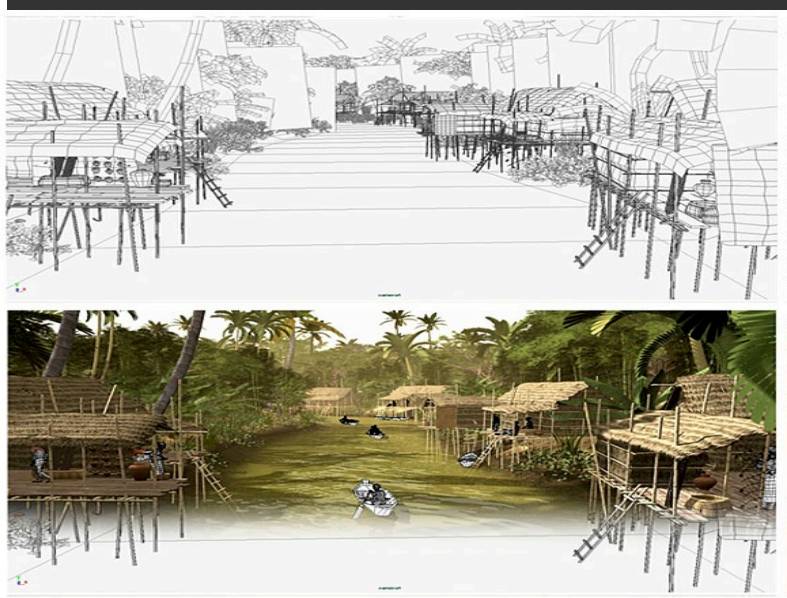
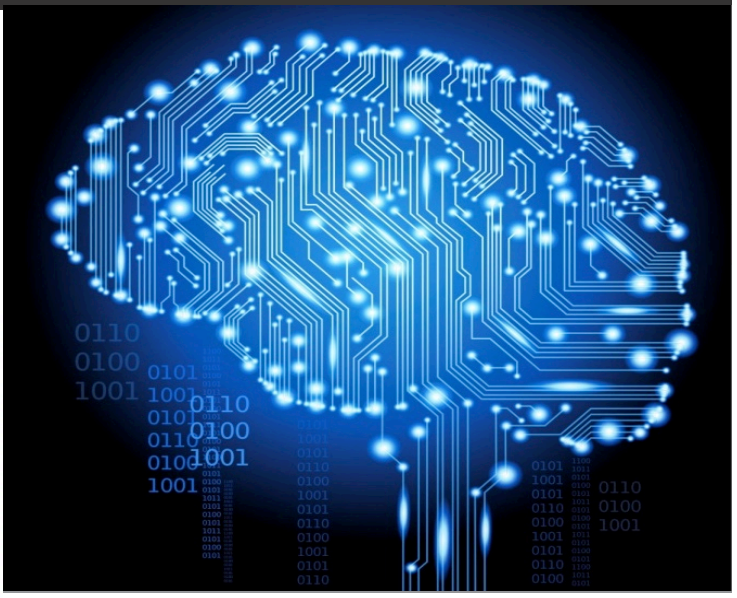
MONASH University

Information Technology

Abstract Data Types and Stack ADT

Prepared by:
Maria Garcia de la Banda
Revised by Pierre Le Bodic





Objectives for this lesson

- **Understand the concepts of**
  - Data Types
  - Abstract Data Types (ADTs)
  - Data Structures
- **Understand the Stack ADT:**
  - Main operations
  - Their complexity
- **To be able to:**
  - Implement Stacks with arrays
  - Use them
  - Modify its operations and
  - Reason about their complexity

MONASH
University

Abstract Data Types

Data Types

- **Common concept in lower level languages (C, Java, …)**
- **Refers to a classification that determines:**
  - The possible values for that type
  - The meaning of those values
  - The operations that can be done on them
  - The way those values are implemented
- **Example: if a Java variable has type `int`**
  - Can take values from -2,147,483,648 to 2,147,483,647
  - Their meaning is that of an integer number
  - Can be used in all integer operations (add, subtract, etc)
  - Implemented using 32 bits and specific bytecode operations

Data Types

- **Knowing the implementation can have advantages**
  - Extra functionality
  - Speed
  - …
- **E.g., some languages implement `False/True` as `0/+`**
  - AND allow programs to use arithmetic with Booleans
- **Using the implementation can have disadvantages:**
  - Lack of portability
  - Poor maintenance (that often lead to bugs)
  - …

Abstract Data Types (ADTs)

- **Often no need to know how types are implemented**
  - Data abstraction
- **An abstract data type:**
  - Provides information regarding:
    - The possible values of the type and their meaning
    - The operations that can done on them
  - BUT *not* on its implementation, i.e. how:
    - The values are stored
    - The operations are implemented
  - Users interact with the data only through the provided operations
- **In some languages, concept of abstraction is mixed with hiding**
  - Like in Java: actively hides implementation (e.g., `private`)

ADTs look very much like classes, right?

Advantages of ADTs

- **Build programs without knowing their implementation**
  - Simplicity!
- **The implementation can change without affecting you**
  - Maintenance!
- **If several ADTs available, you could easily use any**
  - Flexibility and reusability!
- **Different compilers can use different implementations**
  - Portability!

Data Structures

- **At some point we must give ADTs an implementation**
- **Some ADTs (like lists) contain several data fields**
  - How do we organise the data? How do we access it?
- **That is what a data structure provides:**
  - A particular way in which data is physically organised (so that certain operations can be performed efficiently)
- **Example: the array data structure**
  - Fixed size
  - Data items are stored sequentially
  - Each item occupies the same amount of space
- **That looks VERY much like a Python list:**
  - Because Python lists are implemented using arrays

Physical organisation

This allows constant time access to any element (remember your MIPS!)

This is becoming confusing!

- **We have already talked about**
  - Data types
  - Data structures
  - Abstract Data types
- **And this is only part of the picture, we also have:**
  - Primitive (or built-in) data types versus user-defined
    - Readily available in a given programming language or not
  - Simple (or basic, or atomic) versus complex ones
    - Single data versus multiple data fields
- **What is the relation between them? It is all about:**
  - Abstraction level
  - Simple/complex data (single/multiple data)

A way to clarify things a bit (not gospel)

Higher level language (Python, Scribble)	Only ADTs (no details about implementation)	Primitive simple ADTs: integers, booleans,... Primitive complex ADTs: lists, strings, … Non primitive simple/complex ADTs: users can add any ADT they want.
Mid-level (Java)	Primitive data types plus user/library defined ADTs.	Same as below, plus non primitive simple/complex ADTs (implementation is ignored or even hidden)
Lower level language (C, Fortran)	Primitive data types (both simple and complex). Details of implementation are known.	Primitive simple data types: int, short, float… Primitive complex data types (called data structures): arrays, strings Non primitive simple/complex data types: users can add anything like time, linked lists, array lists…
Assembly language instructions	32-bits registers and a few operations on them	Primitive simple data types: 8-, 16-, 32-bit signed/unsigned integer, float.
Hardware implementation	Bits and logic circuits	No real concept of type: bit, bytes, word …

This is for those interested in language evolution, but not examinable

Just remember that in this unit, we say:

- **Abstract Data types provide information about**
  - The possible values for that type
  - The meaning of those values
  - The operations that can be done on them
  - **Example: lists and stacks (however they are implemented – don't care)**
- **Data Types provide the same info plus:**
  - The way those values are implemented
  - **Example: a stack for which I know (and make use of) how it is implemented**
- **Data Structures provide information about:**
  - The way in which data is physically organised in memory
  - The only operation it provides is access
  - **Example: an array in MIPS or a linked node**

This is language independent. Thus, independent of OO concepts

MONASH
University

Stack Abstract Data Type

What is a Stack Abstract Data Type (or Stack ADT)?

- **An ADT that is used to store items?**
- **That is very vague! What makes it a Stack?**
- **The main two factors (which you have seen in MIPS' run-time stack) are:**
  1. Its operations follow a Last In First Out (LIFO) process
     - The last element to be added, is the first to be deleted
  2. Access to any other element is unnecessary (and thus not allowed)
     - If you need to access another element, choose a different ADT…

Main Stack Operations

- **The key operations are:**
  - Create a stack (Stack)
  - Add an item to the top (push)
  - Take an item off the top (pop)
- **Other common operations include:**
  - Look at the item on top, don't alter the stack (top/peek)
  - What is its length?
  - Is the stack empty?
  - Is the stack full?
  - Empty the stack (clear)

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

Implementing the Stack ADT

- **In this block we will learn to define and implement our own Stack ADT**
  - Why on earth? They are already in Python!
- **Because the learning objectives of this unit require you to:**
  - Learn to implement the operations yourself
    - You might need to program on a device with limited memory
  - Reason about the properties of these operations
  - Understand the changes in properties depending on implementation
- **What data structure do we use to implement it?**
  - We will start with arrays (and later we will re-implement with linked nodes)
- **Wait! In an OO context, should we go directly to the array implementation?**
  - No! Better to start with a general parent class that works for all implementations
  - So let's create a base stack class in file **`stack.py`**

A possible base Stack class

`Generic` let's us to use type vars

Remember: CapWords convention for class names

Blue marks method names

`pass` indicates abstract methods

```
from typing import TypeVar, Generic
T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        self.length = 0

    def push(self, item: T) -> None:
        pass

    def pop(self) -> T:
        pass

    def peek(self) -> T:
        pass

    def __len__(self) -> int:
        return self.length

    def is_empty(self) -> bool:
        return len(self) == 0

    def is_full(self) -> bool:
        pass

    def clear(self) -> None:
        self.length = 0
```

Imports two types from `typing`

Uses `TypeVar` to create type variable T; will represent the type of the elements

Only initializes the `length`. The rest will depend on the implementation

Could increment the `length` but does not seem worth it (or clear). Instead we leave it as abstract

Could decrement the `length`, but not returning anything seems wrong

This implementation is complete

So is this one. Note we are calling len (reusing method definitions is good)

This one is abstract

This one might need more, depending on the implementation

Big O of complete methods in Stack

```
from typing import TypeVar, Generic
T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        self.length = 0

    def push(self, item: T) -> None:
        pass

    def pop(self) -> T:
        pass

    def peek(self) -> T:
        pass

    def __len__(self) -> int:
        return self.length

    def is_empty(self) -> bool:
        return len(self) == 0

    def is_full(self) -> bool:
        pass

    def clear(self) -> None:
        self.length = 0
```

All return statements and assignments are always constant, so `__init__` `__len__` and `clear` are O(1)

Integer comparison is also always constant and `__len__` is O(1), so `is_empty` is also O(1)

No element properties can change that, so best = worst!

Also, if the worst-case is O(1), the best case much be O(1) too!

Something does not seem right

- **As defined, the base Stack class is not meant to be instantiated**
  - This should happen only when all methods are implemented by derived classes
- **But we can instantiate it!**
  - `Stack()` does create a stack object
- **Which doesn't behave like a correct stack**
- **How can we avoid this happening? Easy!**
- **First declare the class as abstract**
  - Using the `abc` module
- **Then, declare non fully implemented methods as abstract methods**
  - Using the `@abstractmethod` decorator

```
>>> from stack import Stack
>>> my_stack = Stack()
>>> my_stack
<stack.Stack object at 0x10ac33fd0>
>>> my_stack.push(7)
>>> my_stack.push(5)
>>> print(my_stack.pop())
None
>>>
```

What? It should be 5!...

Abstract base Stack class

Red marks the differences

```
from abc import ABC, abstractmethod
from typing import TypeVar, Generic
T = TypeVar('T')

class Stack(ABC, Generic[T]):
    def __init__(self) -> None:
        self.length = 0

    @abstractmethod
    def push(self, item: T) -> None:
        pass

    @abstractmethod
    def pop(self) -> T:
        pass

    @abstractmethod
    def peek(sel) -> T:
        pass

    def __len__(self) -> int:
        return self.length

    def is_empty(self) -> bool:
        return len(self) == 0

    @abstractmethod
    def is_full(self) -> bool:
        pass

    def clear(self):
        self.length = 0
```

Imports the class and the decorator from **abc**

Indicates the class is abstract; thus it cannot be instantiated

Indicates the method is abstract, and thus its implementation is not defined

```
>>> from abstract_stack import Stack
>>> x = Stack()
Traceback (most recent call last):
…
TypeError: Can't instantiate abstract class
Stack with abstract methods is_full, peek,
pop, push
>>>
```

It now behaves as expected: it cannot be instantiated

MONASH University

Stacks implemented with Arrays

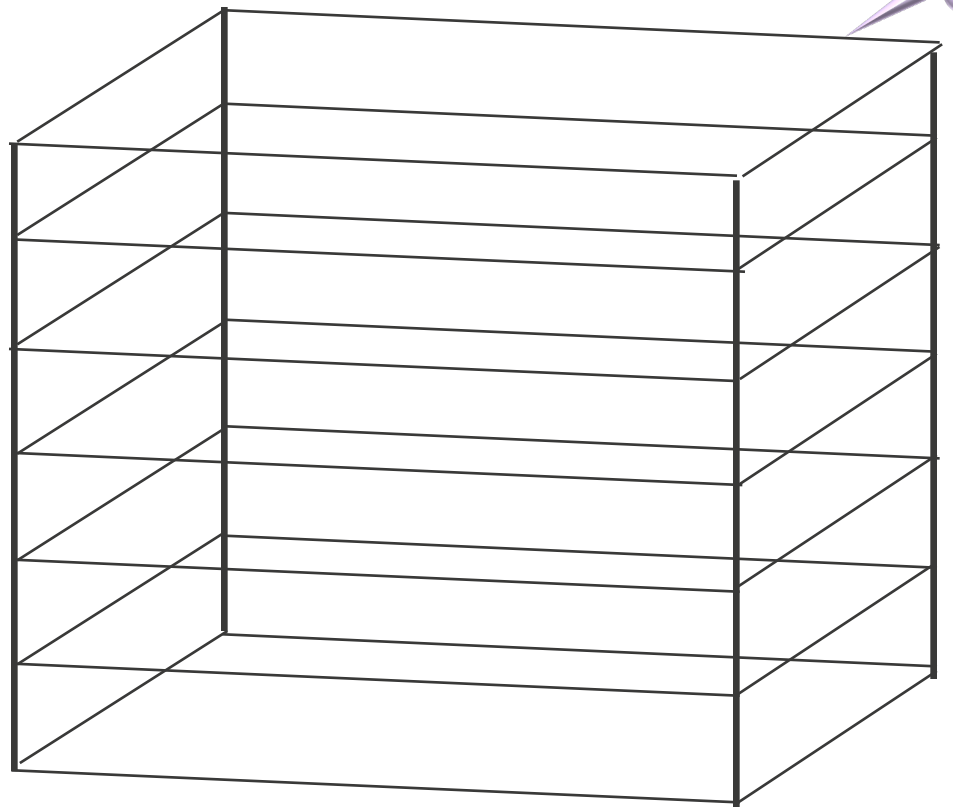
Part I

Let's now implement a Stack using arrays for the data

- **Stacks implemented with arrays often use two elements:**
  - An array to store the items in the order in which they arrive
    - The length of the array indicates the maximum capacity of the Stack
  - An integer indicating the first empty space in the array
    - We can reuse `length`, as the first empty space coincides with the stack's length
- **Invariants of such implementation:**
  - Valid data appears in the `0..length-1` positions of the array (beyond that is "garbage")
  - Element at the top of the stack is the element in position `length-1`
- **Wait, Stacks are an ADT! Shouldn't we hide the implementation?**
  - No, the abstraction comes from the user ignoring the implementation, not form hiding it (e.g., making it private in a ways that blocks outside view)

Stack: basic idea with arrays

- **Create a new stack: initially no elements in the array**

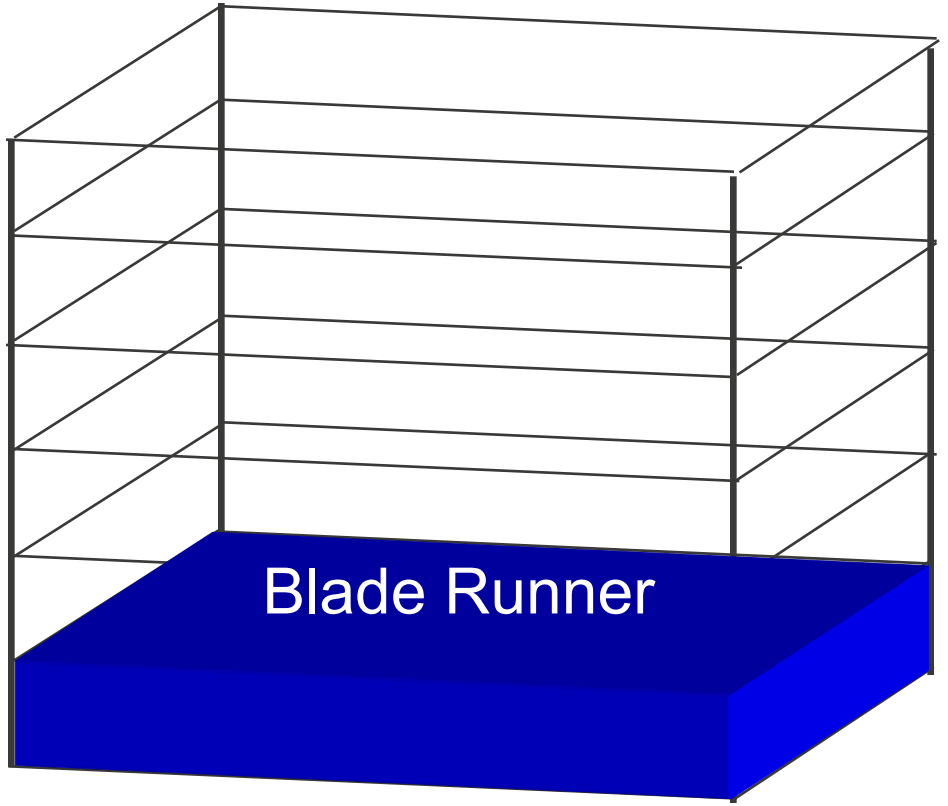


- **Max capacity = 6**
- **Length = 0**

Length of the stack

Stack: basic idea with arrays

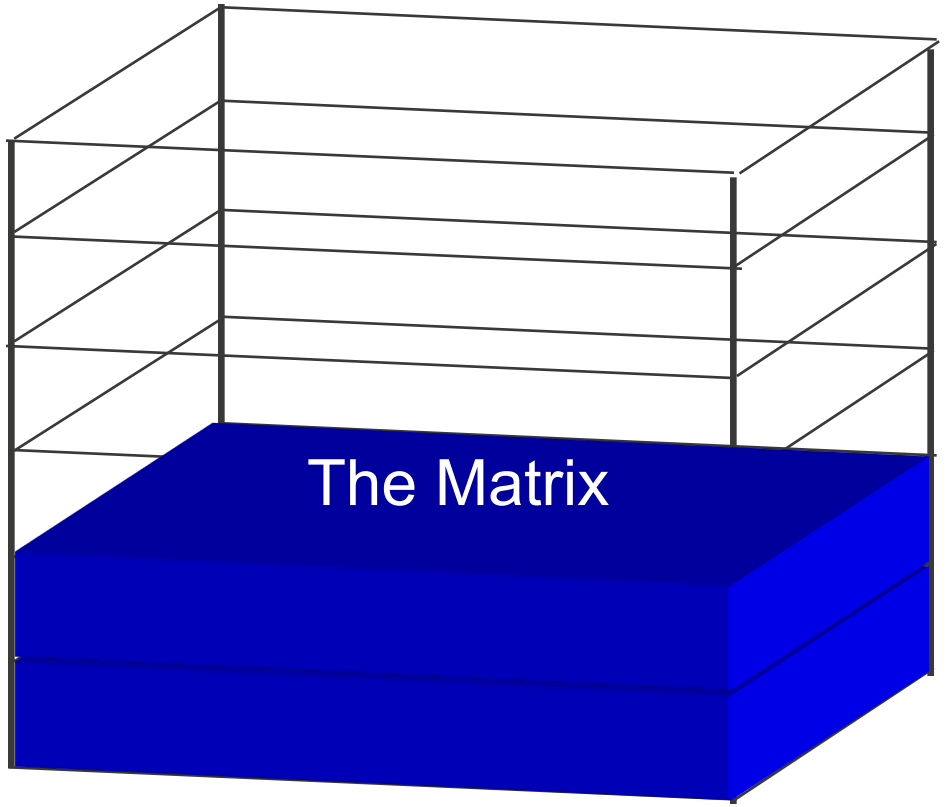
- Add an item to the top (push)



- Max capacity = 6
- Length = 1

Stack: basic idea with arrays

- Add another item to the top (push)

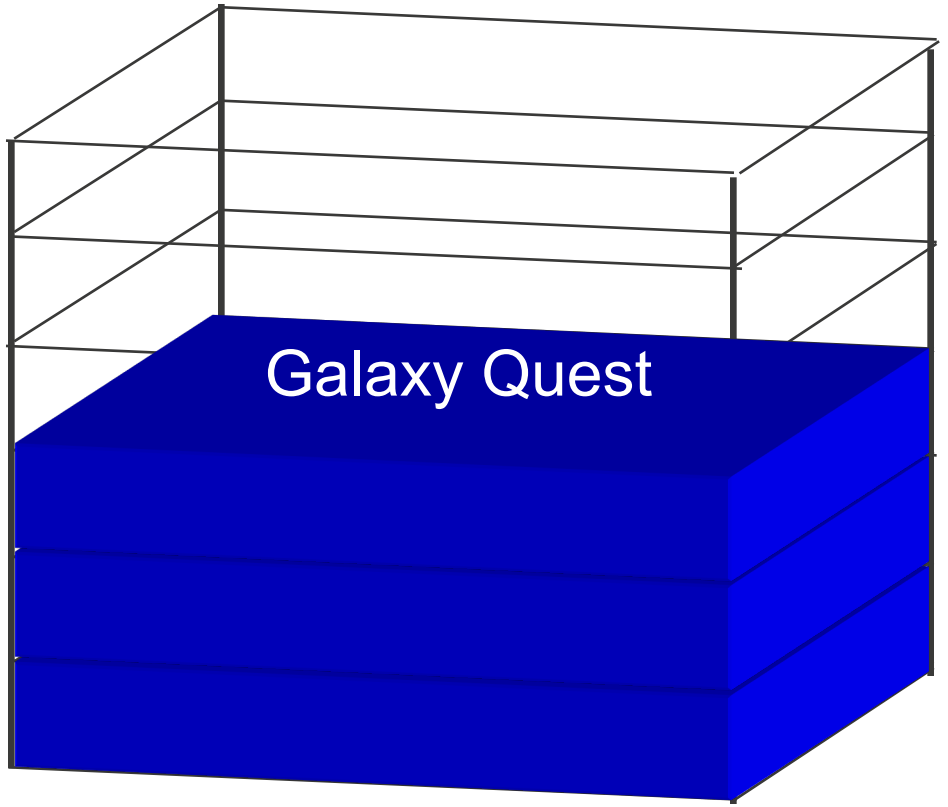


- Max capacity = 6
- Length = 2

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 with arrays

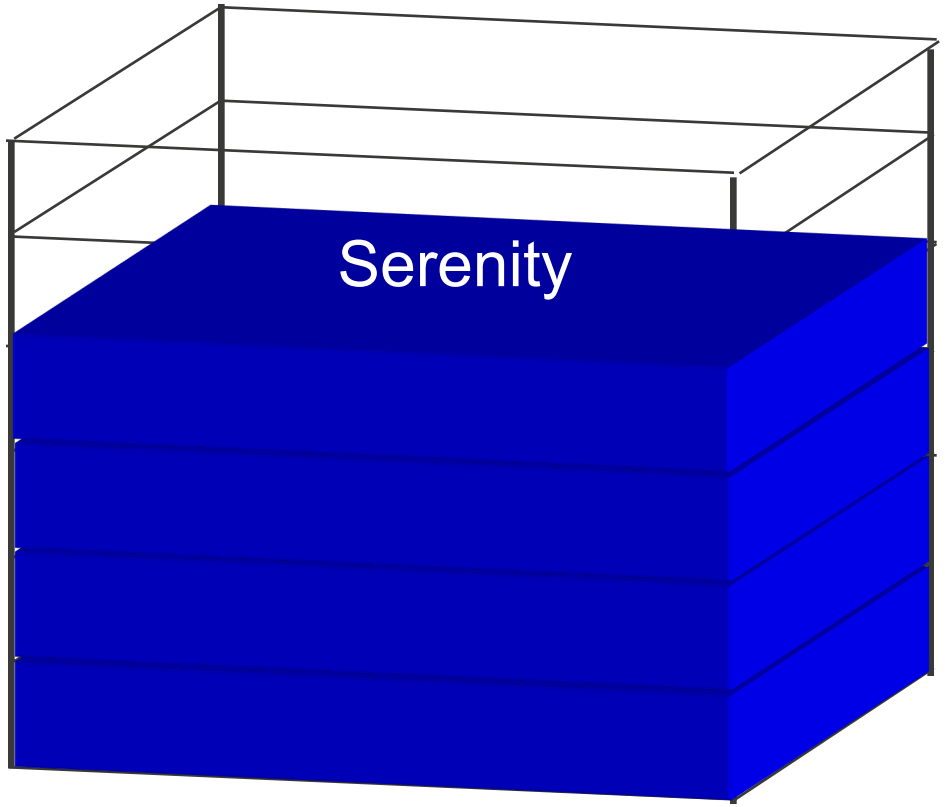
- Add another item to the top (push)



- Max capacity = 6
- Length = 3

Stack: basic idea with arrays

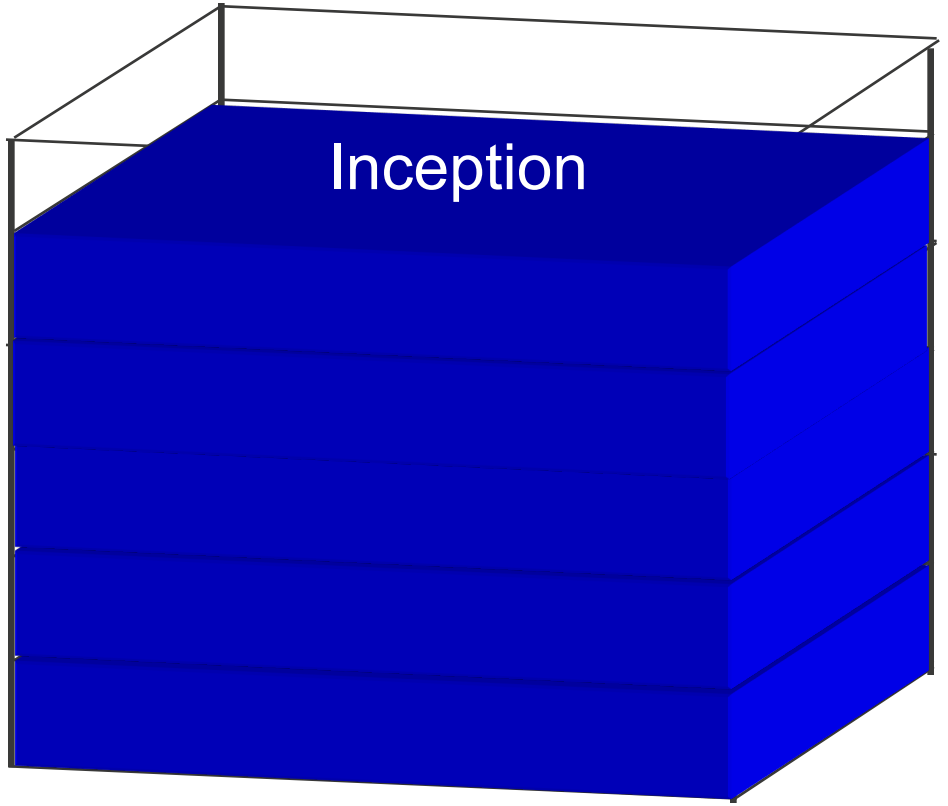
- Add another item to the top (push)



- Max capacity = 6
- Length = 4

Stack: basic idea with arrays

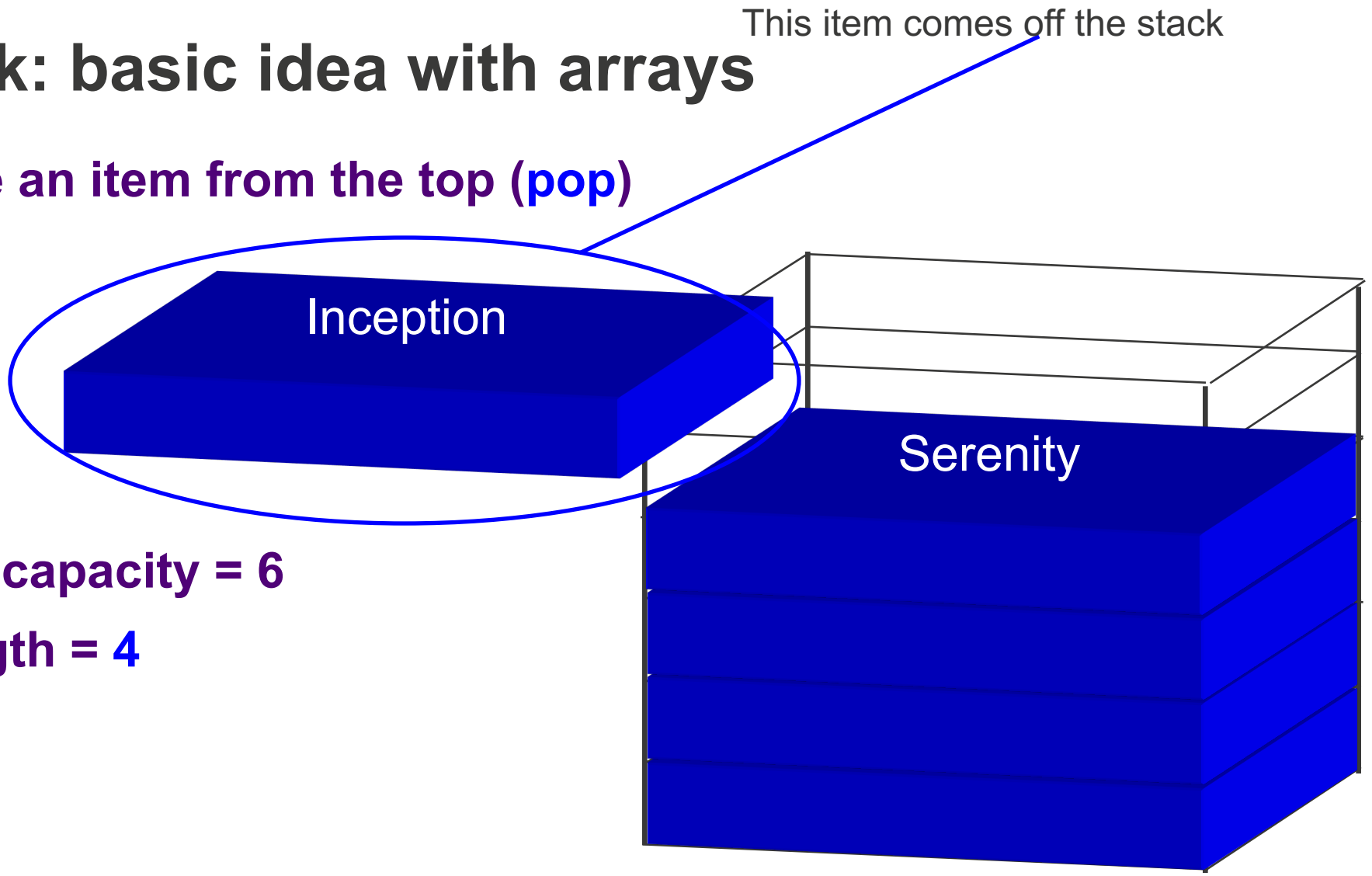
- Add another item to the top (push)



- Max capacity = 6
- Length = 5

Stack: basic idea with arrays

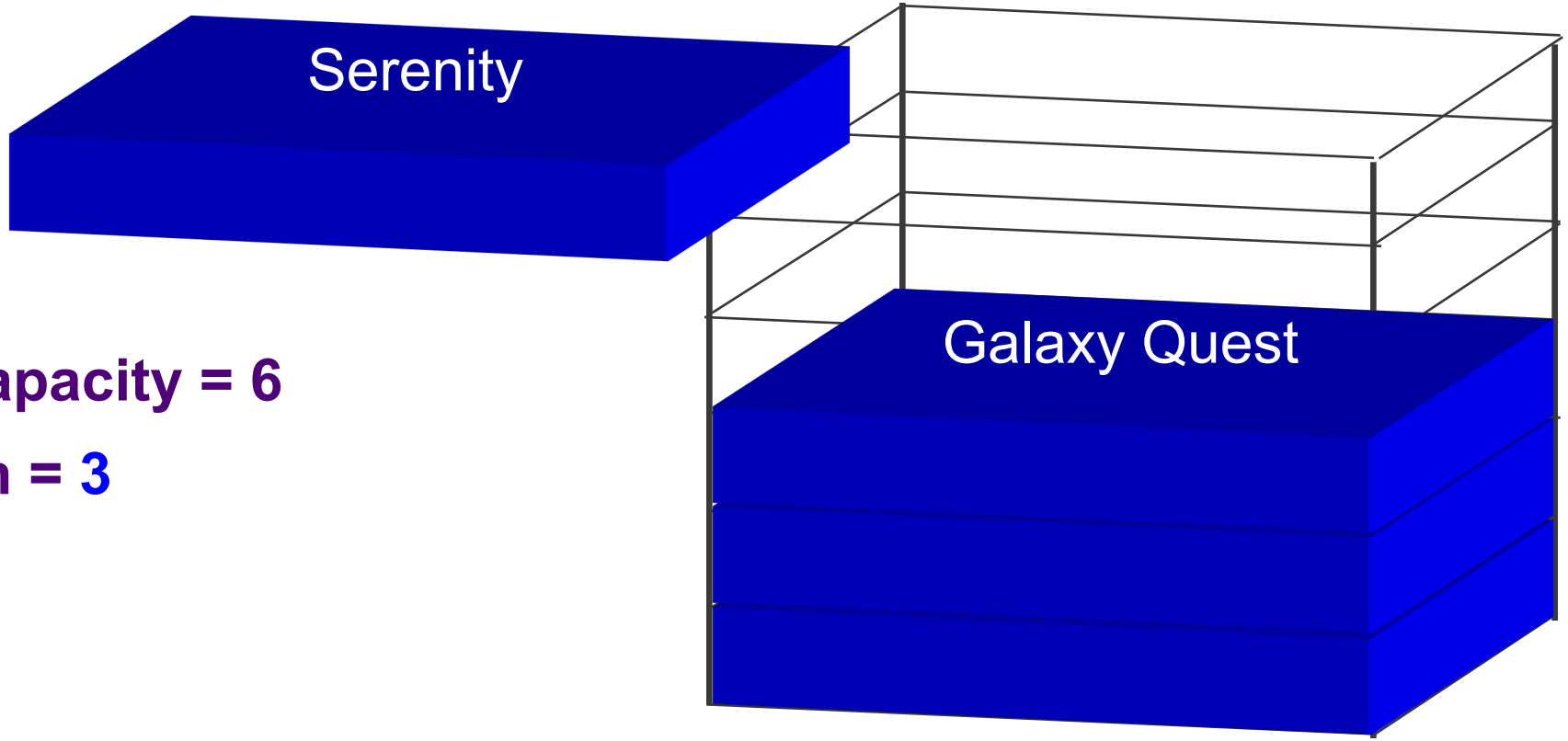
- **Take an item from the top (pop)**



- **Max capacity = 6**
- **Length = 4**

Stack: basic idea with arrays

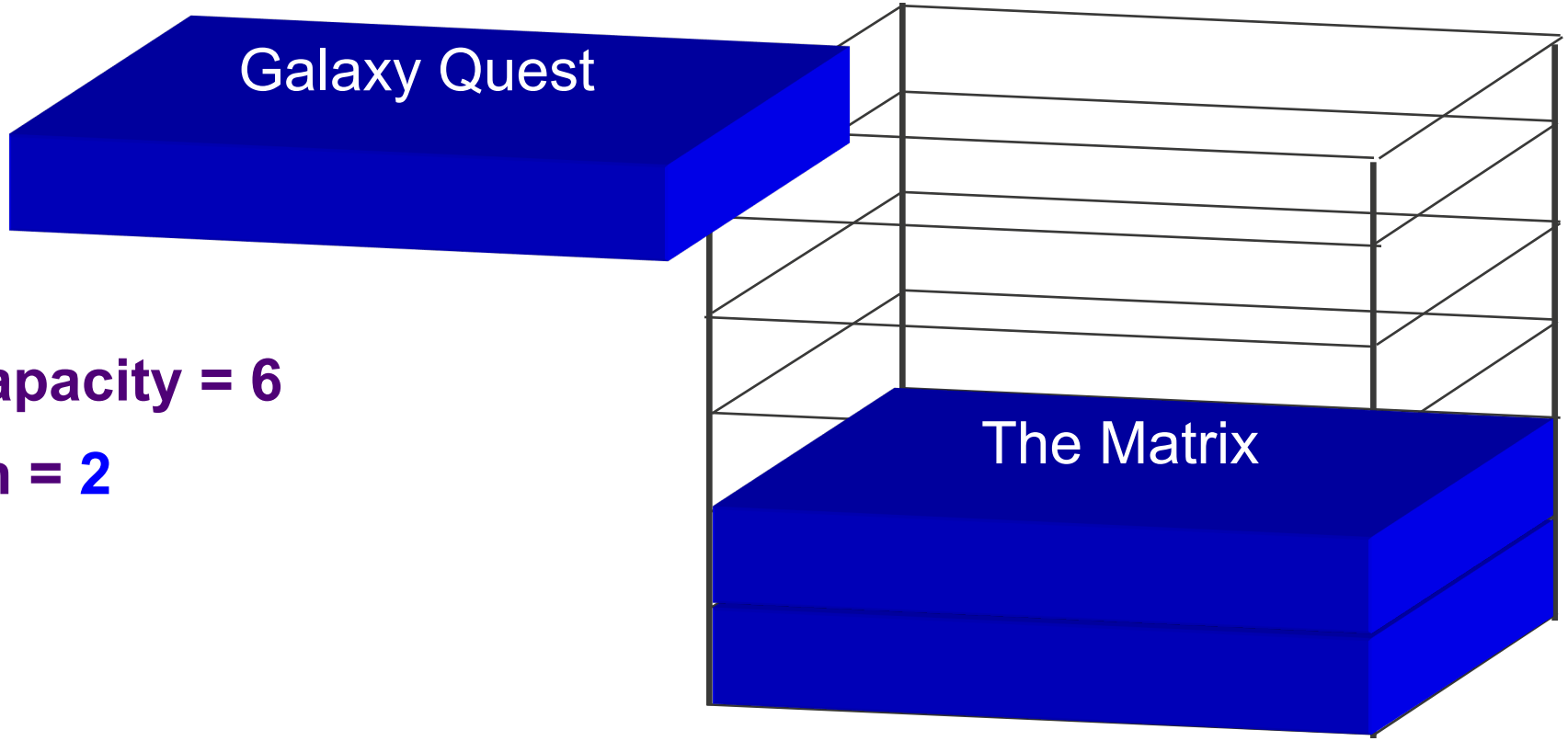
- Take another item from the top (pop)



- Max capacity = 6
- Length = 3

Stack: basic idea with arrays

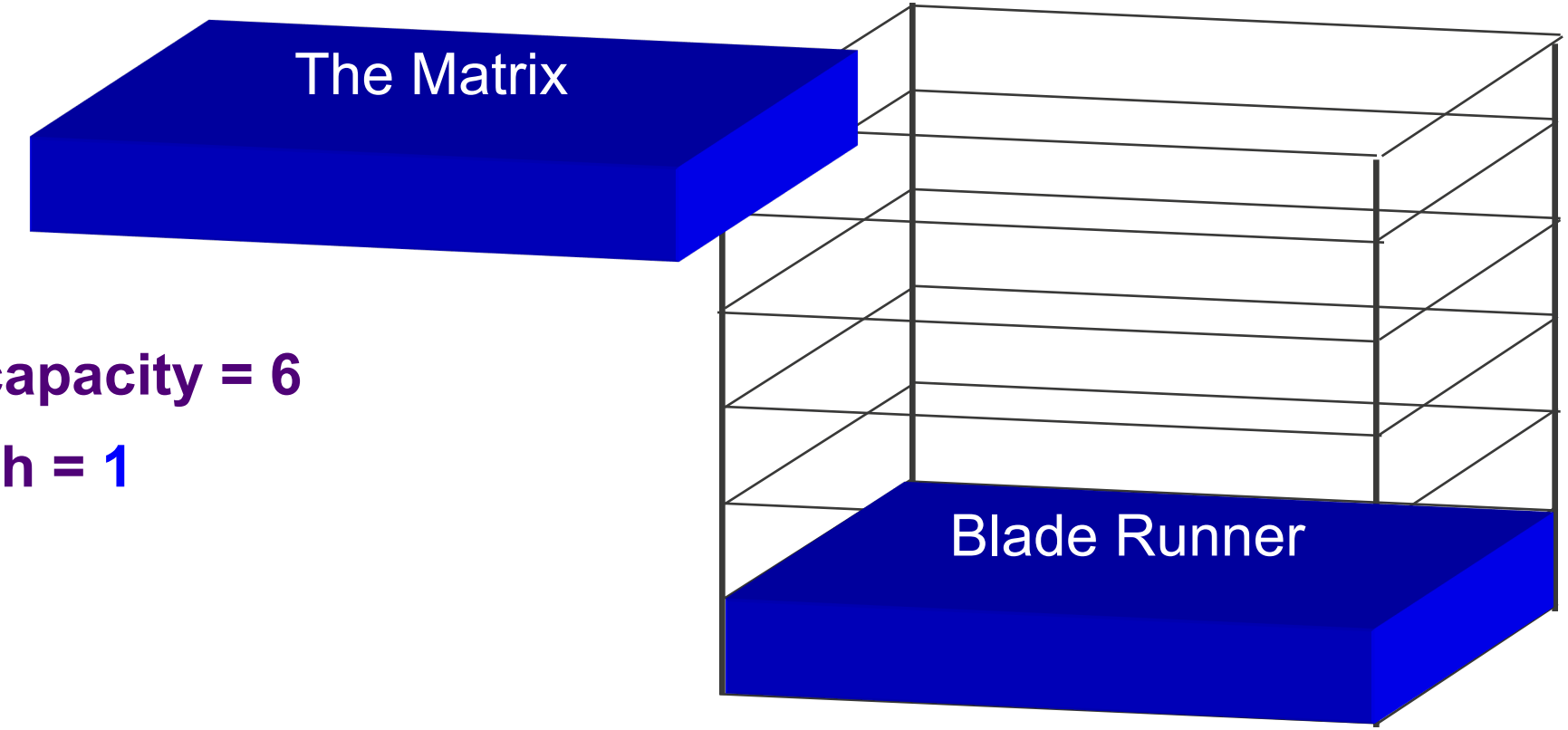
- Take another item from the top (pop)



- Max capacity = 6
- Length = 2

Stack: basic idea with arrays

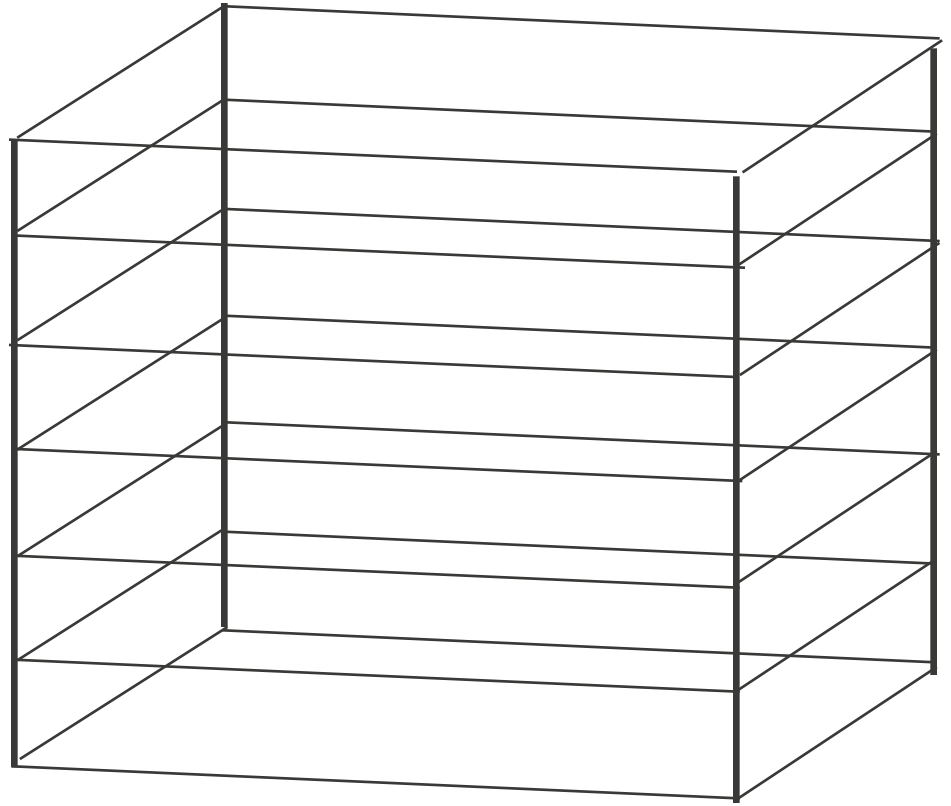
- Take another item from the top (pop)



- Max capacity = 6
- Length = 1

Stack: basic idea with arrays

- Take another item from the top (pop)



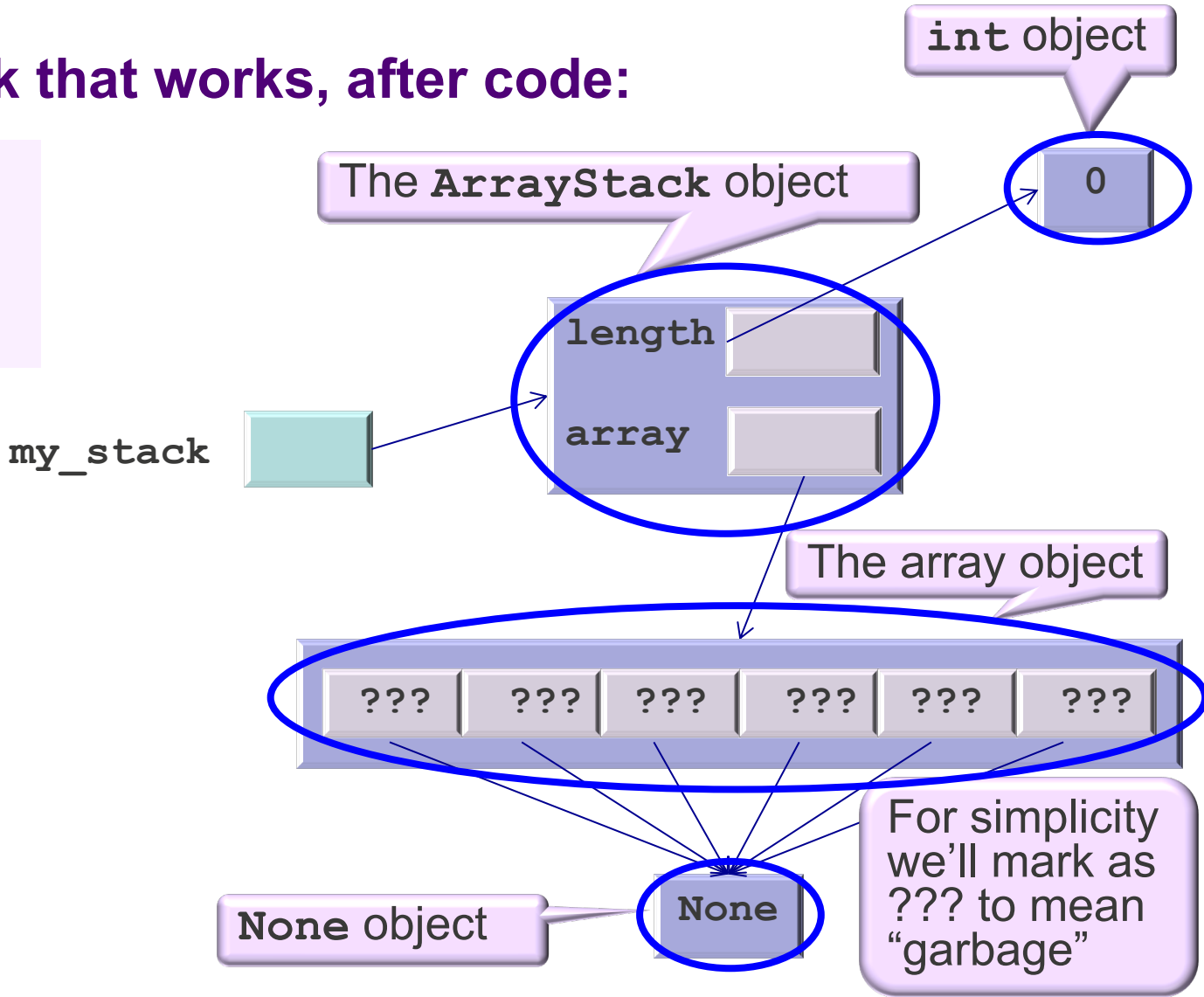
- Max capacity = 6
- Length = 0

A more realistic view of array stacks in memory

- **Given an implemented ArrayStack that works, after code:**

```
>>> from array_stack import ArrayStack
>>> my_stack = ArrayStack(6)
>>>
```

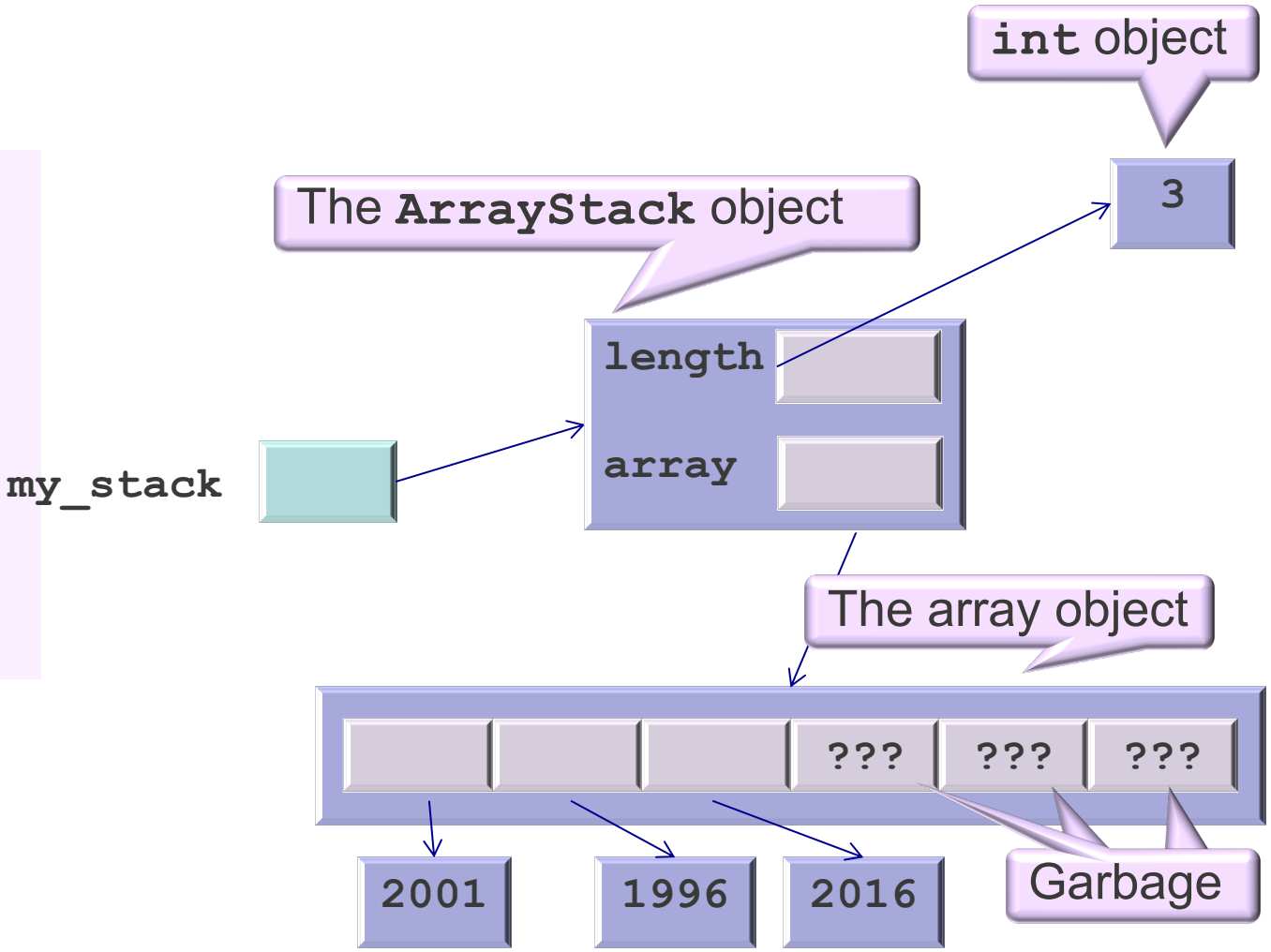
- **We would have something like this (not exactly, but close)**
  - Every element in the array is initialized to object **`None`**
  - Note that there is only one object **`None`** in Python (same as **`True`** and **`False`**)



A more realistic view of array stacks in memory

- **And after the code:**

```
>>> from array_stack import ArrayStack
>>> my_stack = ArrayStack(6)
>>> my_stack.push(2001)
>>> my_stack.push(1996)
>>> my_stack.push(2016)
>>> len(my_stack)
3
>>>
```



- **We would have something like this (again not exactly, but close)**

Let's implement the Stack ADT using arrays

- **Add a class `ArrayStack` derived from `Stack` that implements all methods**
- **Wait! Does Python have traditional arrays? (fixed size, like in MIPS)**
  - Since 3.8, it has the Array class but only for numerical values, which is not ideal
  - We have created an ArrayR class to hold references to any object (not just ints)
- **To use it, first download `referential_array.py` from Moodle**
- **And then import the `ArrayR` class adding to your module the line:**
  - `from referential_array import ArrayR`
- **Form then on, you can use it as follows:**
  - Create an array of a given size (e.g., `array = ArrayR(4)`)
  - Access the element at a given position (e.g., `element = array[2]`)
  - Set the value of the element at a given position (e.g.. `array[3] = value`)
  - Obtain its length (e.g., `length = len(array)`)

Let's start Implementing ArrayStack

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

```
from referential_array import ArrayR
from abstract_stack import Stack, T


class ArrayStack(Stack[T]):
    MIN_CAPACITY = 1

    def __init__(self, max_capacity: int) -> None:
        Stack.__init__(self)
        self.array = ArrayR(max(self.MIN_CAPACITY, max_capacity))

    def is_full(self) -> bool:
        return len(self) == len(self.array)
```

The parent class

Minimum capacity of the array (**ArrayR** cannot handle 0). Created as class variable, since it is the same for all objects. Uppercases because it is a constant

Uses the implementation of the parent class

reusing **len**: good

Notice we do not need to define clear(), we just inherit it!

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

It is still there, but becomes garbage to the stack object

Let's start Implementing ArrayStack

Big O?

```
from referential_array import ArrayR
from abstract_stack import Stack, T


class ArrayStack(Stack[T]):
    MIN_CAPACITY = 1

    def __init__(self, max_capacity: int) -> None:
        Stack.__init__(self)
        self.array = ArrayR(max(self.MIN_CAPACITY, max_capacity))

    def is_full(self) -> bool:
        return len(self) == len(self.array)
```

All return statements, assignments and integer comparisons are always constant

`Stack.__init__` and `len()` are known to be O(1)

Therefore `is_full` is O(1)

What about `__init__`?

It depends on `ArrayR` which is O(max_capacity) since MIN_CAPACITY is a constant and `max` is O(1)

So it is O(max_capacity)

MONASH University

Stacks implemented with Arrays

Part II

Implementing push

- **What do we do if the stack is full?**
  - Let's make it a precondition and raise an exception

We could also decide to return `False` but we then need to change the parent class. So this is a decision that needs to be taken when defining the parent class

Can I just test the precondition with an assertion?

No! since it is an external function. So it must use exceptions to increase robustness

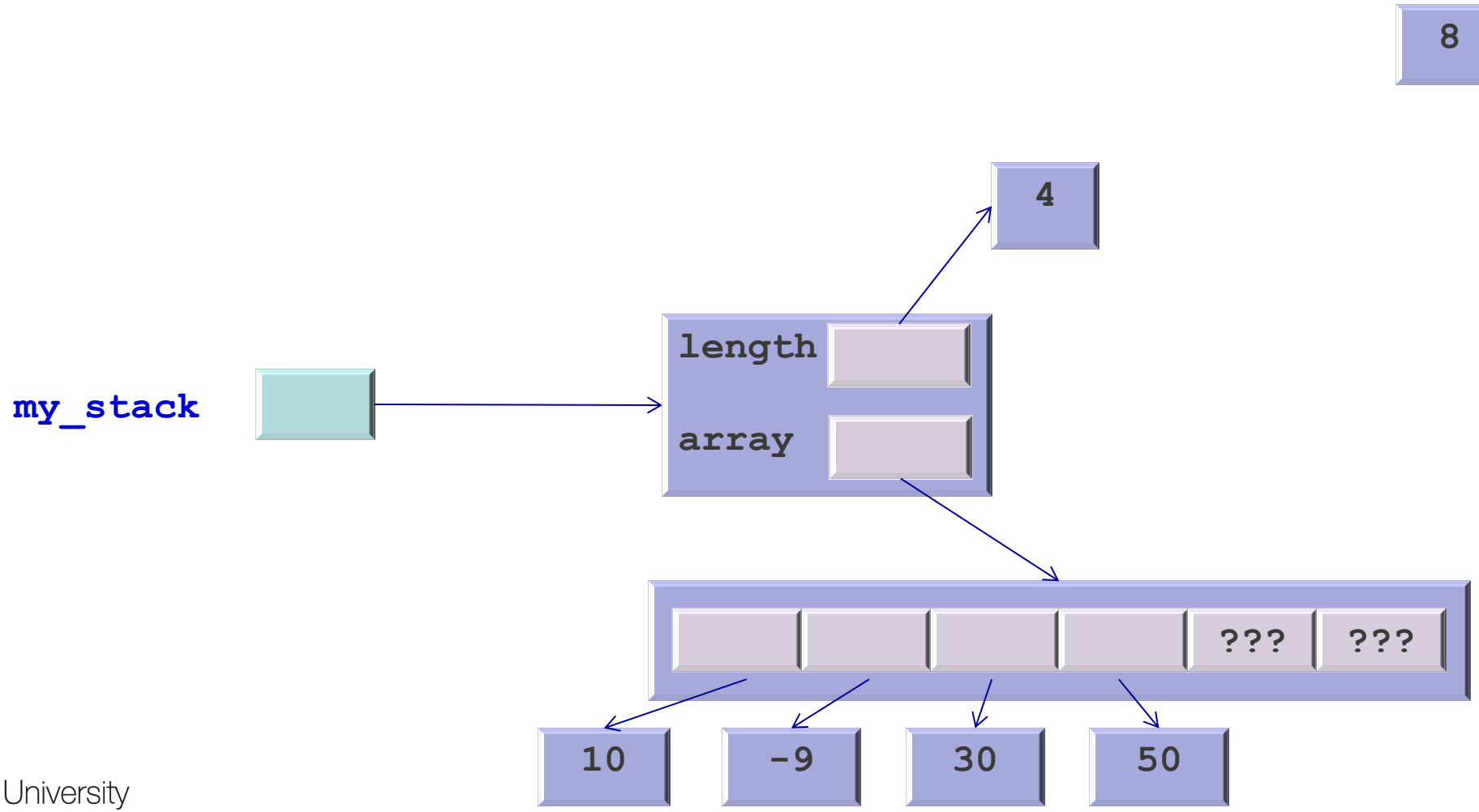
Even better, we could decide that if full, we must increase the size of the array! Leave that to you!

```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

reusing `is_full`: good

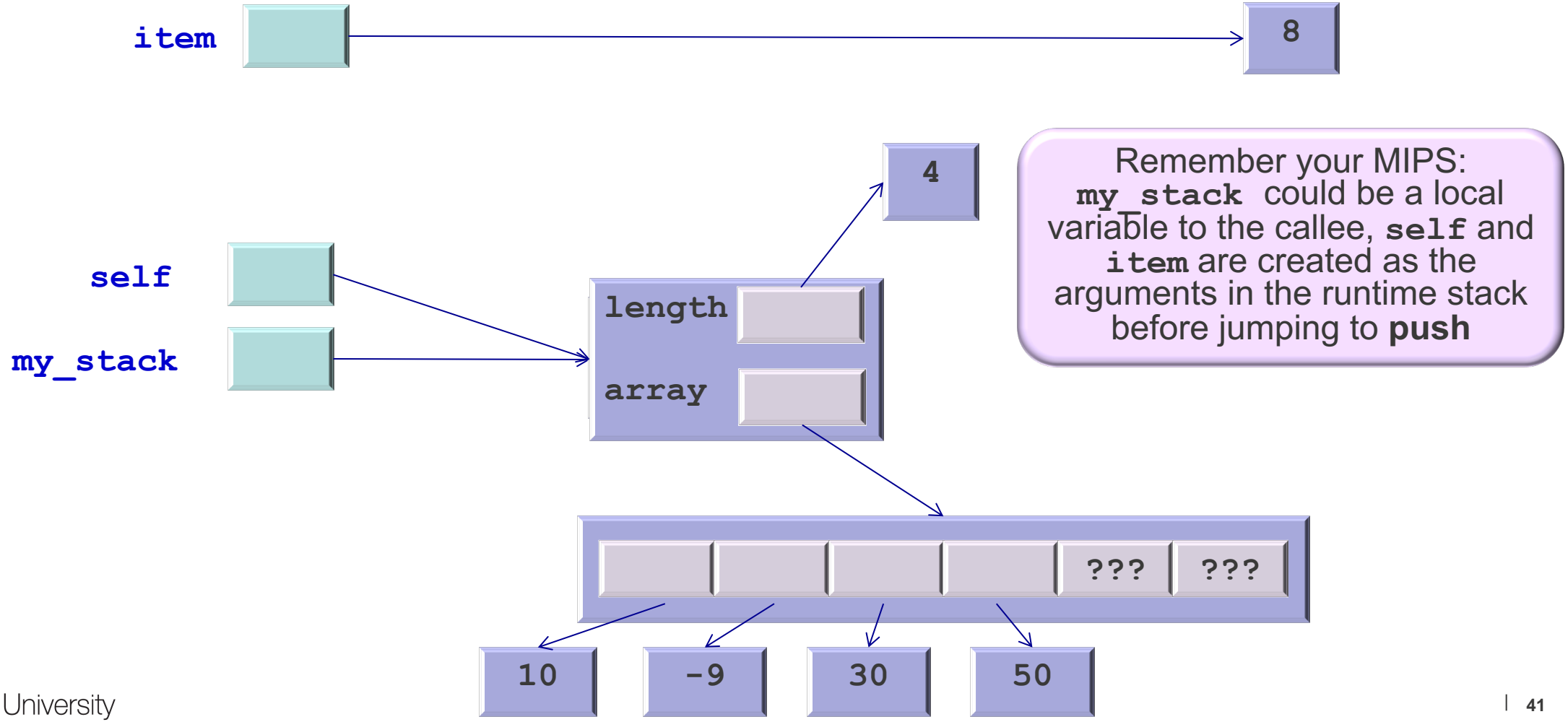
```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

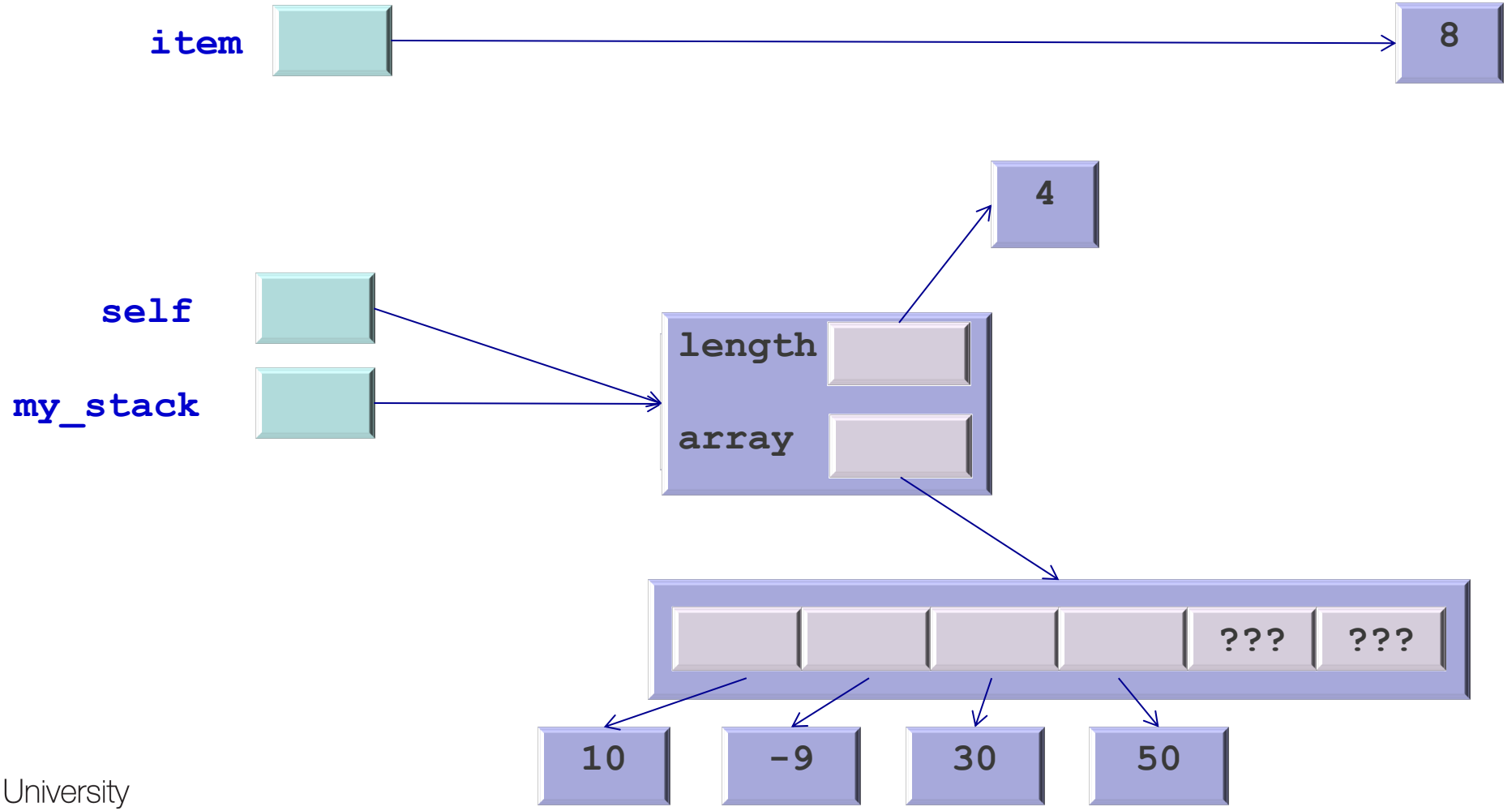
push(my_stack,8)



Remember your MIPS: `my_stack` could be a local variable to the callee, `self` and `item` are created as the arguments in the runtime stack before jumping to **push**

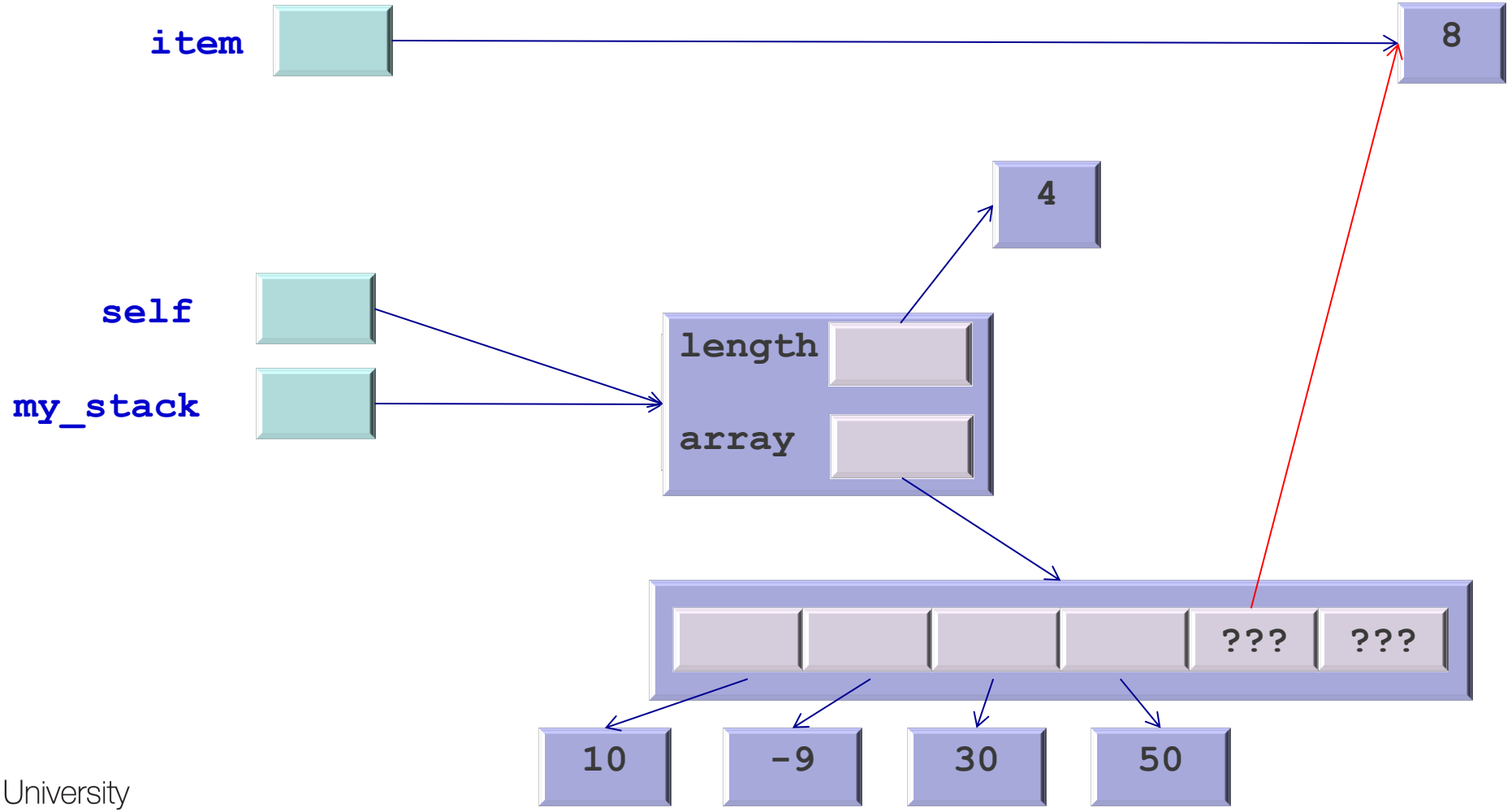
```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



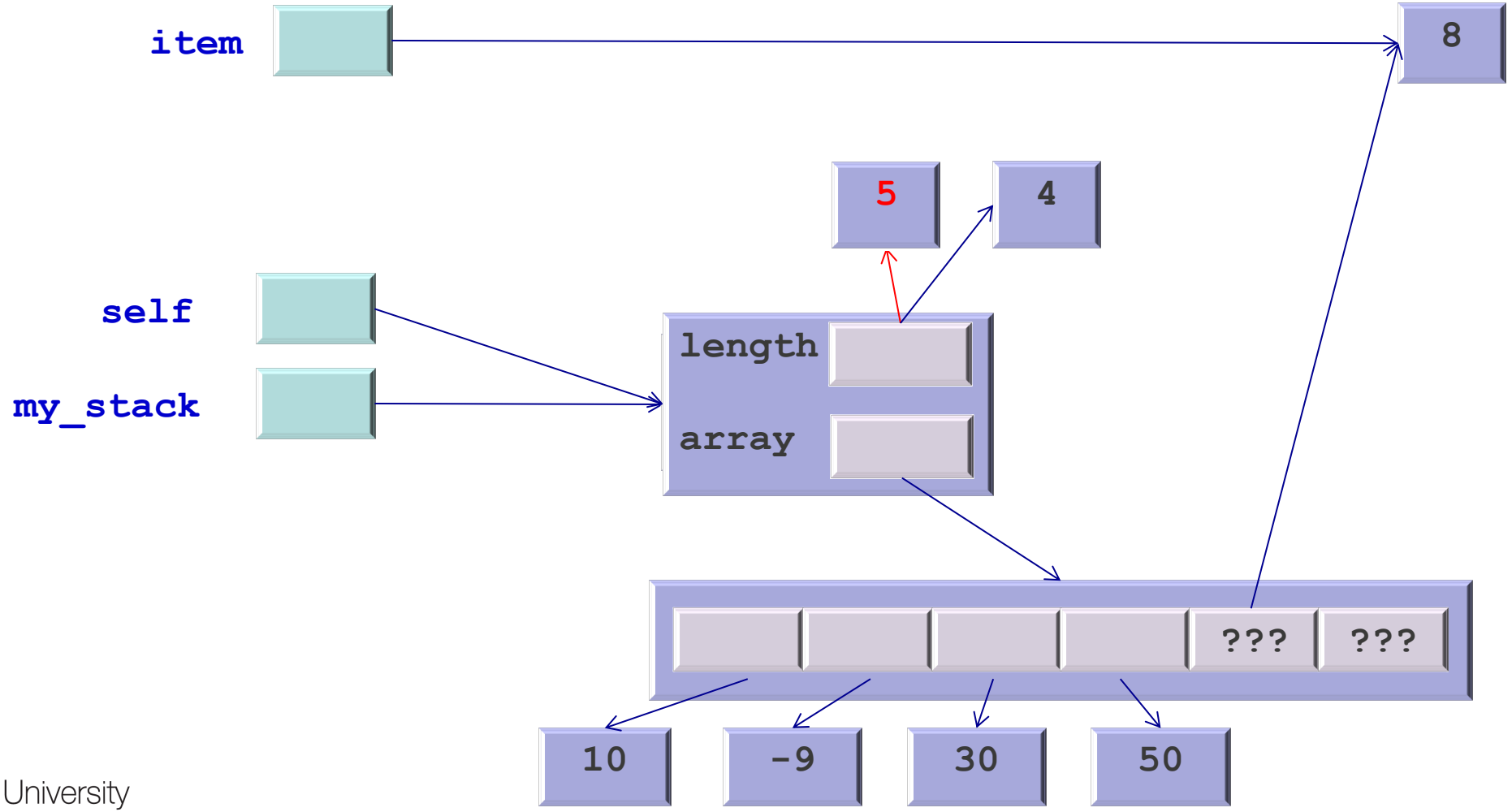
```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



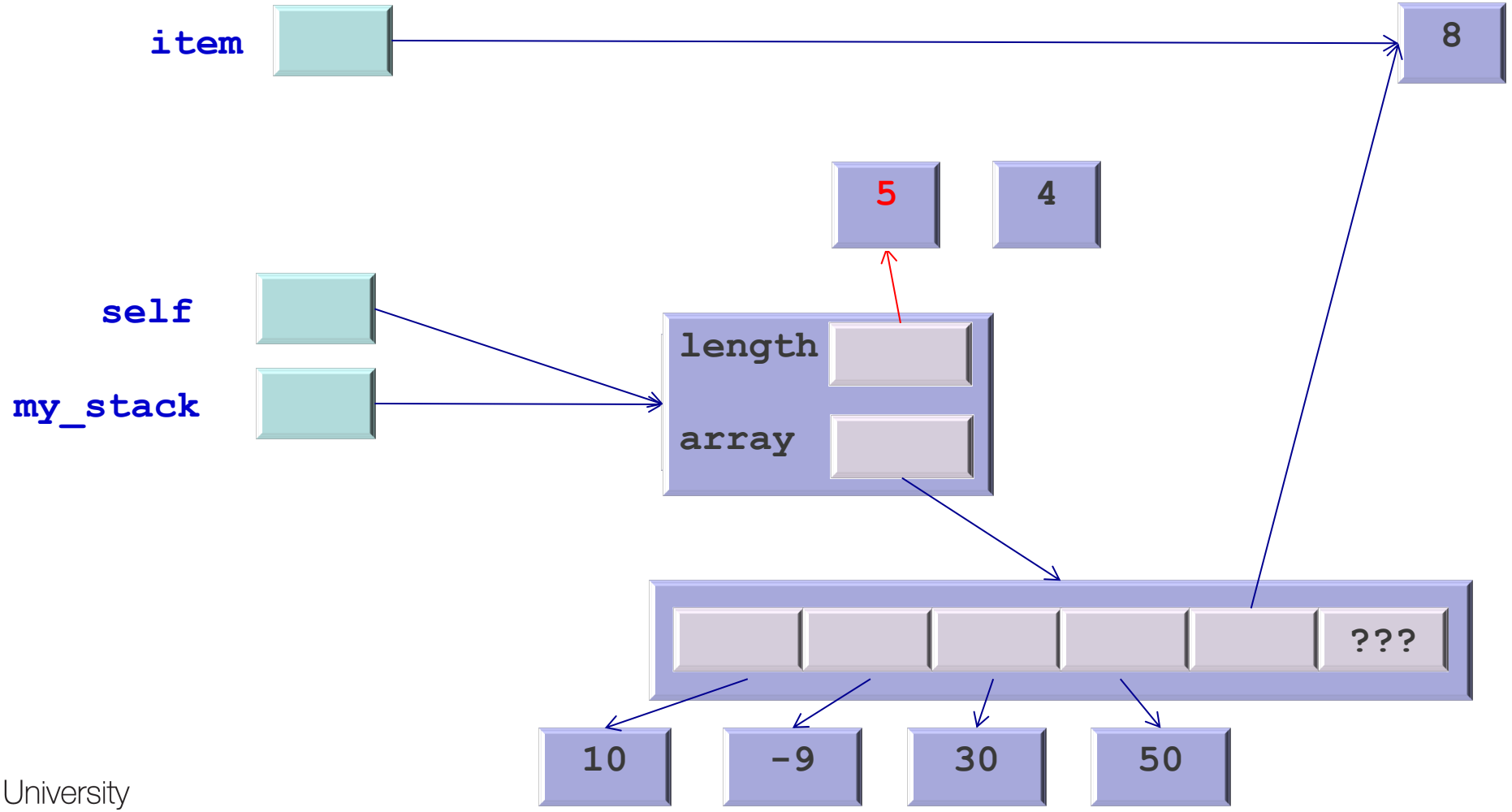
```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



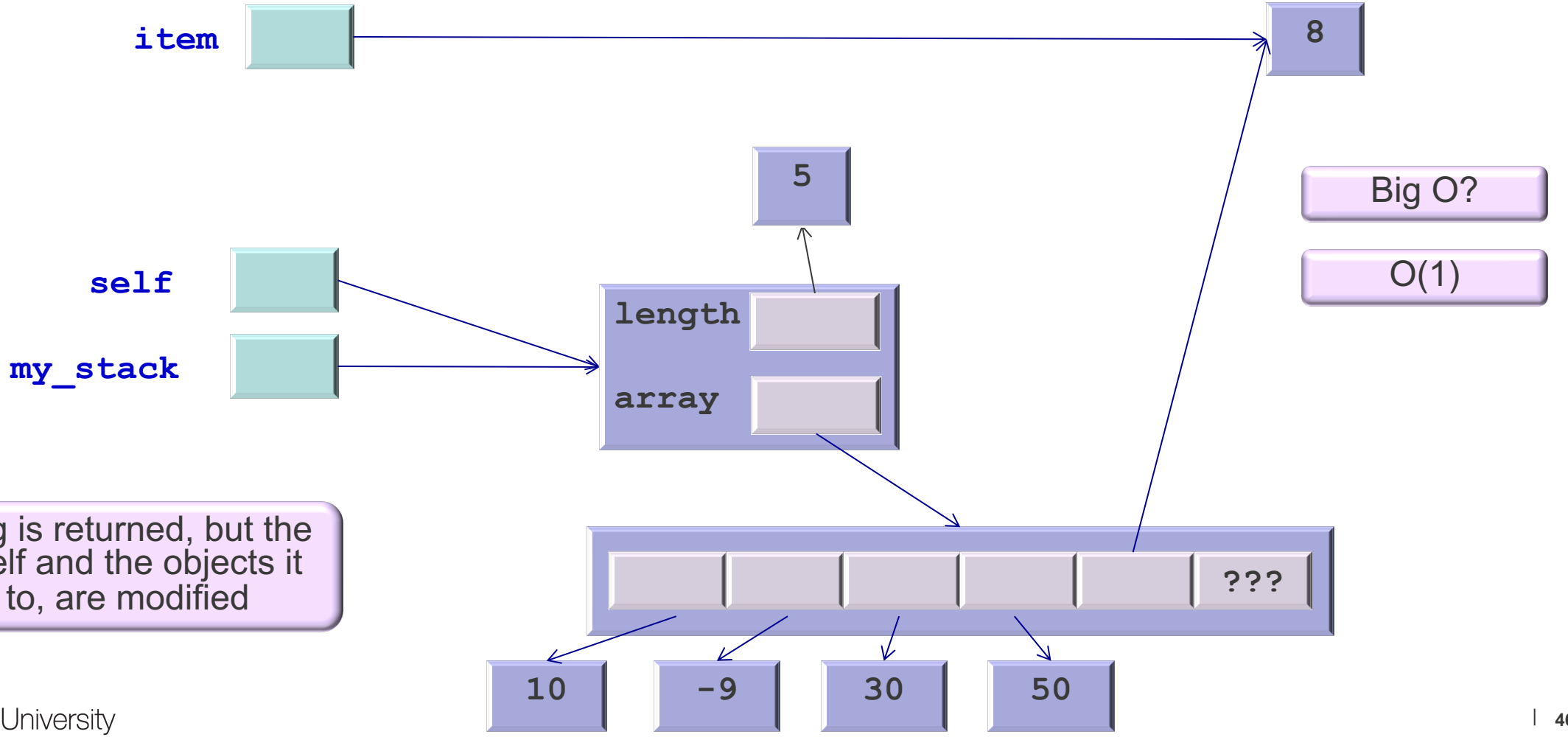
```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



```
def push(self, item: T) -> None:
    if self.is_full():
        raise Exception("Stack is full")

    self.array[len(self)] = item
    self.length += 1
```

push(my_stack,8)



So nothing is returned, but the object itself and the objects it points to, are modified

Implementing pop

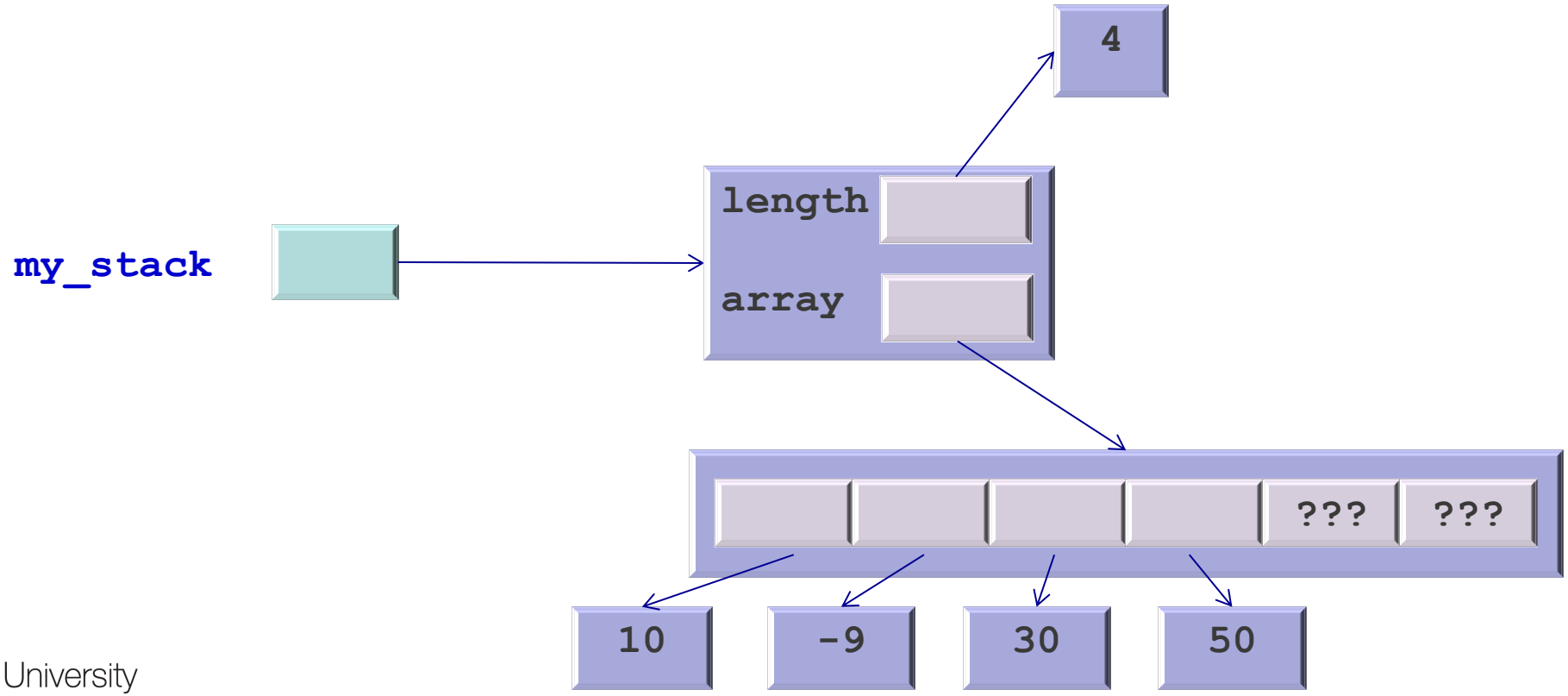
- **What do we do if the stack is empty?**
  - Let's raise an exception again

Another design decision!

```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

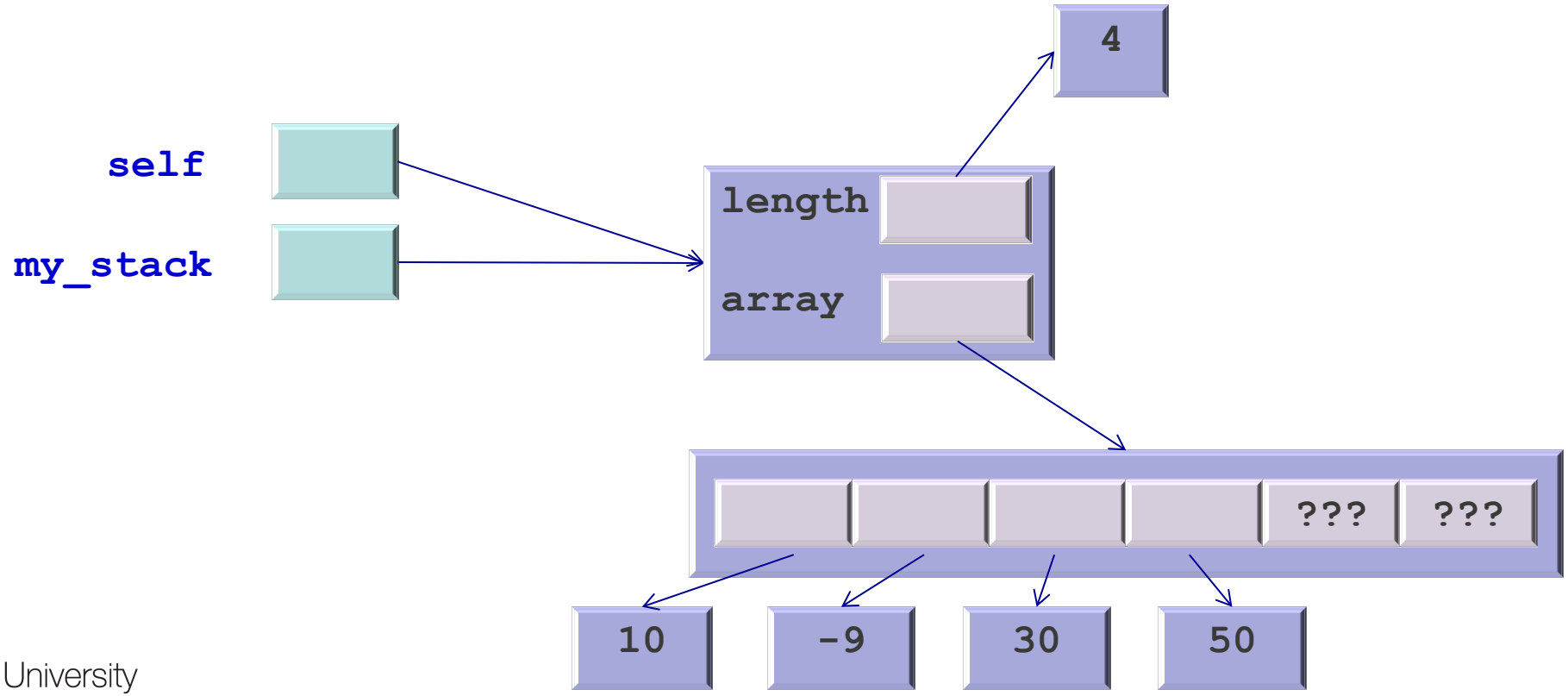
```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

pop(my_stack)



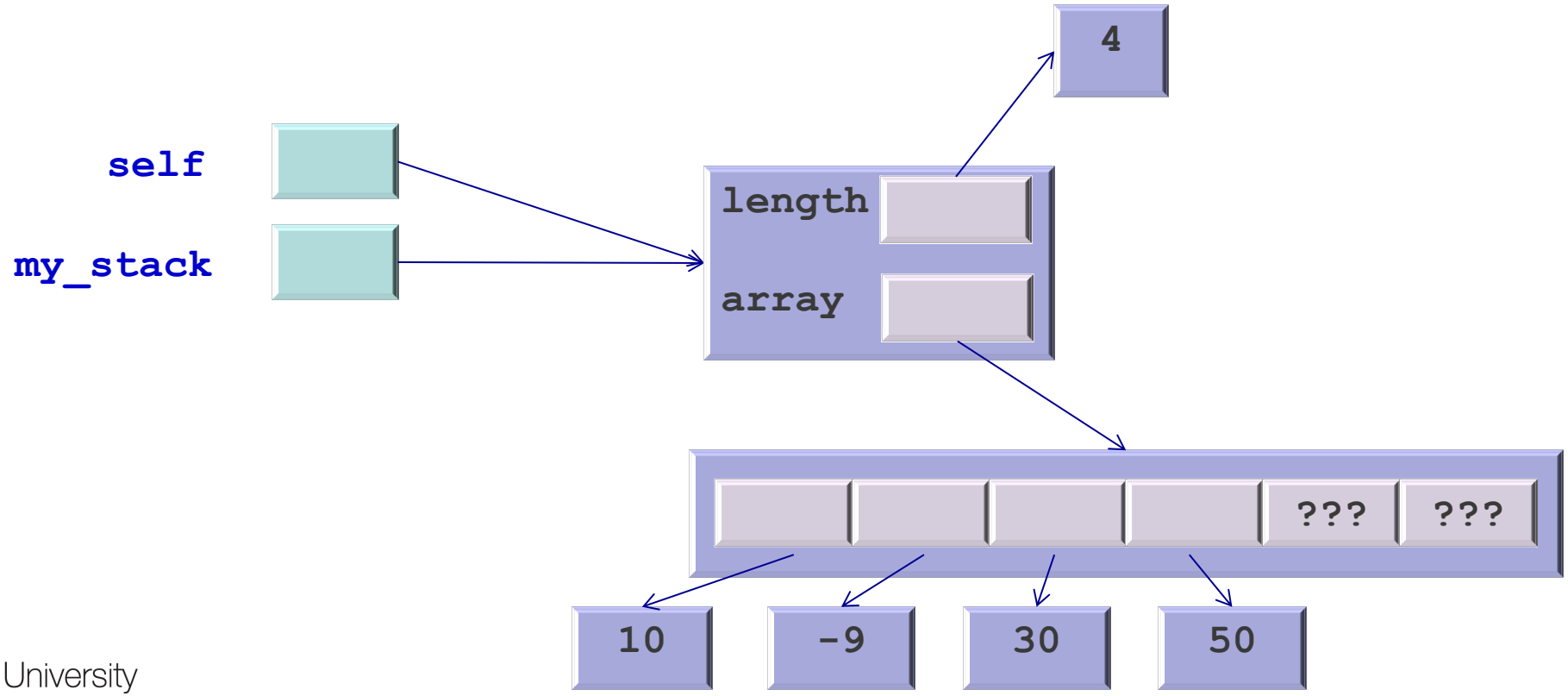
```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

pop(my_stack)



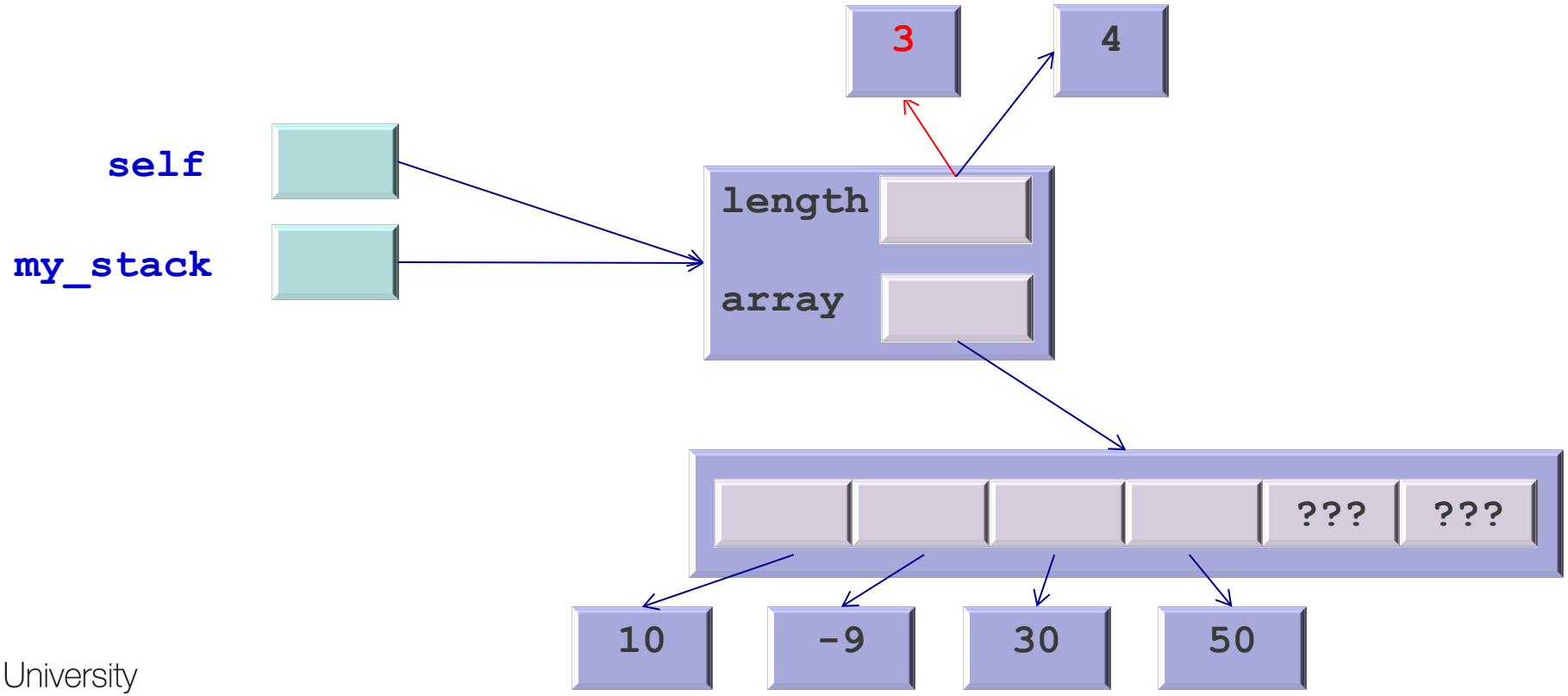
```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

pop(my_stack)



```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

pop(my_stack)

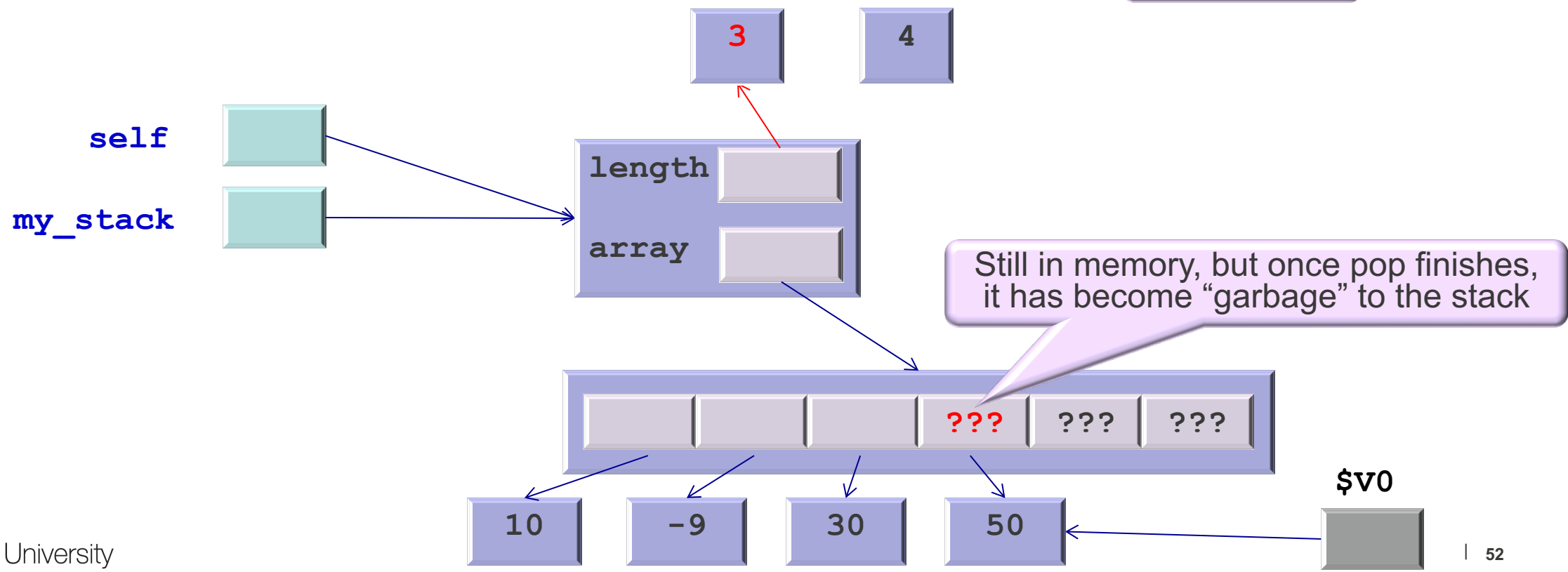


```
def pop(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    self.length -= 1
    return self.array[self.length]
```

pop(my_stack)

Big O?

O(1)



Implementing peek

- **Peek is very similar to pop (same but does not remove the top item)**

```
def peek(self) -> T:
    if self.is_empty():
        raise Exception("Stack is empty")

    return self.array[self.length-1]
```

Big O?

O(1)

MONASH
University

Using our ArrayStack

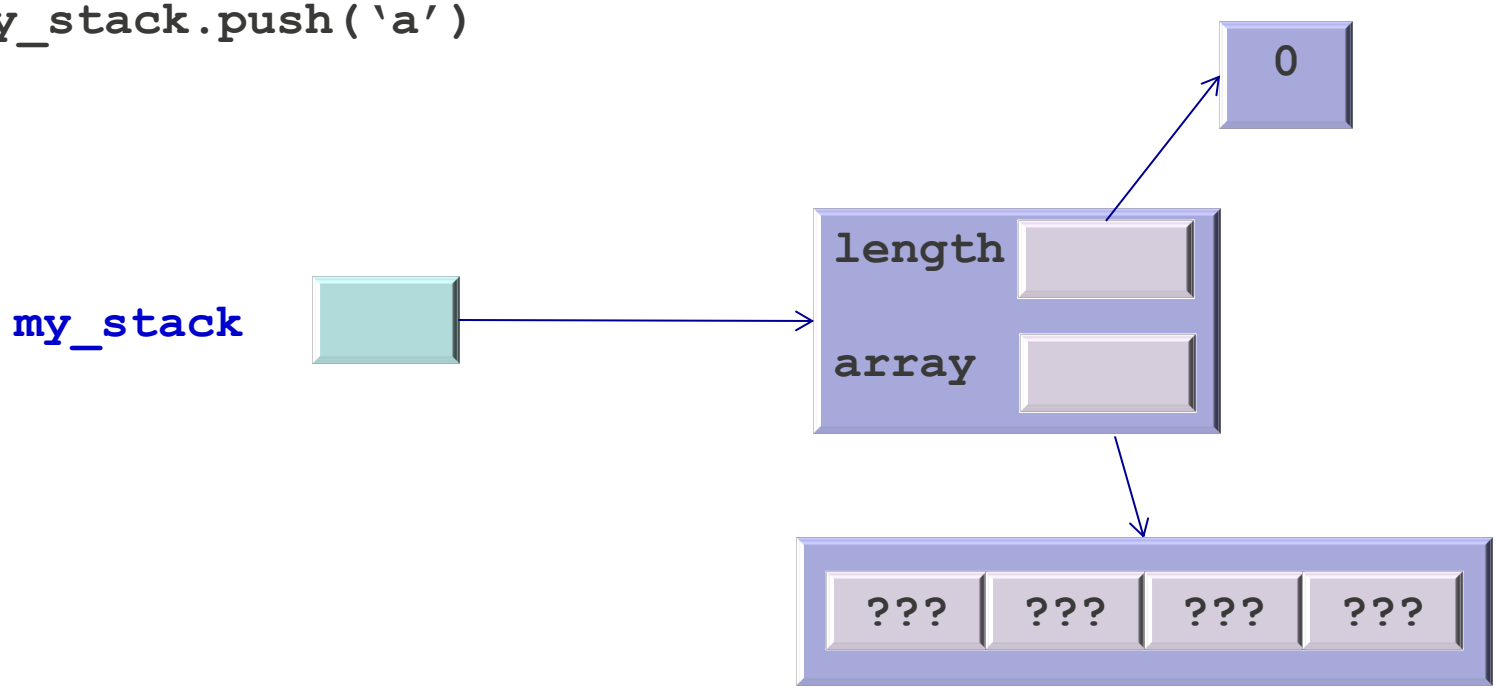
Let's use our new stack: reversing a sequence of items

- **Given a string as input, return one with the characters in reverse order**
  - For example, given "abcd" it should return "dcba"
- **A possible algorithm is:**
  - 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
  - Return the new (and now reversed) string
- **Of course you can do this without a stack, but it is a simple way to use it**
  - Useful when you do not know the length of the input objects to be reversed
  - Think about undoing the actions when editing a file

Example: reversing a sequence of chars

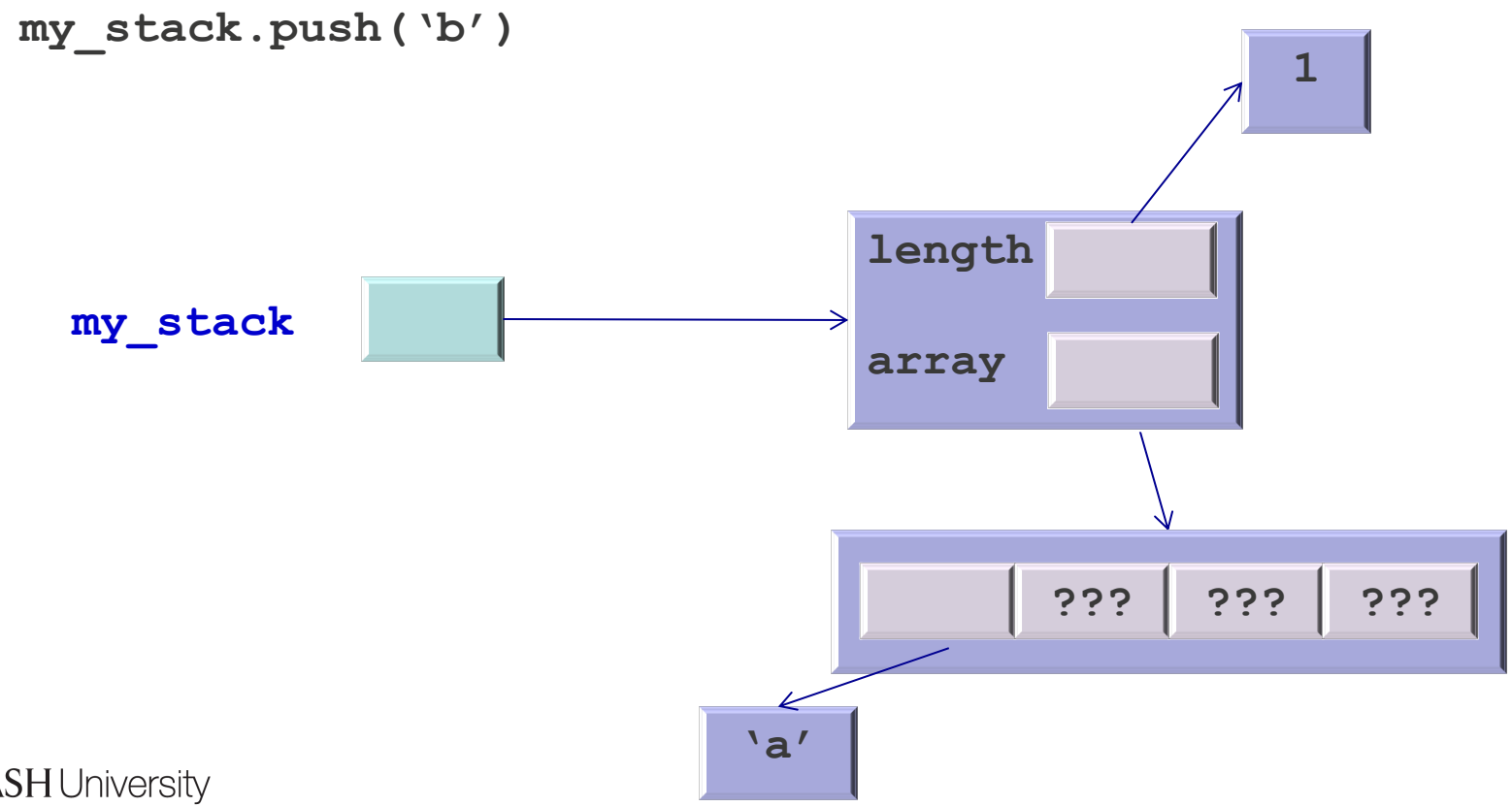
input string: “a b c d”

```
my_stack.push('a')
```



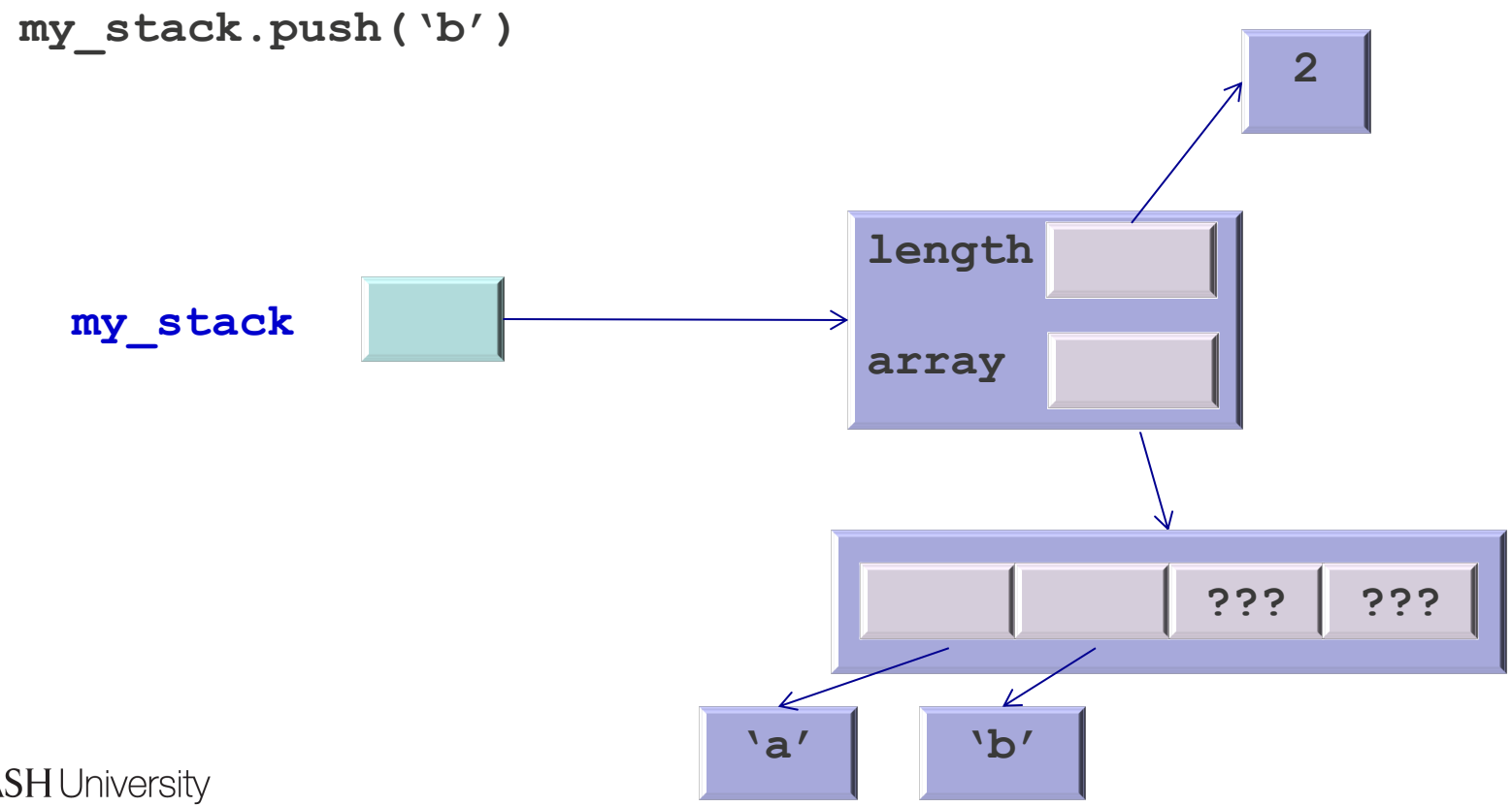
Example: reversing a sequence of chars

input string: “a b c d”



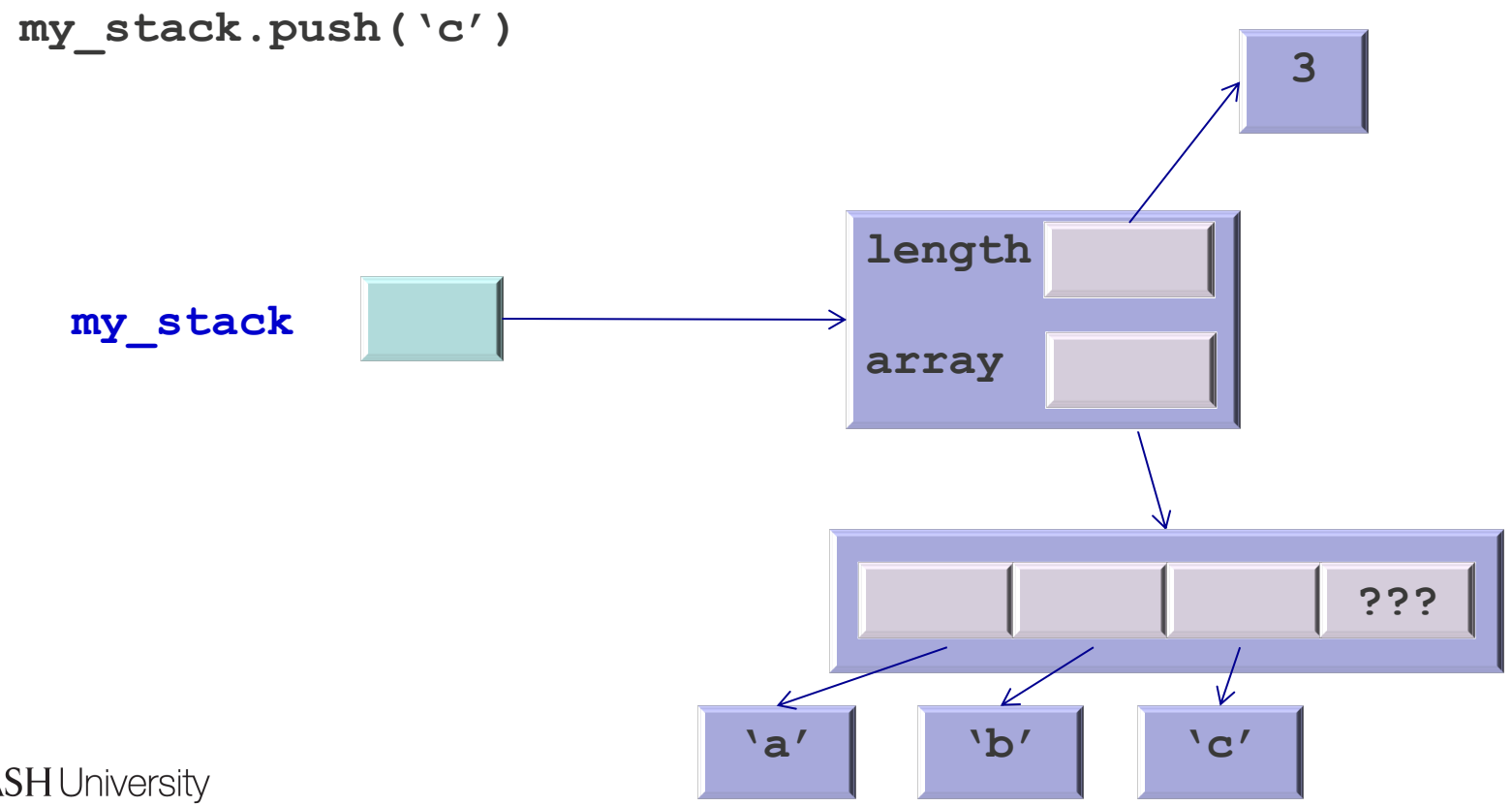
Example: reversing a sequence of chars

input string: “a b c d”



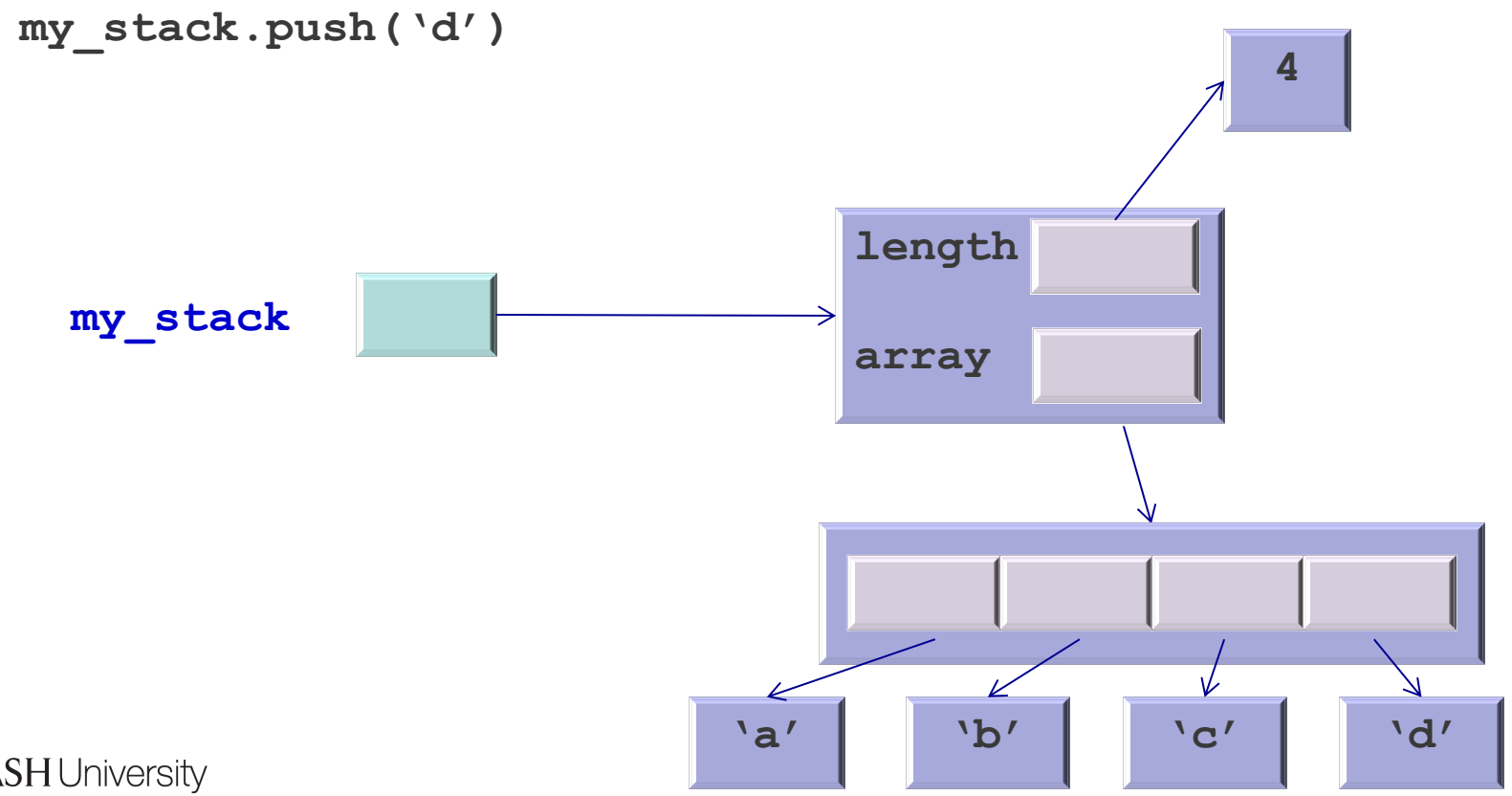
Example: reversing a sequence of chars

input string: “a b c d”



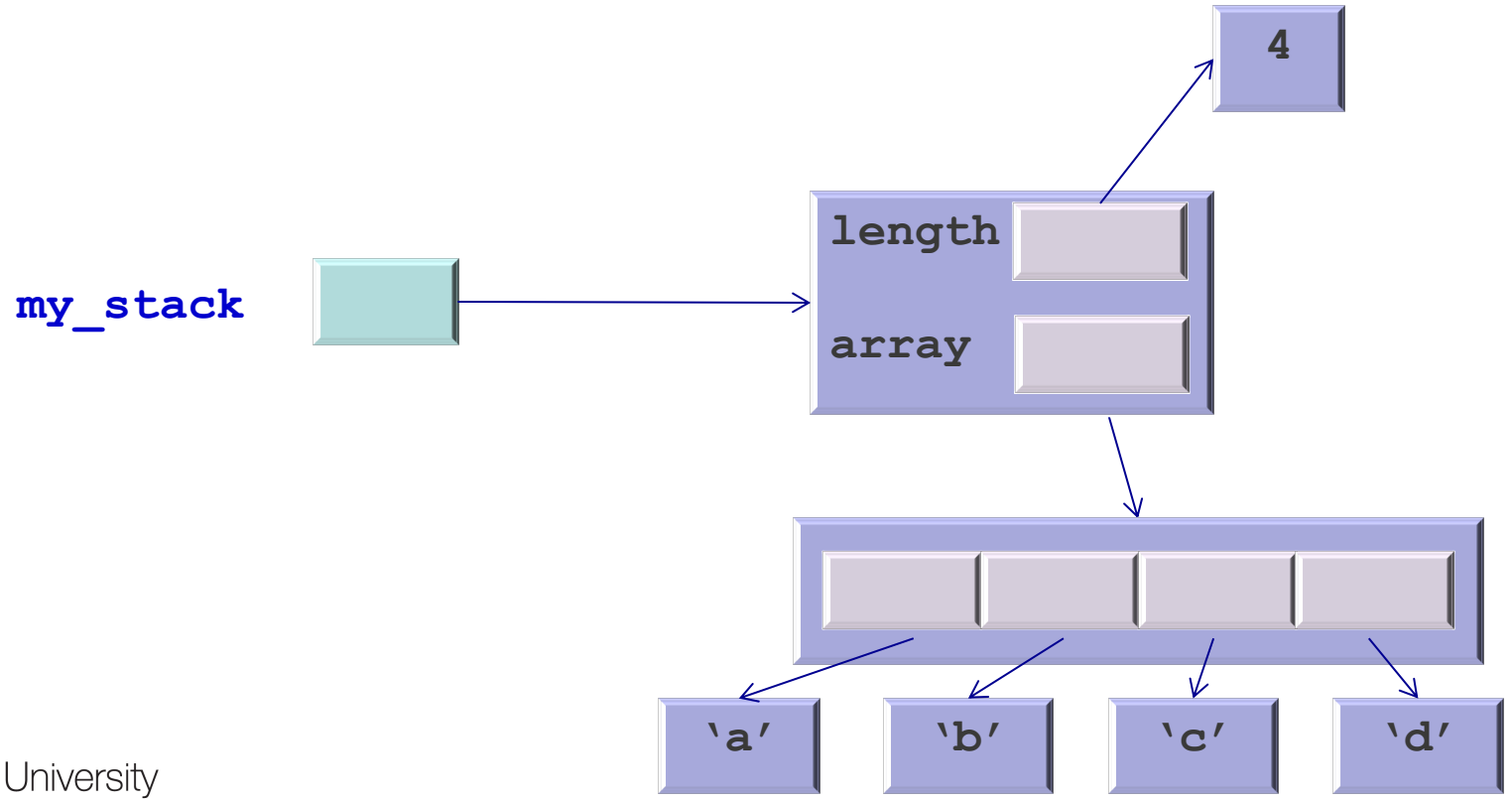
Example: reversing a sequence of chars

input string: “a b c d”



Example: reversing a sequence of chars

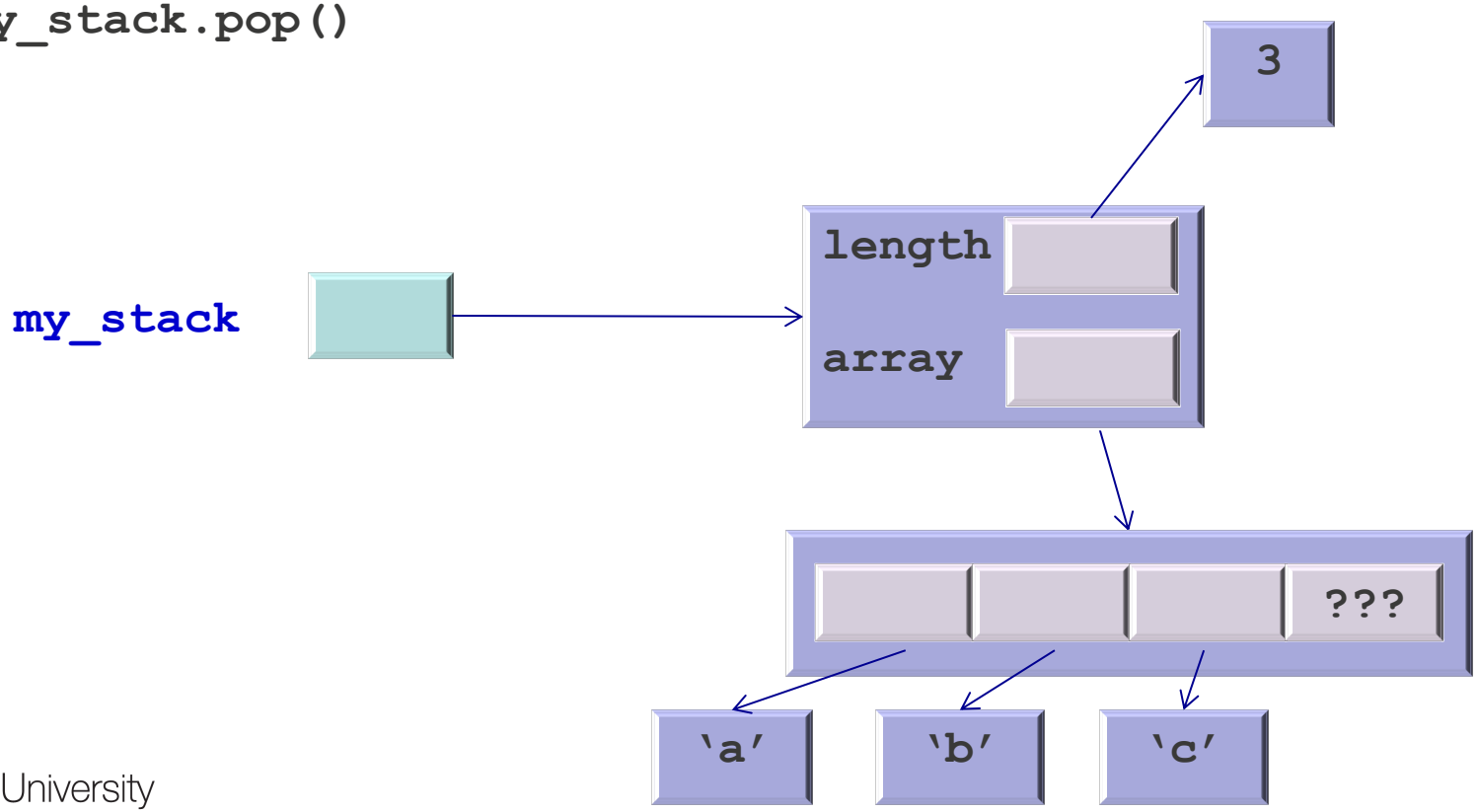
output string: “”



Example: reversing a sequence of chars

output string: “d”

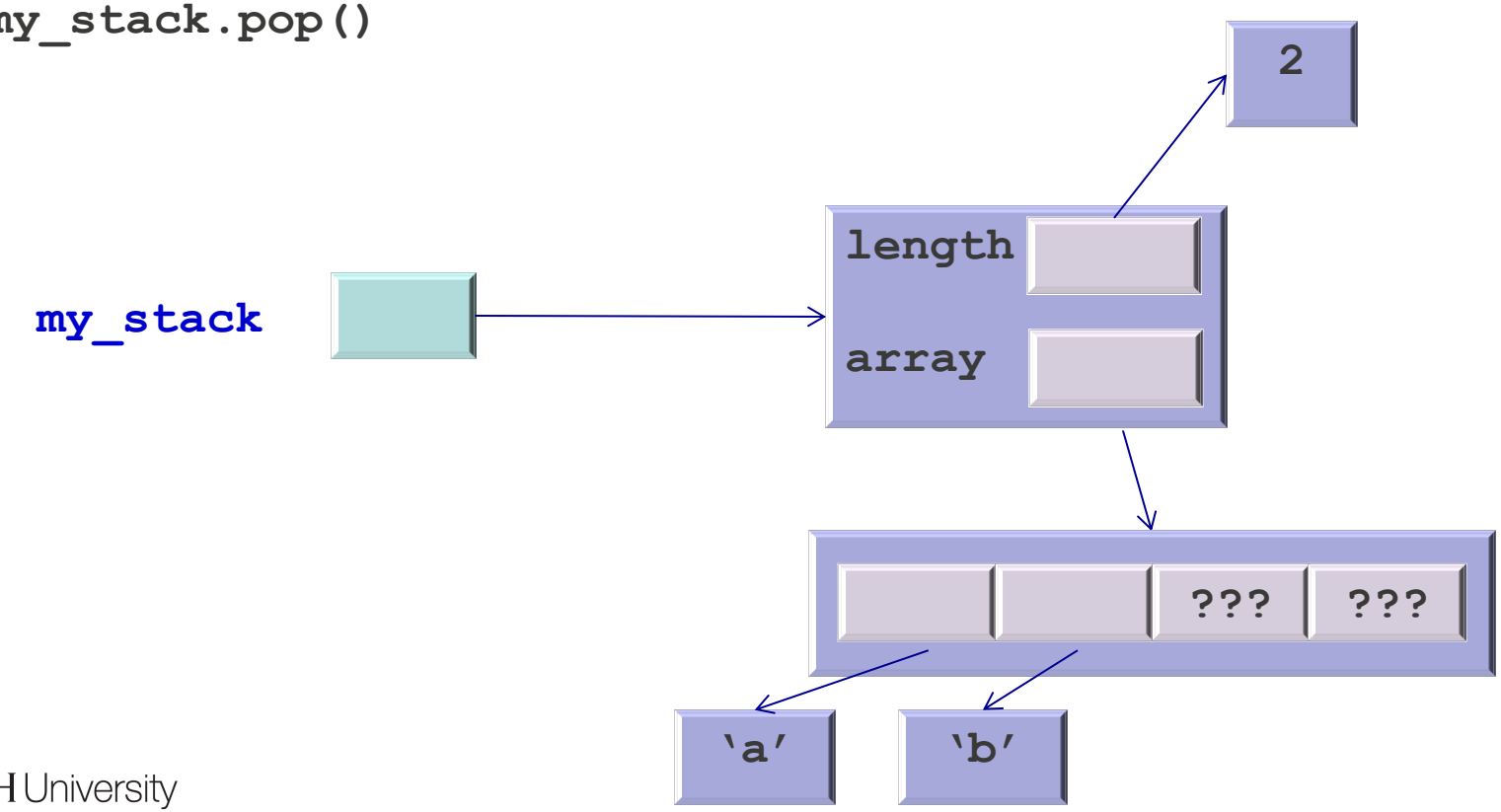
```
my_stack.pop()
```



Example: reversing a sequence of chars

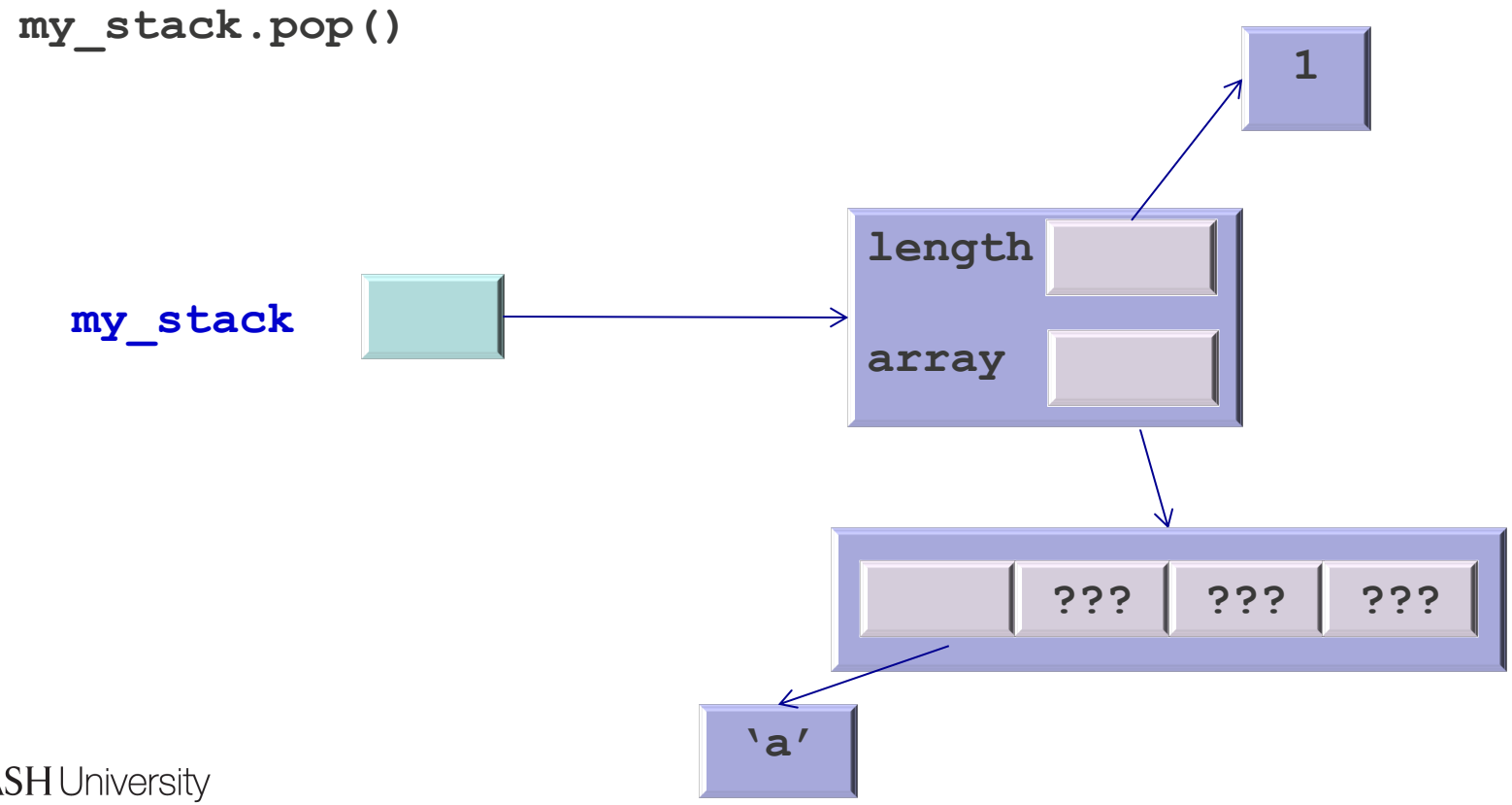
input string: “dc”

```
my_stack.pop()
```



Example: reversing a sequence of chars

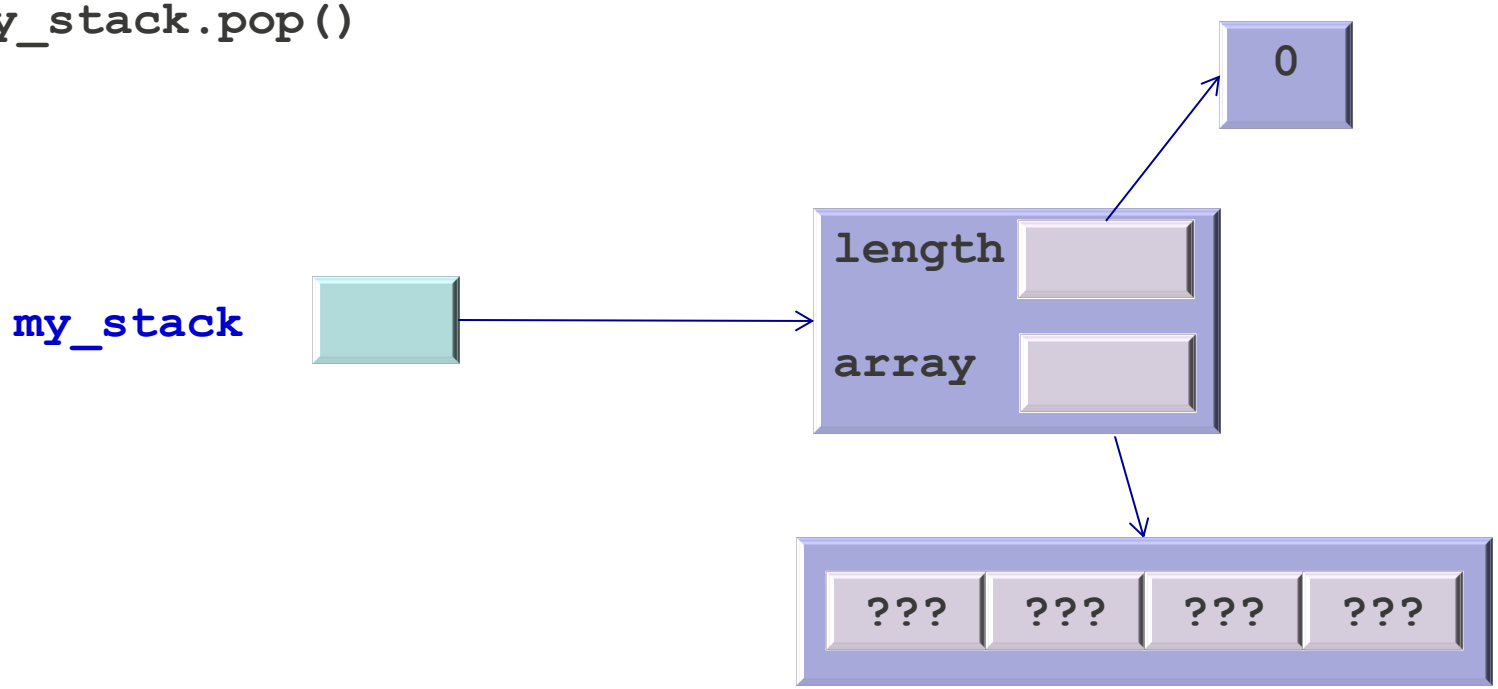
input string: “dcb”



Example: reversing a sequence of chars

input string: “dcba”

```
my_stack.pop()
```



Algorithm for reversing a sequence of chars

- **Create a stack with enough capacity**
  - 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(string: str) -> str:
    my_stack = ArrayStack(len(string))

    for char in string:
        my_stack.push(char)

    output = ""

    while not my_stack.is_empty():
        char = my_stack.pop()
        output += char

    return output
```

pushing

len(string) times

Could be `for _ in string` as the number of elements in the stack and chars in the string are the same

popping

len(string) times

Big O?

Complexity of reverse

- **Two loops, both executed n times (n is the length of the string)**
- **One executed after the other, so we add the complexity (don't multiply)**
- **Inside the:**
  - First loop, the number of operations is constant (push is constant)
  - Second loop is all constant except for + which we do not know, so $Comp_+$ (integer addition is constant, as we saw in MIPS)
- **We get n + n*$Comp_+$ so O(n*$Comp_+$)**
- **Each loop is always performed exactly n times, regardless of the string**
  - best = worst
- **Which gives best = worst = O(n *$Comp_+$)**

```
def reverse(string: str) -> str:
        my_stack = ArrayStack(len(string))
        for char in string:
            my_stack.push(char)
        output = ""
        while not my_stack.is_empty():
            char = my_stack.pop()
            output += char
        return output
```

Another exercise, this time for you

- **Given a string that includes parenthesis of the form { ([ ]) }**
  - Determine if the string has the parenthesis balanced or not
- **Examples:**

```
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

(does not match }

- **Hint: separate the parenthesis into those that open, and those that close.**

```
{ [ (        ) ] }
```
Open **Close**

and relate this to push and pop…

Some Stacks Applications

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

Summary

- **We now understand the concepts of**
  - Data Types
  - Abstract Data Types (ADTs)
  - Data Structures
- **We also know what a Stack ADT is and:**
  - Its main operations
  - Their complexity
- **We are able to:**
  - Implement Stacks using a base abstract class and a derived one using arrays
  - Use them
  - Modify its operations and
  - Reason about their complexity