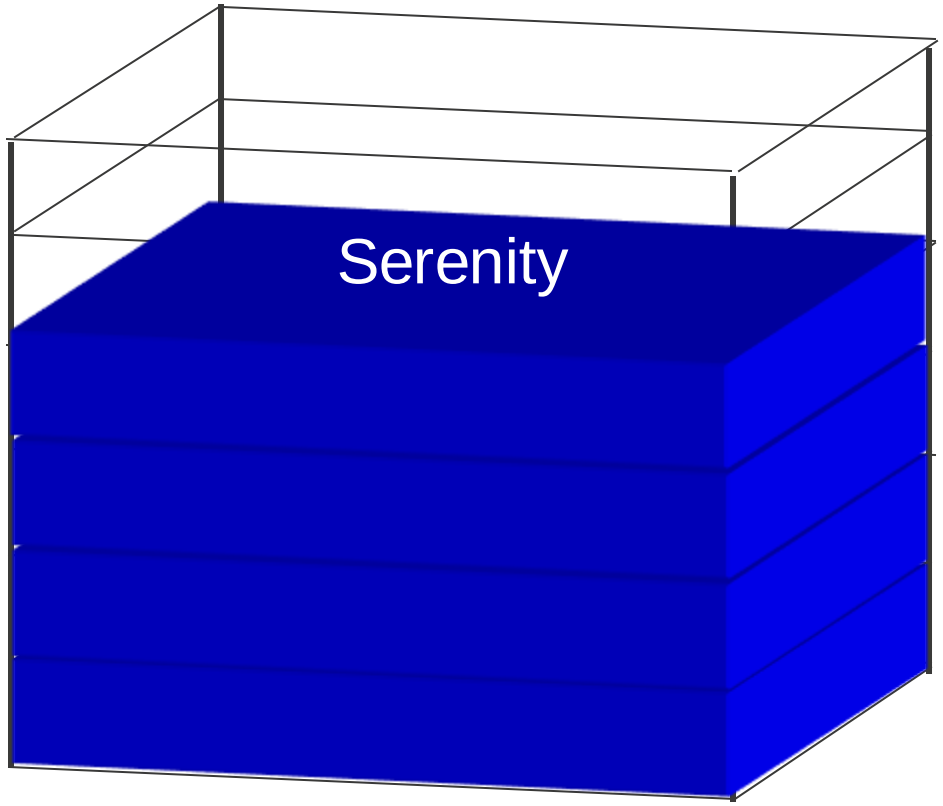
- Add another item to the top (push)

- Max size = 6
- Used = 3



Stack: basic idea

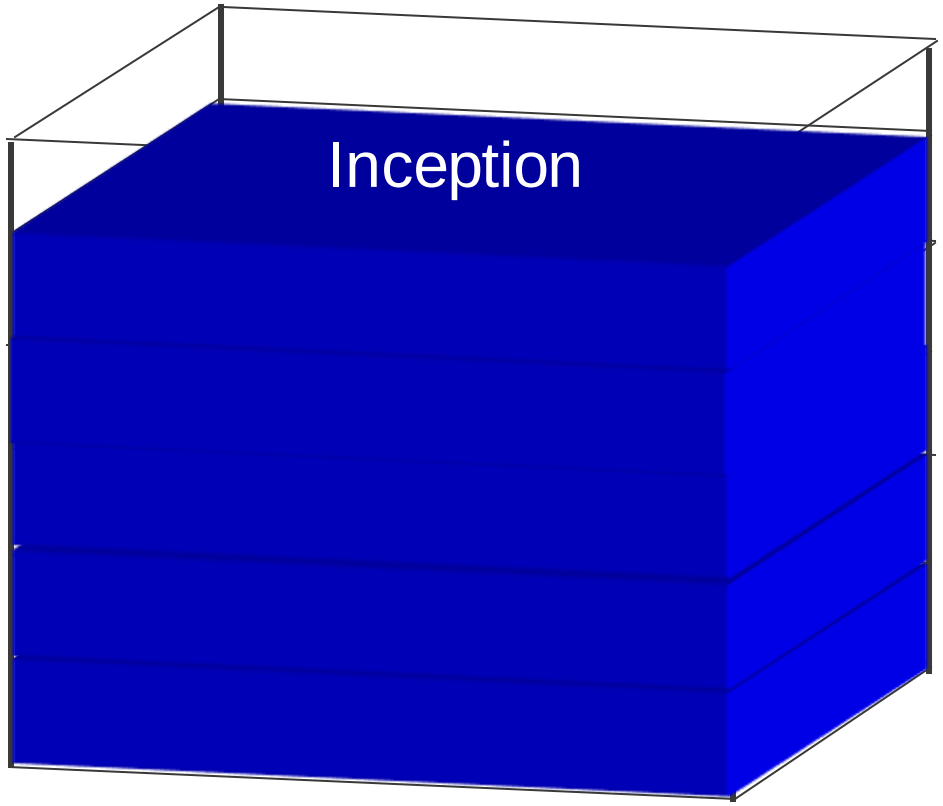
- Add another item to the top (push)



- Max size = 6
- Used = 4

Stack: basic idea

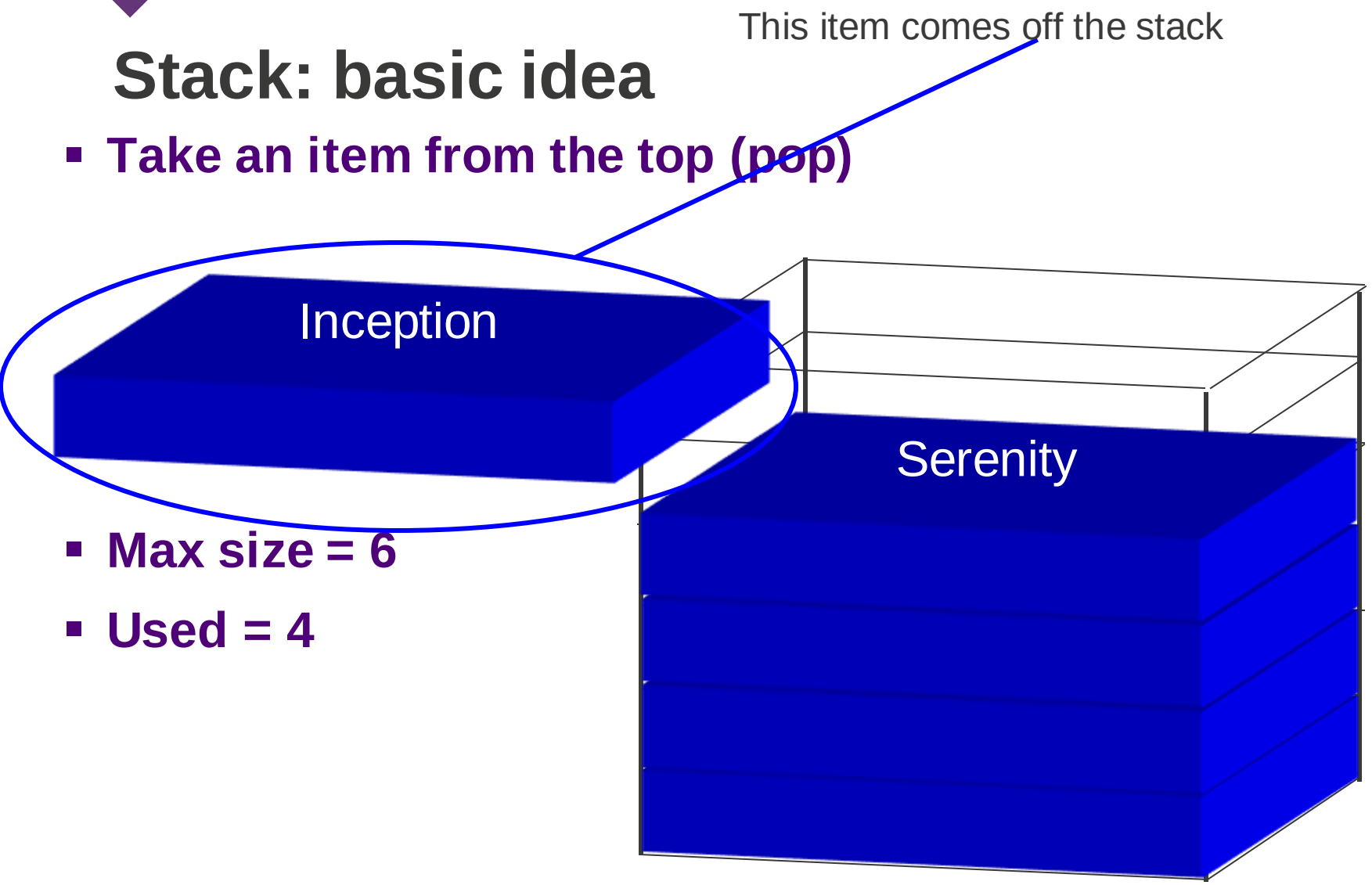
- Add another item to the top (push)



- Max size = 6
- Used = 5

Stack: basic idea

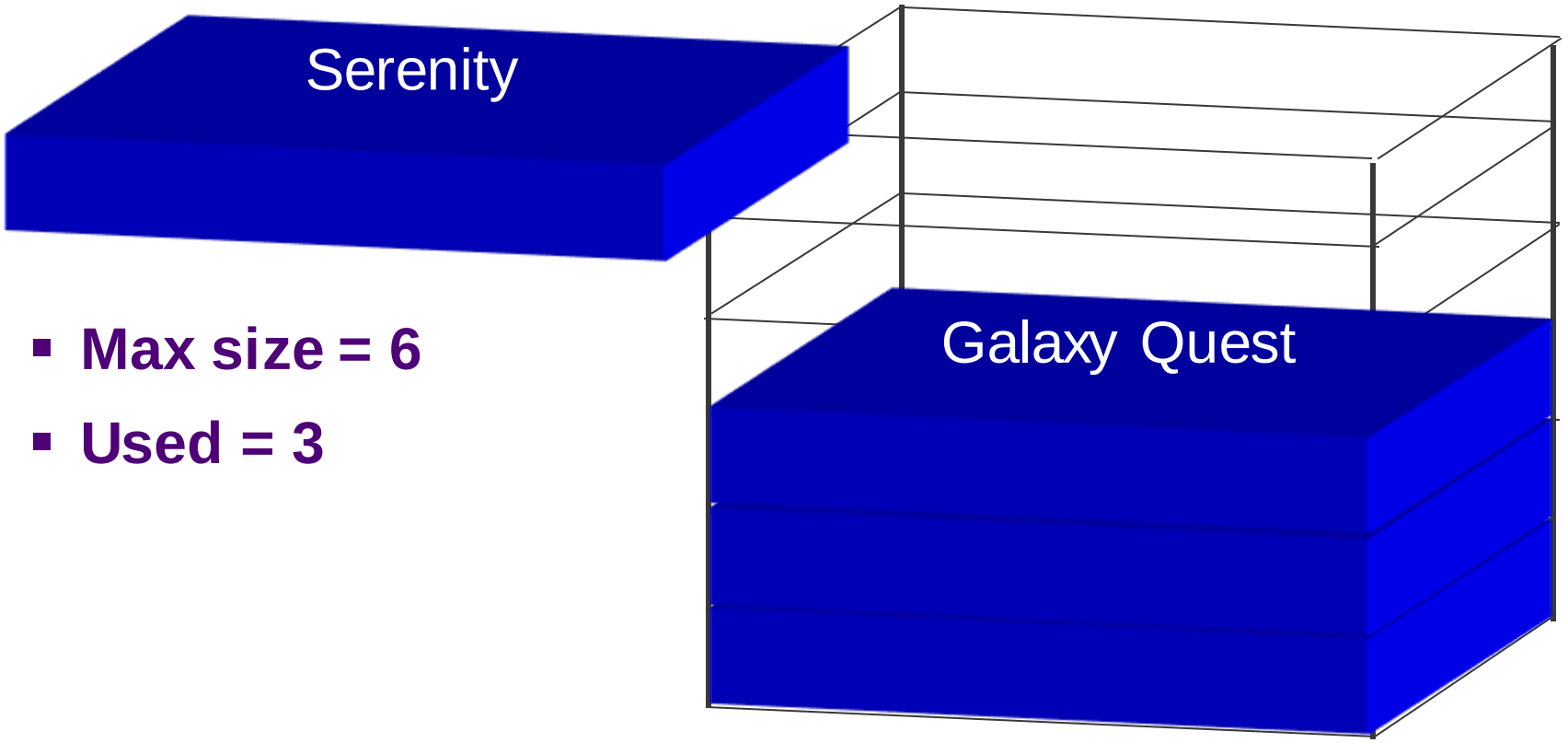
- **Take an item from the top (pop)**



- **Max size = 6**
- **Used = 4**

Stack: basic idea

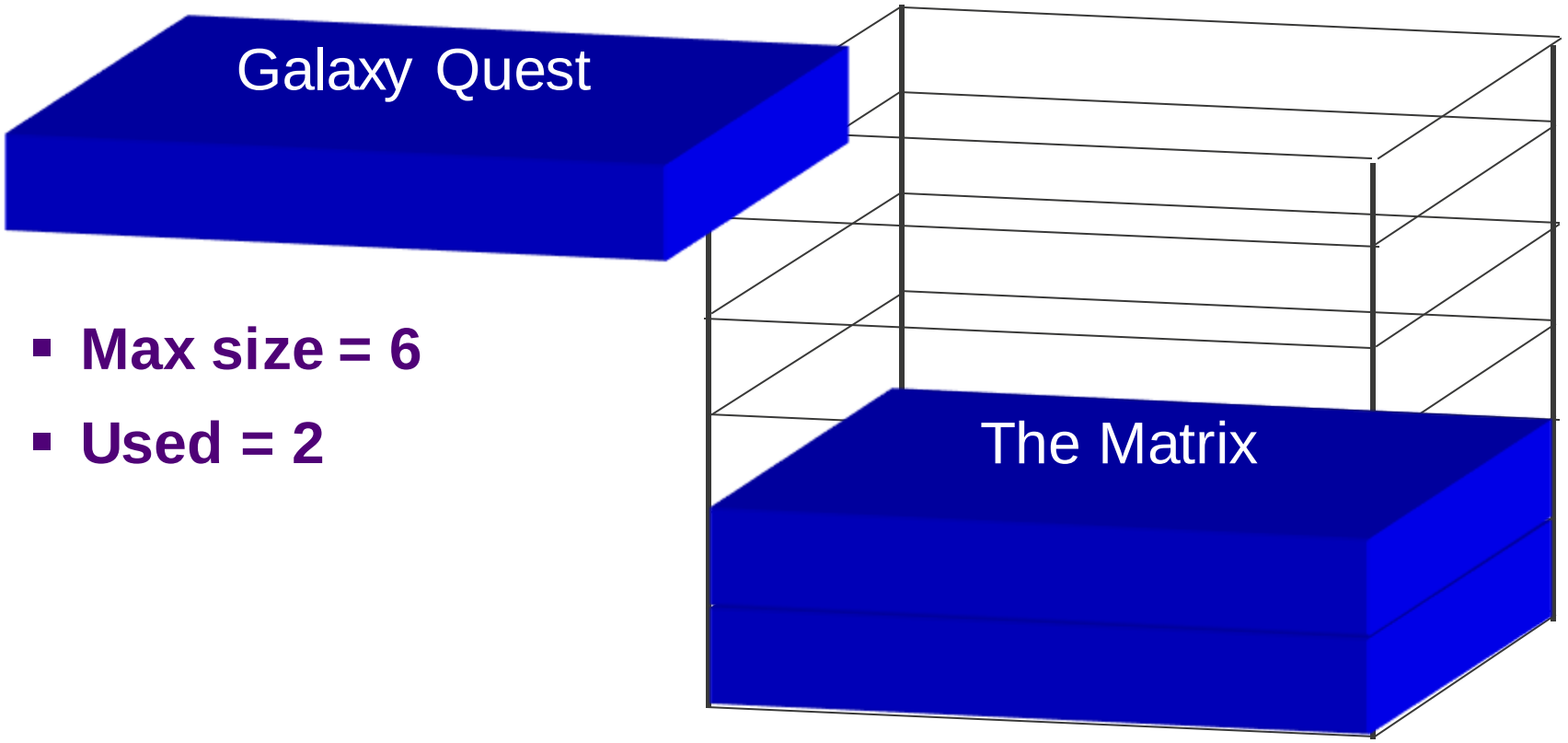
- Take another item from the top (pop)



- Max size = 6
- Used = 3

Stack: basic idea

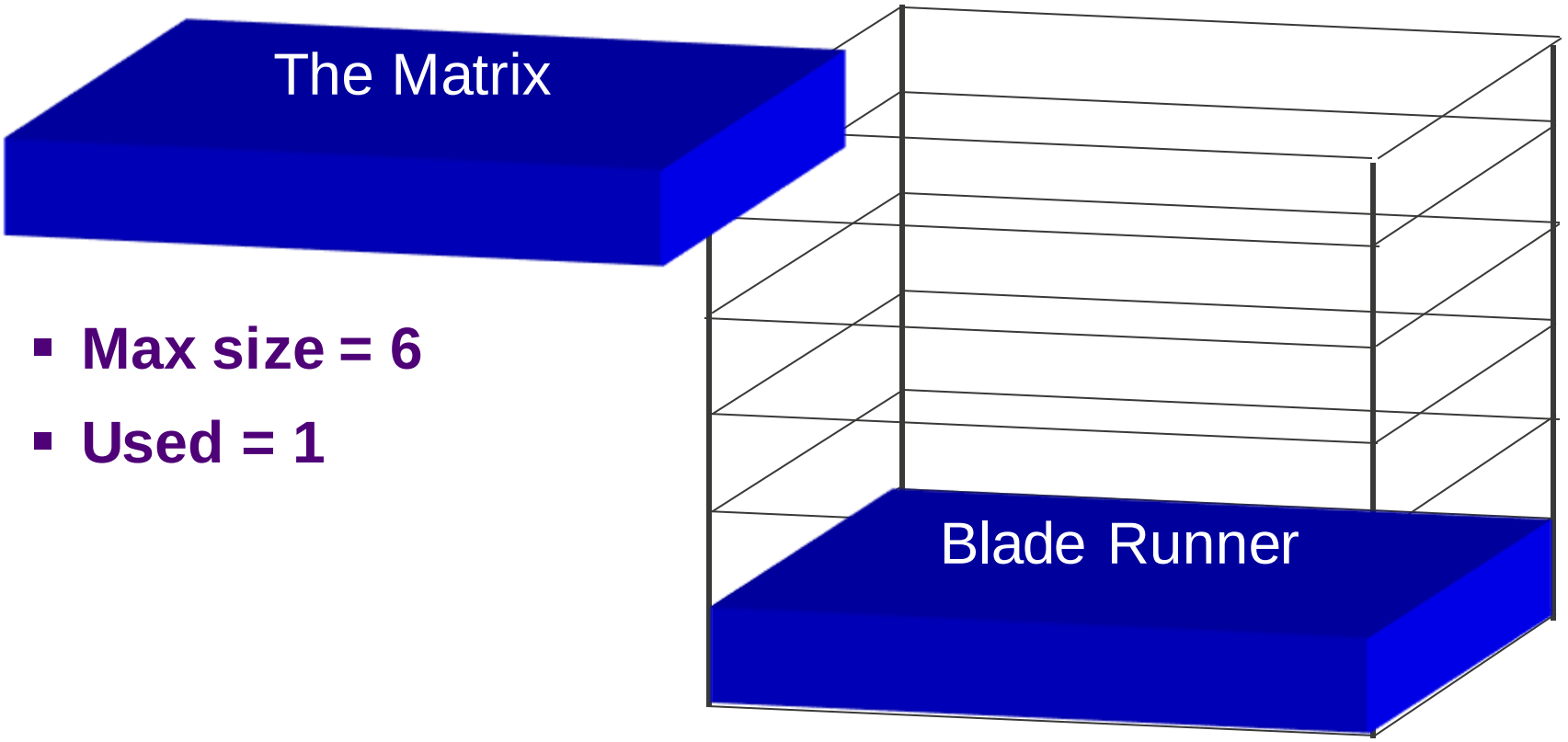
- Take another item from the top (pop)



- Max size = 6
- Used = 2

Stack: basic idea

- Take another item from the top (pop)



- Max size = 6
- Used = 1

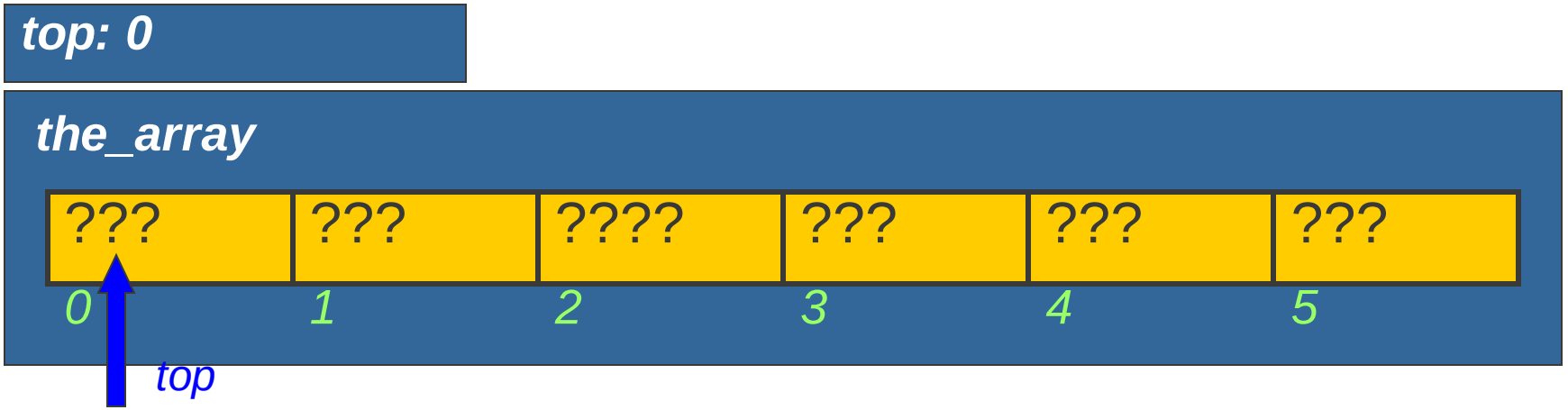
Common Stack implementation

- **Stacks are often implemented using two elements:**
  - An array to store the items in the order in which they arrive.
  - An integer indicating the first empty space in the array (top: which is also the number of elements in the stack)
- **Invariant: valid data in the `0..top-1` positions**
- **Pretty similar to lists, so what is the difference?**
  - The operations provided!
  - **`Stack, is_empty, is_full, size`**
  - **`push, pop, peek`** }

These are the different ones

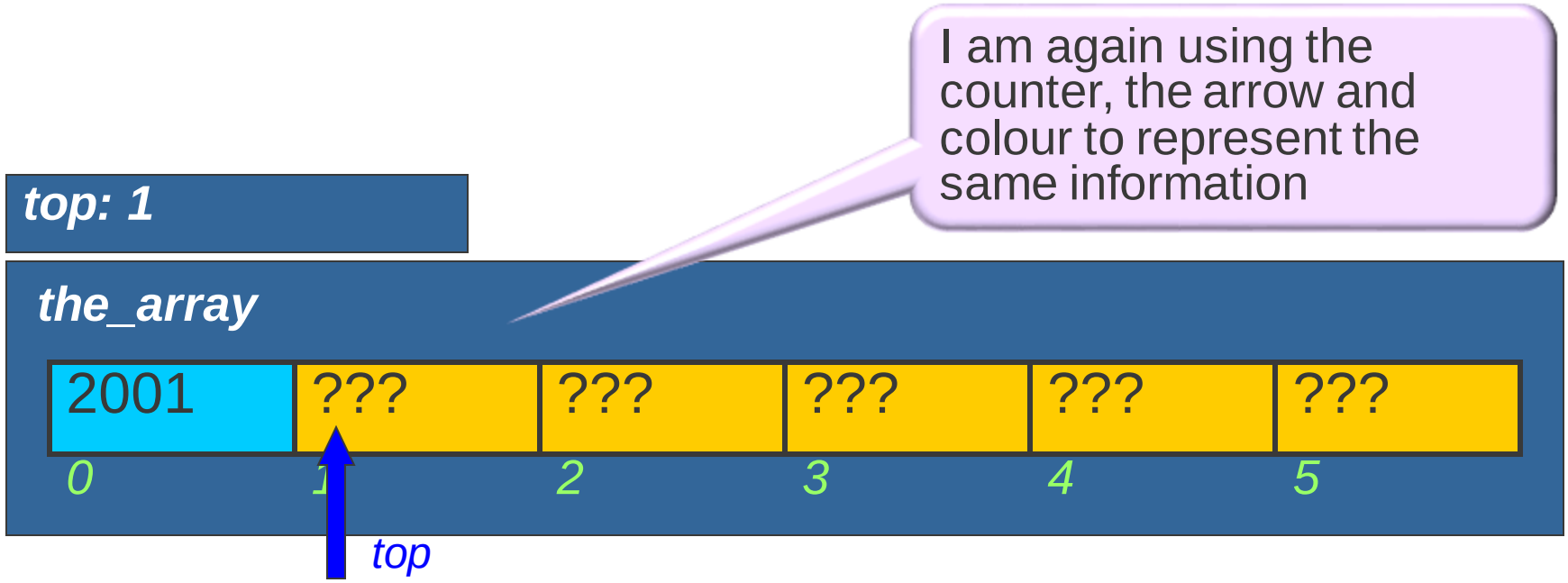
Using the stack

```
the_stack = Stack(6)
```



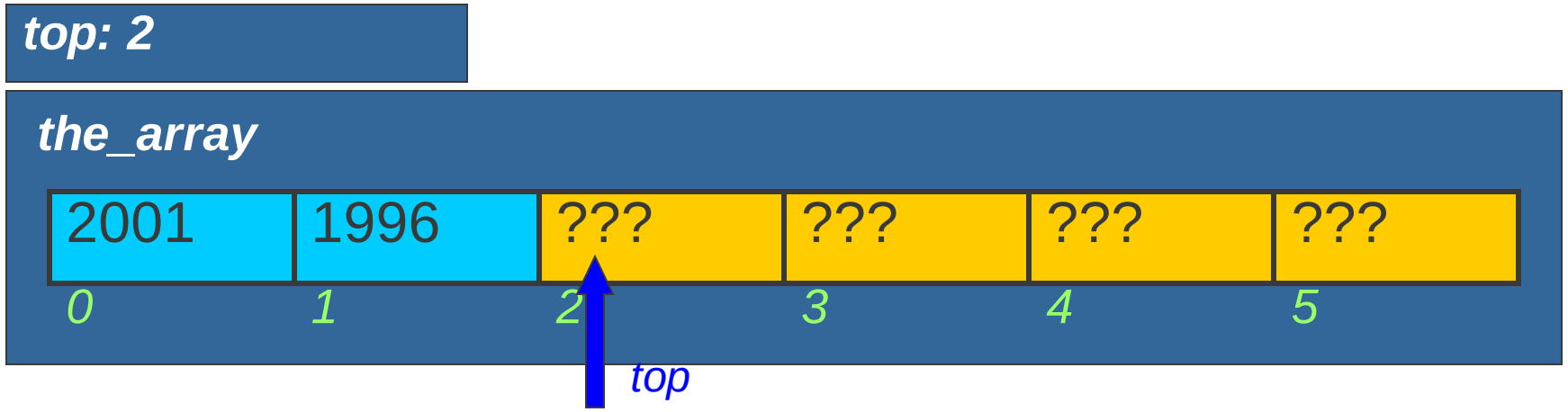
Using the stack

```
the_stack = Stack(6)
push(the_stack, 2001)
```



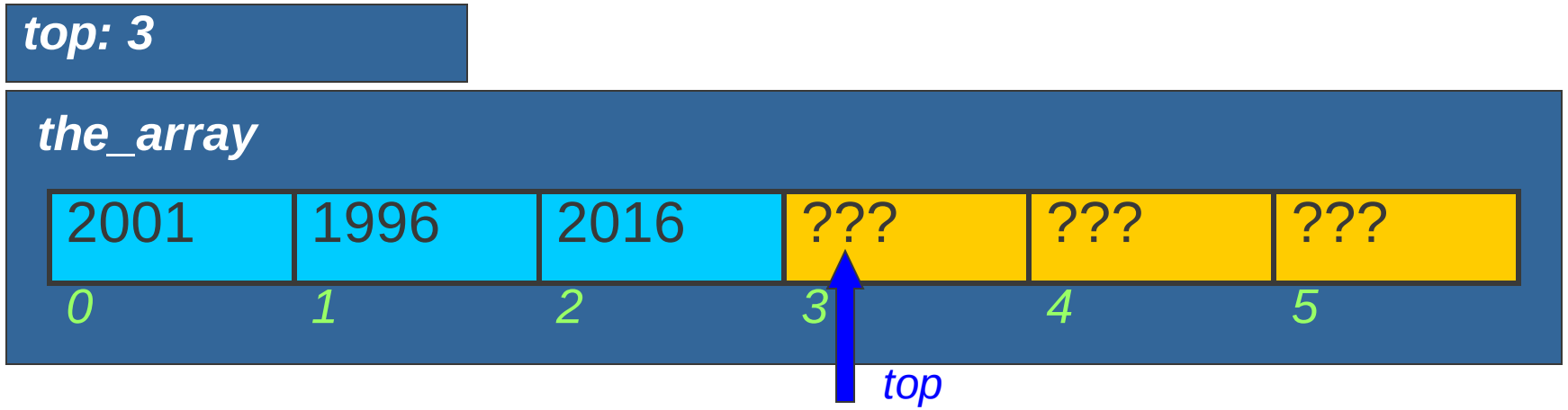
Using the stack

```
the_stack = Stack(6)
push(the_stack, 2001)
push(the_stack, 1996)
```



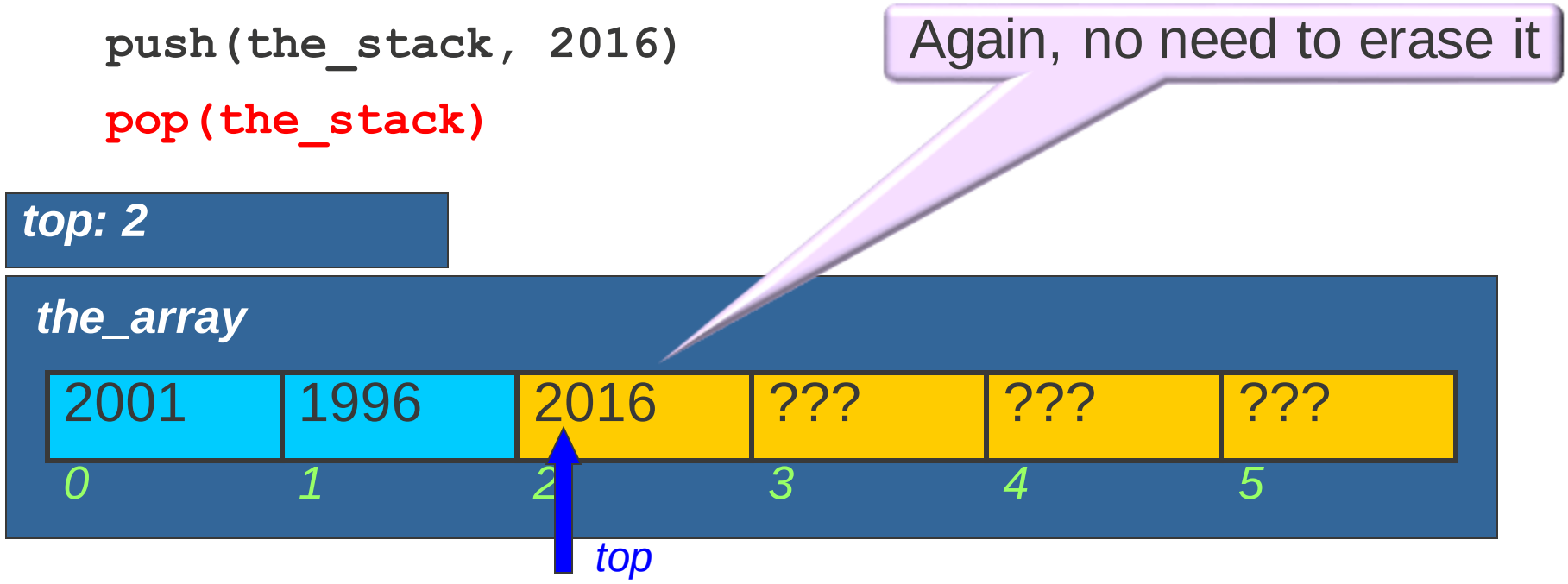
Using the stack

```
the_stack = Stack(6)
push(the_stack, 2001)
push(the_stack, 1996)
push(the_stack, 2016)
```



Using the stack

```
the_stack = Stack(6)
push(the_stack, 2001)
push(the_stack, 1996)
push(the_stack, 2016)
pop(the_stack)
```



Implementation for an array Stack

- **Let's start defining:** `Stack, size, is_empty, is_full`

```
def Stack(size):
    the_array = [None] * size
    top = 0
    return [top, the_array]

def size(the_stack):
    [top,_] = the_stack
    return top

def is_empty(the_stack):
    return size(the_stack) == 0

def is_full(the_stack):
    [top,the_array] = the_stack
    return top == len(the_array)
```

Same as we did for lists

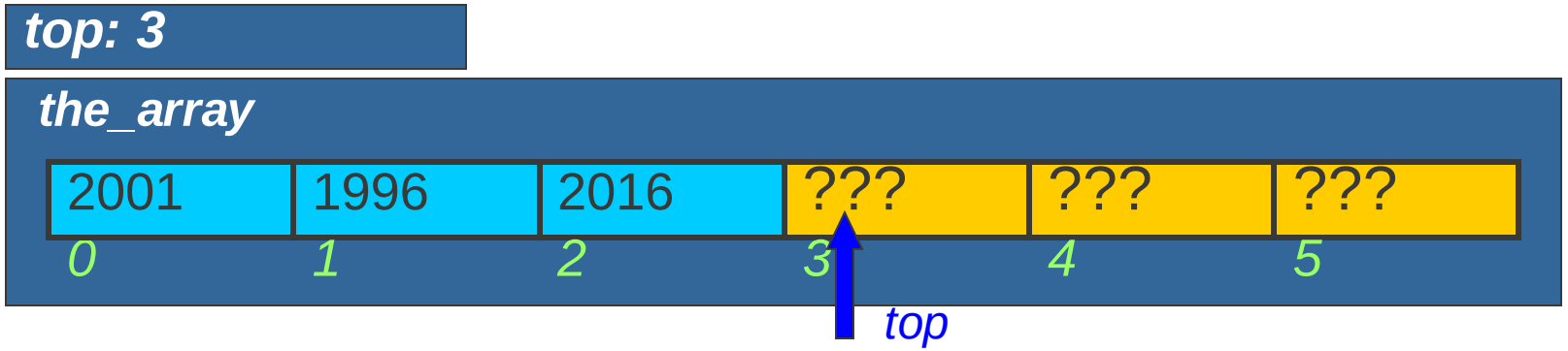
Same unpacking as for lists

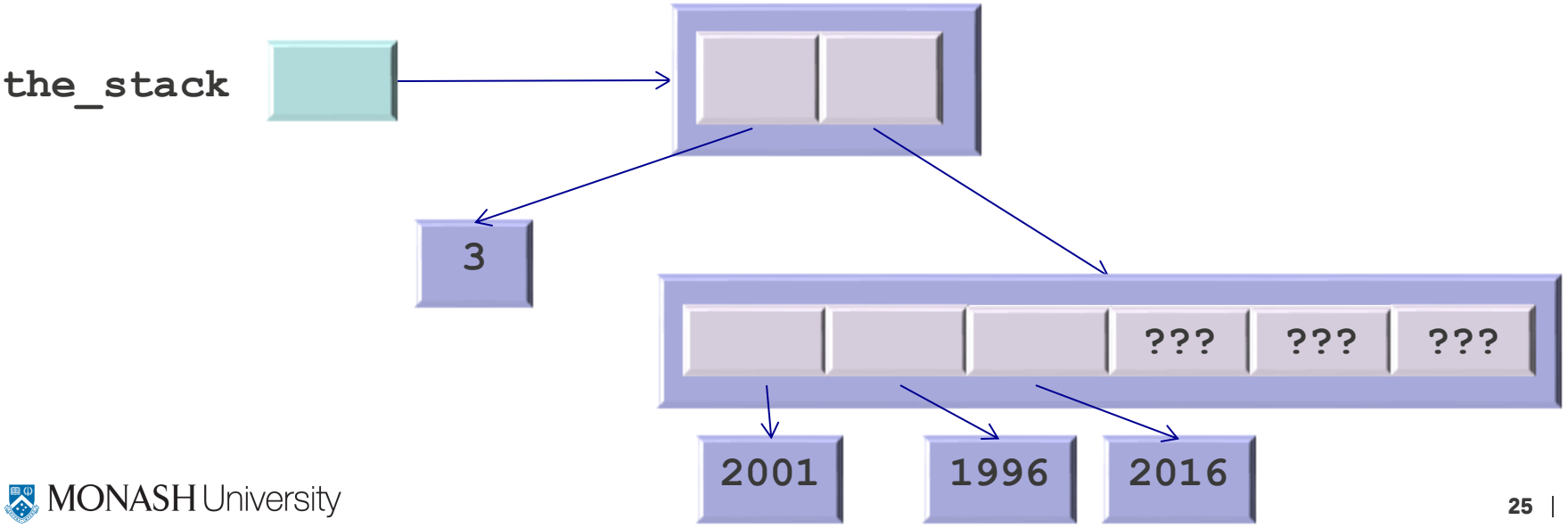
reusing `size`

Big O?

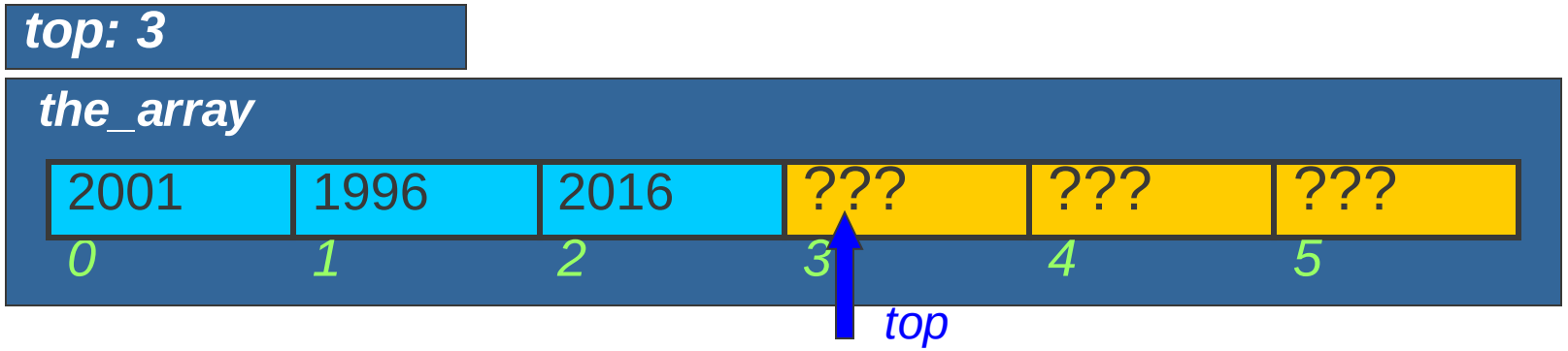
Important: no comments in slides due to lack of space.
BUT YOUR CODE MUST HAVE GOOD COMMENTS

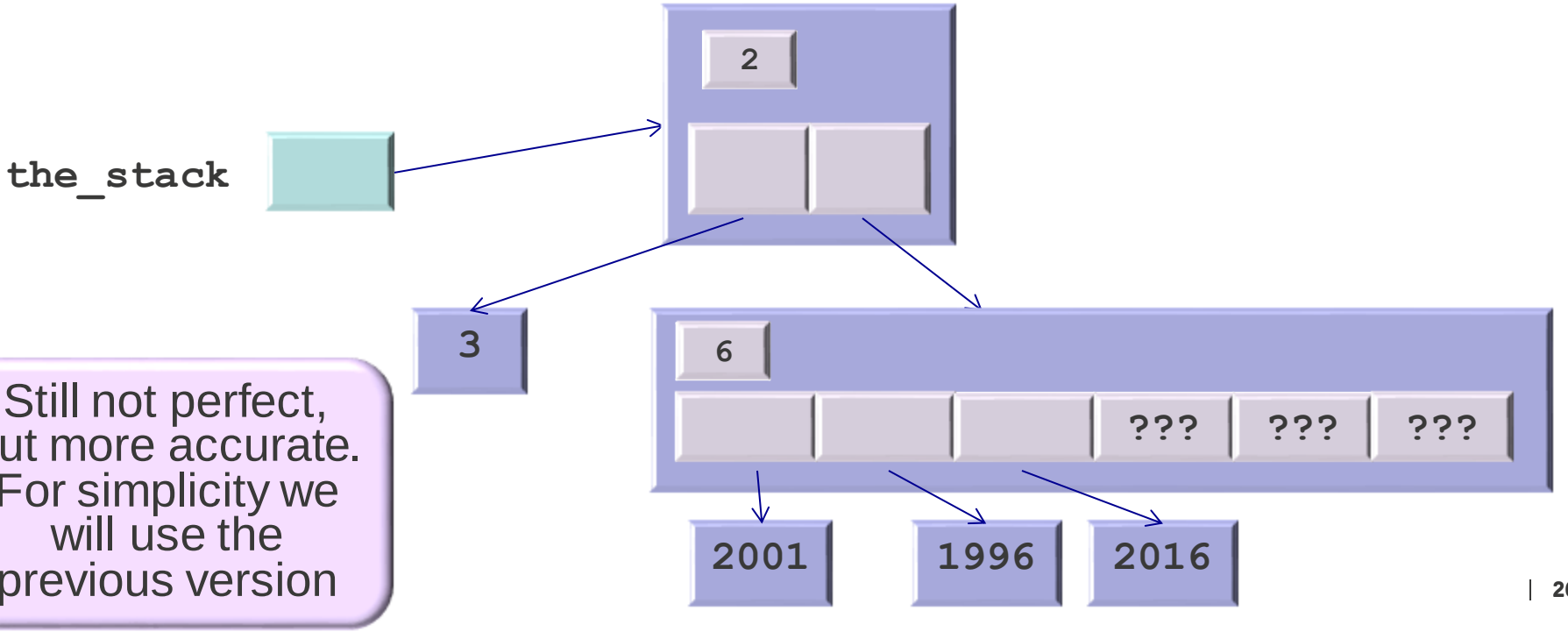
How is `the_stack` represented in memory?





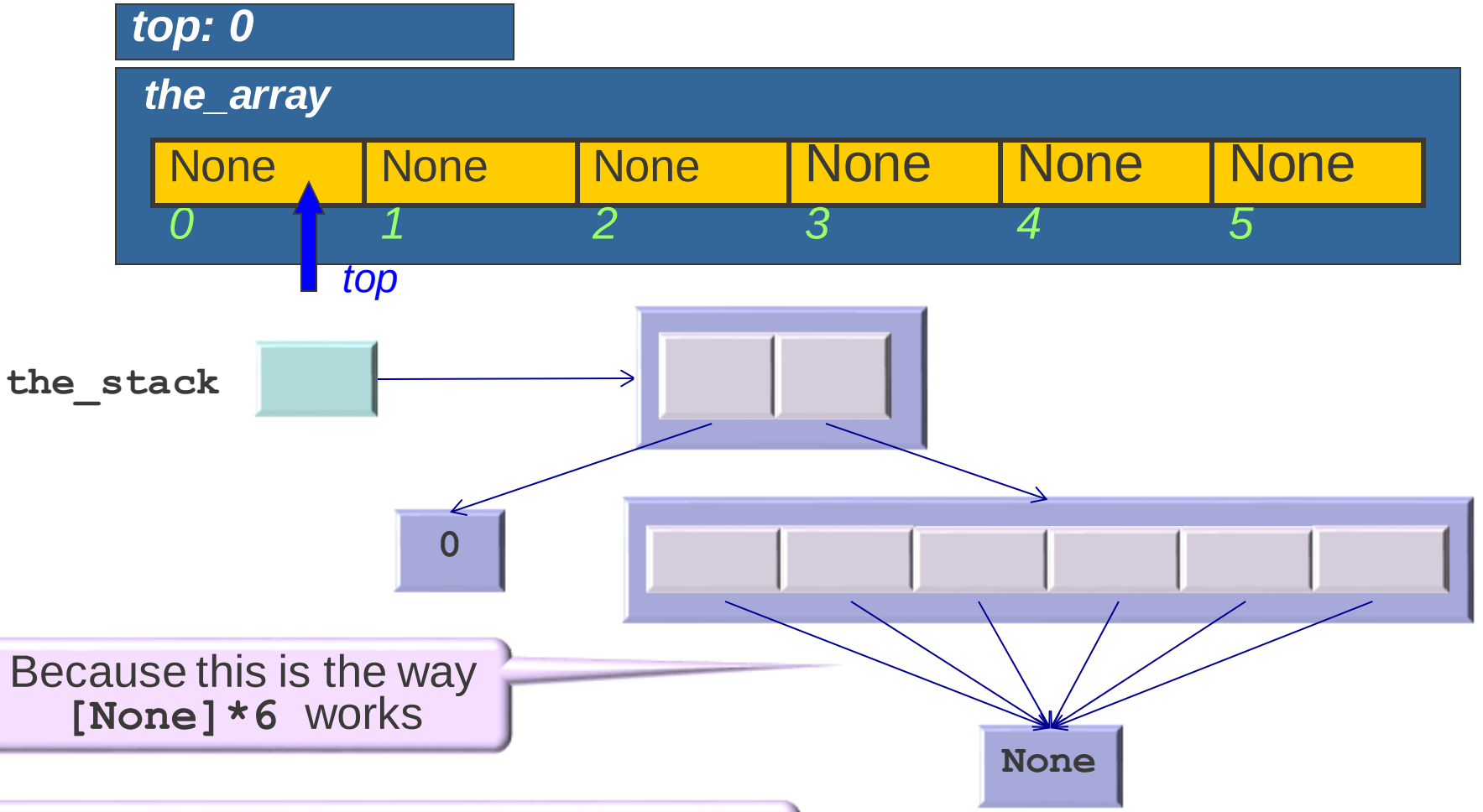
A more accurate representation





Still not perfect, but more accurate. For simplicity we will use the previous version

And the stack returned by `Stack(6)`?



Implementation for an array Stack

- **Let's now define `push`**
- **What do we do if the stack is full?**
  - Let's raise an exception

Why not return `False` as for lists?

Why not take as precondition that it cannot be full (and then use and assertion)?

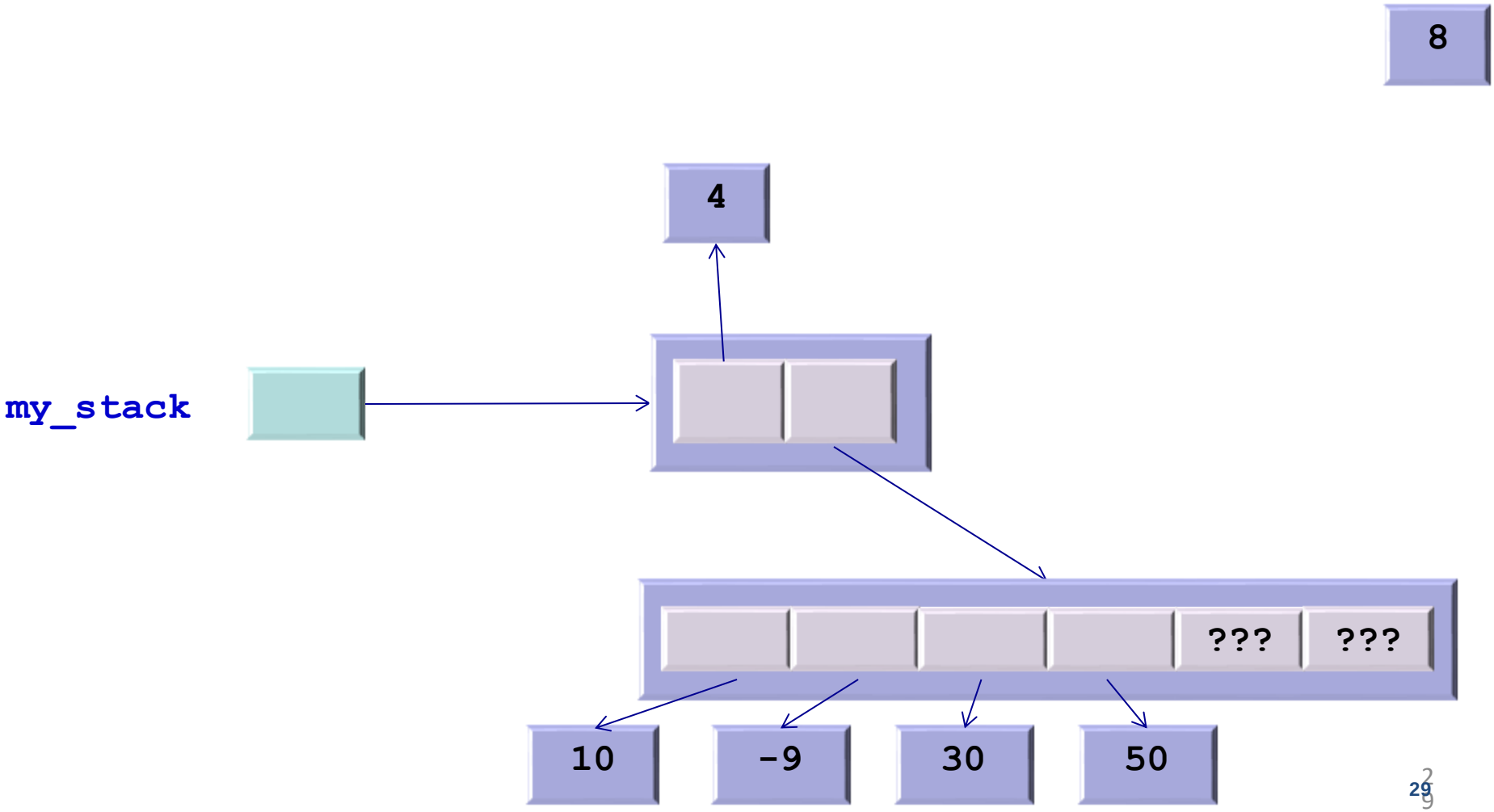
It is a design decision but, regarding preconditions: in real life, any pre-condition that you do not control (is not internal to your code) is better handled with exceptions

```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

reusing `is_full`: good

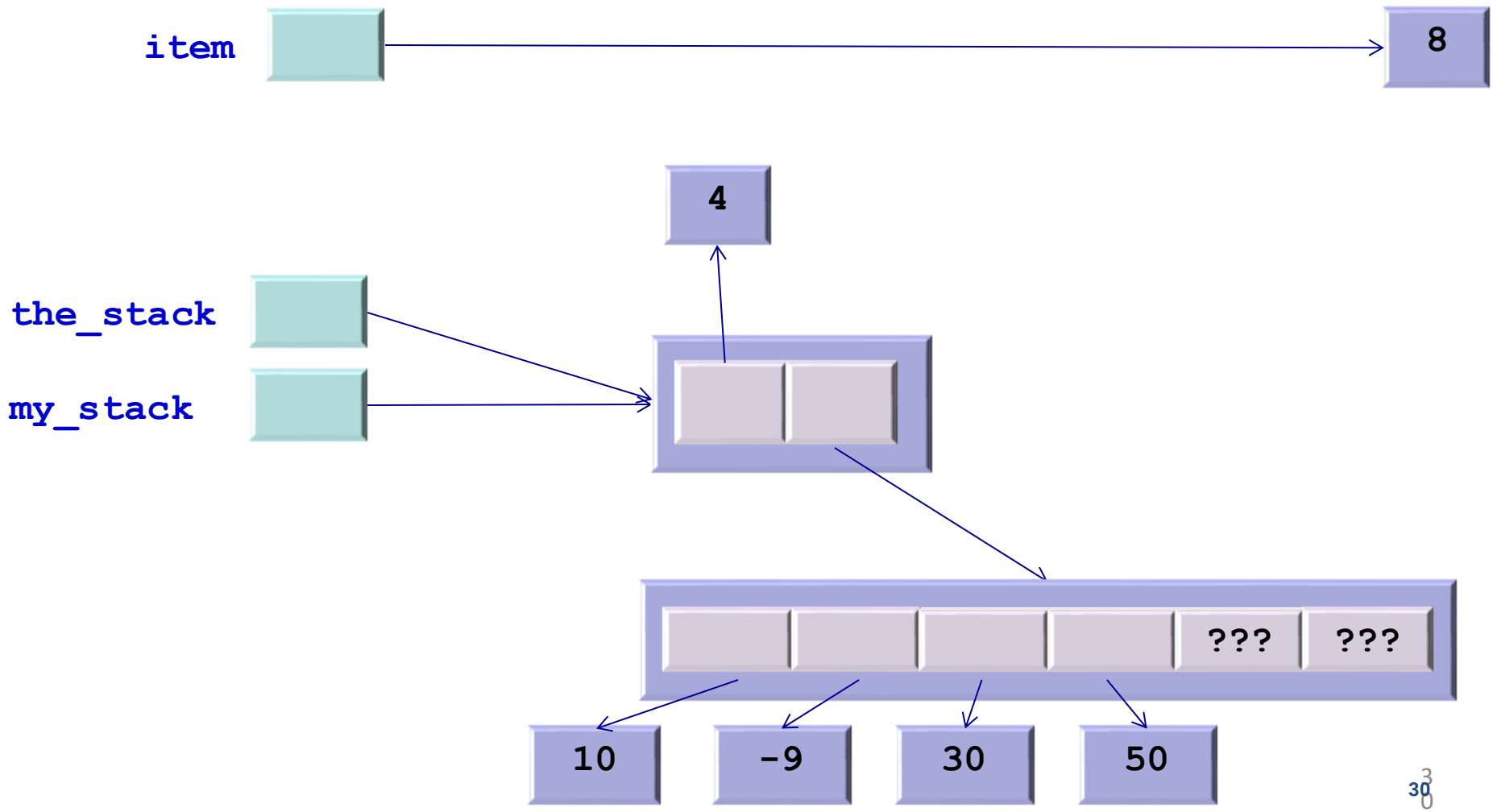
```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

push(my_stack,8)



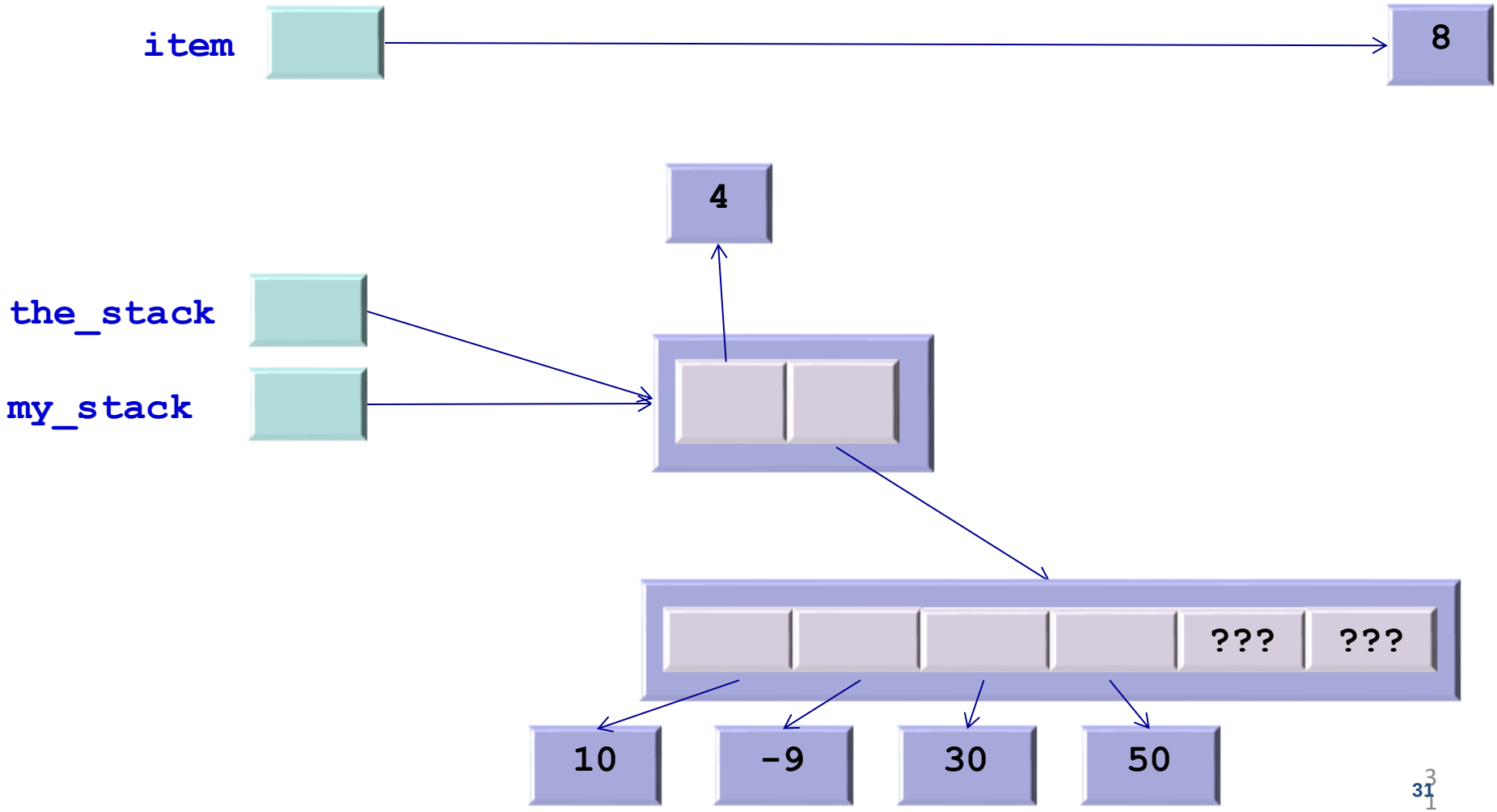
```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

push(my_stack,8)



```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

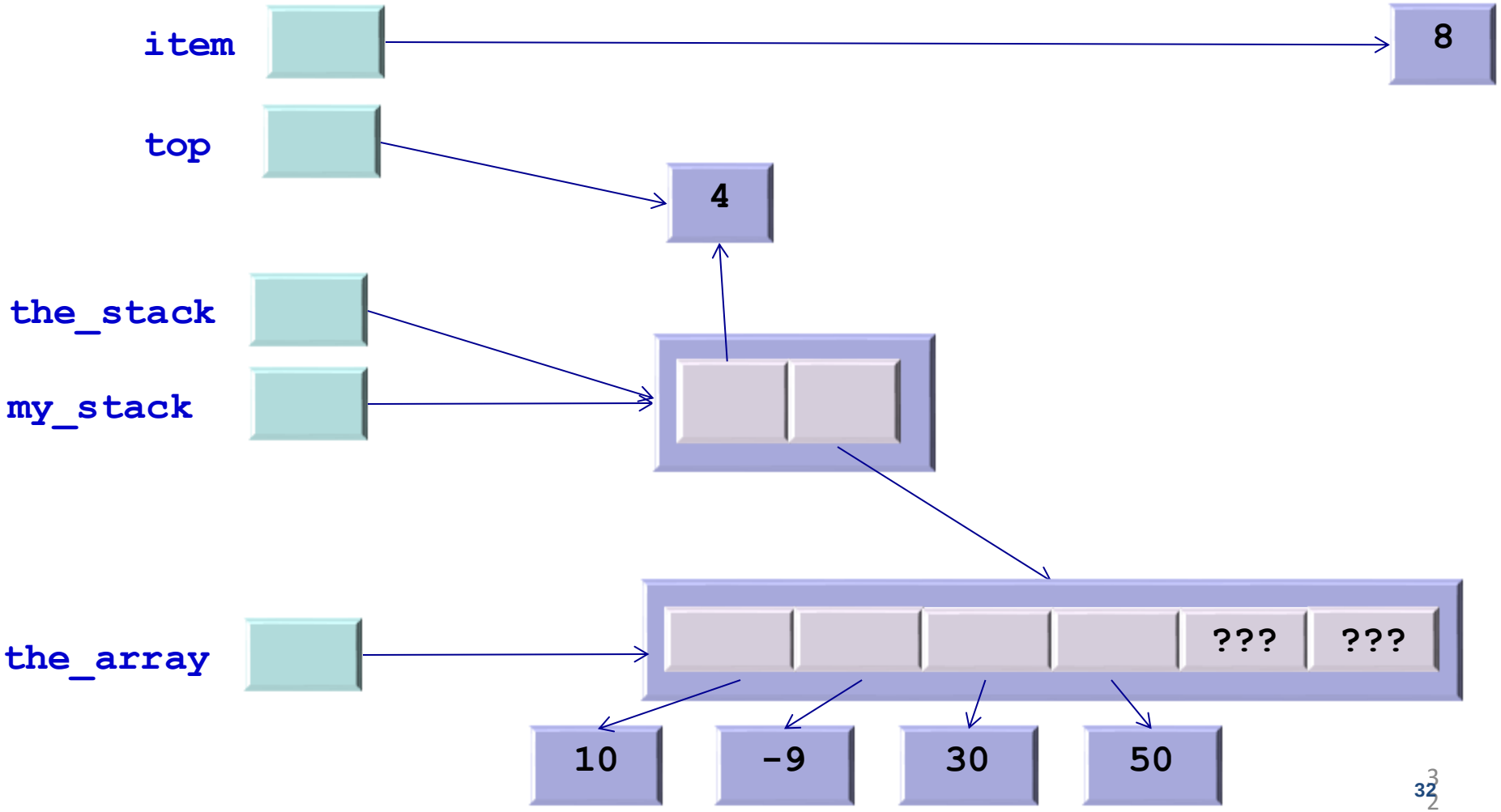
push(my_stack,8)



```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")
```

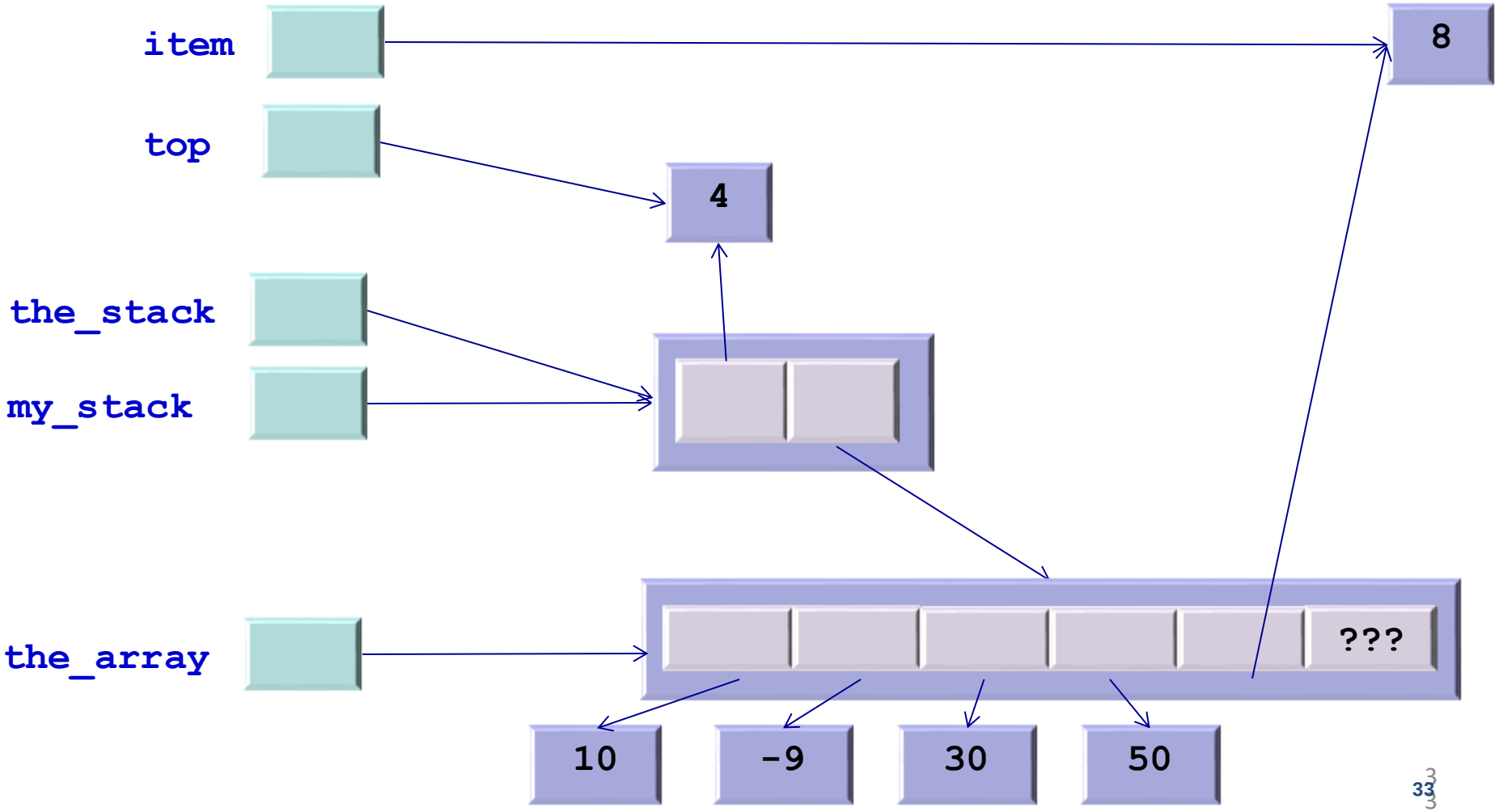
push(my_stack,8)

```
    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```



```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

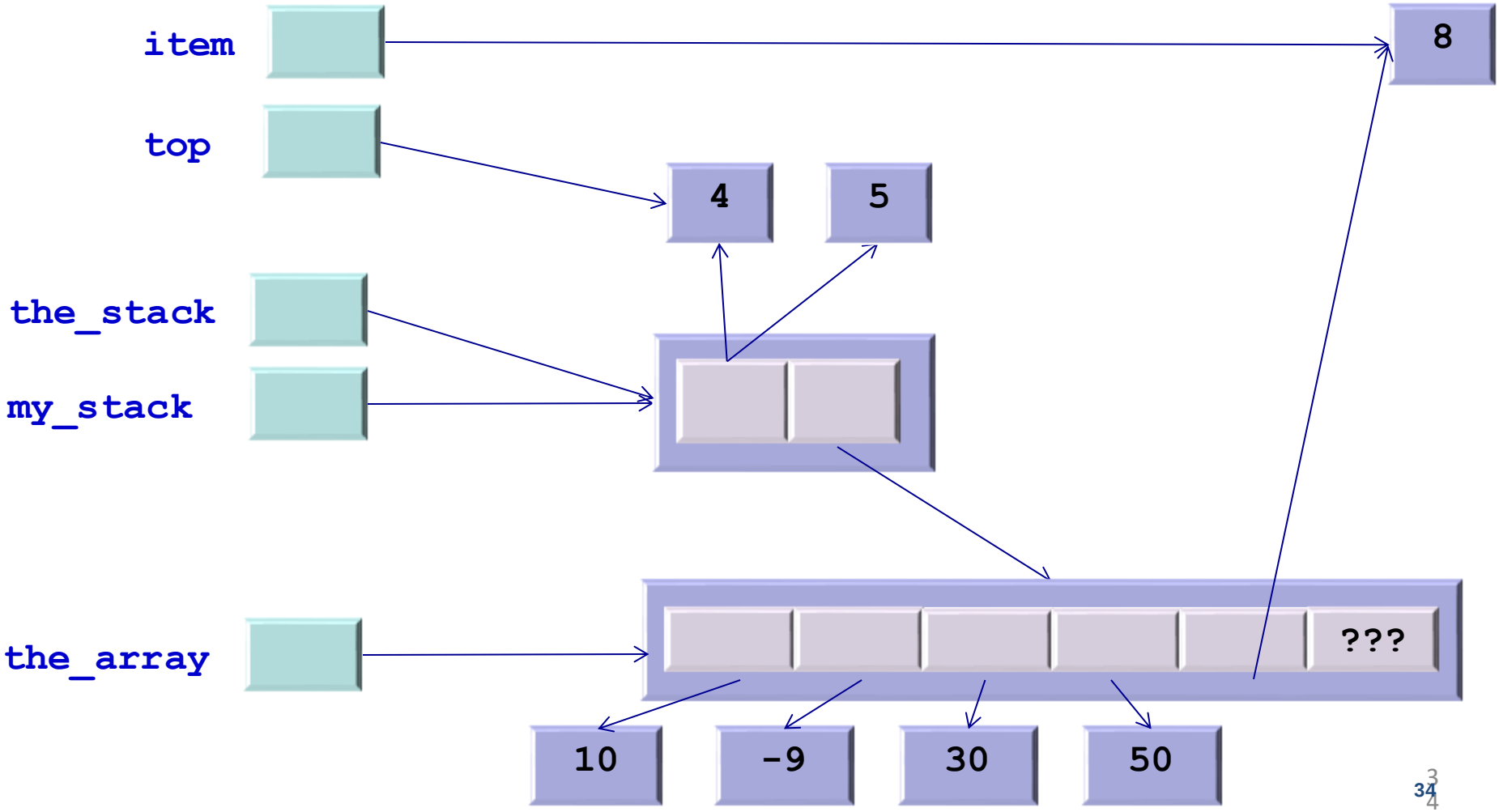
push(my_stack,8)



```
def push(the_stack, item):
    if is_full(the_stack):
        raise Exception("Stack is full")

    [top,the_array] = the_stack
    the_array[top] = item
    the_stack[0] = top + 1
```

Big O?

push(my_stack,8)



Implementation for an array Stack

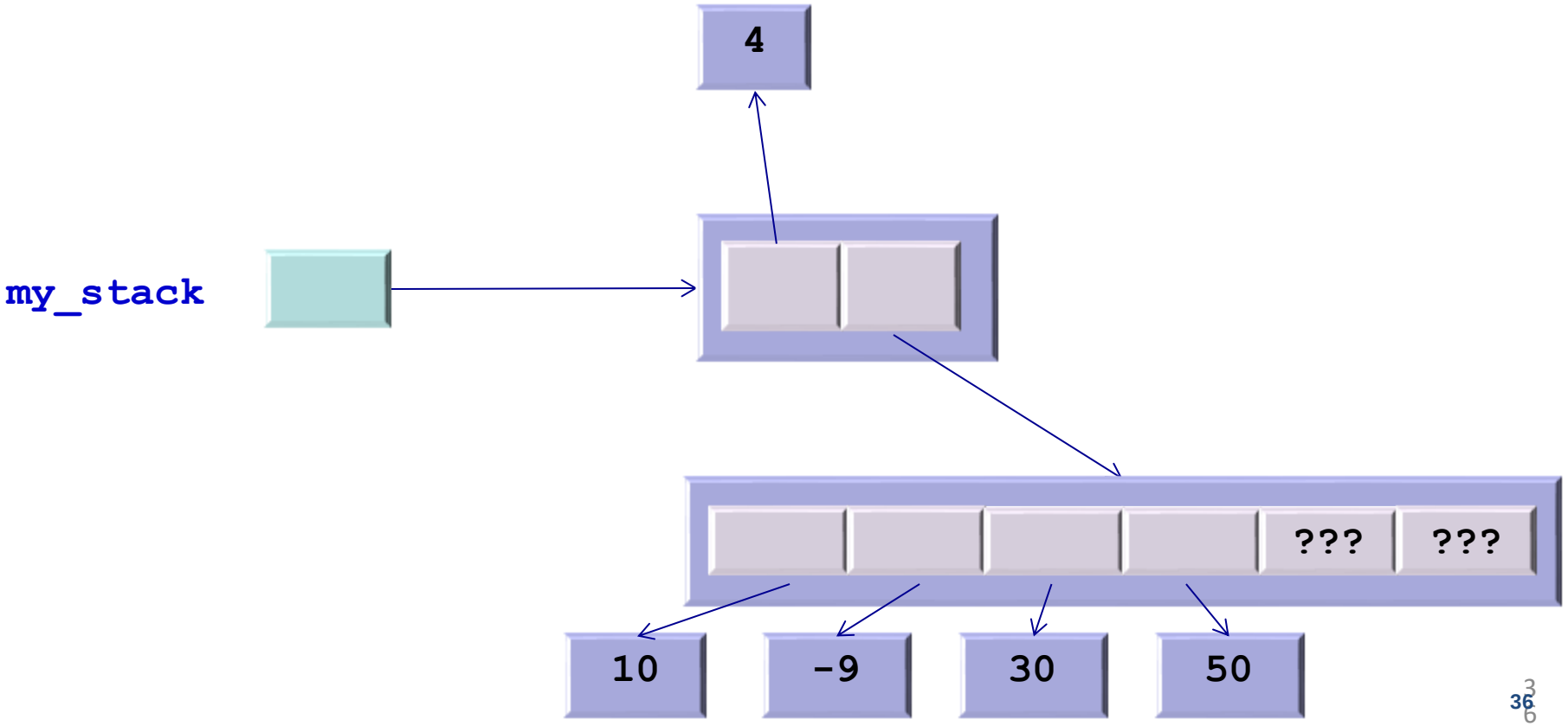
- **Let's now define `pop`**
- **What do we do if the stack is empty?**
  - Let's raise an exception

Again: design decision!

```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")

    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

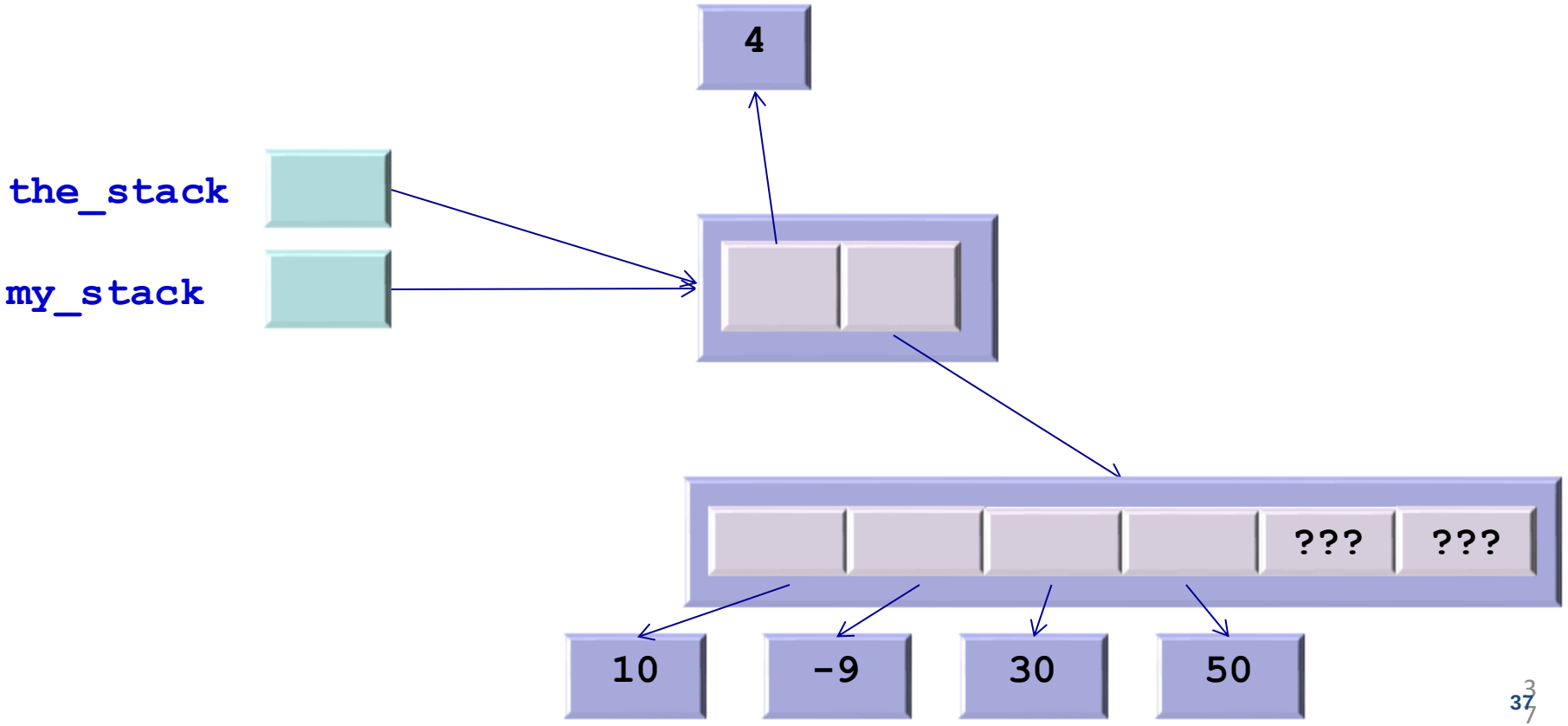
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty”)
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



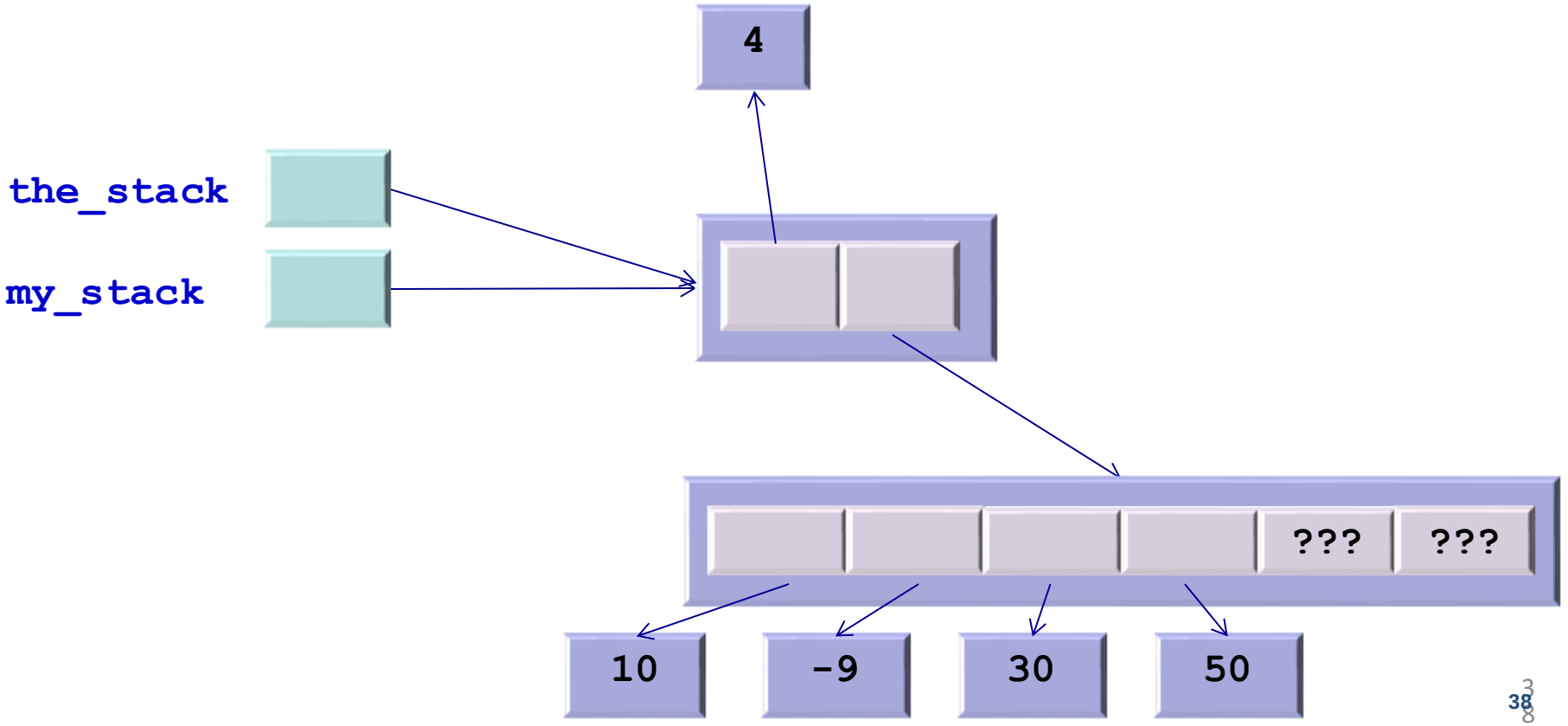
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty”)
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



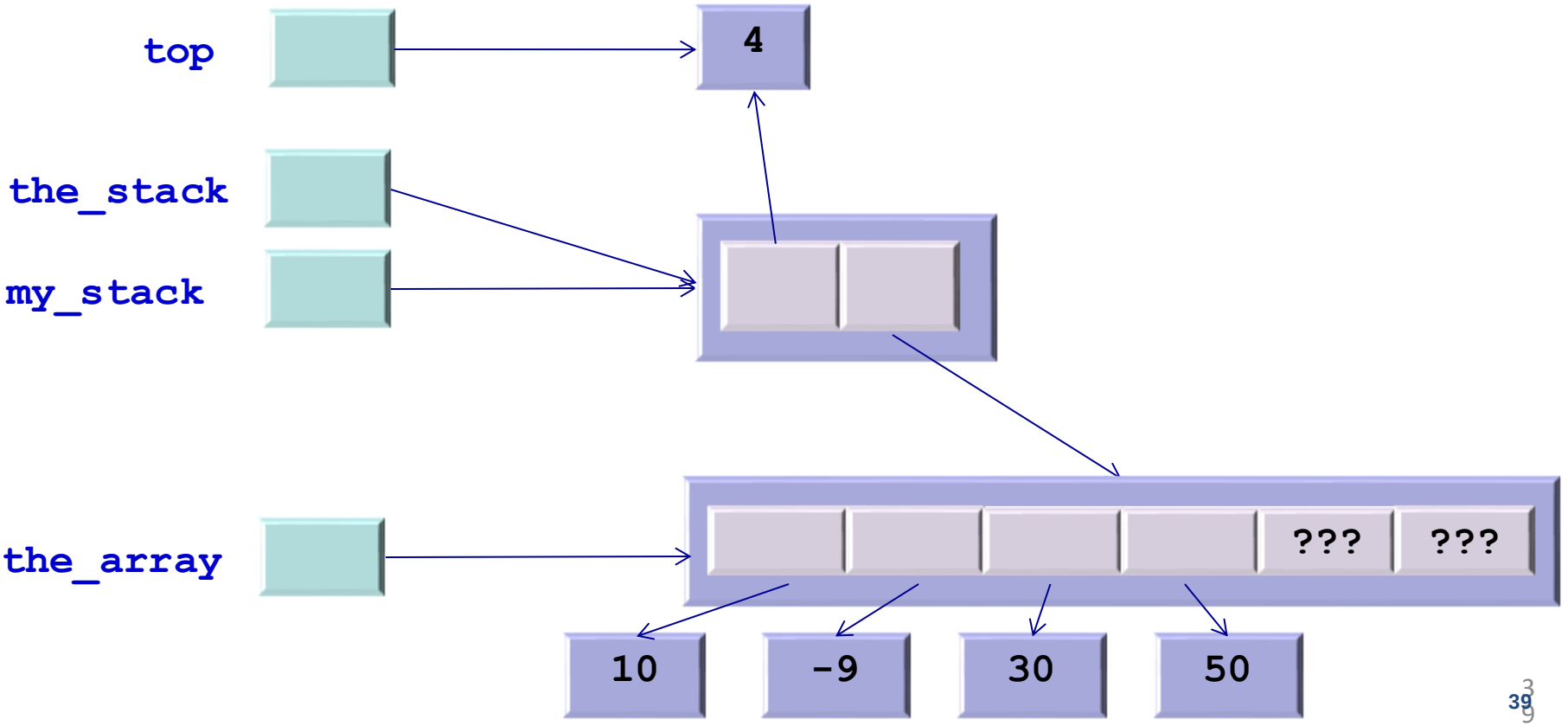
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty”)
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



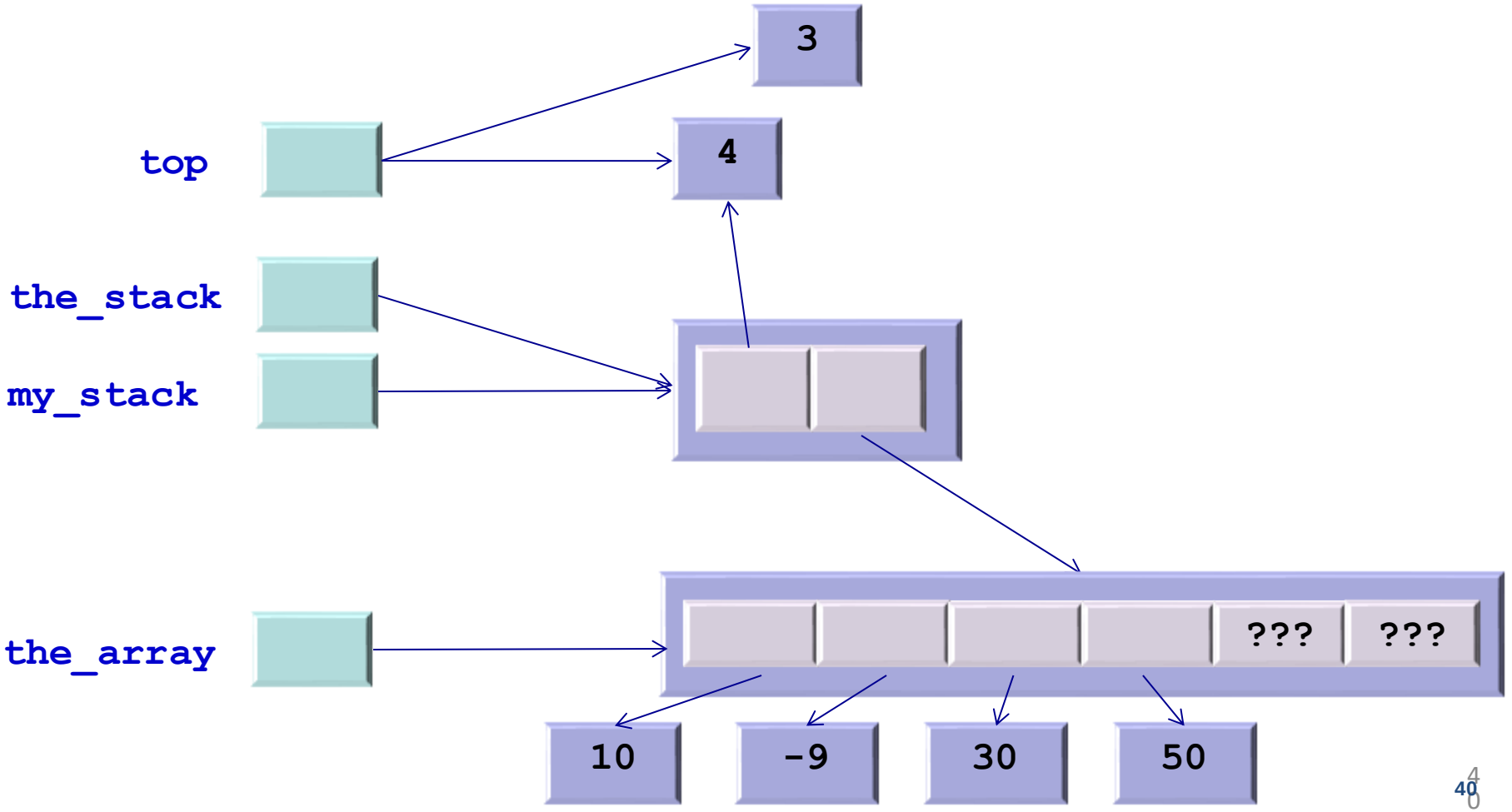
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty”)
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



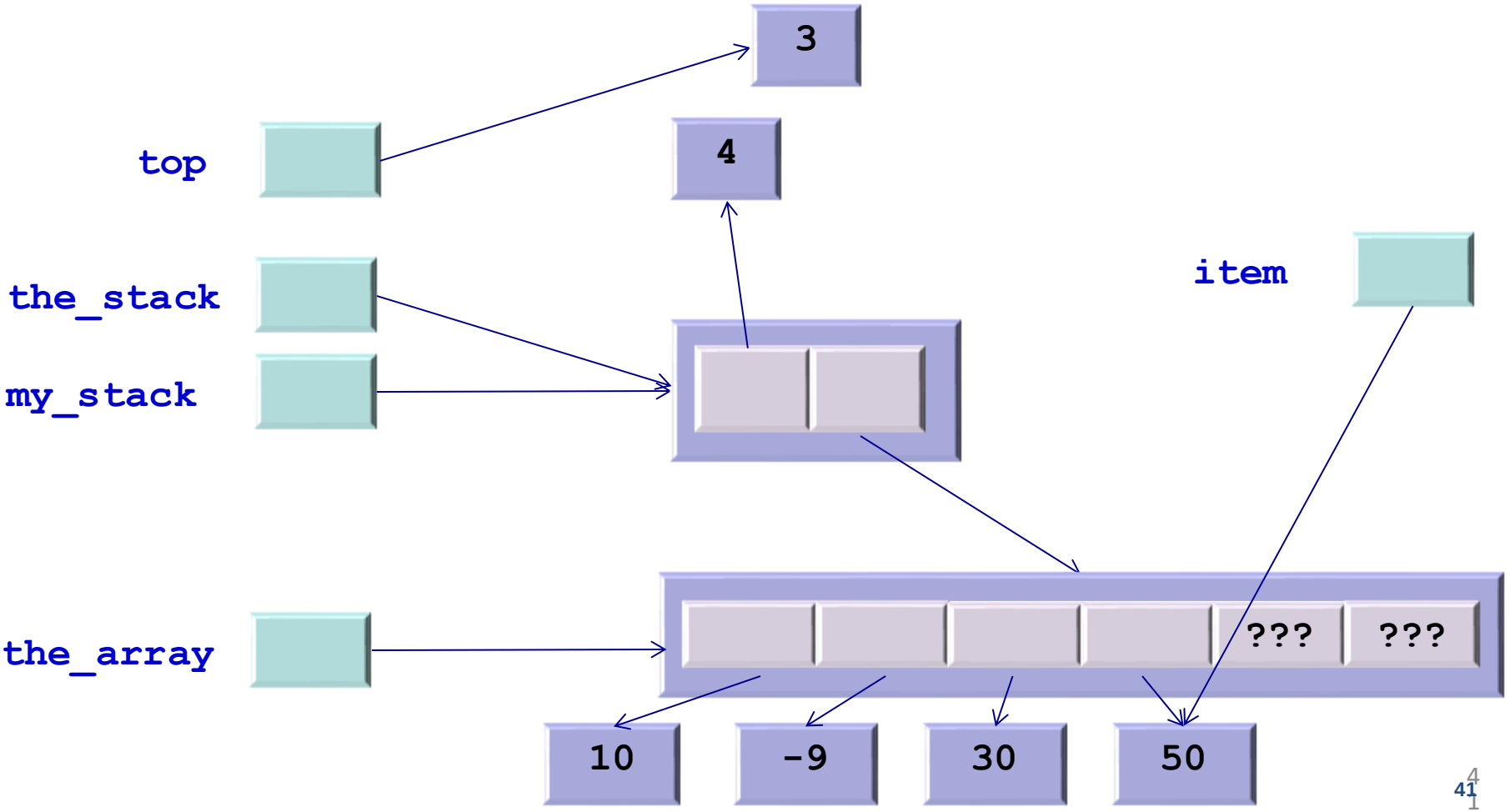
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



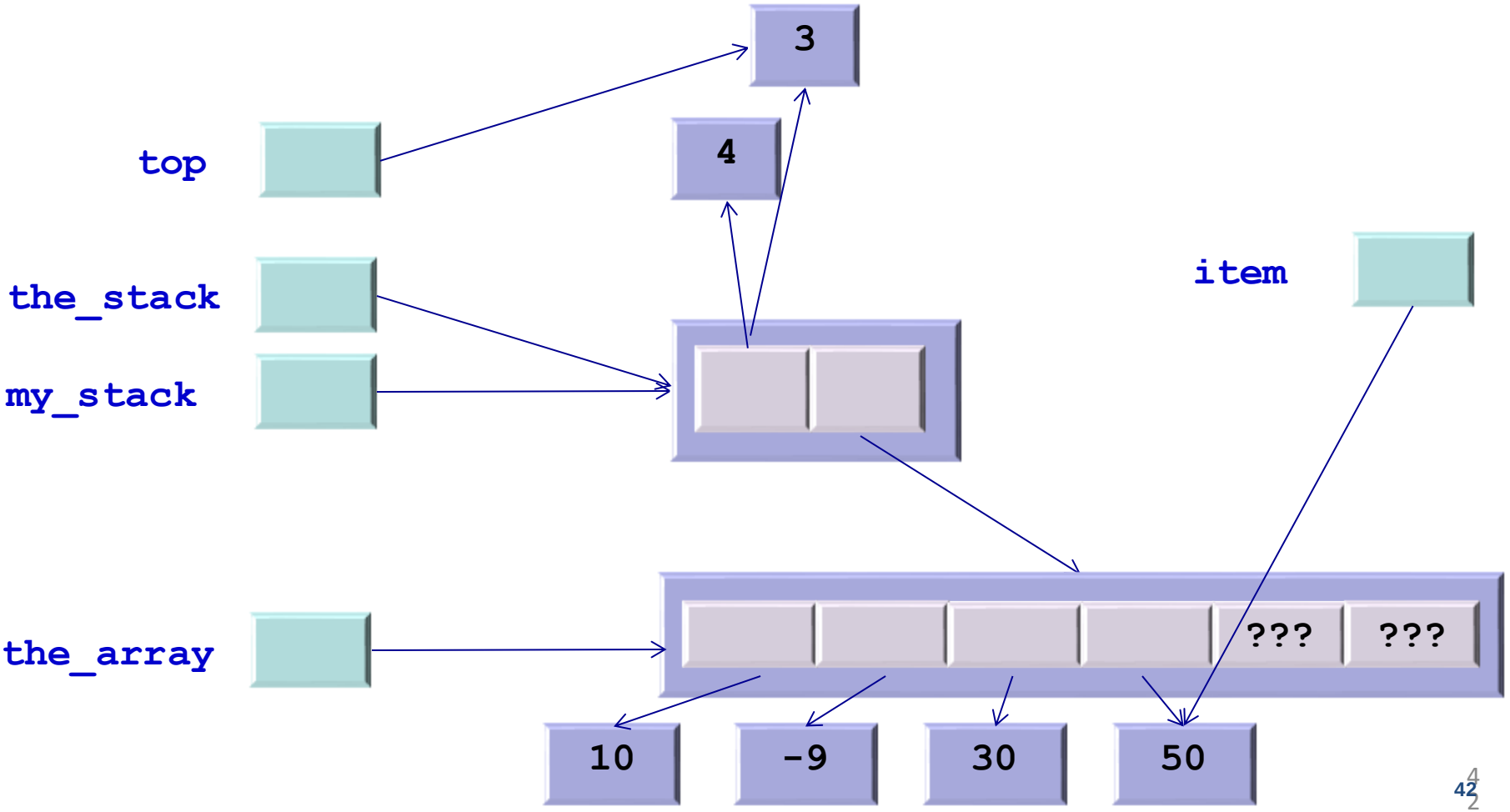
```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

pop(my_stack)



```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

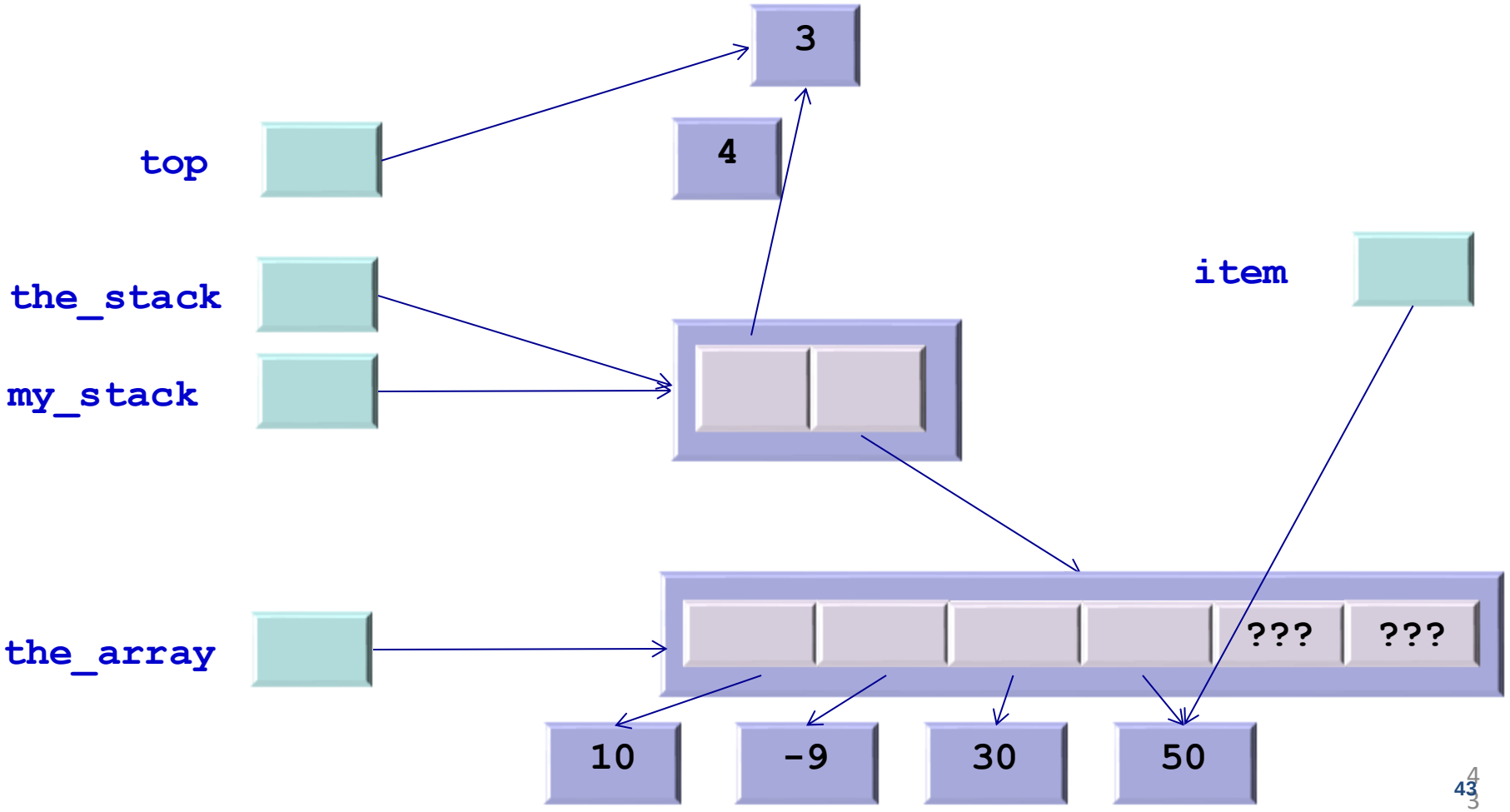
pop(my_stack)



```
def pop(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")
    [top,the_array] = the_stack
    top = top-1
    item = the_array[top]
    the_stack[0] = top
    return item
```

Big O?

pop(my_stack)



Implementation for an array Stack

- **Let's now define `peek` and `reset`**

```
def peek(the_stack):
    if is_empty(the_stack):
        raise Exception("Stack is empty")

    [top,the_array] = the_stack
    return the_array[top-1]
```

Big O?

```
def reset(the_stack):
    the_stack[0] = 0
```

What happens with all the info that was in the stack?

Example: reversing a sequence of items

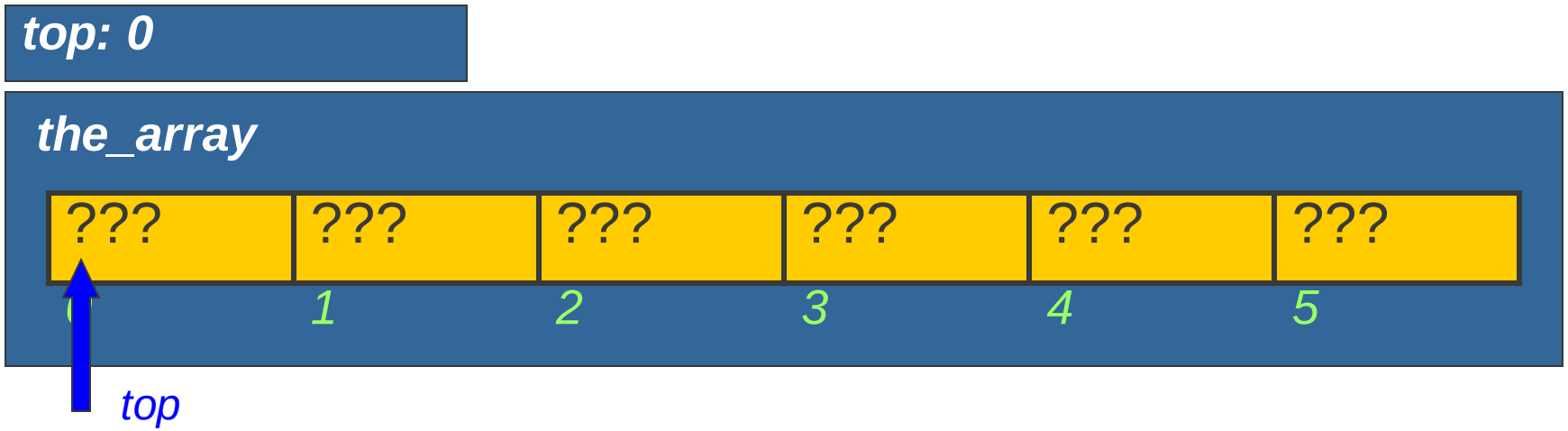
- Given a string as input

- Use a stack ADT to push each char onto the stack
- Once finished pushing all characters, pop each and concatenate it to a new string

- The resulting string has the same characters in reverse order

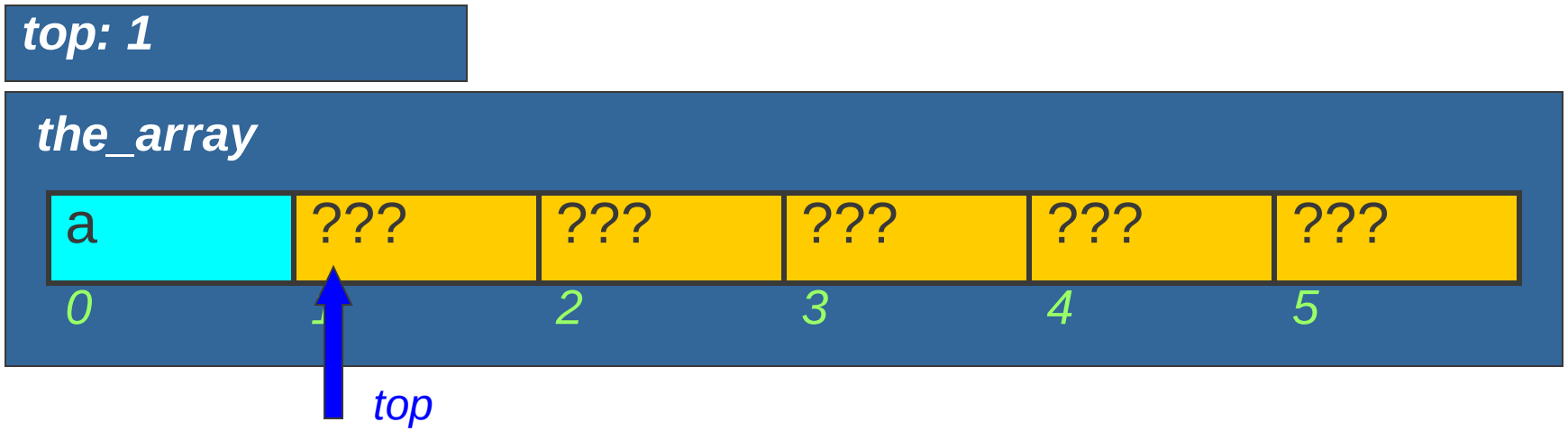
Example: reversing a sequence of chars

input string: “a b c d”



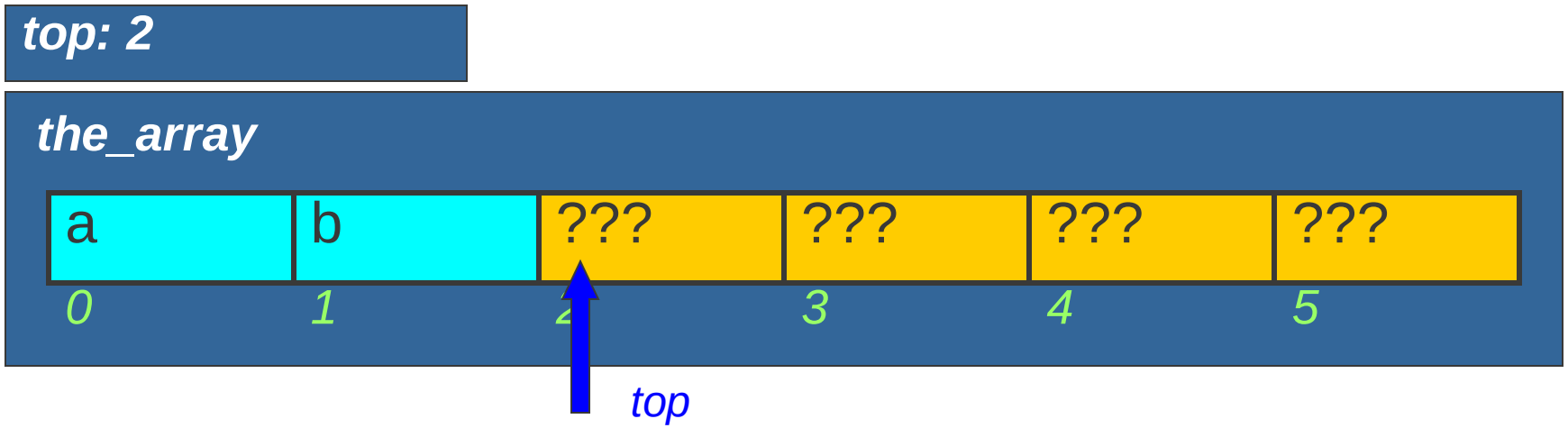
Example: reversing a sequence of chars

input string: “a b c d”



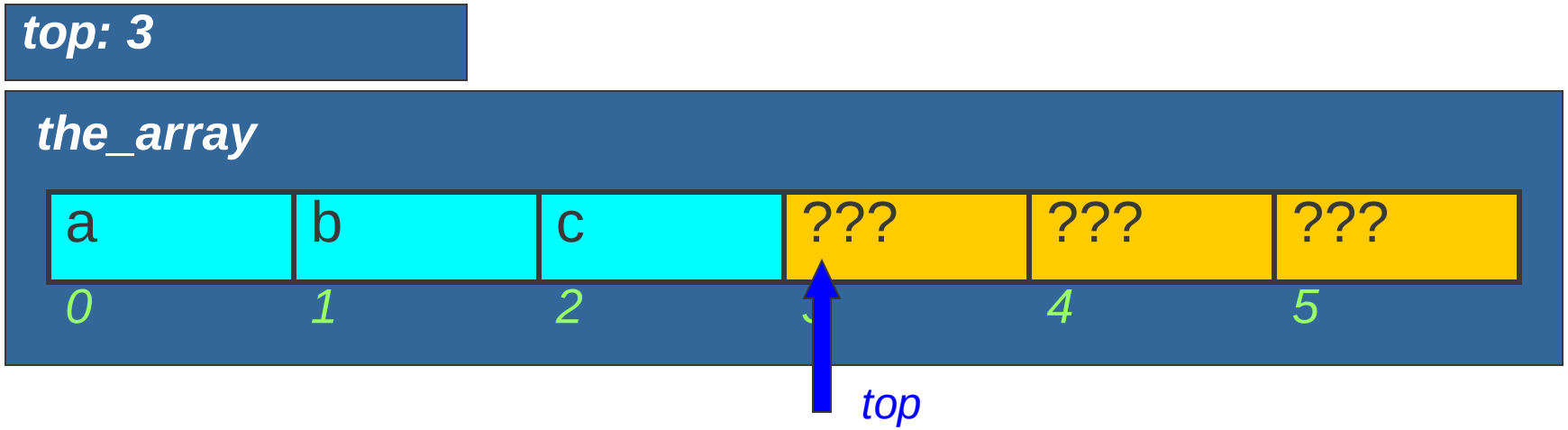
Example: reversing a sequence of chars

input string: “a b c d”



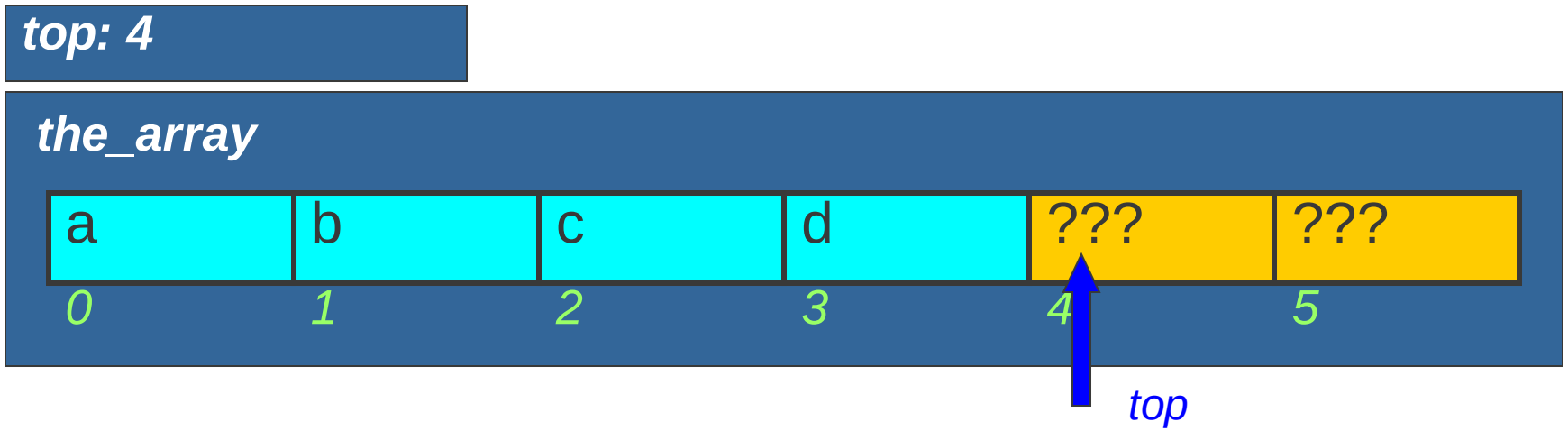
Example: reversing a sequence of chars

input string: “a b c d”



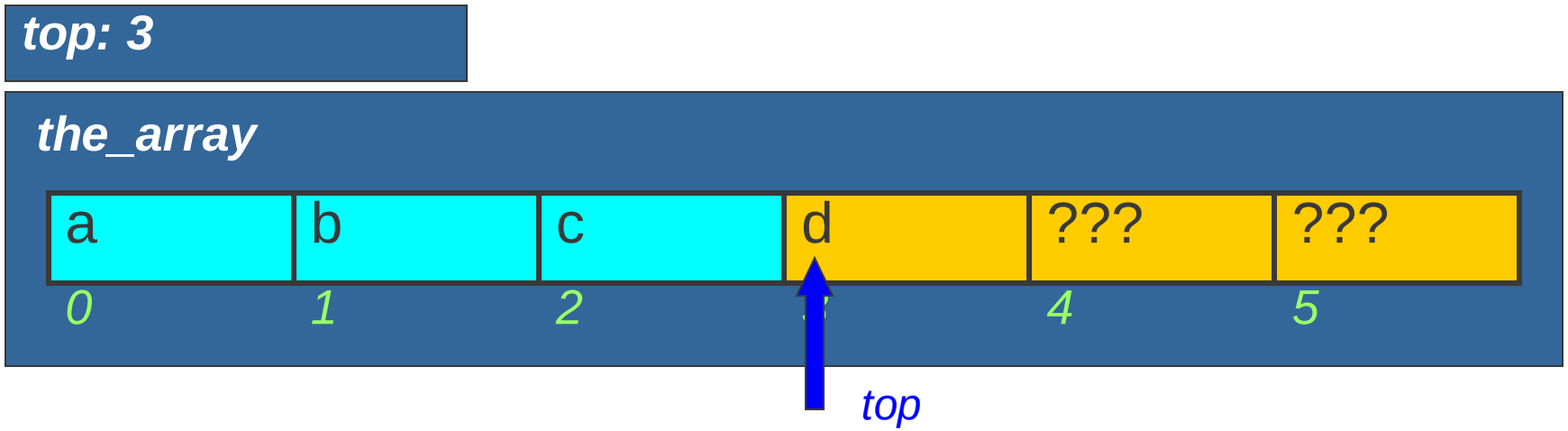
Example: reversing a sequence of chars

input string: “a b c d”



Example: reversing a sequence of chars

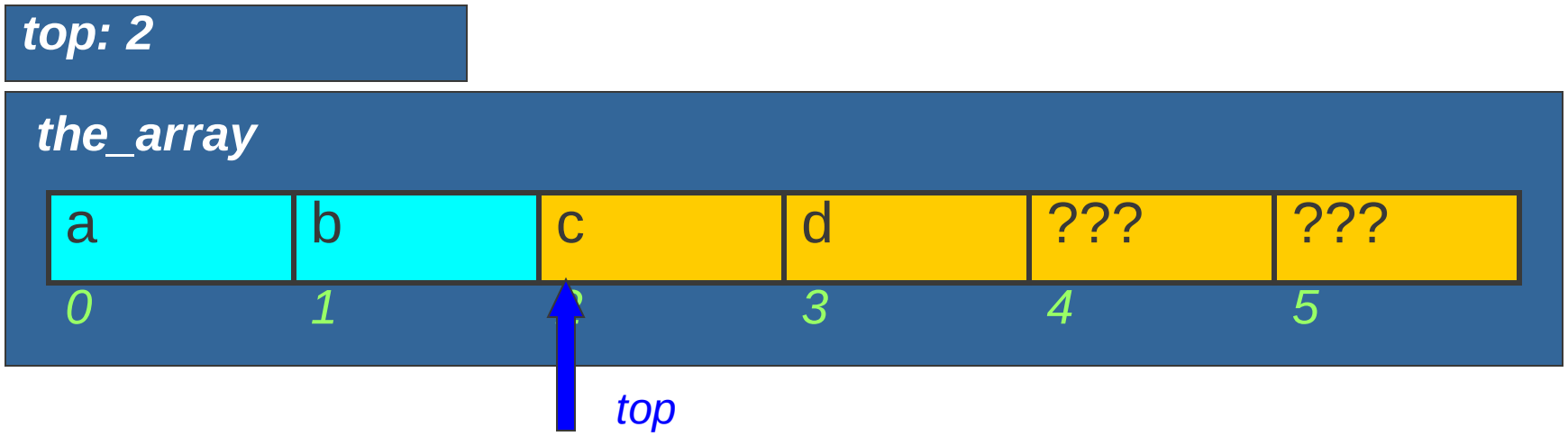
input string: “a b c d”



Output string: “d”

Example: reversing a sequence of chars

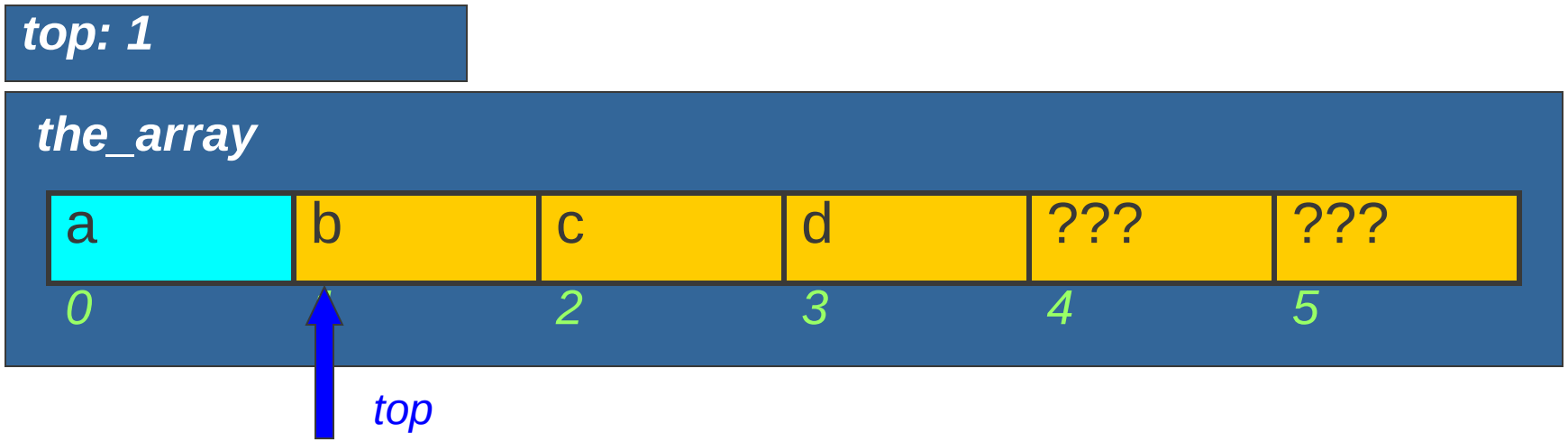
input string: “a b c d”



Output string: “d c”

Example: reversing a sequence of chars

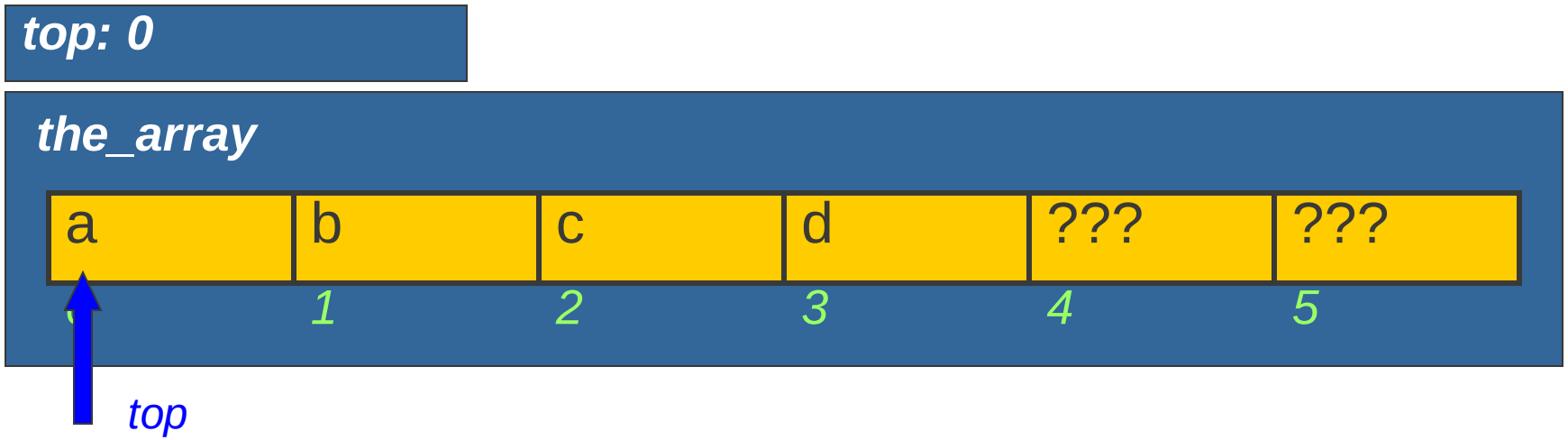
input string: “a b c d”



Output string: “d c b”

Example: reversing a sequence of chars

input string: “a b c d”



Output string: “d c b a”

Example: reversing a sequence of chars

- **Create a stack of the appropriate size**
  - Use `len(string)` to compute the length of the input string
- **Traverse the input string pushing each char onto the stack**
  - You can traverse strings as you can traverse lists
  - `for element in string`
- **Initialise the return string to empty " "**
- **Pop each element from the stack and concatenate it to the return string**
  - Remember: in Python, string concatenation is “+”
- **And remember: for this function you are a user of the stack ADT**
  - That is: you have no idea how it is implemented
  - Which means you can only use its functions
    - NOT the knowledge about how it is implemented with arrays

Example: reversing a sequence of chars

```
def reverse(input_string):
    stack_size = len(input_string)
    the_stack = Stack(stack_size)

    for item in input_string:
        push(the_stack, item)

    output = ""

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

    return output
```

pushing

stack_size times

Could also have been `for _ in input_string`

popping

stack_size times

Big O?

Complexity of reverse

- Two loops
- One executed after the other (+)
- Inside each loop, the number of operations is fixed (assuming + is O(1); if not, multiply by O_+)
- We get $K_1*n + K_2*n \approx n$
- Each loop is always performed exactly n times, regardless of the string
  - best = worst
- Which gives best = worst = O(n)

```
def reverse(input_string):
    stack_size = len(input_string)
    the_stack = Stack(stack_size)
    for item in input_string:
        push(the_stack, item)
    output = ""
    while not is_empty(the_stack):
        item = pop(the_stack)
        output += str(item)
    return output
```

Another Exercise

`a { b c ( d e f [ g ] h t ) l m [ o ] p }` **Balanced**

`a { b c ( d e f } g [ h ] ) l m } o p` **Not balanced**

`{ [ (  ) ] }`

Open **Close**

Some Stacks Applications

- **Undo editing (think about your lab …)**
- **Parsing**
  - Reverse polish notation
  - Delimiter matching
- **Run-time memory management**
  - Stack oriented programming languages (MIPS)
  - Virtual machines
  - Function calling
- **Implement recursion**

Summary

- **Stacks ADT:**
  - Array implementation
  - Basic operations
  - Their complexity
  - Applications