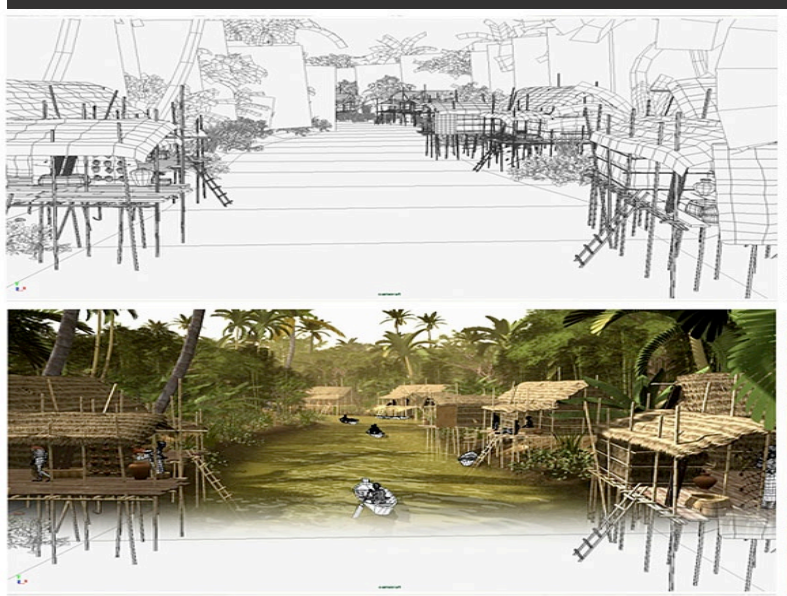
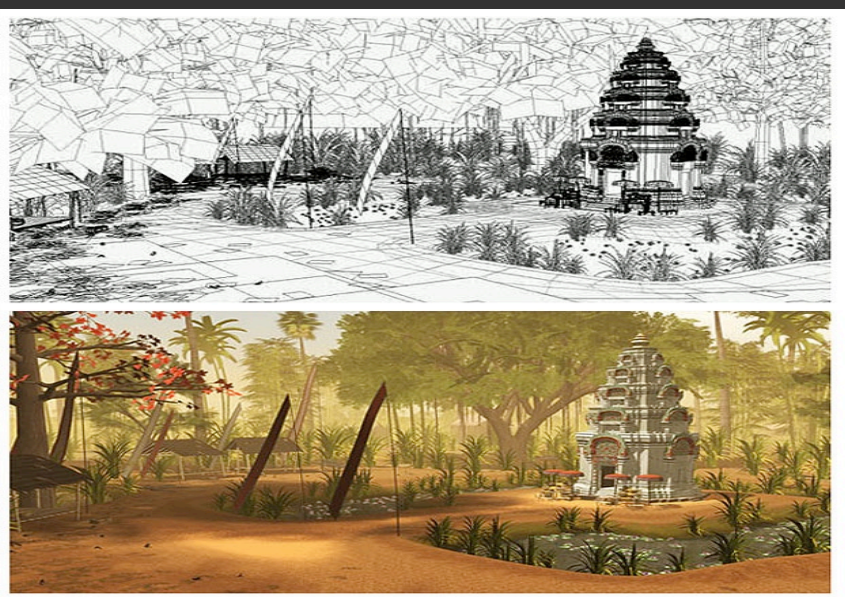
MONASH University

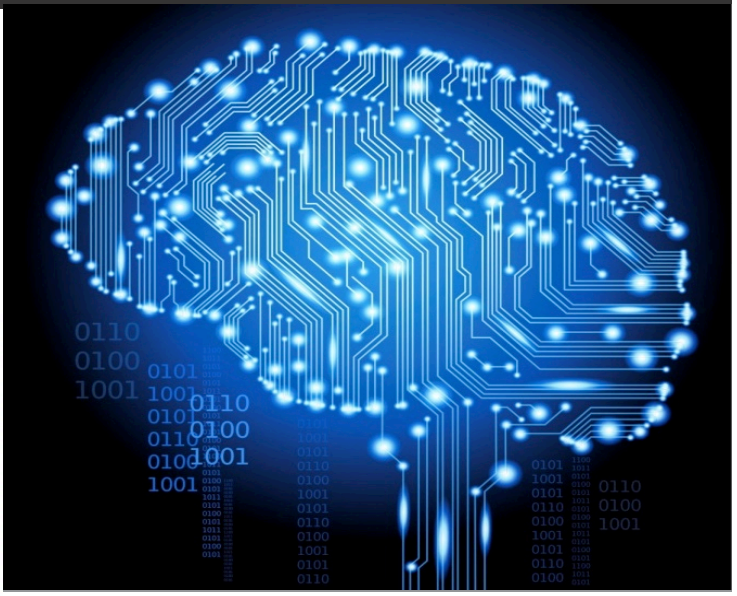
Information Technology

Queue ADT implemented with Arrays

Prepared by:
Maria Garcia de la Banda







Objectives for this lesson

- **Understand the Queue ADT:**
  - Main operations
  - Their complexity
- **To be able to:**
  - Implement Queues with arrays using Linear and a Circular Queue implementations
  - Use them
  - Modify its operations and
  - Reason about the complexity of their operations
  - Reason about the advantages of Circular versus Linear implementations
- **To be able to decide when it is appropriate to use a Stack and a Queue**

MONASH University

Queue ADT

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

- **An ADT that is used to store items**
- **And its operations follow a First In First Out (FIFO) process**
  - The first element to be added, is the first to be deleted (to be processed)
- **And access to any other element is unnecessary (and not allowed)**
  - If you need to access another element, choose a different ADT…

Image by <a href="https://pixabay.com/users/sheadquarters-5187/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=50119">Sabine

Main Queue Operations

- **The key operations are:**
  - Create the queue (Queue)
  - Add an item to the back (append)
  - Take an item off the front (serve)
- **Other common operations include:**
  - What is its length?
  - Is the queue empty?
  - Is the queue full?
  - Empty the queue (clear)

- **Remember: you can only access the element at the front of the queue (first item inserted that is still in)**

Abstract base Queue class

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

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

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

    @abstractmethod
    def serve(self) -> 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
```

Very similar to the Stack class: changes in blue

Main differences: methods append and serve

```
>>> from abstract_queue import Queue
>>> x = Queue()
Traceback (most recent call last):
…
TypeError: Can't instantiate abstract class
Queue with abstract methods append, is_full,
clear and serve
>>>
```

It behaves as expected: it cannot be instantiated

MONASH University

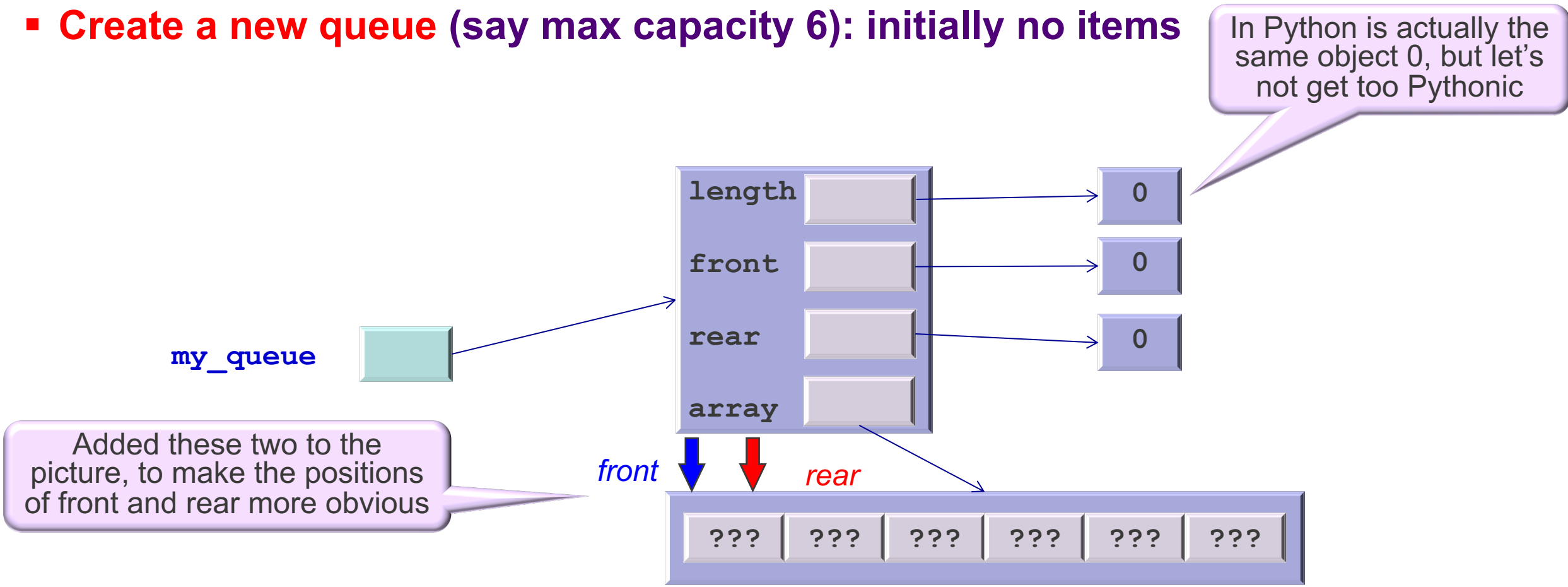
Linear Queues

Possible array implementation: linear queue

- **We need to:**
  - Add items at the rear of the queue (append)
  - Take items from the front (serve)
- **Can we use length to mark the rear? As we will see, no we cannot.**
- **We will implement queues using:**
  - An array to store the items in the order they arrive
  - An integer marking the front of the queue
    - Points to the first element to be served
  - An integer marking the rear of the queue
    - Points to the first empty cell at the rear
  - An integer indicating the number of items in the queue (the inherited length)
- **Invariant: valid data appears from front to rear-1, and rear-front equals the length**

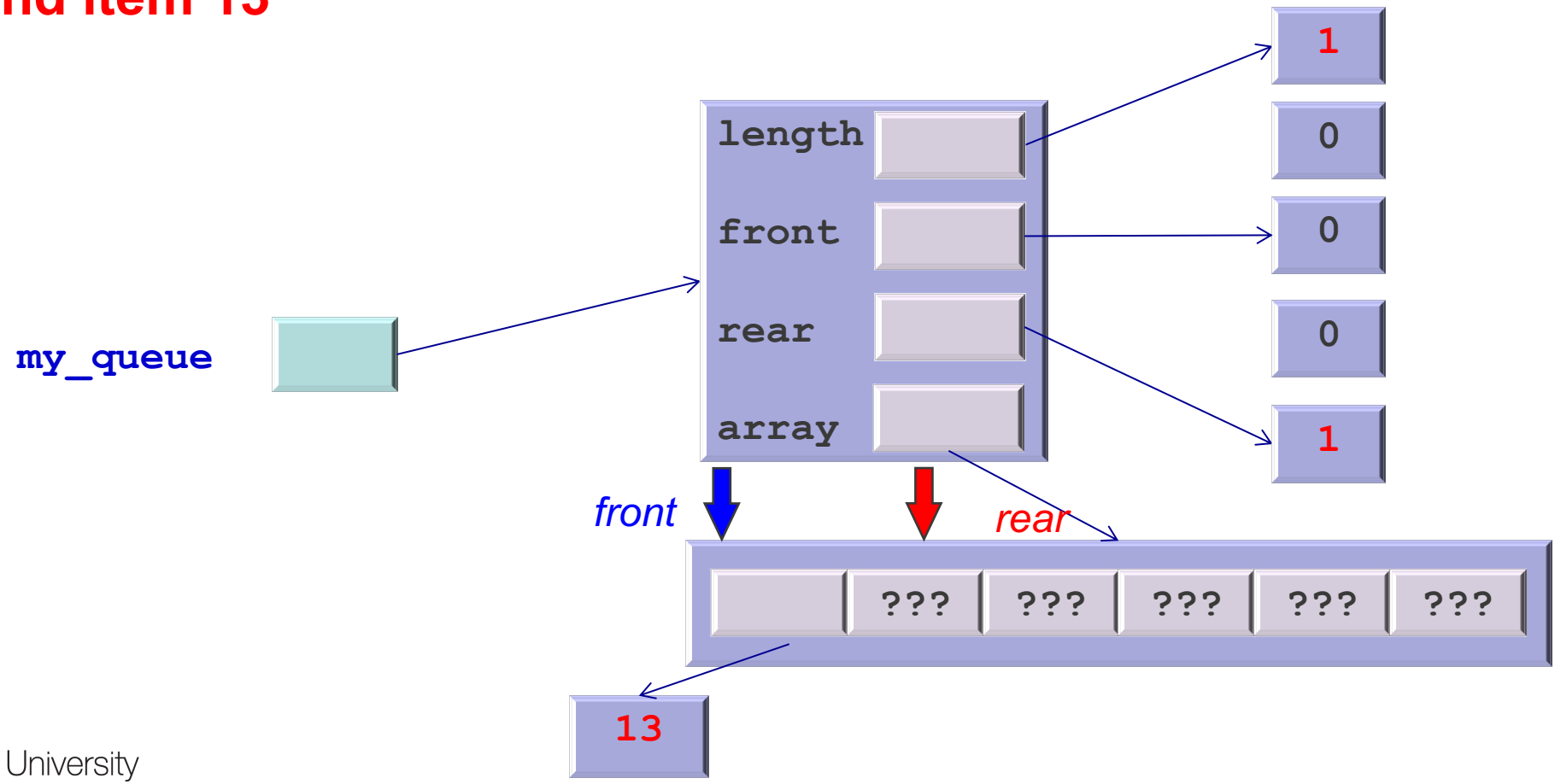
Possible array implementation: linear queue

- **Create a new queue (say max capacity 6): initially no items**



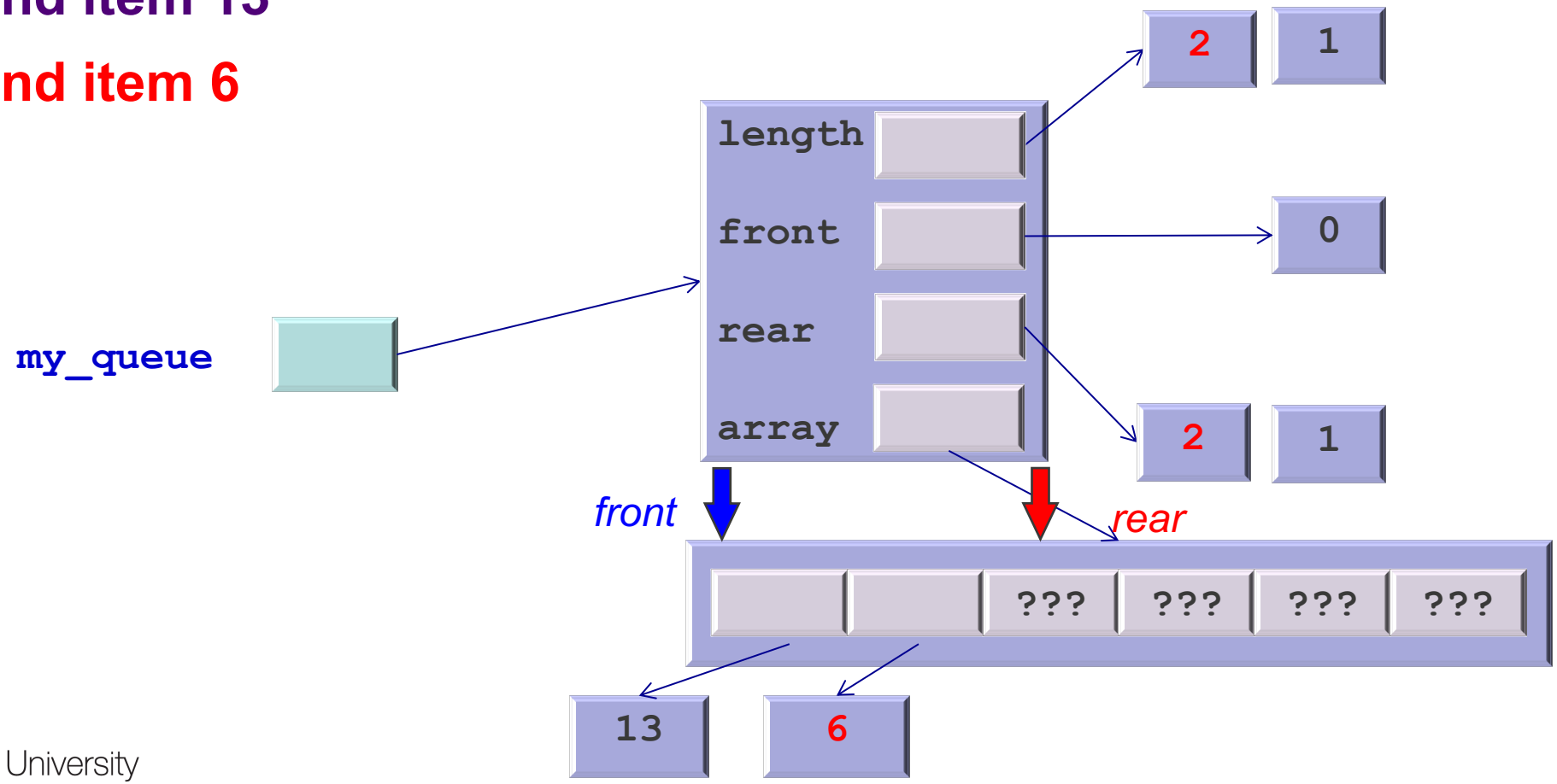
Possible array implementation: linear queue

- Create a new queue (say max capacity 6): initially no items
- Append item 13



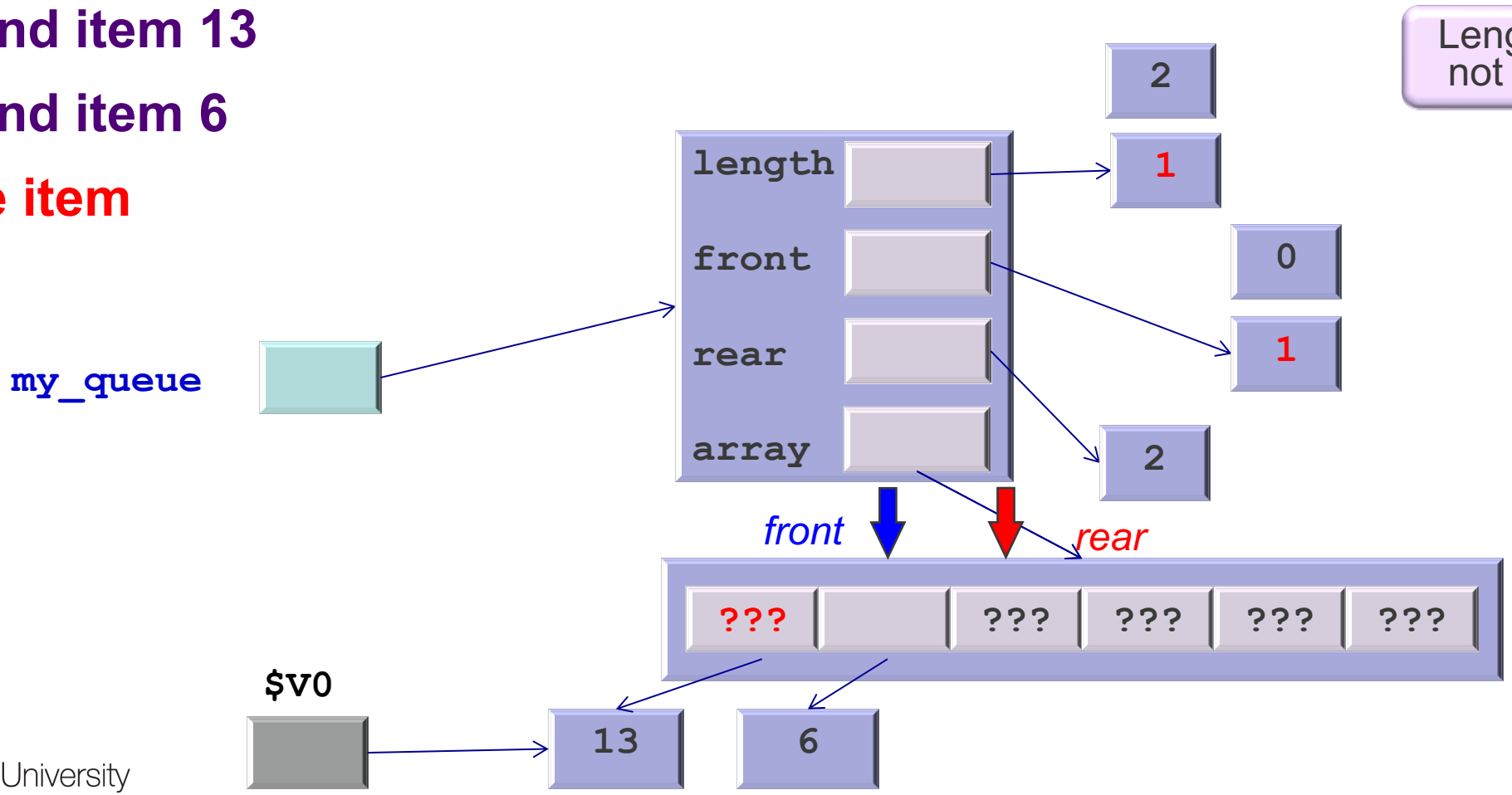
Possible array implementation: linear queue

- Create a new queue (say max capacity 6): initially no items
- Append item 13
- Append item 6



Possible array implementation: linear queue

- Create a new queue (say max capacity 6): initially no items
- Append item 13
- Append item 6
- Serve item



MONASH University

Linear Queue implementation

Let's start Implementing LinearQueue

```
from referential_array import ArrayR
from abstract_queue import Queue, T


class LinearQueue(Queue[T]):
    MIN_CAPACITY = 1

    def __init__(self,max_capacity:int) -> None:
        Queue.__init__(self)
        self.front = 0
        self.rear = 0
        self.array = ArrayR(max(self.MIN_CAPACITY,max_capacity))

    def clear(self) -> None:
        Queue.__init__(self)
        self.front = 0
        self.rear = 0
```

Big O?

All return statements, assignments and integer comparisons are always constant

`Queue.__init__` is also known to be O(1)

Therefore `clear` 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)

The parent class

Minimum capacity of the array again

Uses the method of the parent class

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

Implementing append

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

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

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```

Again, we could 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

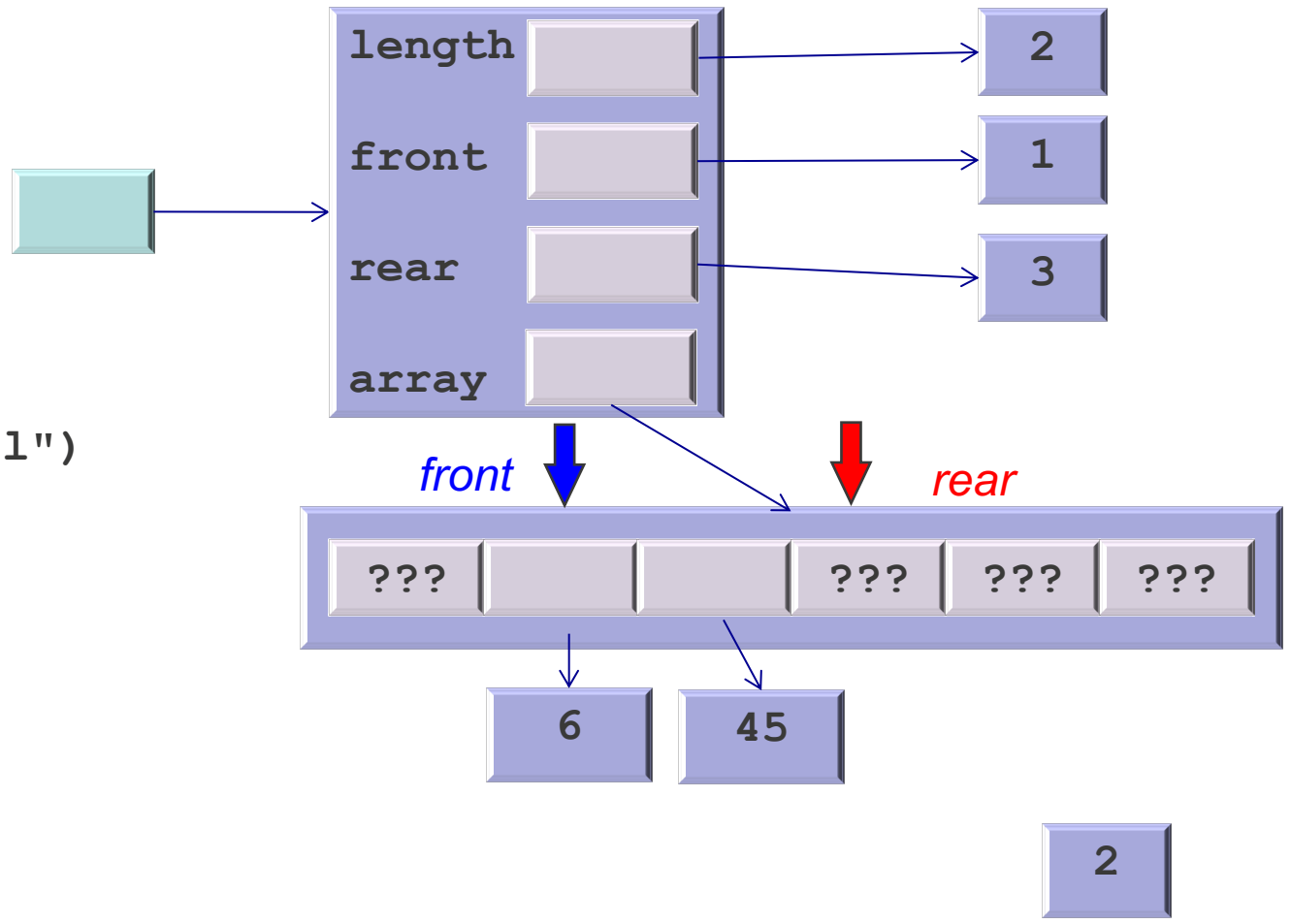
Even better, we could decide that if full, we must increase the size of the array! Leave that to you!

reusing `is_full`: good

Executing append

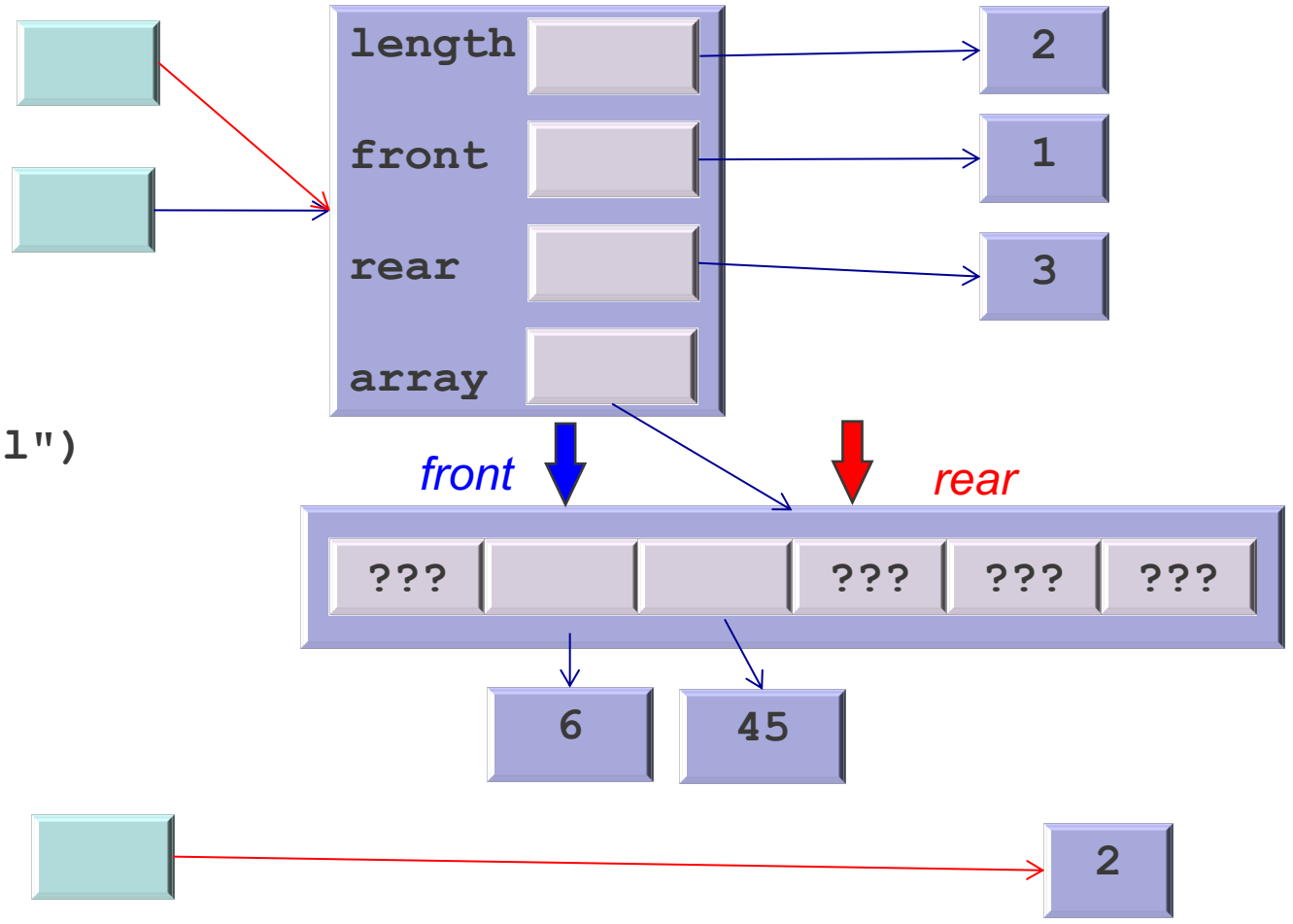
`append(my_queue,2)`

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

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```



Executing append

```
append(my_queue,2)
```

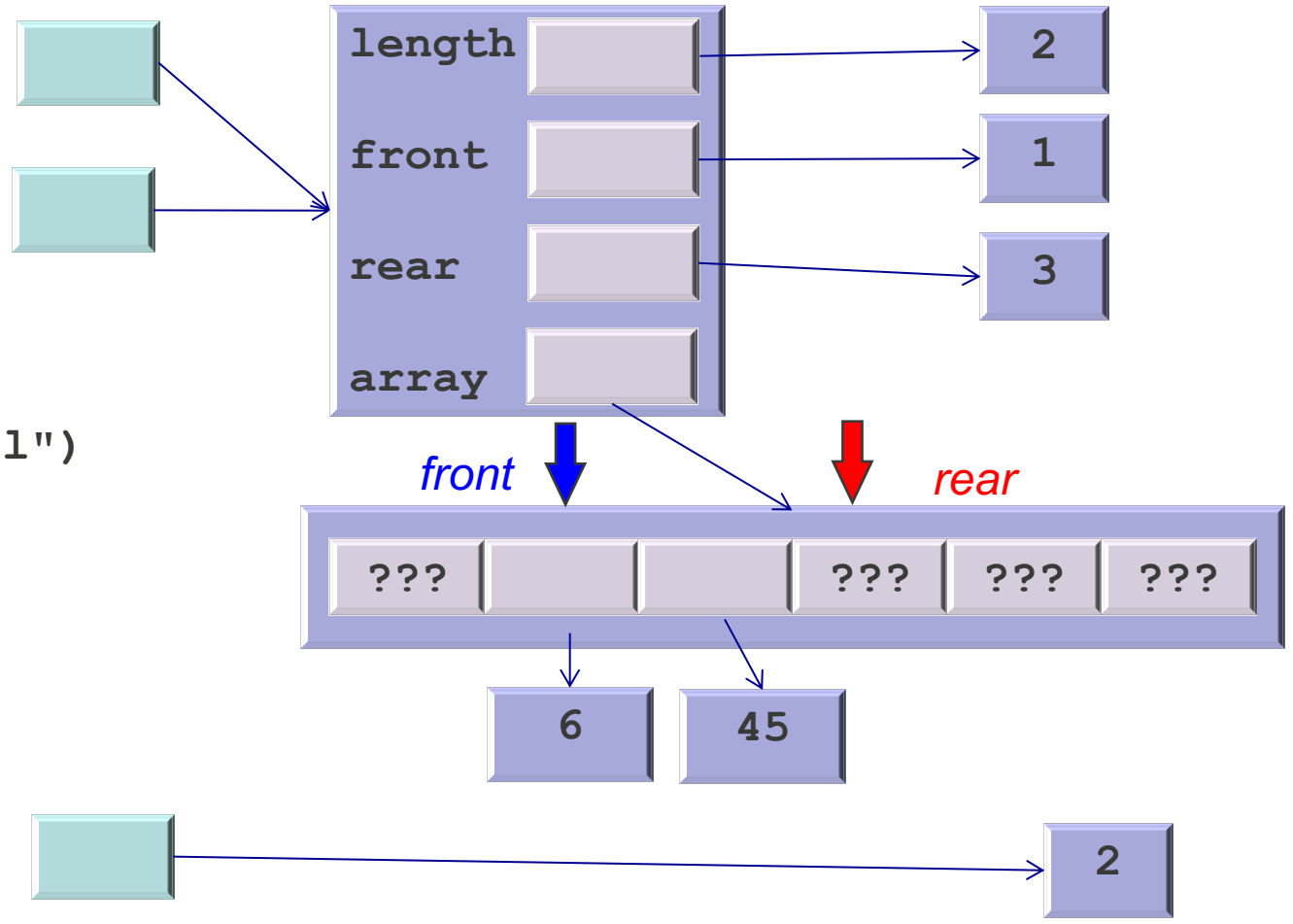



```
def append(self, item: T) -> None:
    if self.is_full():
        raise Exception("Queue is full")

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```

Executing append

```
append(my_queue,2)
```

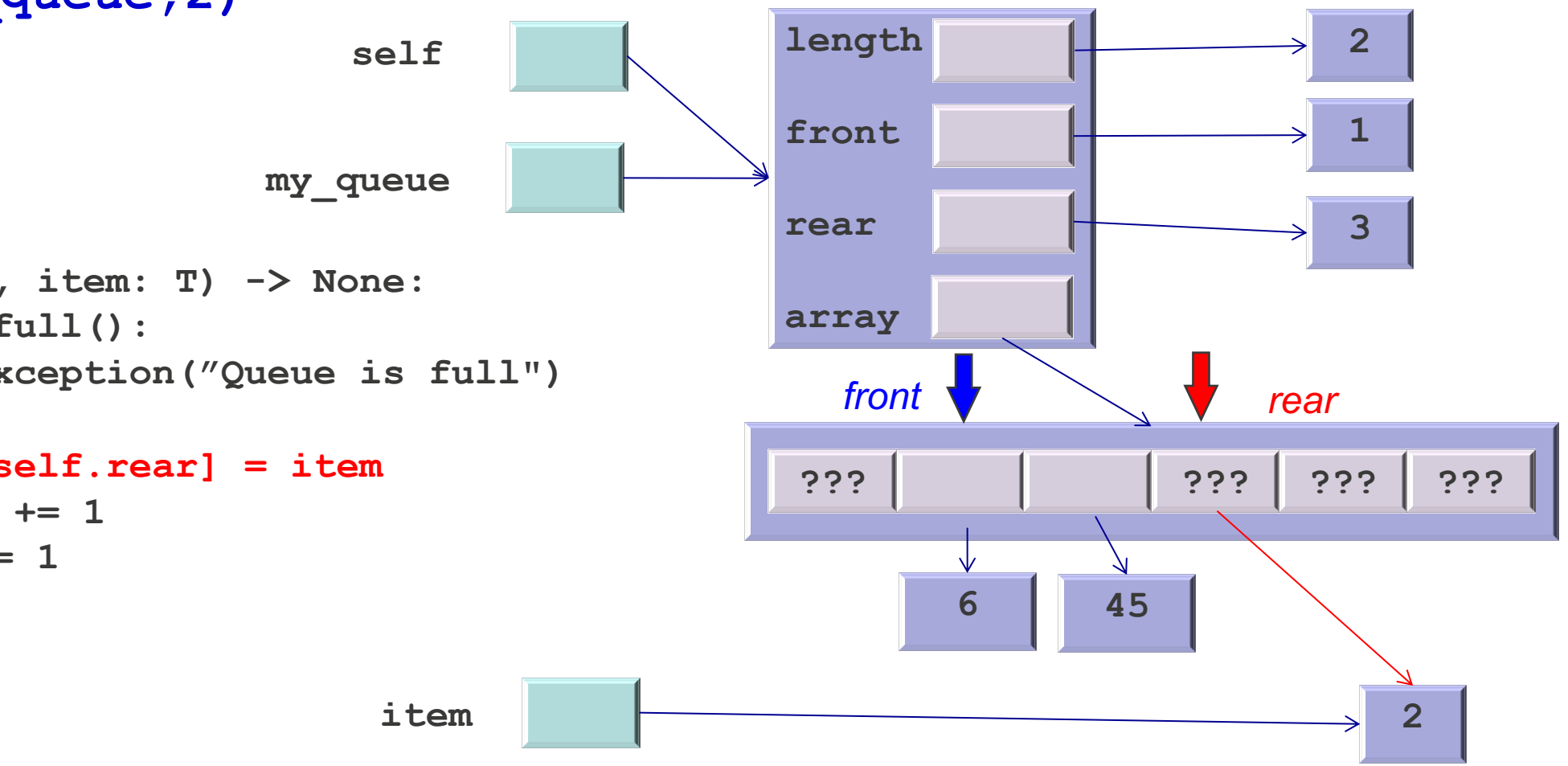
```
def append(self, item: T) -> None:
    if self.is_full():
        raise Exception("Queue is full")

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```



Executing append

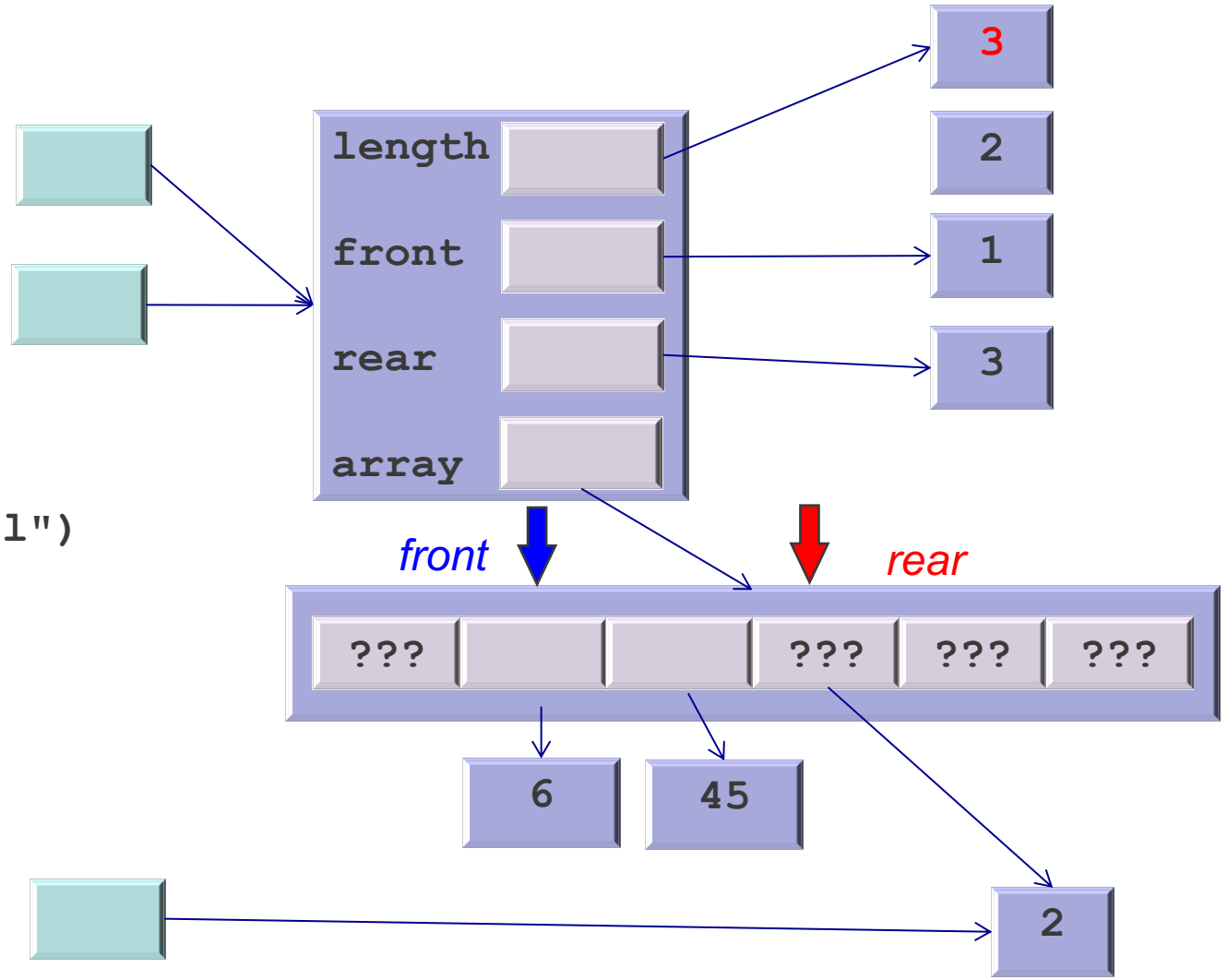
```
append(my_queue,2)
```

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

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```



Executing append

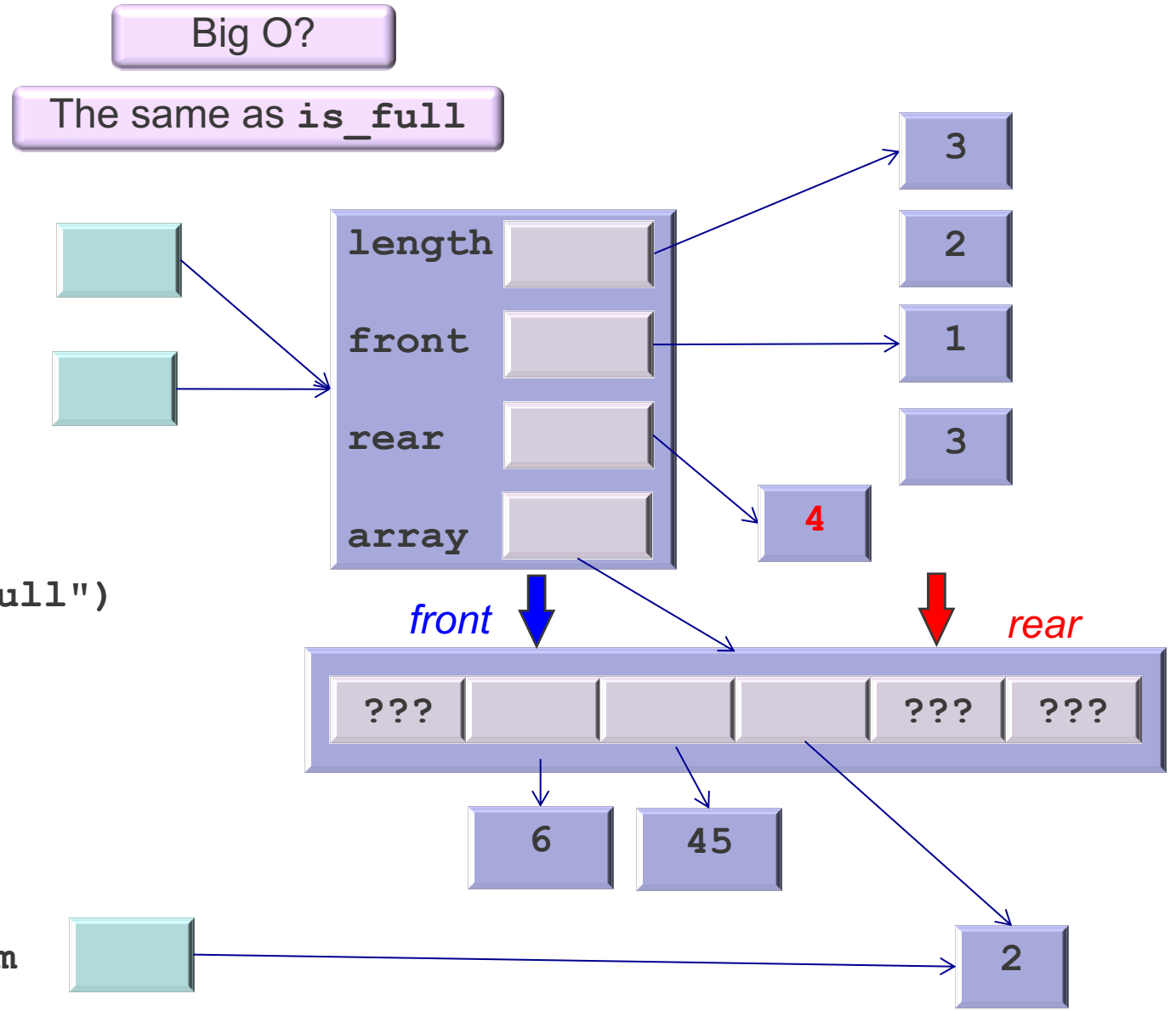
```
append(my_queue,2)
```




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

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```

Executing append

```
append(my_queue,2)
```




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

    self.array[self.rear] = item
    self.length += 1
    self.rear += 1
```

Implementing serve

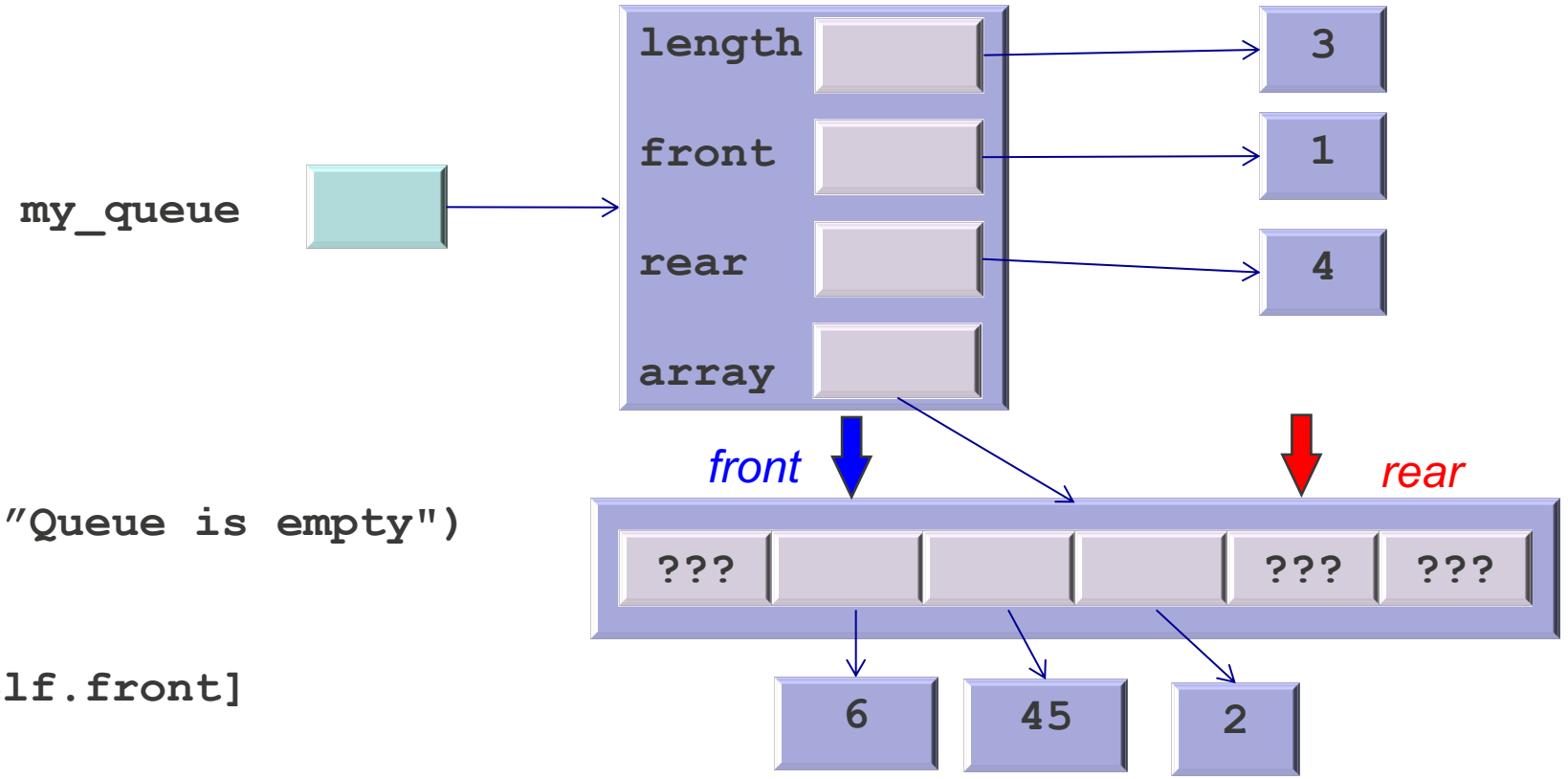
- **What do we do if the queue is empty?**
  - Let's raise an exception again

Another design decision!

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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

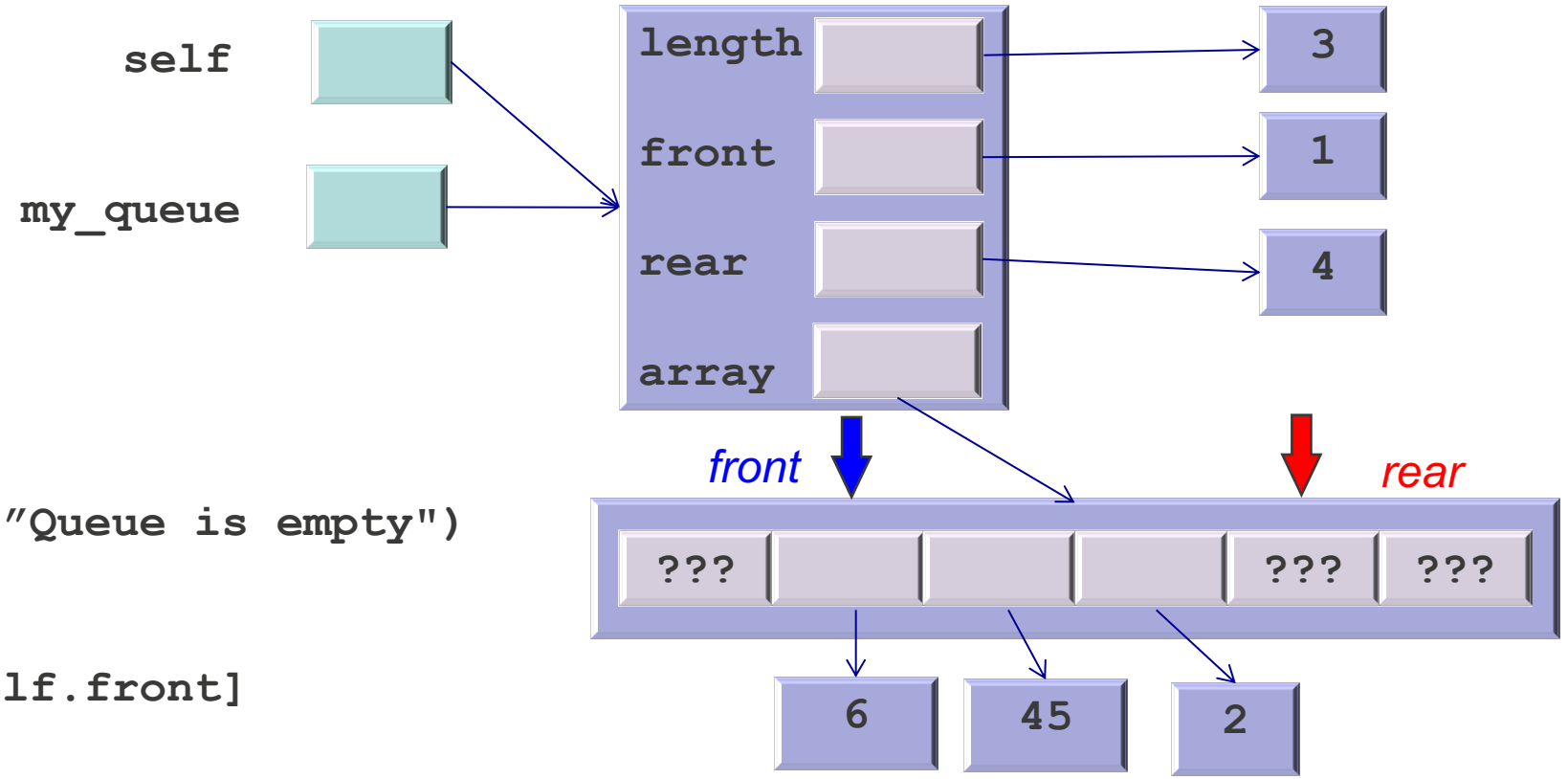
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

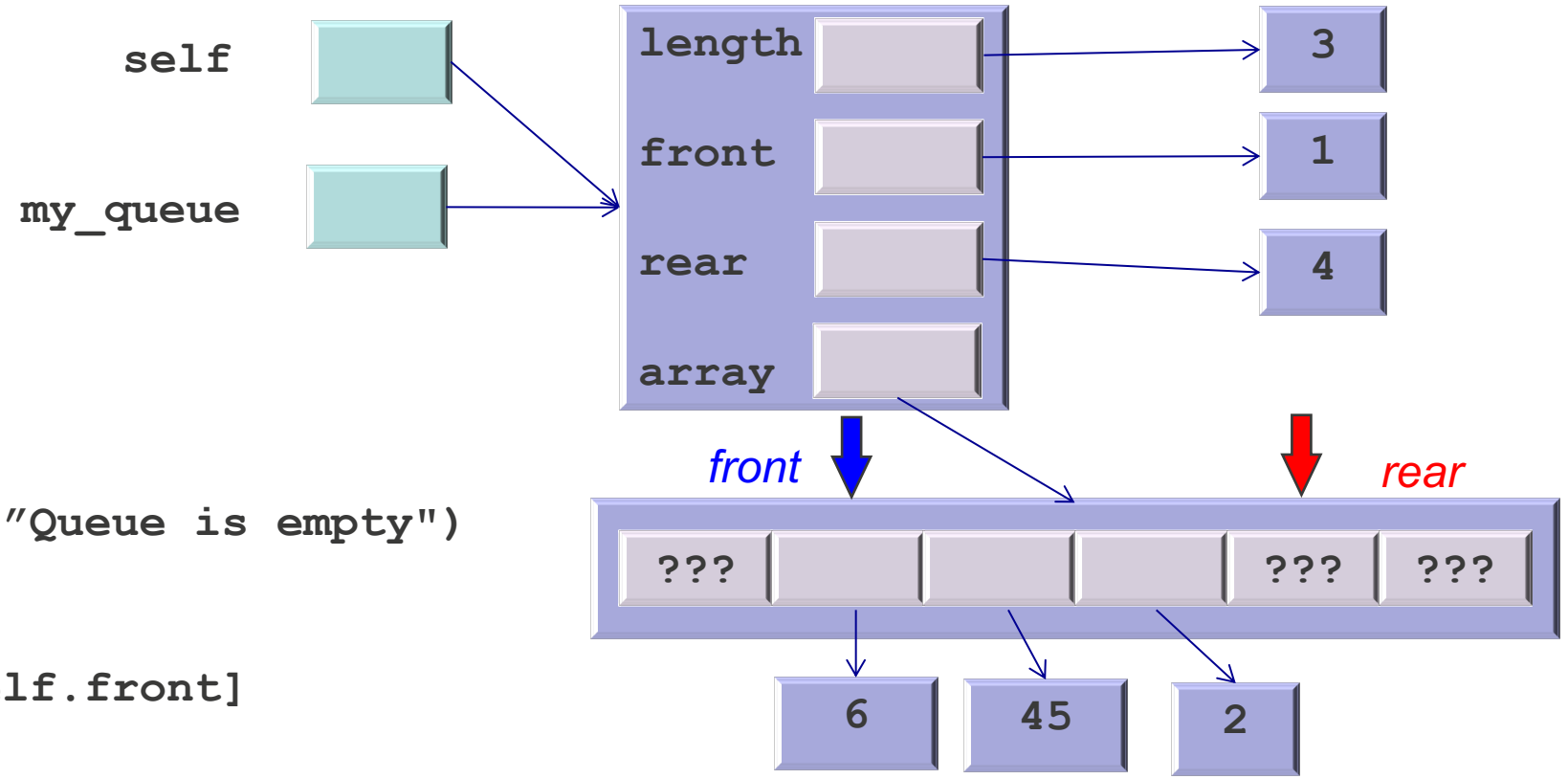
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

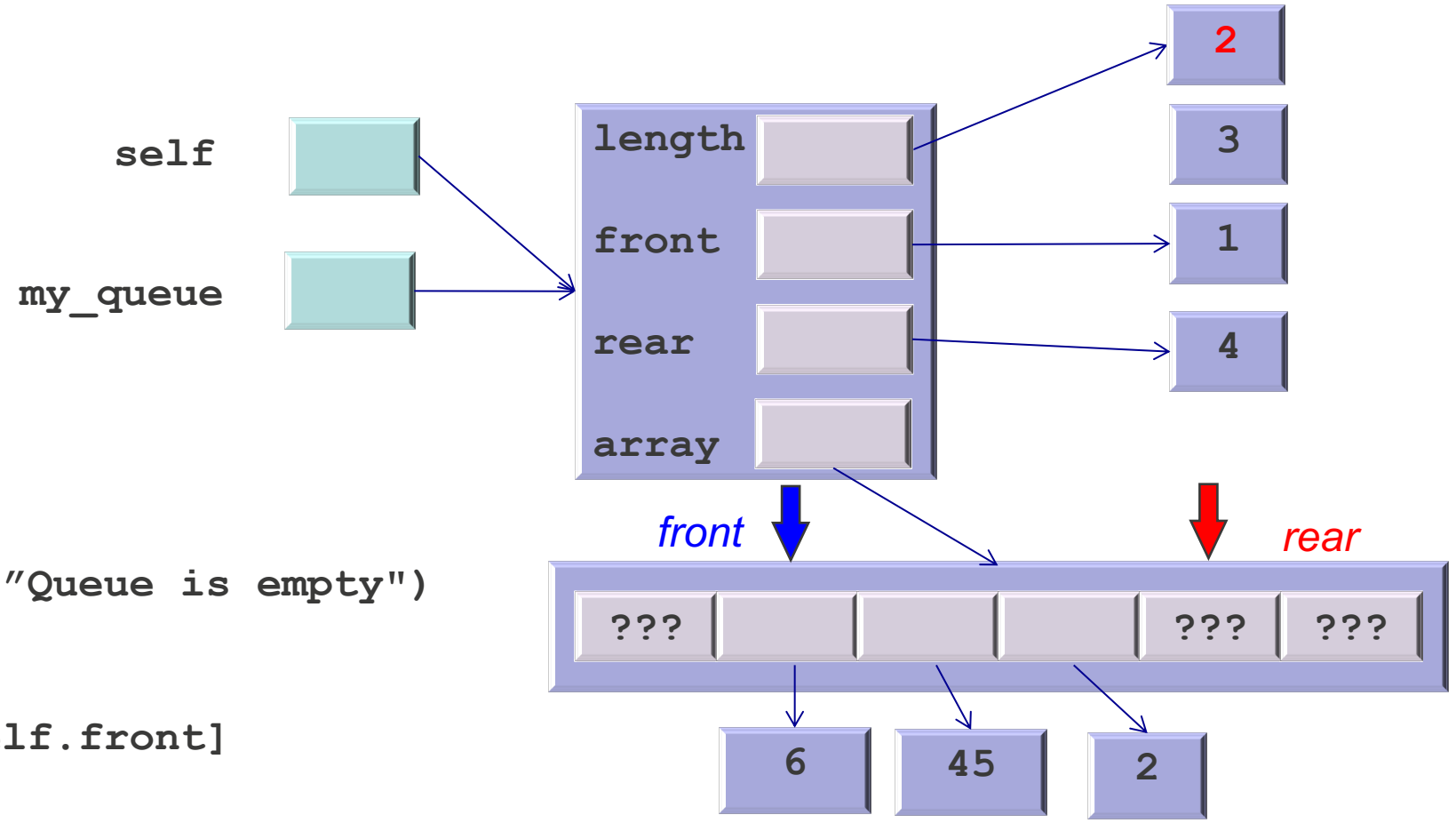
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

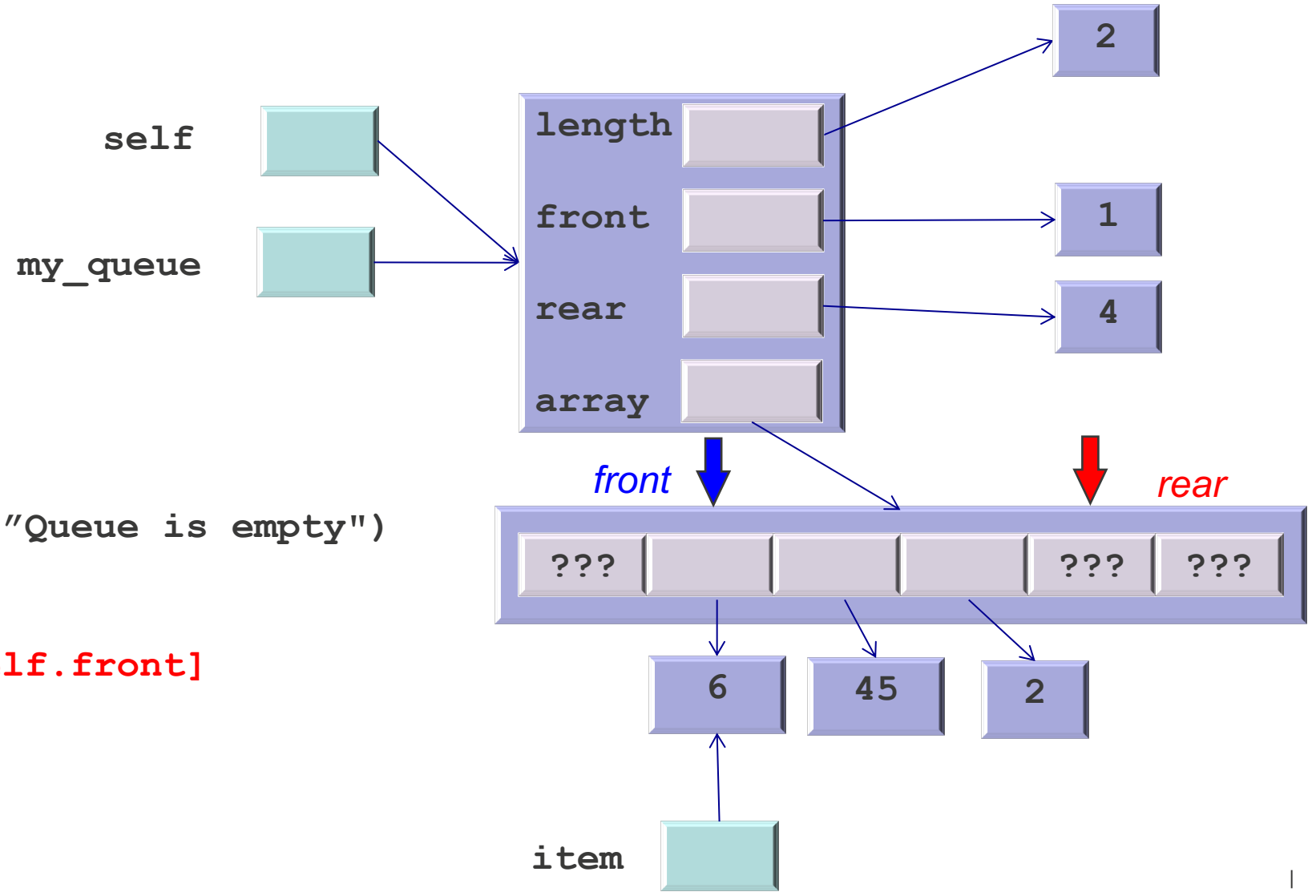
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

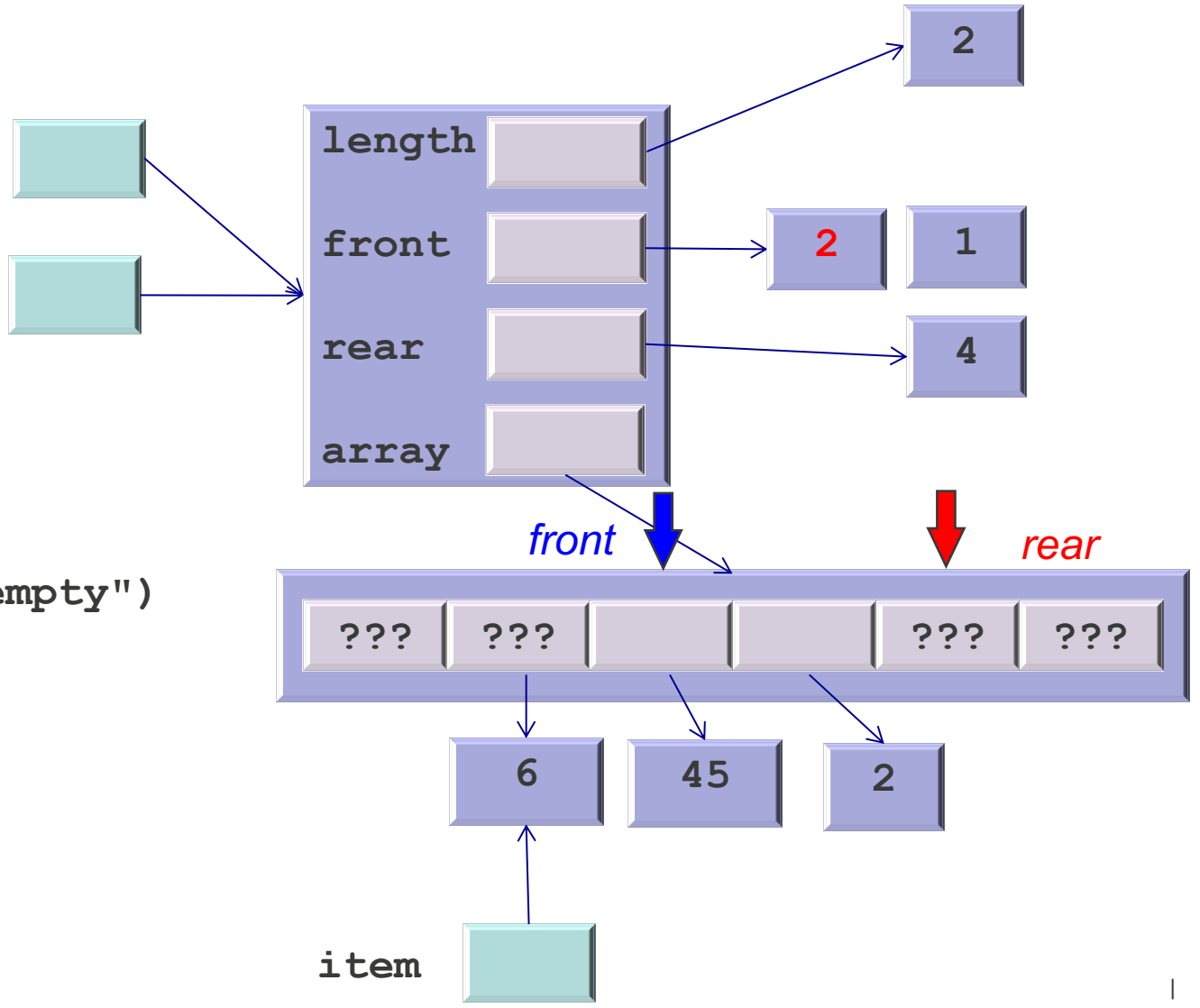
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

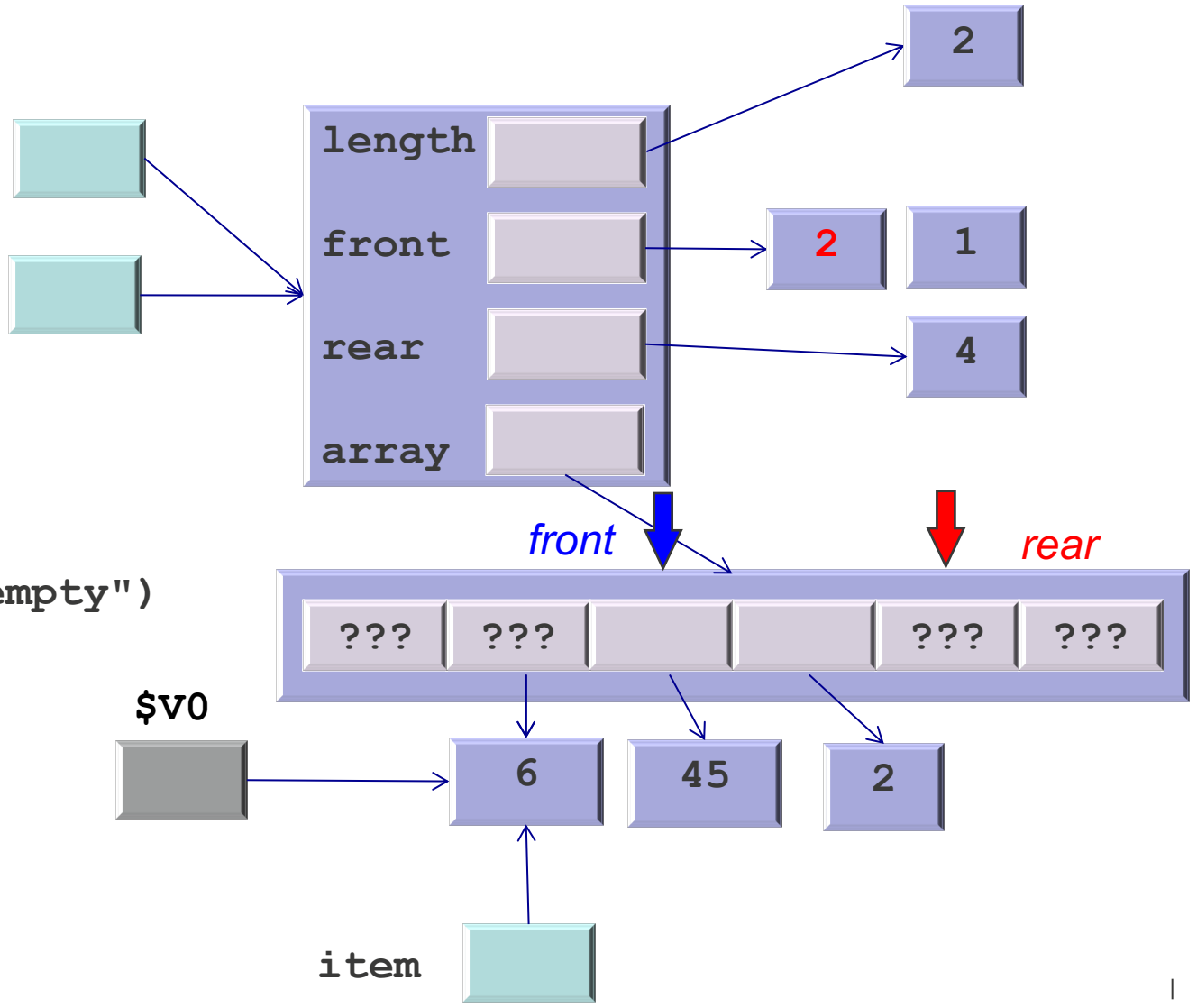
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Executing serve

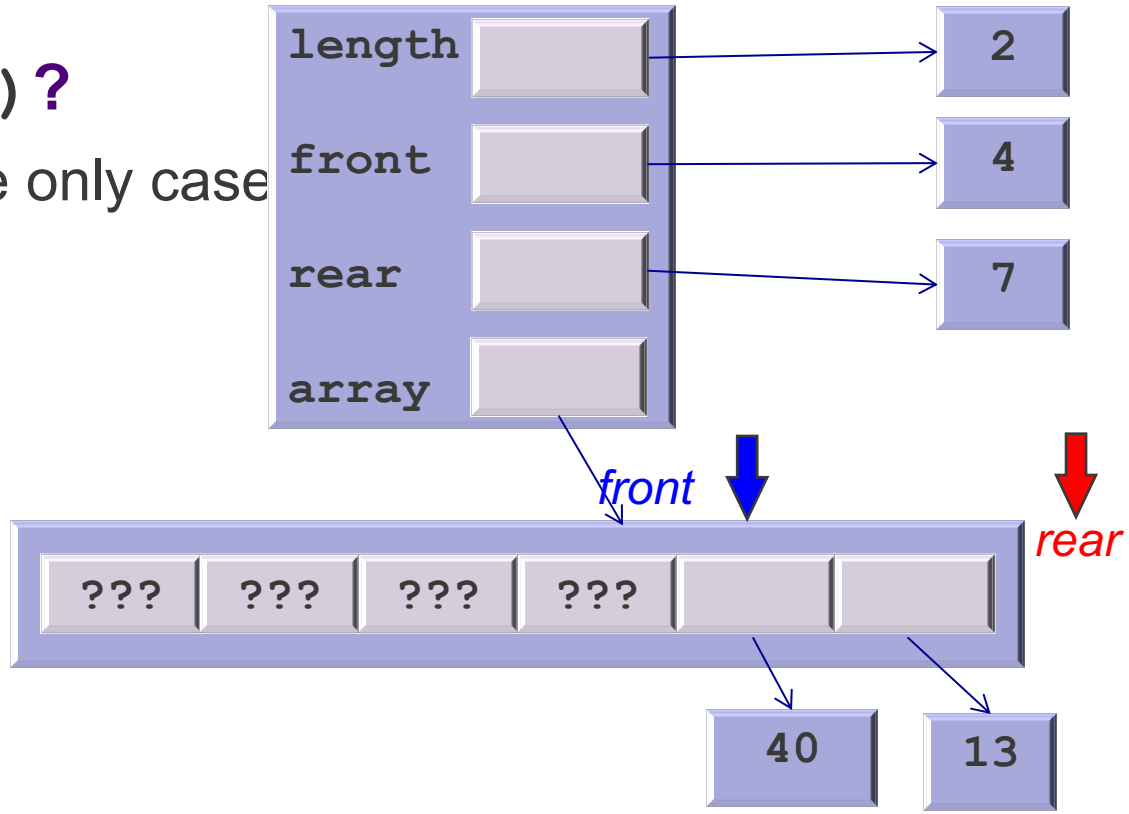
serve(my_queue)




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

    self.length -= 1
    item = self.array[self.front]
    self.front += 1
    return item
```

Implementing is_full

- **How do we know the queue is full?**
- **When `len(self) == len(self.array)`?**
  - In that case is indeed full, but that is not the only case
- **What other cases can you think of?**
  - Whenever rear is pointing out of the array
  - There is not more space left!



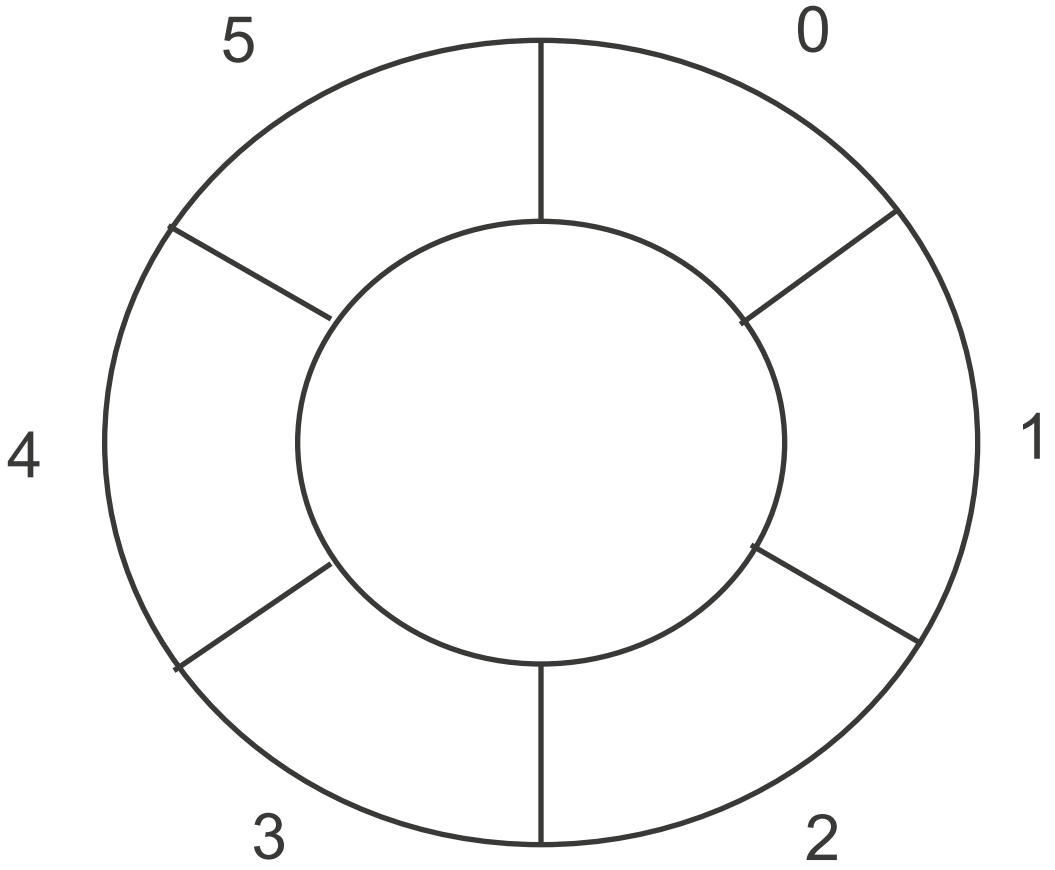

```
def is_full(self) -> T:
    return self.rear == len(self.array)
```

Waste of space!
There are many empty cells!

MONASH
University

Circular Queues

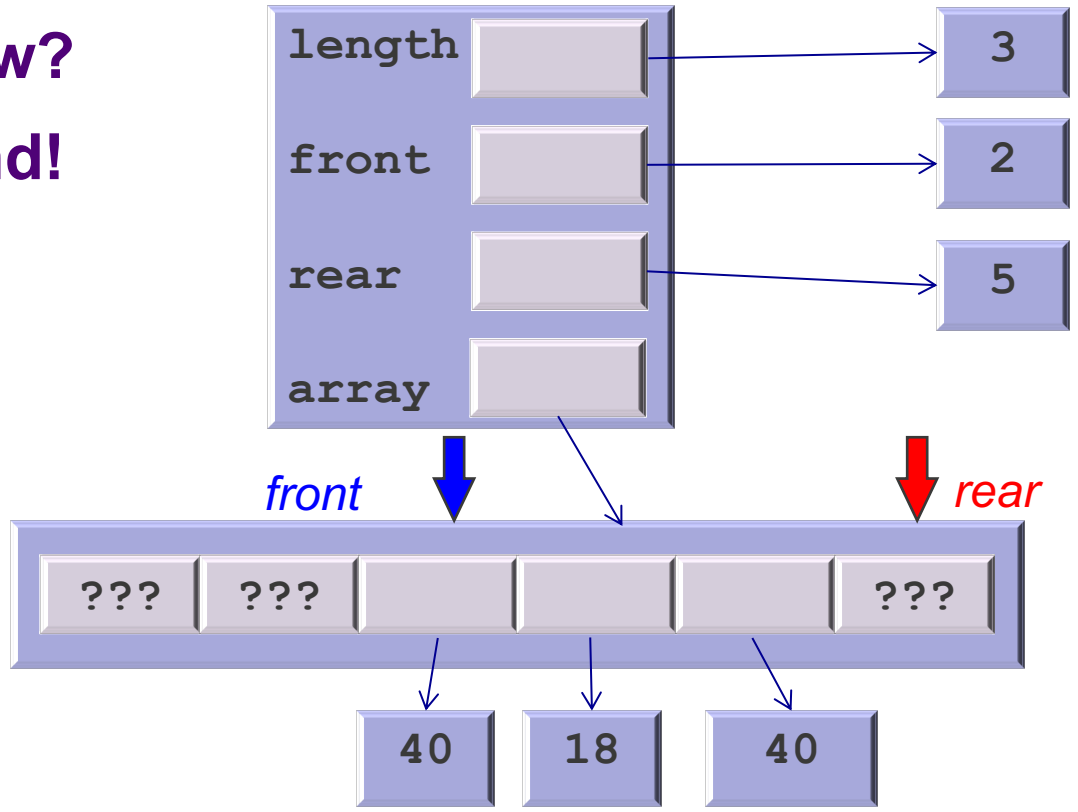
Solution: Circular Queues



Simulated by allowing rear and front to **wrap around** each other

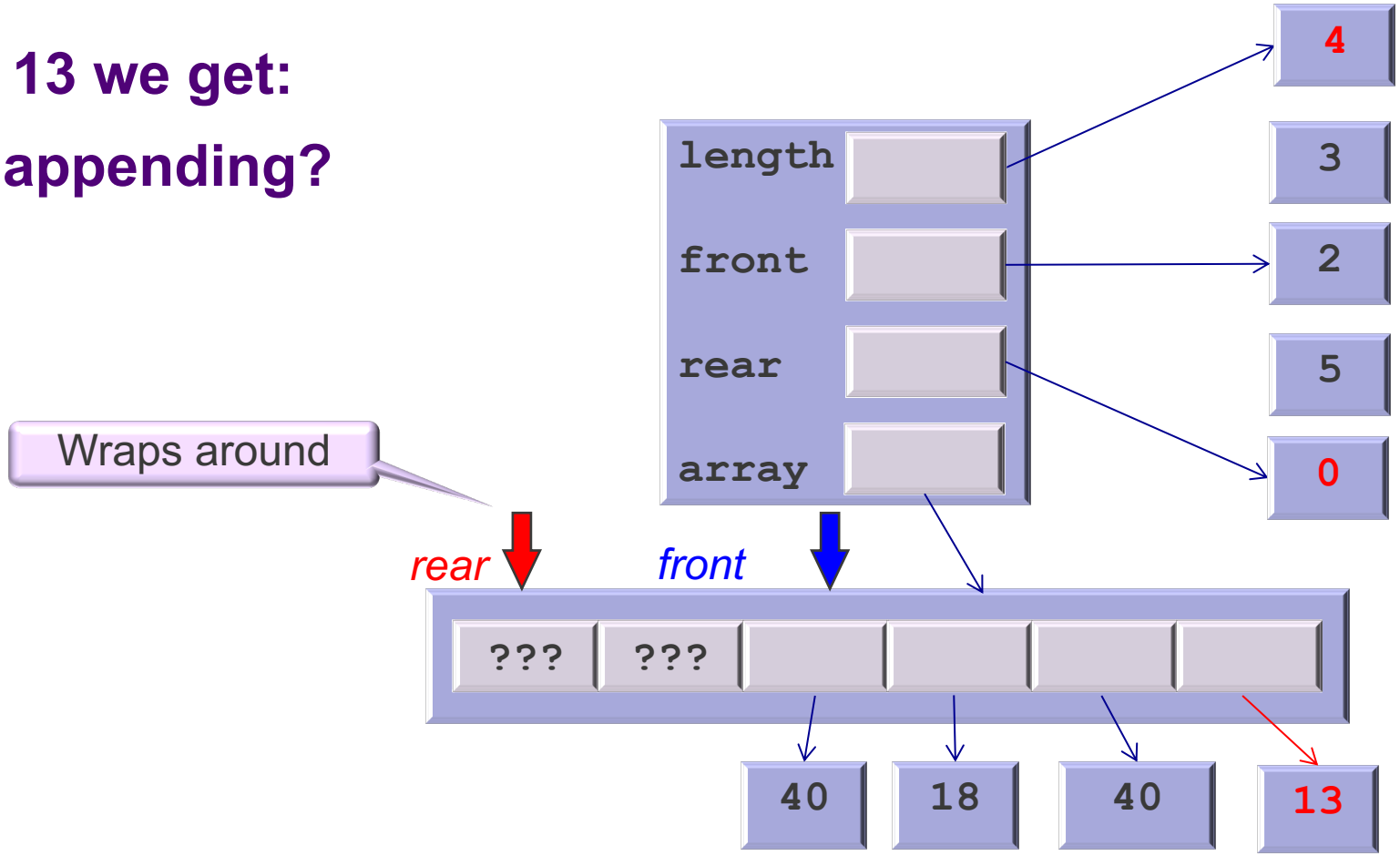
Appending in a Circular Queue

- Assume we have the queue in the figure:
- What happens if we append now?
- We need to wrap the rear around!



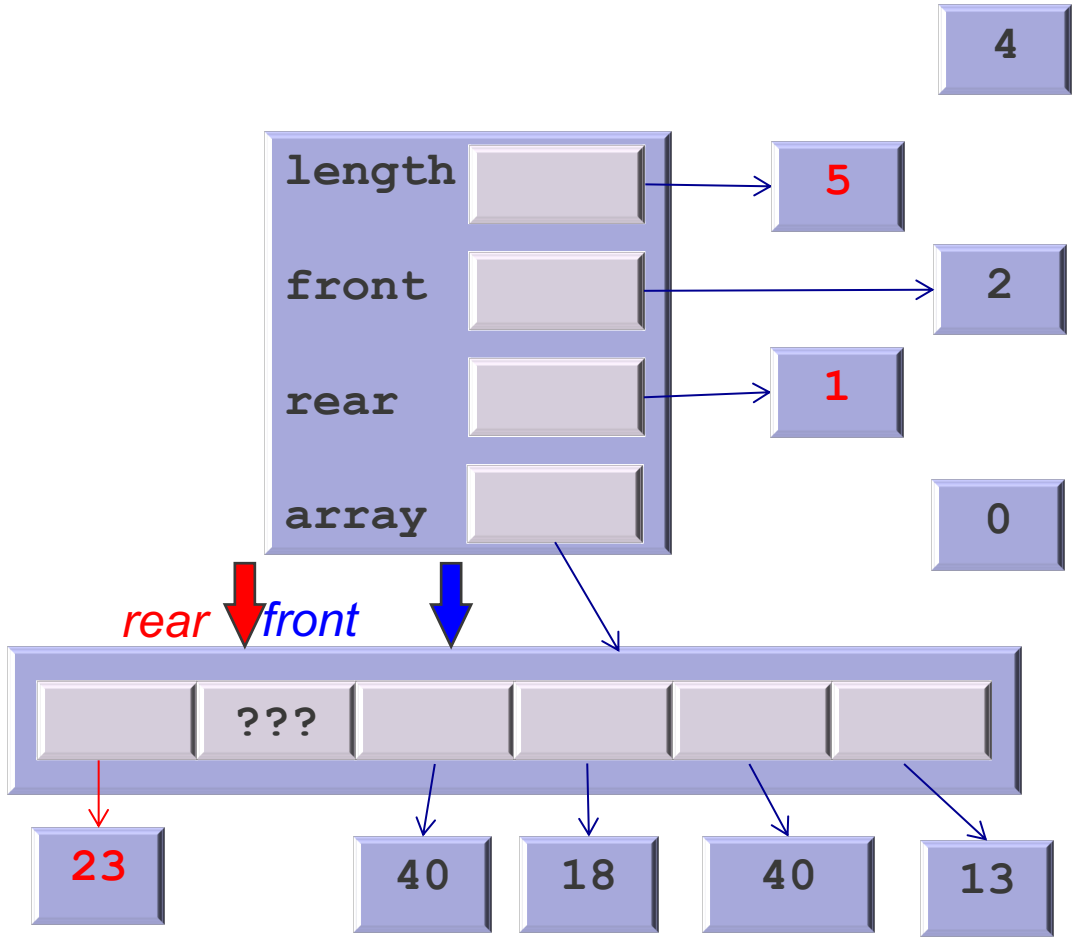
Appending in a Circular Queue

- **After appending 13 we get:**
- **What if we keep appending?**



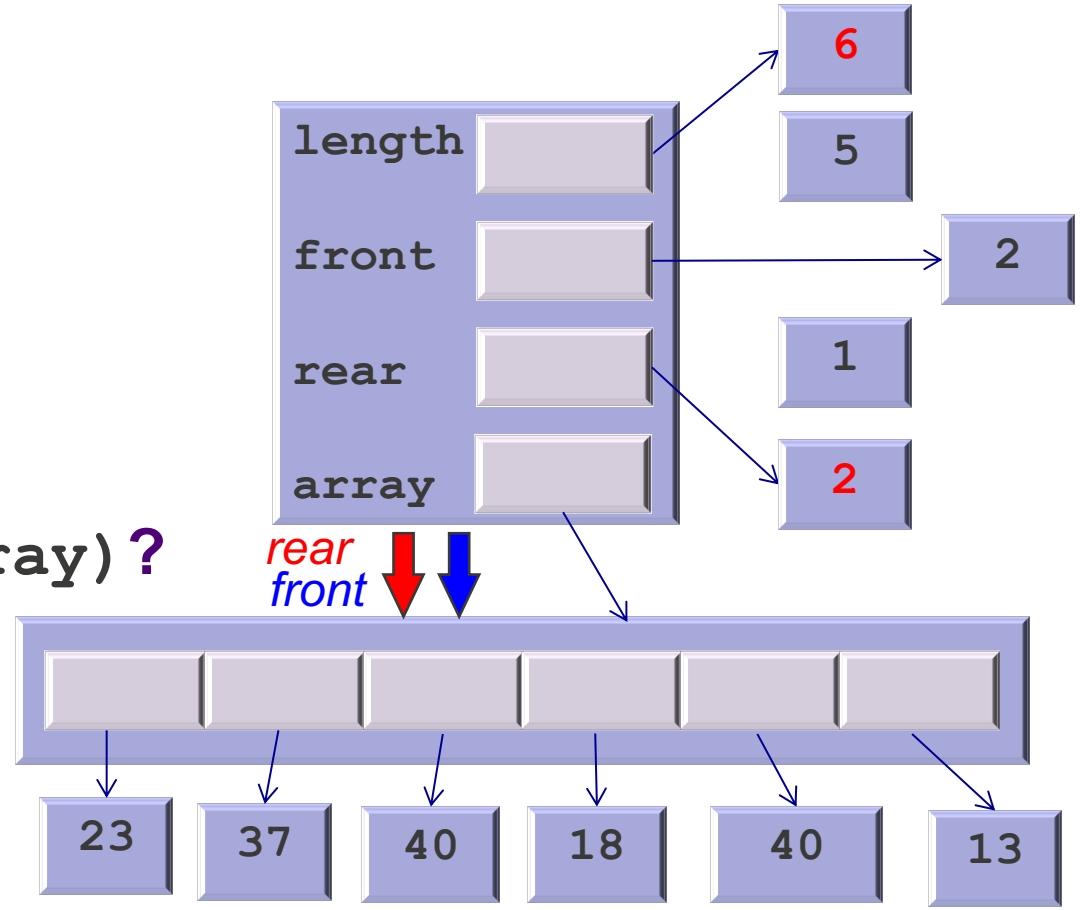
Appending in a Circular Queue

- **After appending 23 we get:**



Appending in a Circular Queue

- **And after appending 37 we get:**
- **The queue is now full**
- **How do we know?**
- `self.rear == self.front`?
- **No, that can also mean empty!**
- `len(self) == len(self.array)`?
- **Yes! That is now correct!**
- **Implementation differs in:**
  - append, serve and is_full



MONASH
University

Circular Queue ADT Implementation

Implementation for a circular Queue

```
from referential_array import ArrayR
from abstract_queue import Queue, T

class CircularQueue(Queue[T]):
    MIN_CAPACITY = 1

    def __init__(self,max_capacity:int) -> None:
        Queue.__init__(self)
        self.front = 0
        self.rear = 0
        self.array = ArrayR(max(self.MIN_CAPACITY,max_capacity))

    def clear(self) -> None:
        Queue.__init__(self)
        self.front = 0
        self.rear = 0

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

Differences in red

Big O?

Same as **LinearQueue** methods

Implementation for a circular Queue (cont)

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

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

Differences in red

Big O?

All return statements, assignments and integer comparisons are always constant

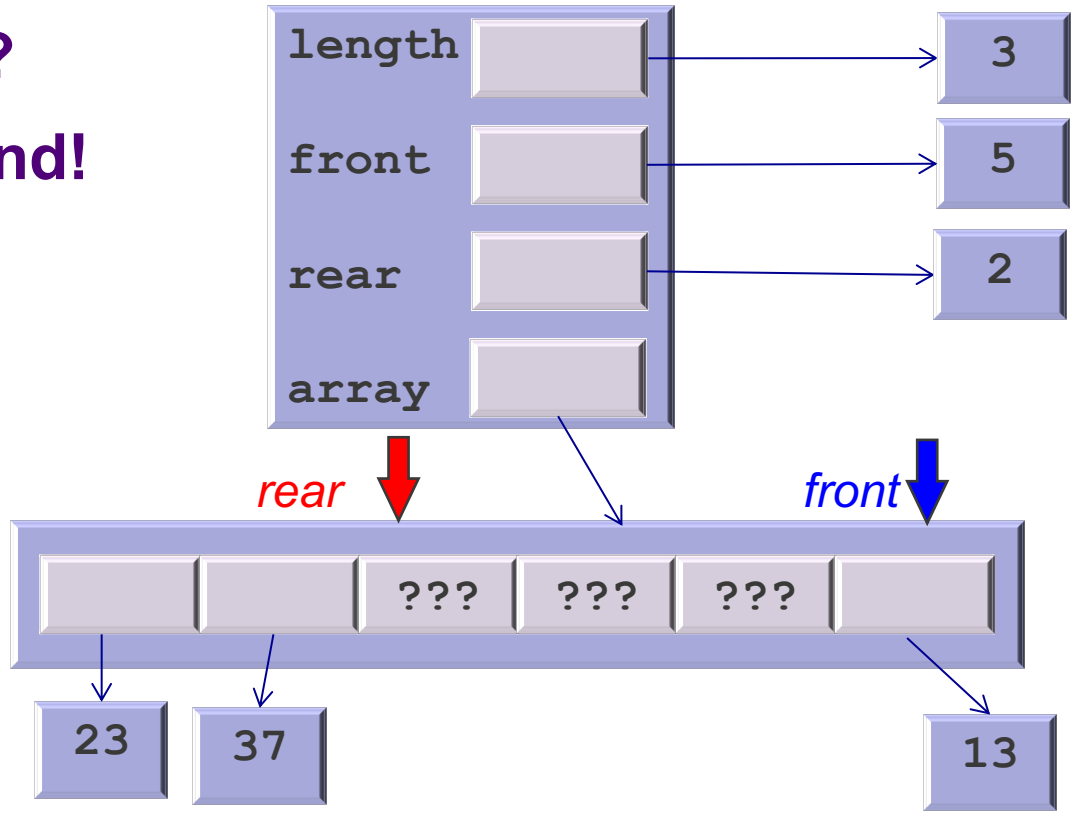
O(1)

Wraps over the end of the array

- **What about serve? What happens if front reaches the end of the array?**

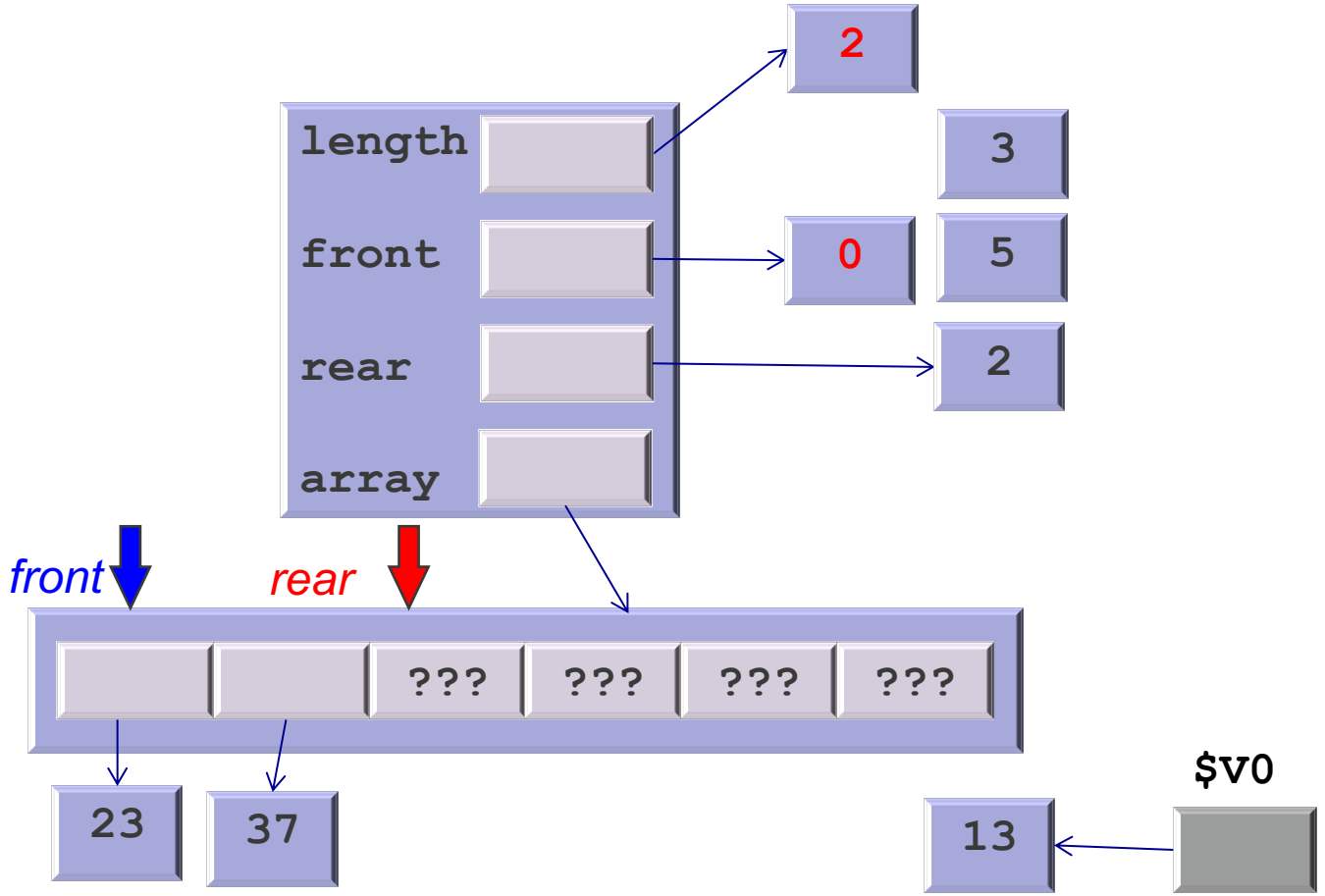
Serving in a Circular Queue

- Assume we have the queue in the figure:
- What happens if we serve now?
- We need to wrap the front around!



Serving in a Circular Queue

- **After serving, we get:**



Implementation for a circular Queue (cont)

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

    self.length -= 1
    item = self.array[self.front]
    self.front = (self.front+1) % len(self.array)
    return item
```

Differences in red

Big O?

All return statements, assignments and integer comparisons are always constant

O(1)

Wraps over the end of the array

- **Invariant: valid data appears from**
  - If front <= rear: from front to rear-1, in that order
  - Else: from front to the end of the array and from 0 to rear-1, in that order

MONASH University

Extending and using our Queue

Extending the class to print elements from front to rear

- **Let's implement it as a method within the `CircularQueue` ADT**
  - This means that the definition has access to the implementation
- **Do not modify the queue, just print its elements**

```
def print_items(self) -> None:
    index = self.front
    for _ in range(len(self)):
        print(self.array[index])
        index = (index + 1) % len(self.array)
```

Could you say:
`for item in self.array`?

Think about the invariant…

The first item would always be at position 0, rather than the first item in the queue. So, no!

Complexity of `print_items`

- **Single loop that is always executed `len(self)` times**
  - Best = worst
- **Inside the loop, the number of operations is fixed except for …**
  - **`print`**: its the number of operations for print depends on the size m of the item, e.g., the length of a string, the number of integers in a matrix, etc.
- **len(self)*(K*m) ≈ len(self)*m**
- **Which gives best = worst = O(len(self)*m)**

In general, particularly when you do not know the implementation of the function, you could just say that assuming the complexity of print is $Comp_{print}$ then $O(len(self)*Comp_{print})$

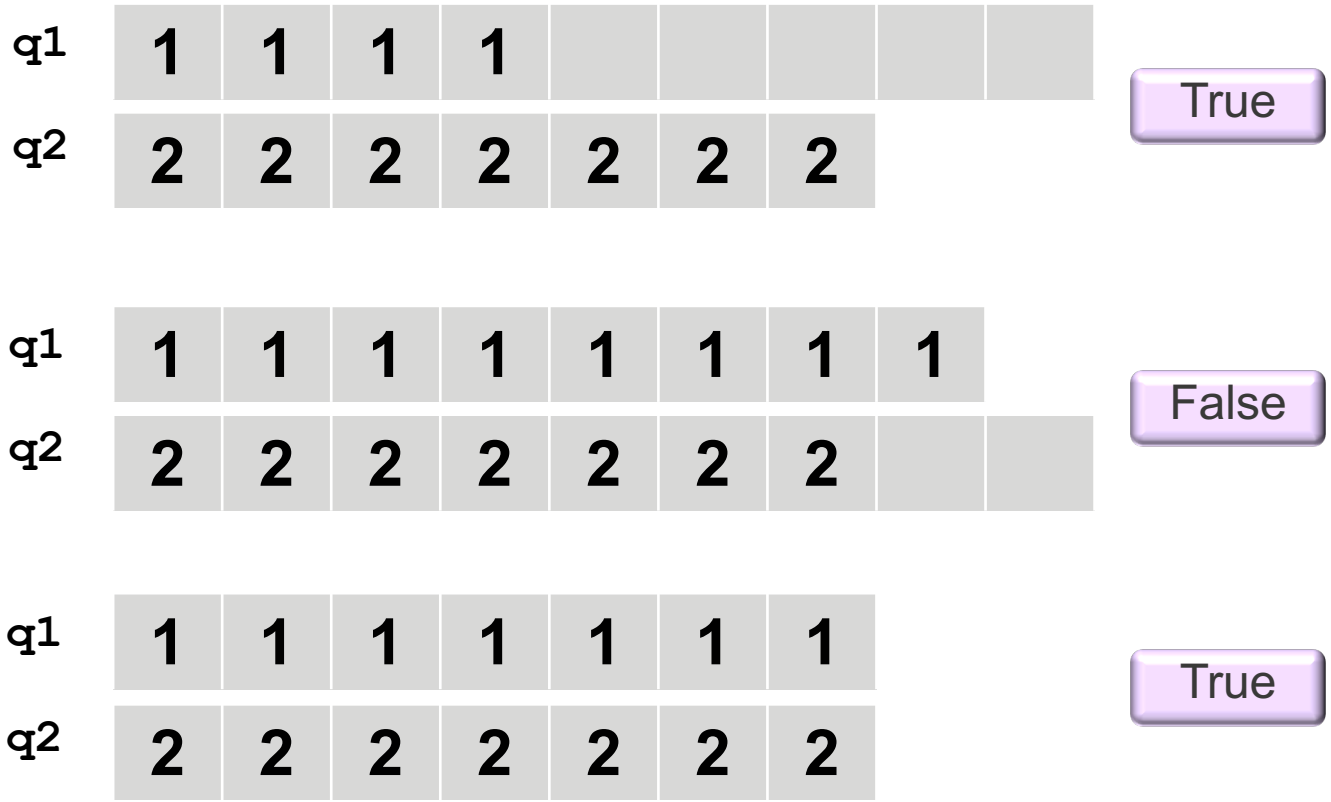
```
def print_items(self) -> None:
    index = self.front
    for _ in range(len(self)):
        print(self.array[index])
        index = (index + 1) % len(self.array)
```

Using the Queue

- Define a function:

```
def greater(q1: Queue,q2: Queue) -> bool:
```

- That returns true if and only if
  - q2 is at least as long as q1
  - AND every element in q1 is less or equal than the element in the same position in q2
- You are allowed to modify the input queues



Using the Queue (cont)

- Make sure you use it as an ADT (no access to the implementation)
- Try using only `is_empty` and `serve`:

```
def greater(q1:Queue,q2: Queue) -> bool:
    while not q1.is_empty() and not q2.is_empty():
        if q1.serve() > q2.serve():
            return False
    return is_empty(q1)
```

- Try now using `len()`:

```
def greater(q1:Queue,q2: Queue) -> bool:
    if len(q1) <= len(q2):
        for _ in range(len(q1)):
            if q1.serve() > q2.serve():
                return False
        return True
    return False
```

Which one is better?

Better as in what? Faster? More scalable? Clearer?

For me the first one is clearer, the second one faster

For you?

Common Queue Applications

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

Summary

- **We now know what a Queue ADT is and:**
  - Its main operations
  - Their complexity
- **We are able to:**
  - Implement Queues using a base abstract class
- **We understand the implementation of both Linear and Circular queues**
  - And the reasons why the second is better
- **We can implement both types and can:**
  - Use them
  - Modify its operations and
  - Reason about their complexity