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

Lecture 17
Stacks & Queues
(Array Implementation)

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

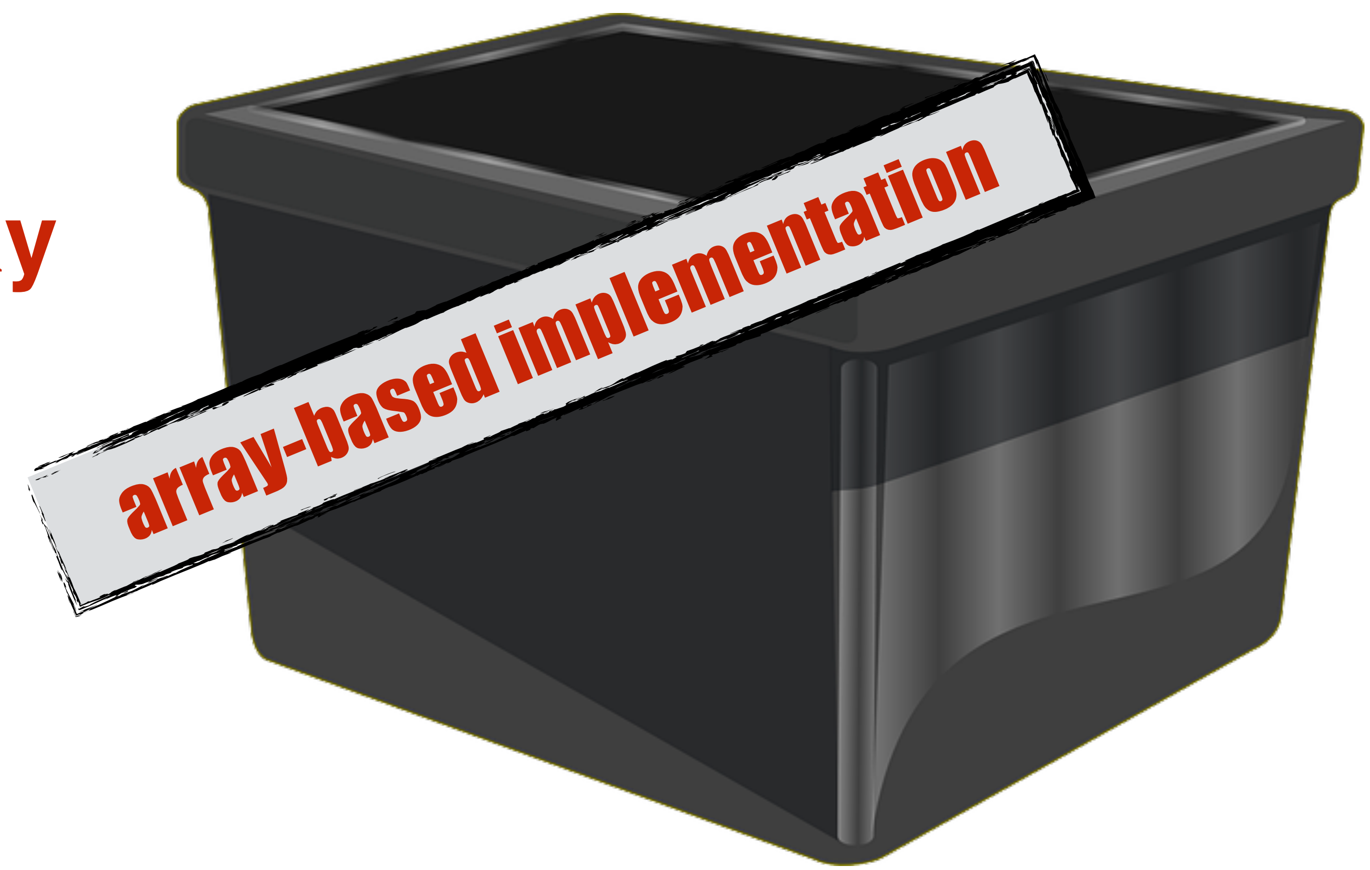
Container ADTs

- **Stores** and removes items **independent of contents.**

- **Examples** include:
  - List ADT
  - Stack ADT
  - Queue ADT.

Today

- Core **operations**:
  - ➡ add item
  - ➡ remove item



LIFO

- **LIFO** (Last In First Out): The last element to arrive, is the first to be processed.

- The last element to be added, is the first to be deleted

- Access to any other element is unnecessary (and thus not allowed).


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 implementation

- Stacks will have the following elements:
  - An **array** to store the items in the order in which they arrive.
  - An **integer** indicating how many items are in the stack.
  - An **integer** indicating which is the top item in the stack.

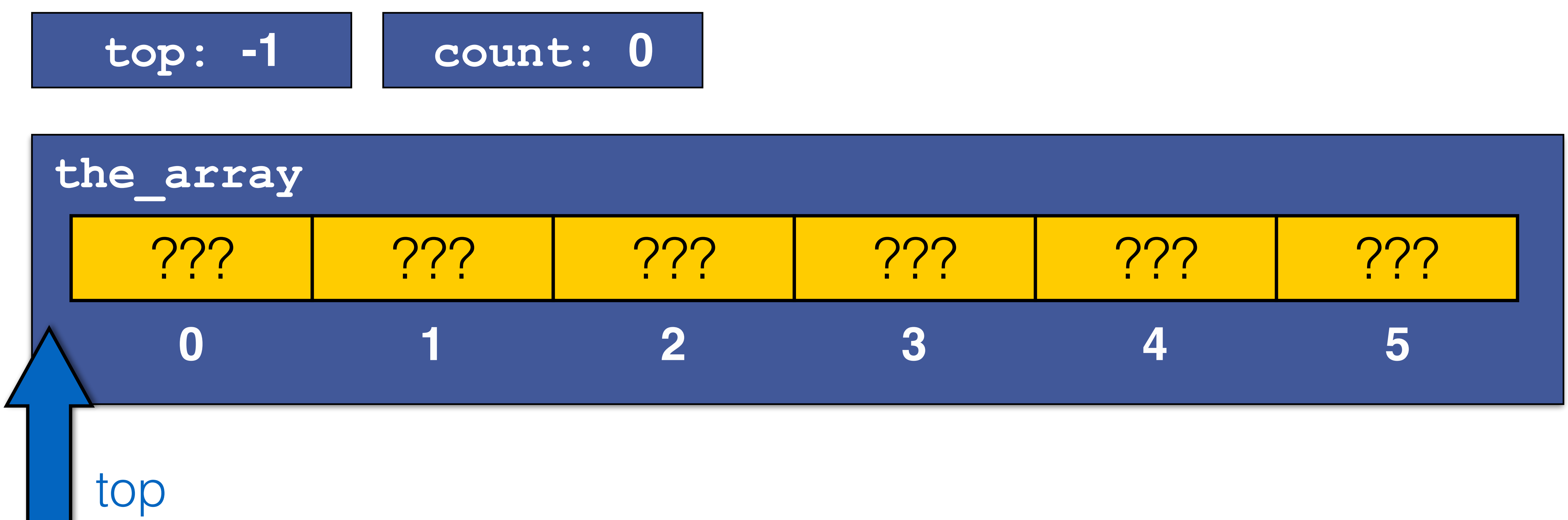
- **Invariant**: valid data in the `0..count-1` positions

- Pretty similar to lists, so what is the difference? The operations provided!
  - **`Stack, is_empty, is_full, size`**
  - **`push, pop, peek`**

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

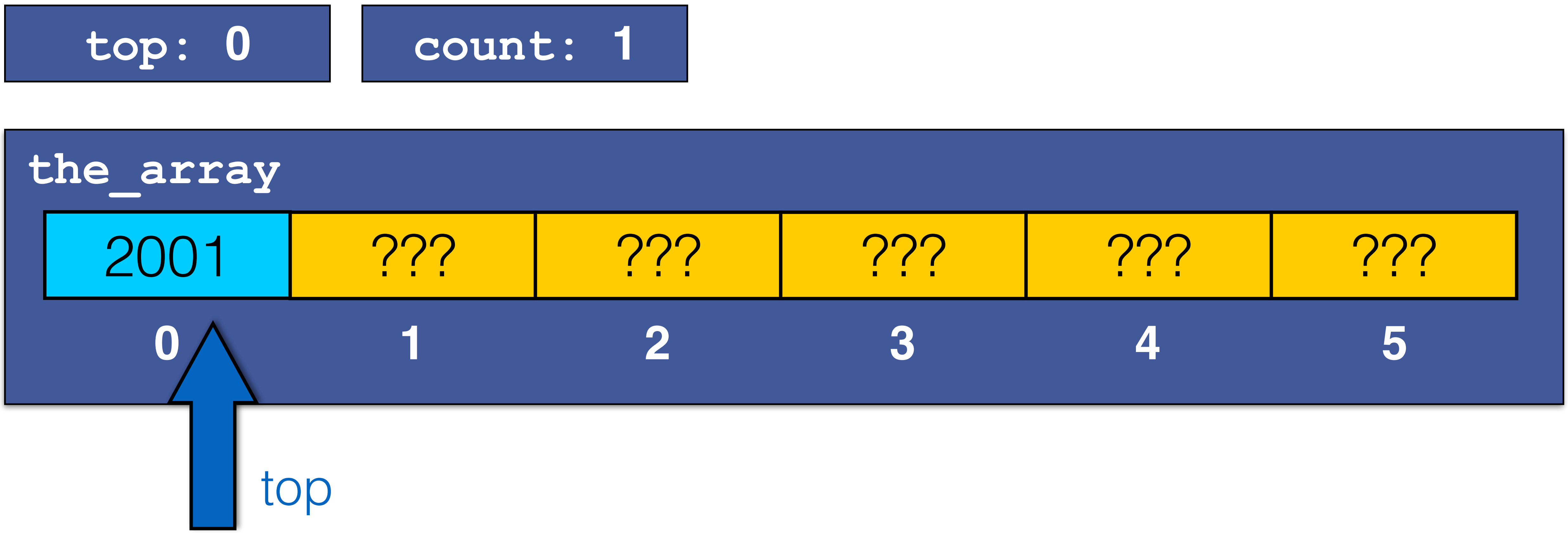
size of underlying array

the_stack = Stack(6)

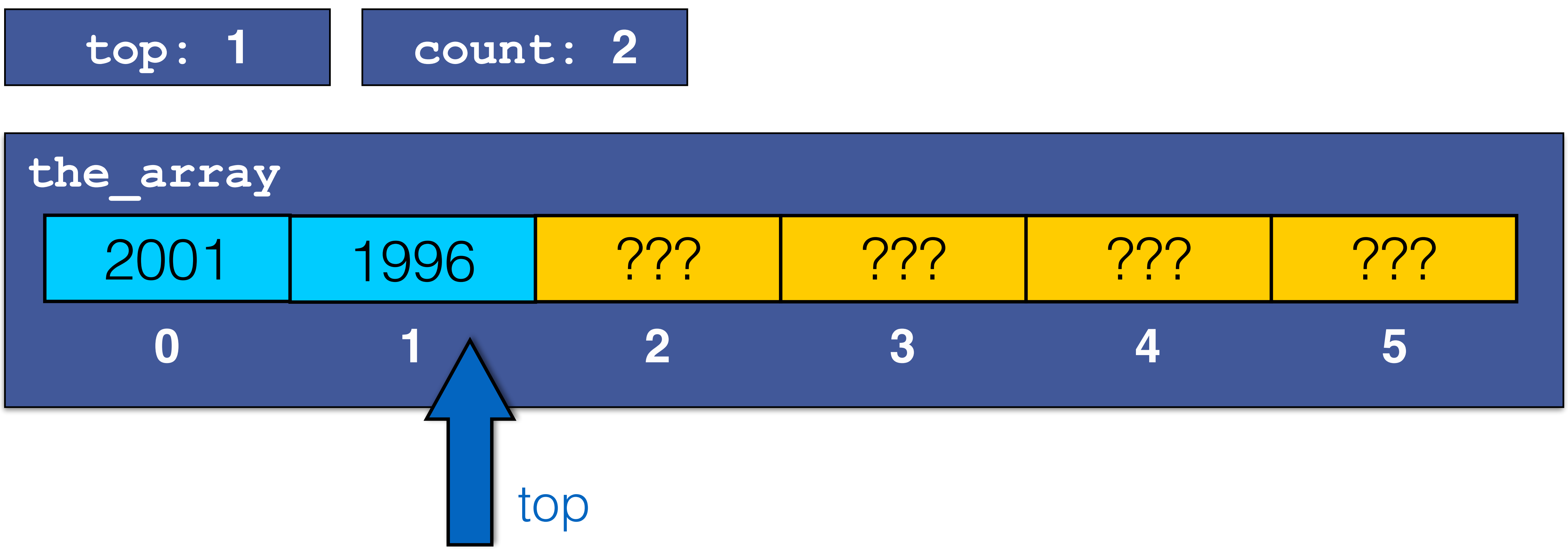


the_stack = Stack(6)

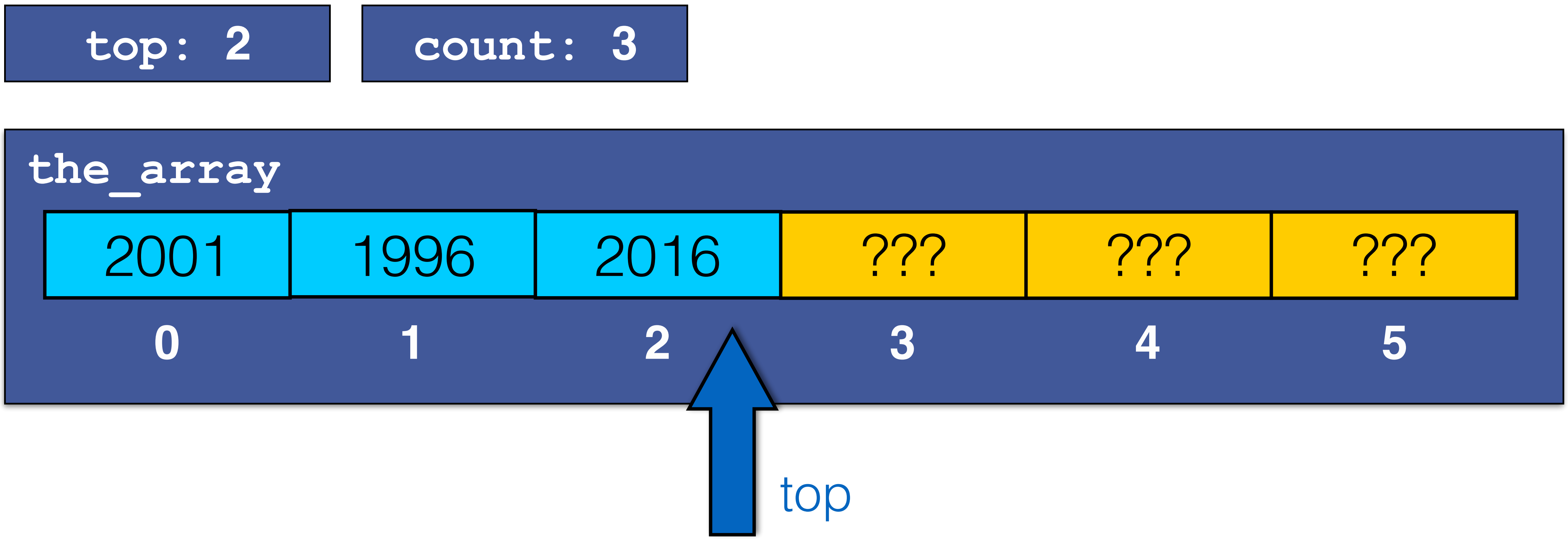
the_stack.push(2001)



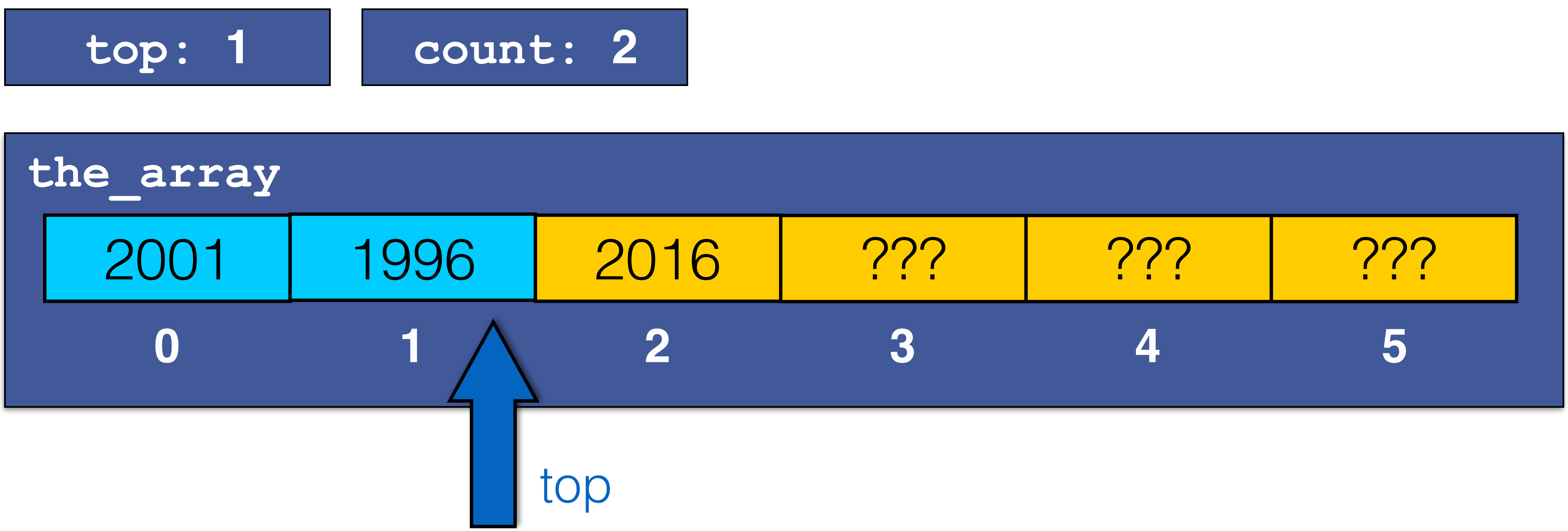
the_stack = Stack(6)
the_stack.push(2001)
the_stack.push(1996)



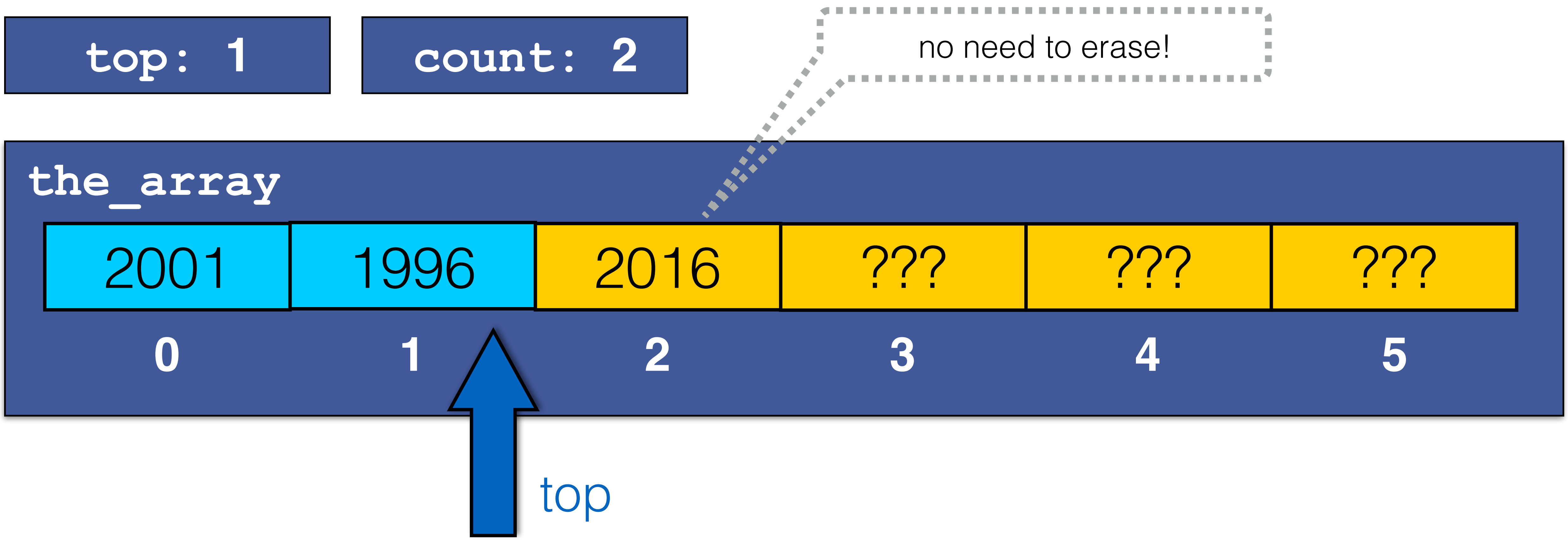
```
the_stack = Stack(6)
the_stack.push(2001)
the_stack.push(1996)
the_stack.push(2016)
```



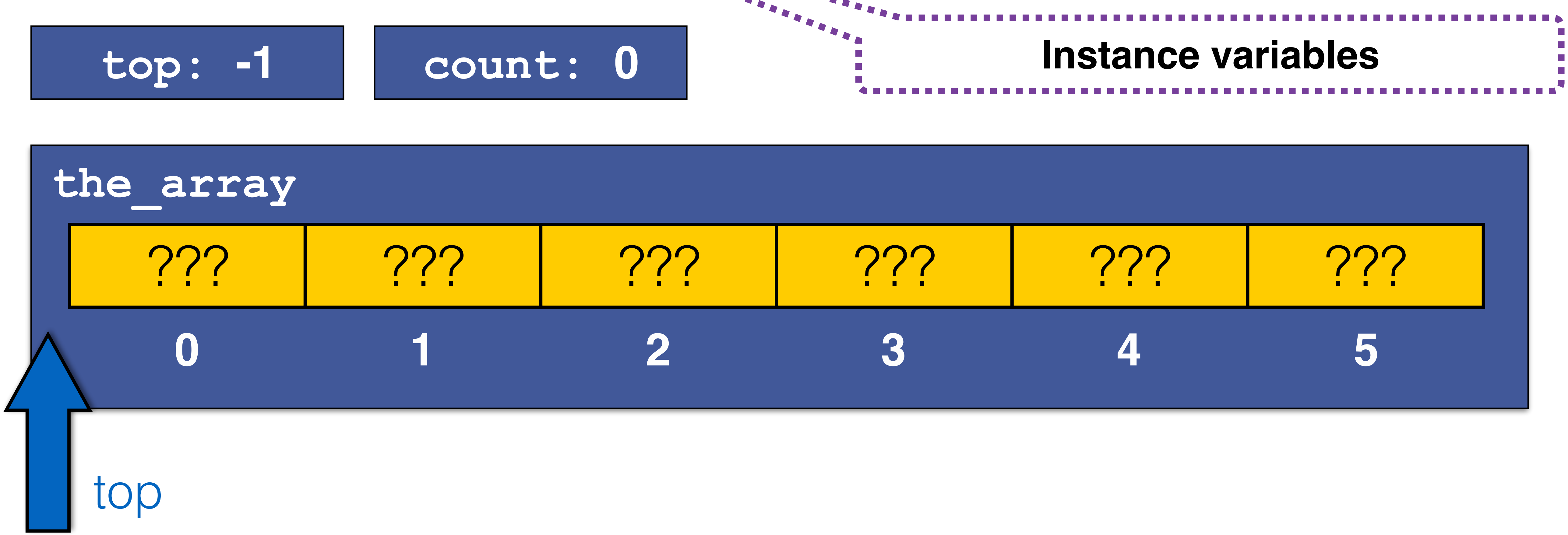
```
the_stack = Stack(6)
the_stack.push(2001)
the_stack.push(1996)
the_stack.push(2016)
the_stack.pop()
```

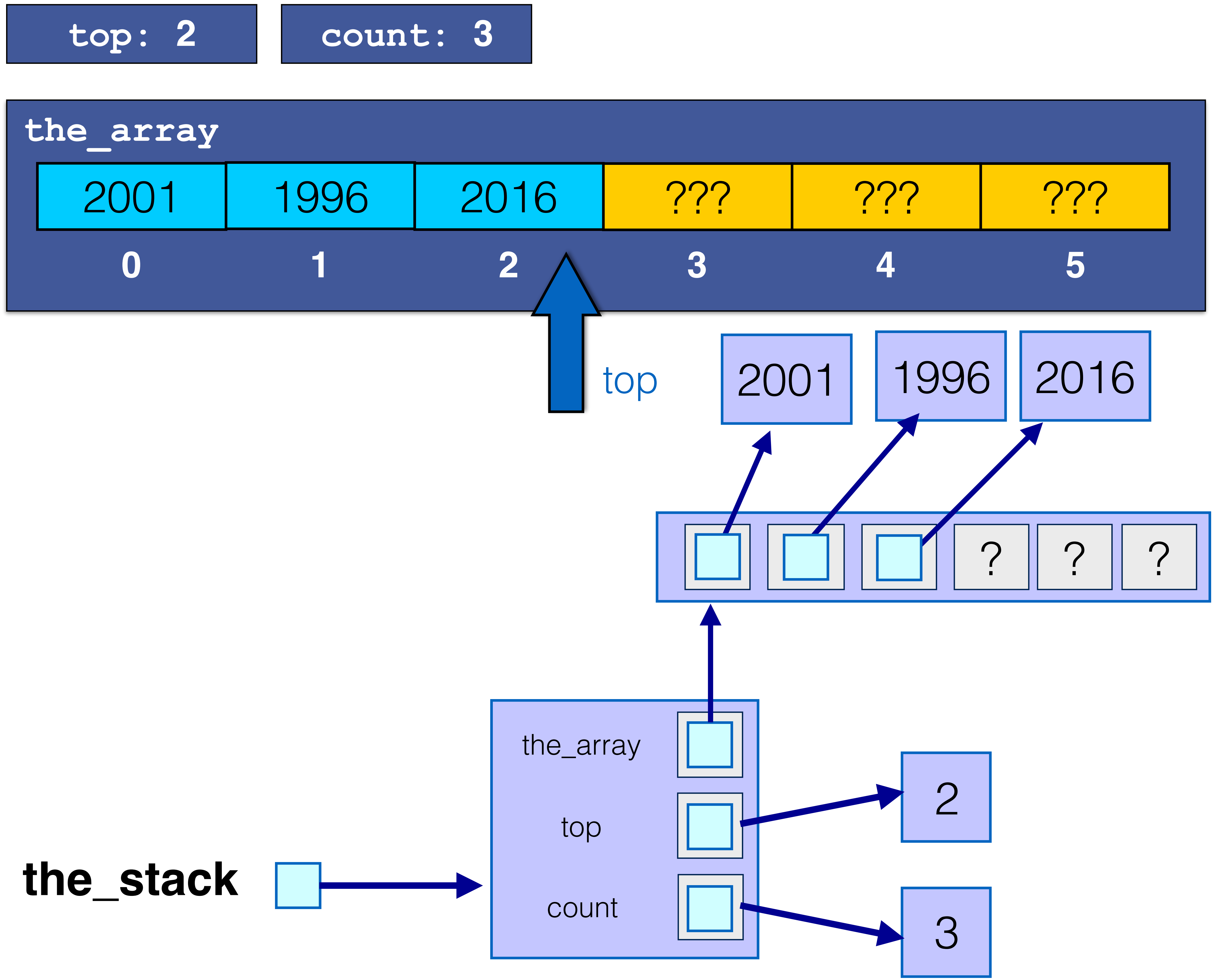


```
the_stack = Stack(6)
the_stack.push(2001)
the_stack.push(1996)
the_stack.push(2016)
the_stack.pop()
```



```python
class Stack:

    def __init__(self, max_capacity):
        if max_capacity <= 0:
            raise ValueError("Size should be positive")
        self.array = build_array(max_capacity)
        self.count = 0
        self.top = -1
```





top: 2
count: 3
the_array
2001
1996
2016
???
???
???
0
1
2
3
4
5
top
2001
1996
2016
?
?
?
the_array
top
count
2
3
the_stack

```python
def push(self, new_item):
    if self.is_full():
        raise Exception("The stack is full")
    self.top+=1
    self.the_array[self.top] = new_item
    self.count +=1
```

```python
def pop(self):
    if self.is_empty():
        raise Exception("The stack is empty")
    item = self.the_array[self.top]
    self.top-=1
    self.count-=1
    return item
```

Complexity is O(1)

Other simple methods

```
def size(self):
    return self.count

def is_empty(self):
    return self.size() == 0

def is_full(self):
    return self.size() >= len(self.the_array)

def reset(self):
    self.count = 0
    self.top = -1
```

Using a Stack.

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**
- Initialise the *output* **string** to empty " "
- **Pop** each element from the stack and **concatenate** it to the output string
- You are **a user** of the stack ADT
  - You have no idea how it is implemented
  - You use methods, NOT the knowledge about how it is implemented with arrays

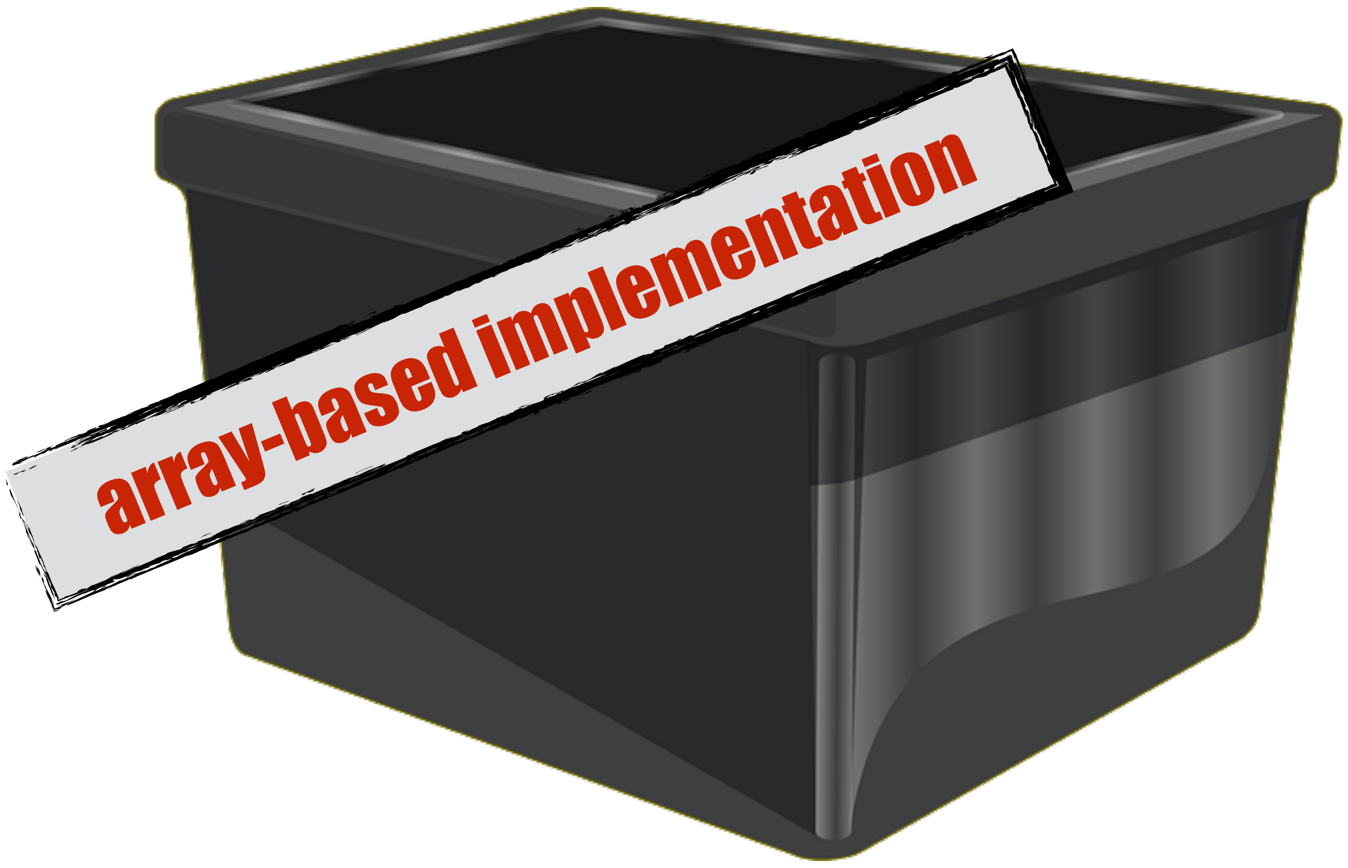
```python
from lecture_17 import Stack


def reverse_string(my_string):
    # create a stack of appropriate size
    string_size = len(my_string)
    my_stack = Stack(string_size)
    # push each character into the stack
    for i in range(0, string_size):
        my_stack.push(my_string[i])

    # create empty output string
    ans = ""
    # pop from the stack
    while not my_stack.is_empty():
        ans = ans + my_stack.pop()
    # ans contains the reversed string
    return ans
```

Some Stacks Applications

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

Container ADTs

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



FIFO

- **FIFO** (First In First Out): The first element to arrive, is the first to be processed
- Data: The first element to be **added**, is the first to be deleted (or **served**)
- Access to any other element is unnecessary (and thus not allowed)

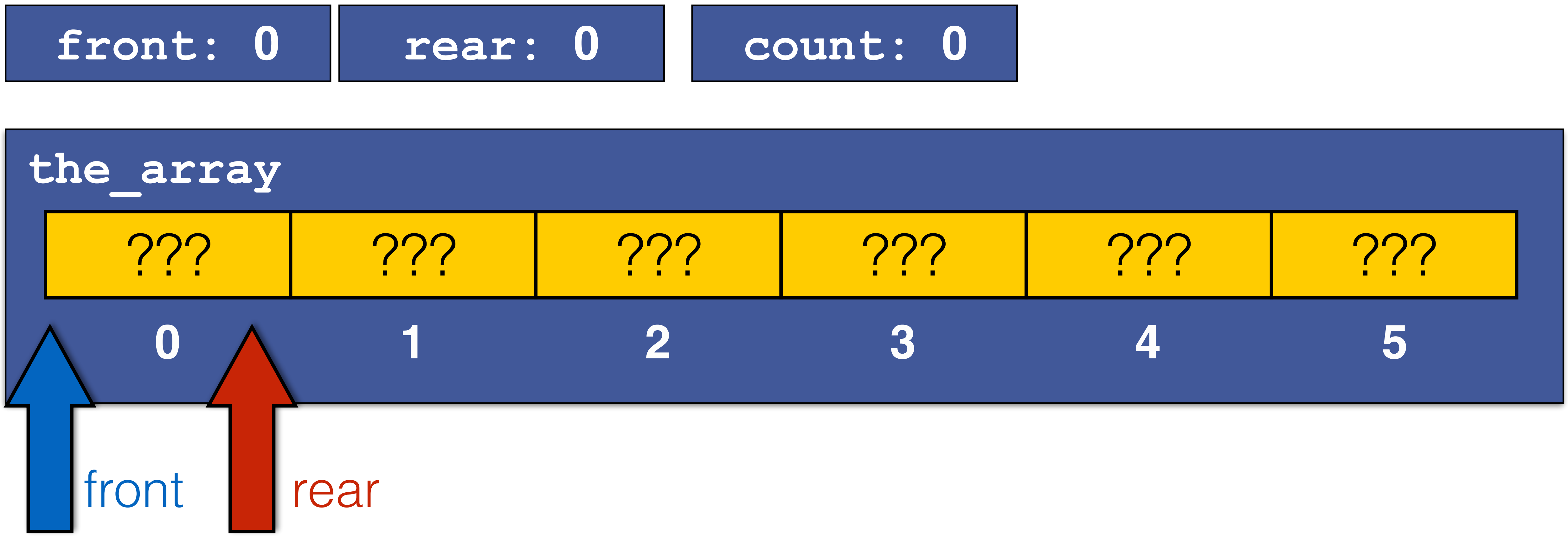
Possible implementation: linear queue

- We need to: **add items** at the **rear**. **take** items from the **front.**

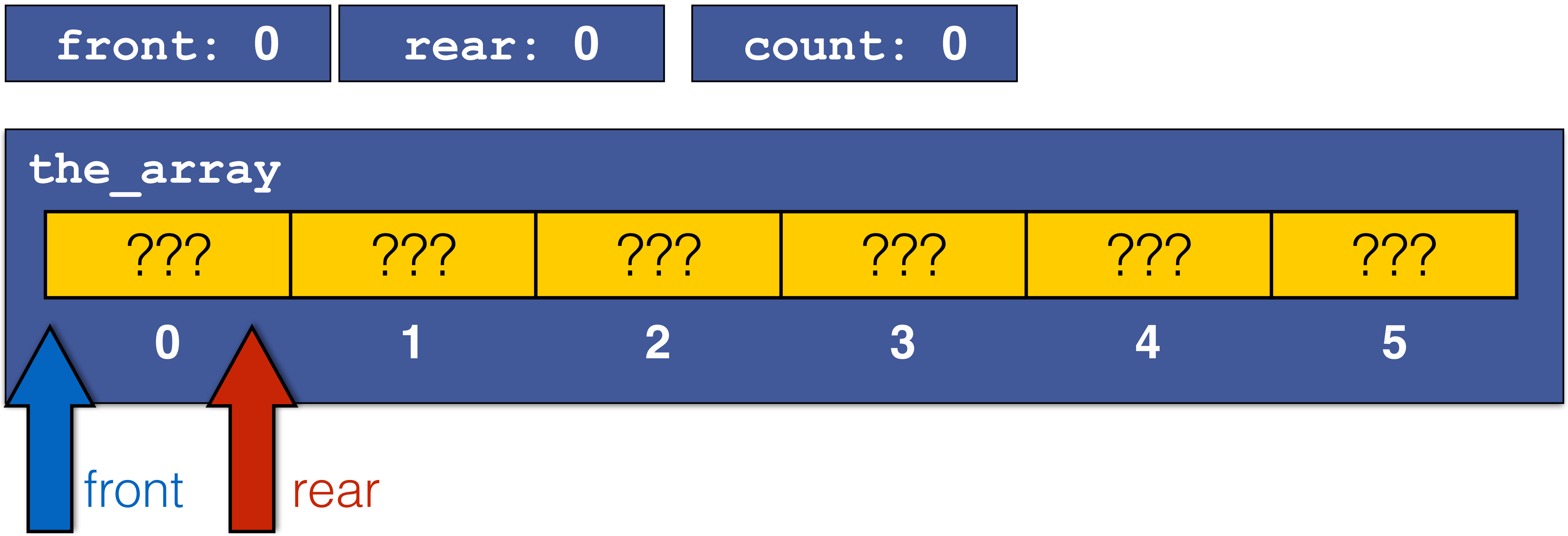
  A single marker is not going to be enough.

- Lets try implementing queues using:
  - An **array** to store the items in the order they arrive.
  - An **integer** marking the front of the queue. Refers to the first element to be served.
  - An **integer** marking the rear of the queue. Refers to the first empty slot at the rear.
  - An integer **count** keeping track of the number of items.

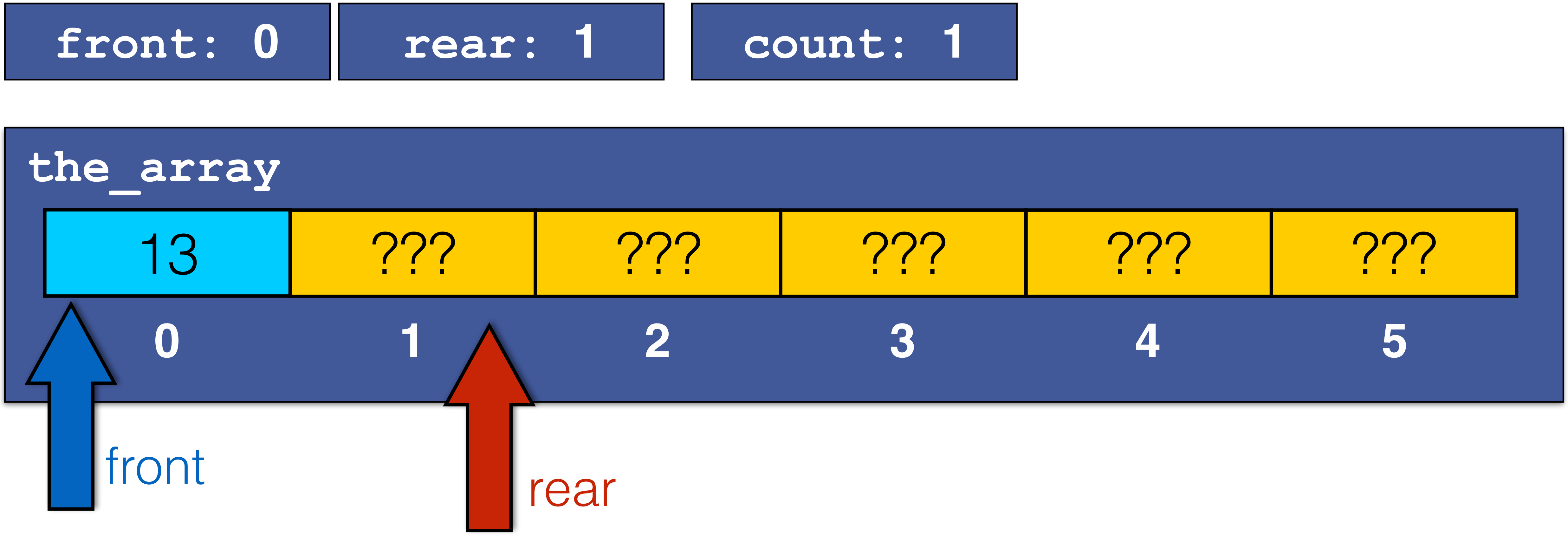
- **Invariant**: valid data appears in `front` … `rear-1` positions


front: 0
rear: 0
count: 0
the_array
???
???
???
???
???
???
0
1
2
3
4
5
front
rear

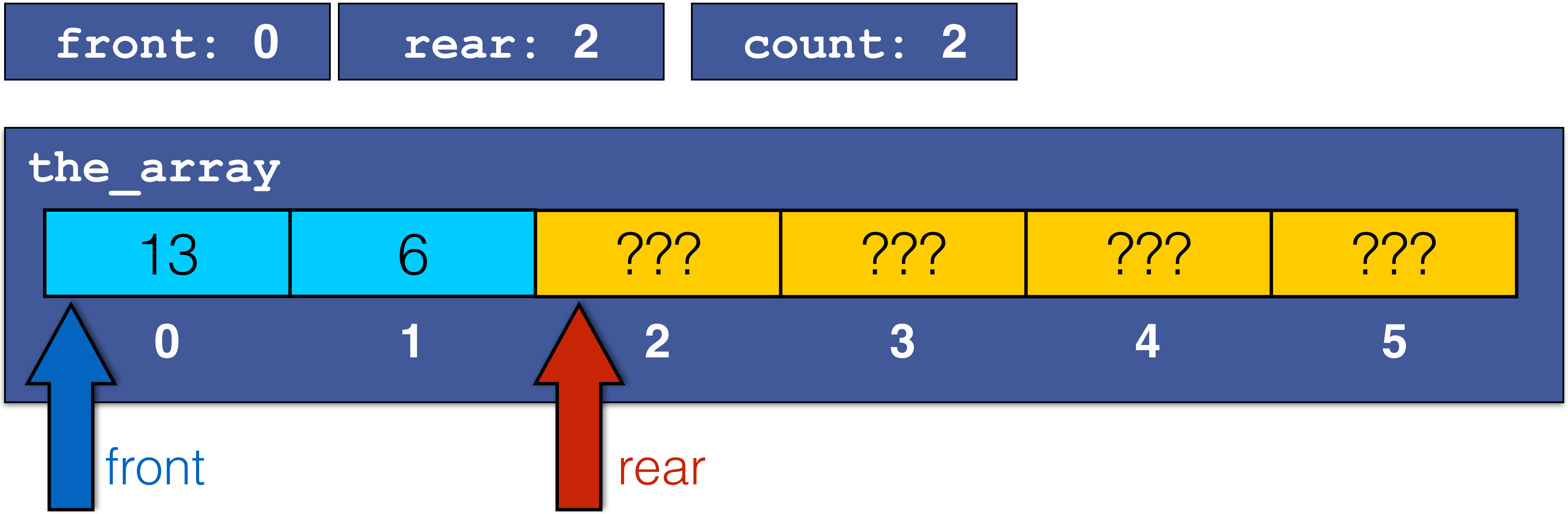
- Create a new queue: no items

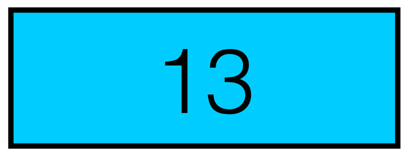


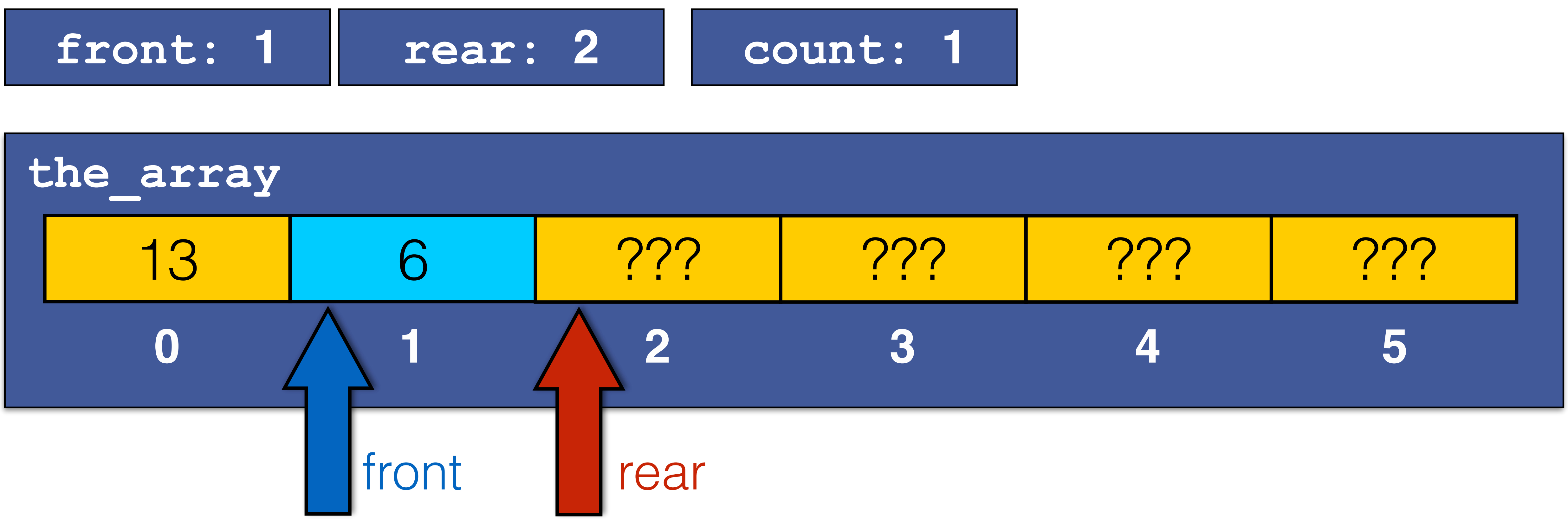
- Create a new queue: no items
- Append item 13



- Create a new queue: no items
- Append item 13
- Append item 6



- Create a new queue: no items
- Append item 13
- Append item 6
- Serve item 



Creating a Queue

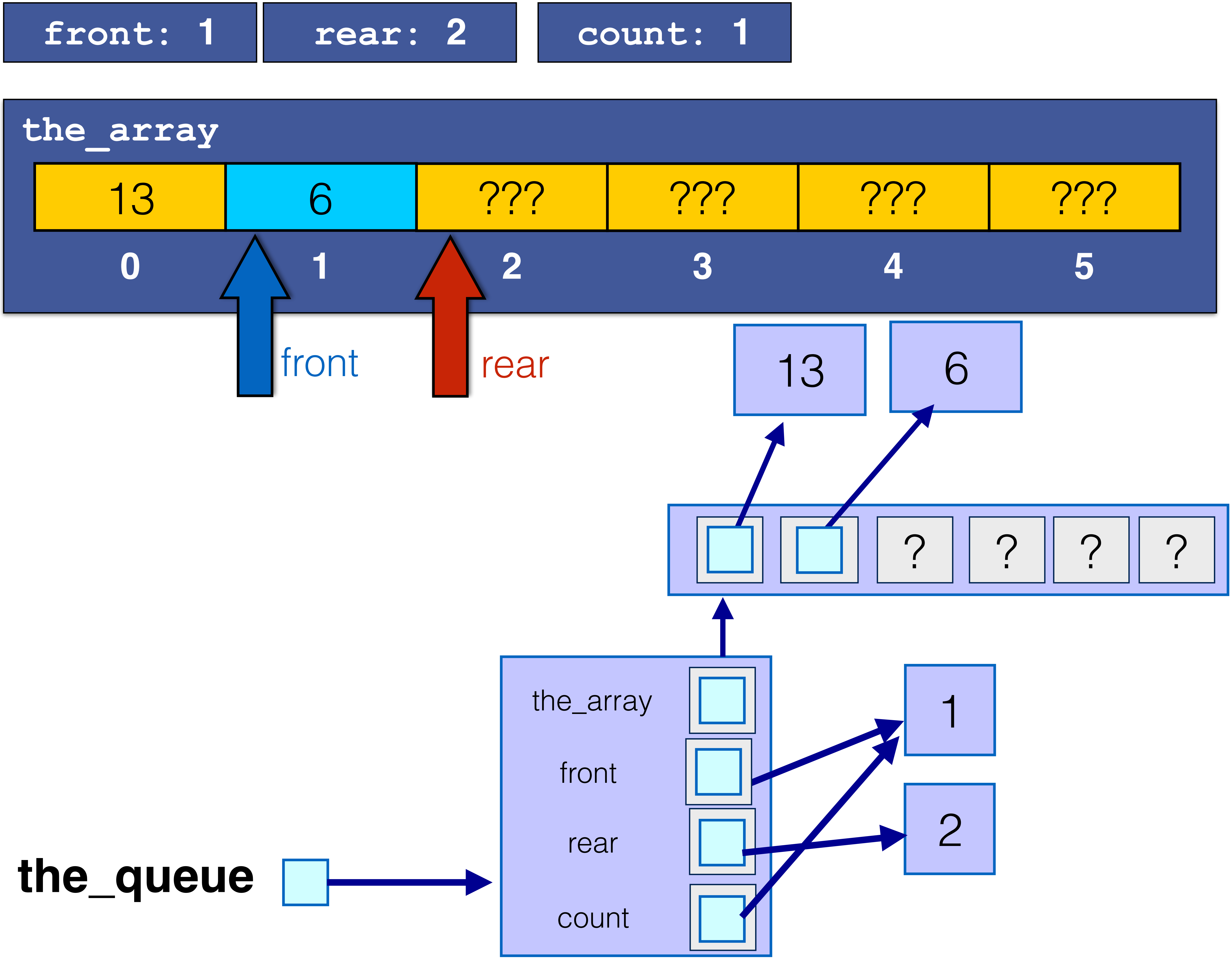
```python
from referential_array import build_array


class Queue:

    def __init__(self, maximum_capacity):
        if maximum_capacity <= 0:
            raise ValueError("Size should be positive")
        self.array = build_array(maximum_capacity)
        self.front = 0
        self.rear = 0
        self.count = 0
```

Creating a Queue

```python
from referential_array import build_array


class Queue:

    def __init__(self, maximum_capacity):
        if maximum_capacity <= 0:
            raise ValueError("Size should be positive")
        self.array = build_array(maximum_capacity)
        self.front = 0
        self.rear = 0
        self.count = 0
```

Instance variables

Complexity is O(N)



front: 1
rear: 2
count: 1
the_array
13
6
???
???
???
???
0
1
2
3
4
5
front
rear
13
6
?
?
?
?
the_array
front
rear
count
1
2
the_queue

Implementing Append

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    self.count += 1
```

Complexity is O(1)

Implementing Serve

```
def serve(self):
    assert not self.is_empty(), "Queue is empty"
    item = self.the_array[self.front]
    self.front +=1
    self.count -=1
    return item
```

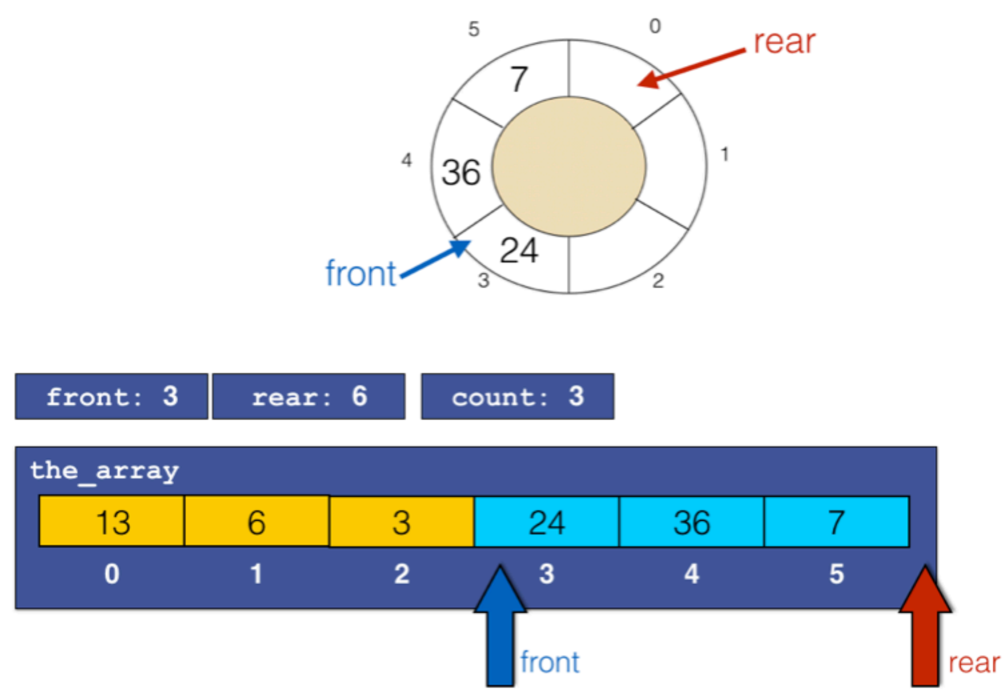
Complexity is O(1)

Some Queue Applications

- Scheduling and buffering
  - Printers
  - Keyboards
  - Executing asynchronous procedure calls

Linear queues waste space…
How to fix this?

A **circular queue** allows for the rear and front to “wrap around” each other



Summary

- Queues
  - Array implementation
    - Linear
    - Circular
  - Basic operations
  - Their complexity