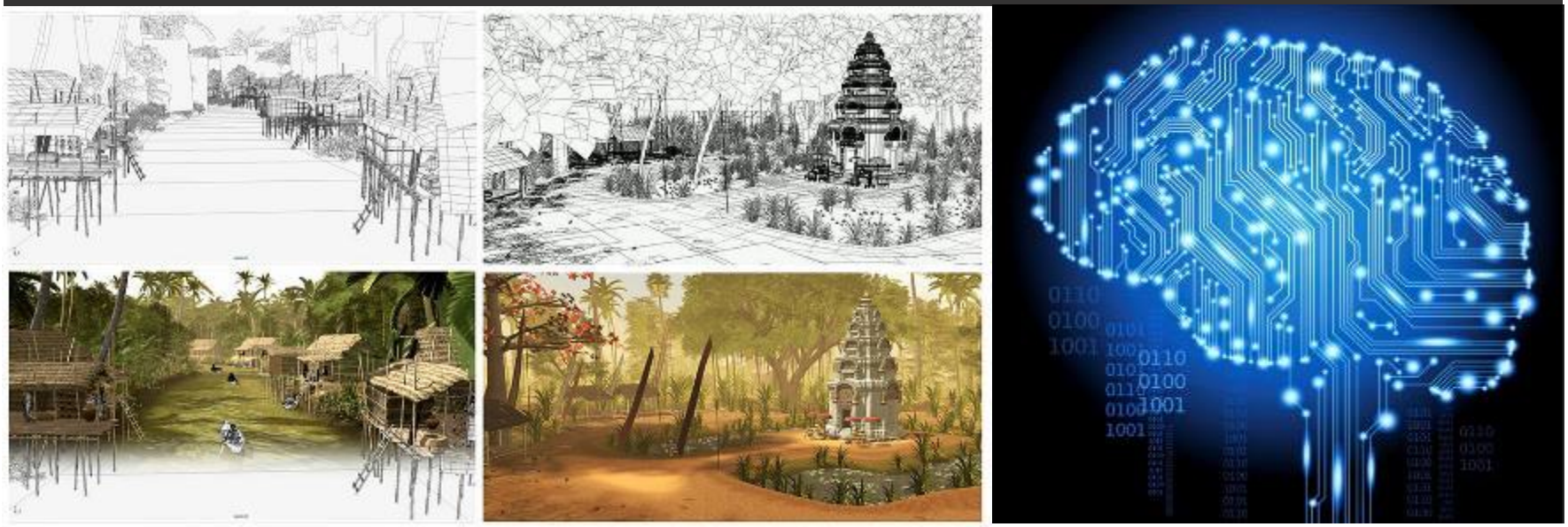
MONASH University

Information Technology

FIT1008&2085 Lecture 18

Prepared by: M. Garcia de la Banda

Queues with Arrays



Where were we at?

- **Last lecture we saw the Stack abstract data type:**
  - Main operations
  - Their complexity
- **To be able to**
  - Implement them with arrays, use them, modify their operations and reason about their complexity

Objectives for the this lecture

- To do the same for the Queue ADT
- To be able to decide when it is appropriate to use a Stack and a Queue

FIFO

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

The Queue Data Type

- **Follows the FIFO model**
- **Its operations are defined in its interface:**
  - Create the queue (Queue)
  - Add an item to the back (append)
  - Take an item off the front (serve)
  - Is the queue empty?
  - Is the queue full?
  - Empty the queue (reset)
- **Remember: you can only access the element at the front of the queue (first item inserted that is still in)**

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

Possible implementation: linear queue

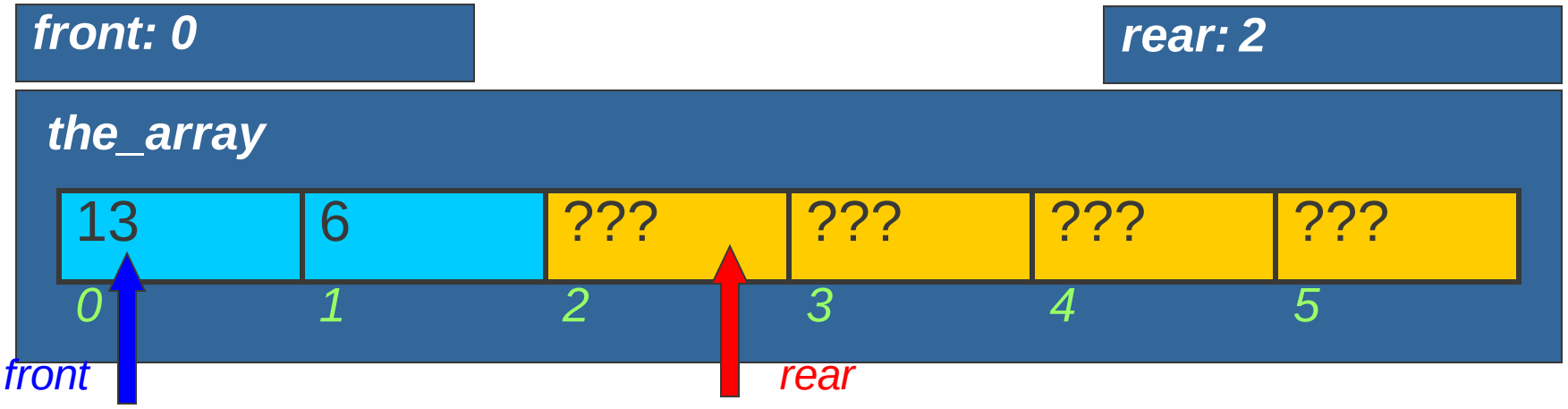
- **Create a new queue: initially no items**

Possible implementation: linear queue

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

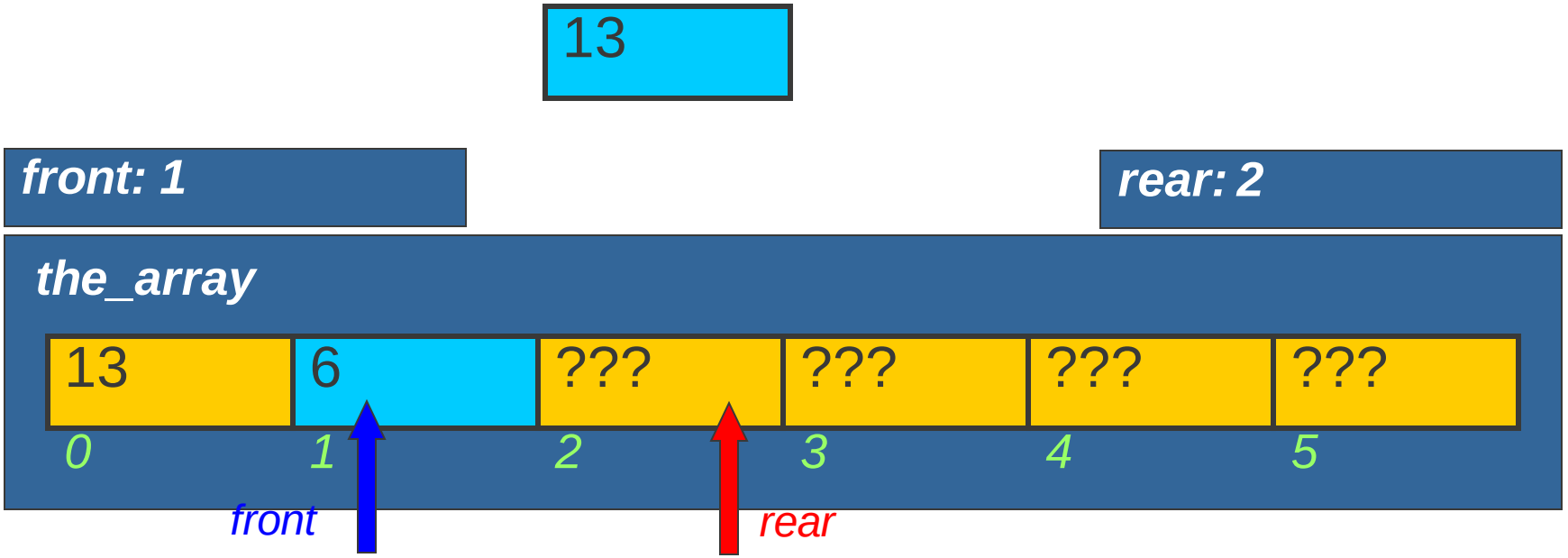
Possible implementation: linear queue

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



Possible implementation: linear queue

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



Implementation for a Linear Queue

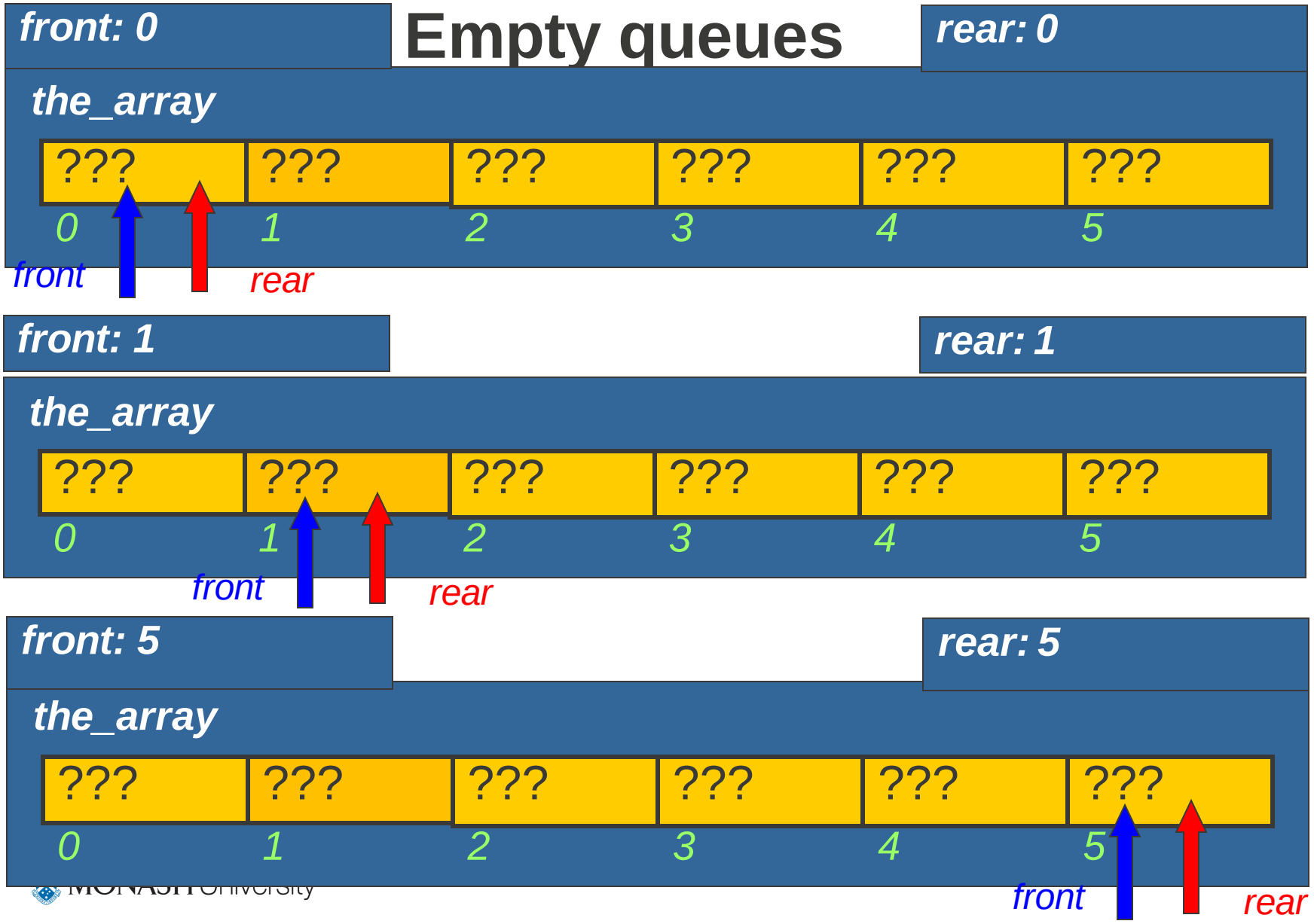
```
def Queue(max_size):
    the_array = [None] * max_size
    front = 0
    rear = 0
    return [front, rear, the_array]


def size(the_queue):
    [front,rear,_] = the_queue
    return front - rear


def is_empty(the_queue):
    [front,rear,_] = the_queue
    return rear == front
```

front and **rear** must be equal

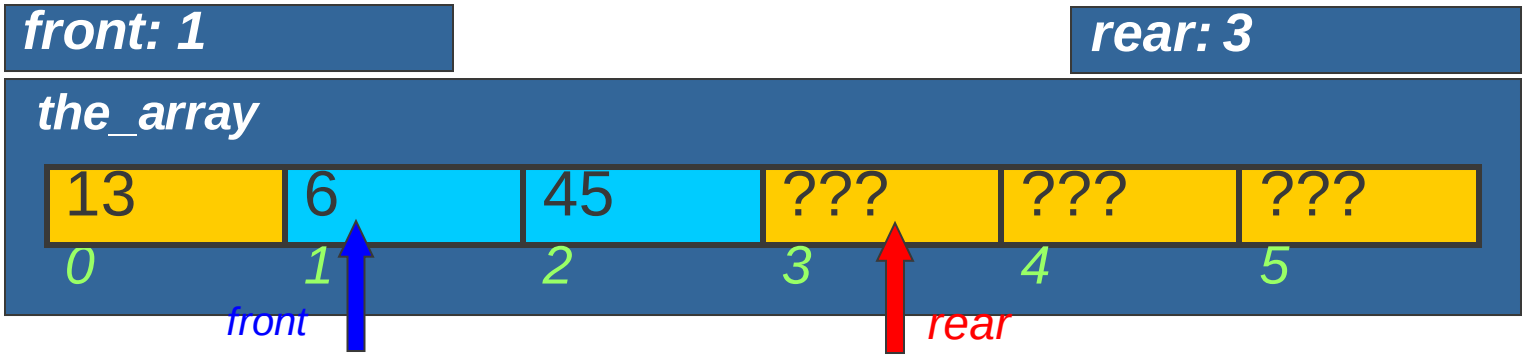
Why? Remember the invariant

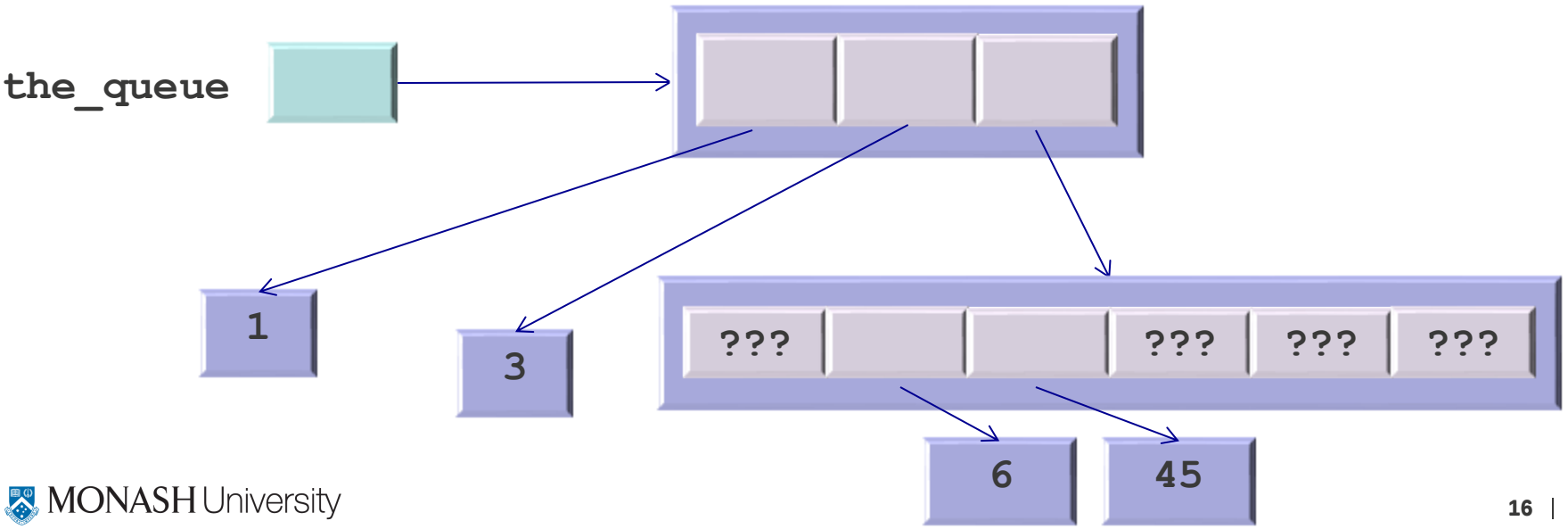
Big O?

Empty queues



How is `the_queue` represented in memory?





Implementation for a linear Queue

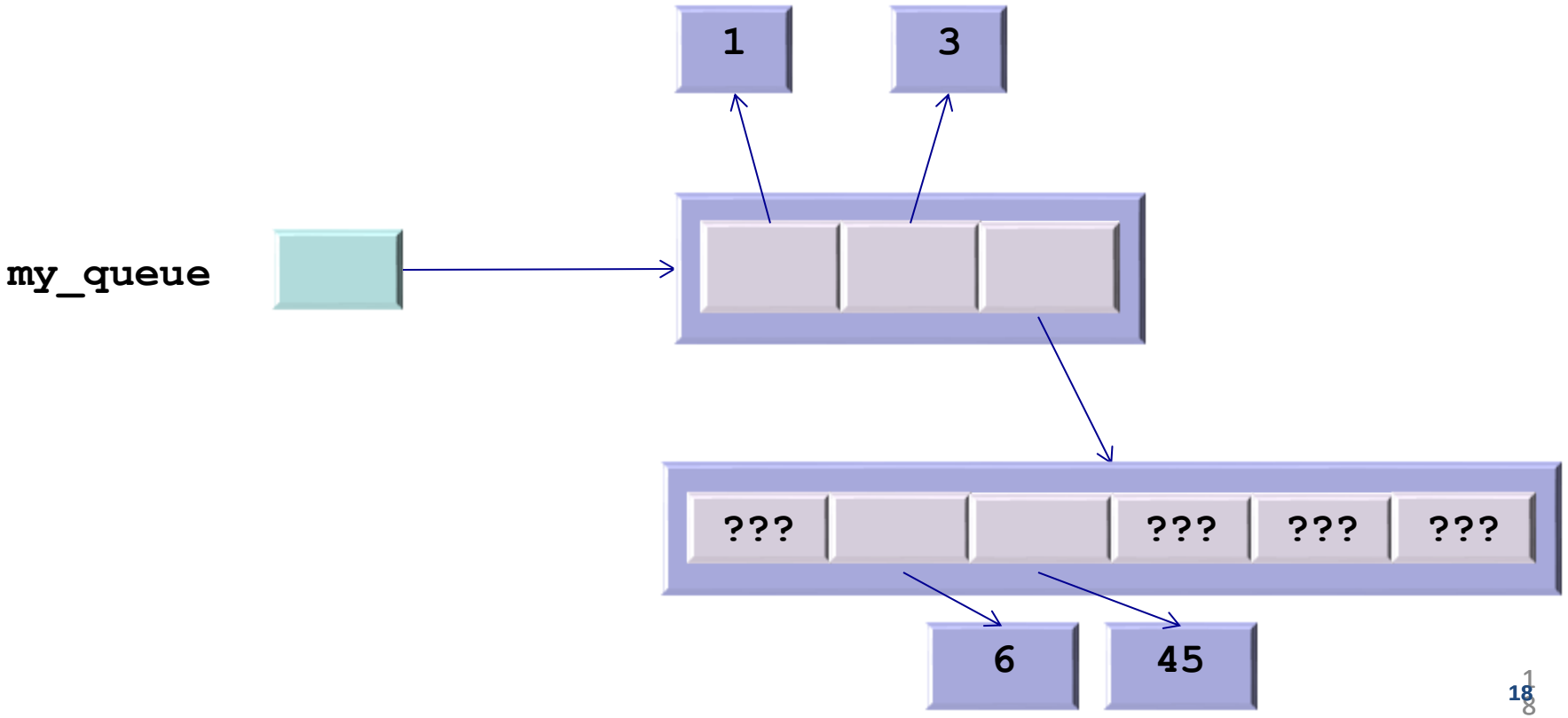
```
def is_full(the_queue):
    [_,rear,the_array] = the_queue
    return rear == len(the_array)


def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

Big O?

```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

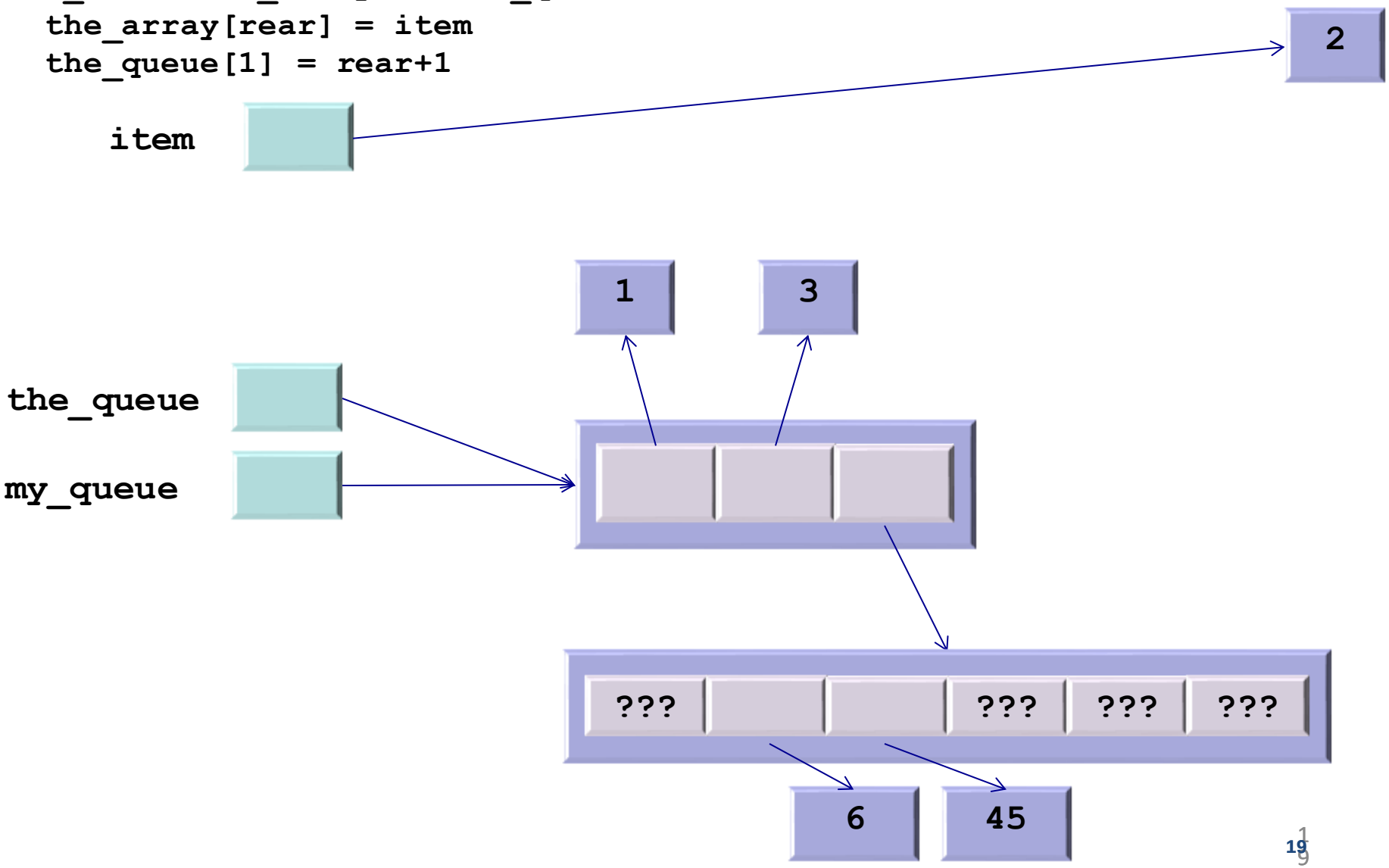
append(my_queue,2)

2



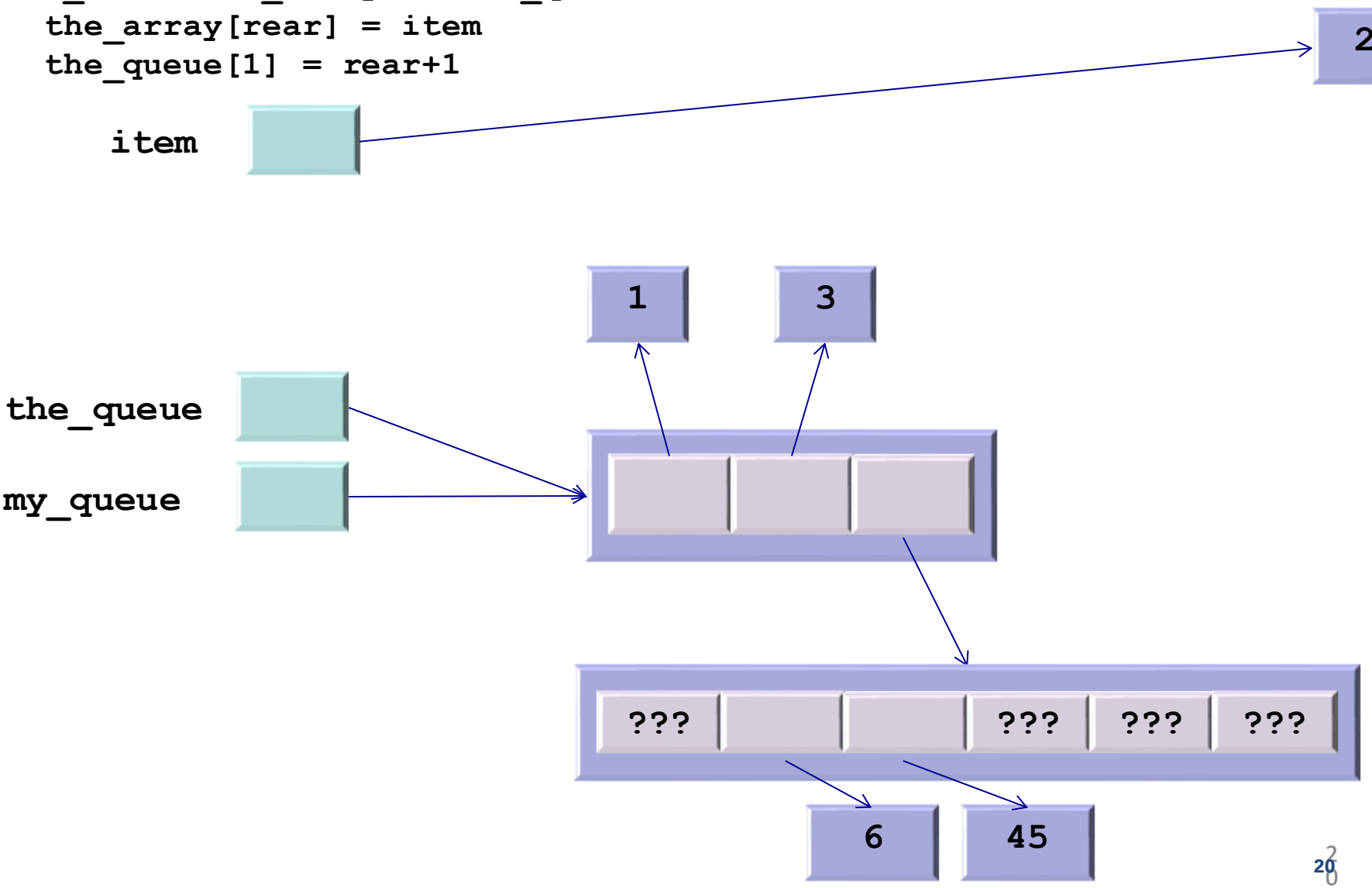
```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

append(my_queue,2)



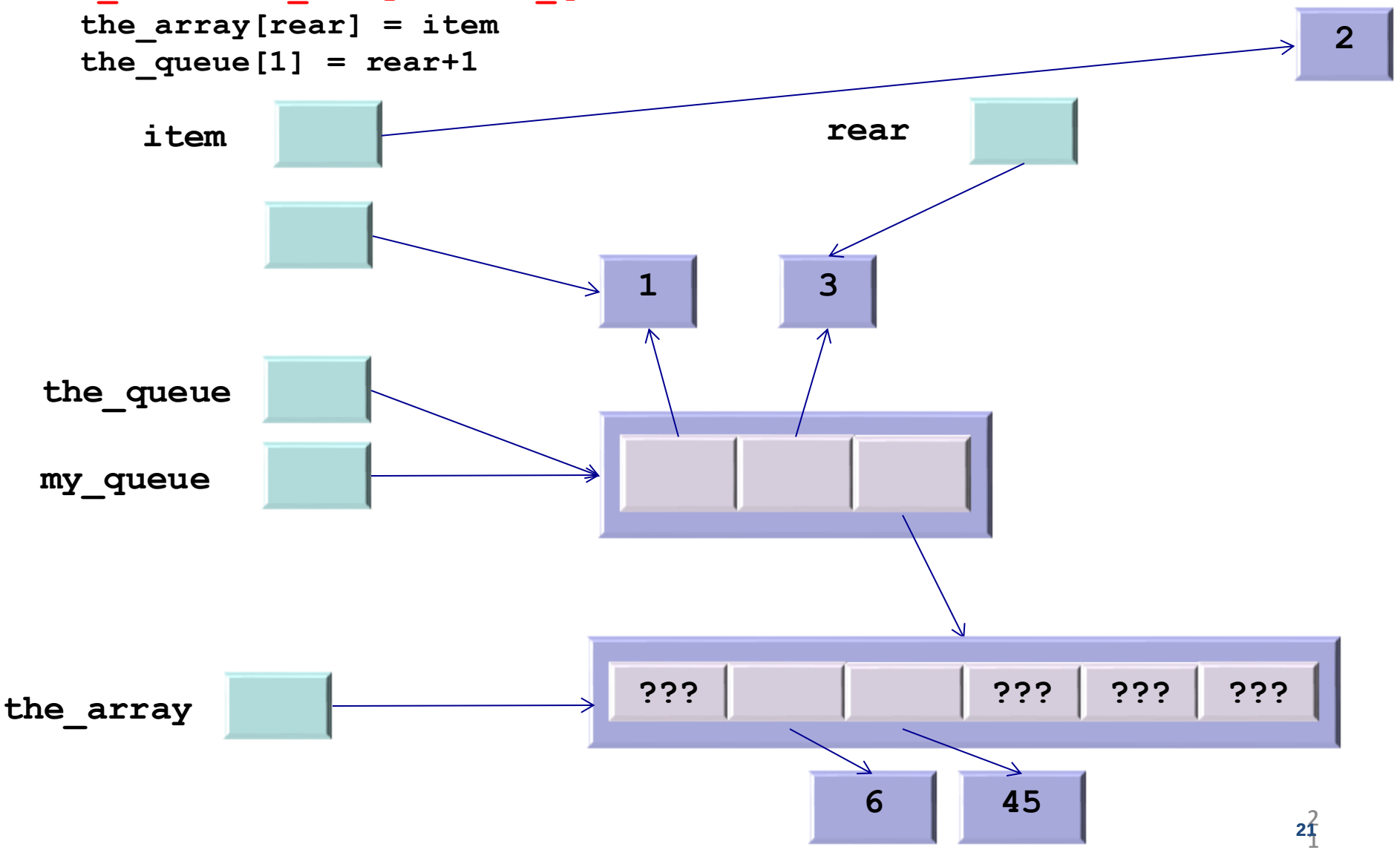
```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

append(my_queue,2)



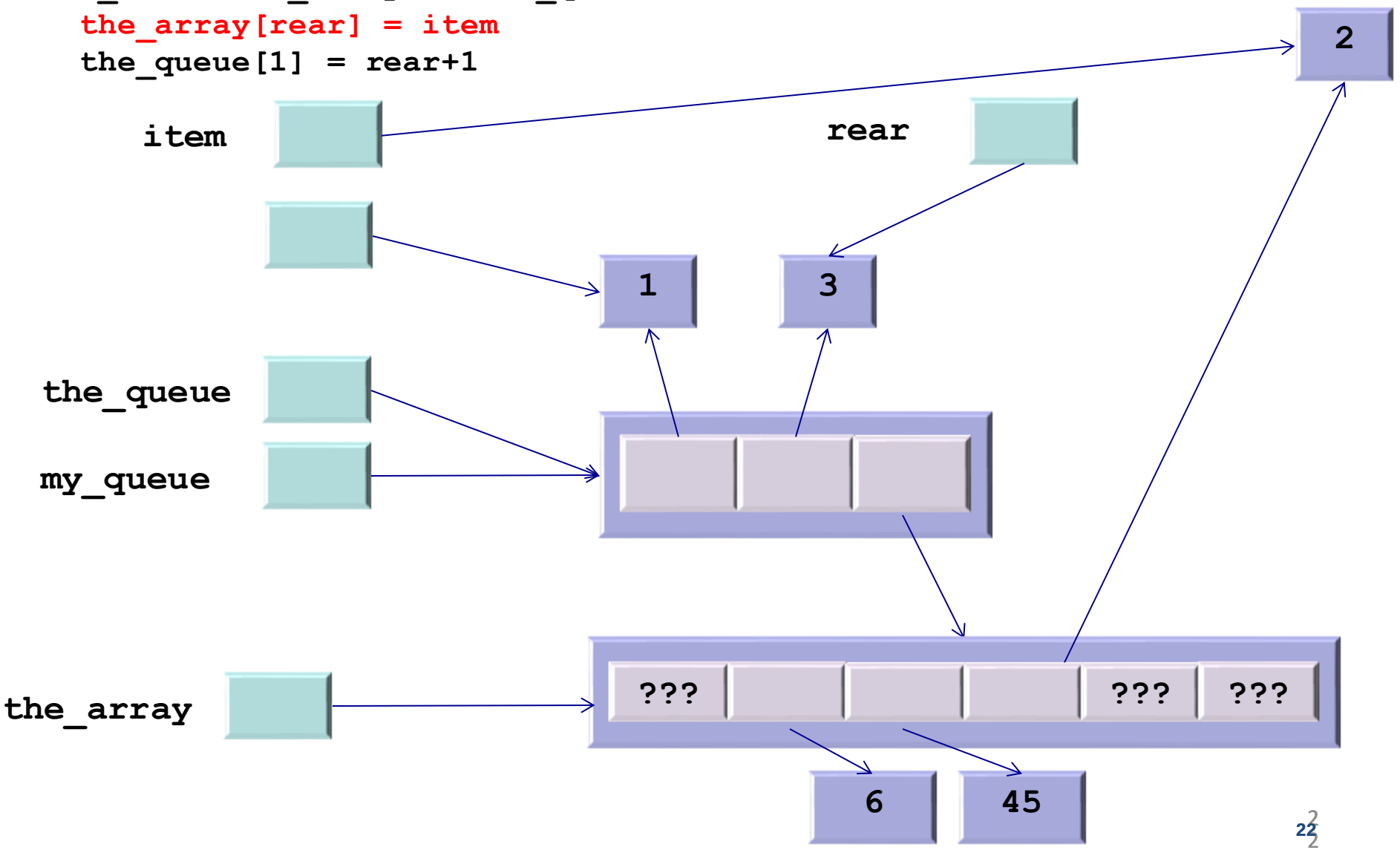
```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

append(my_queue,2)



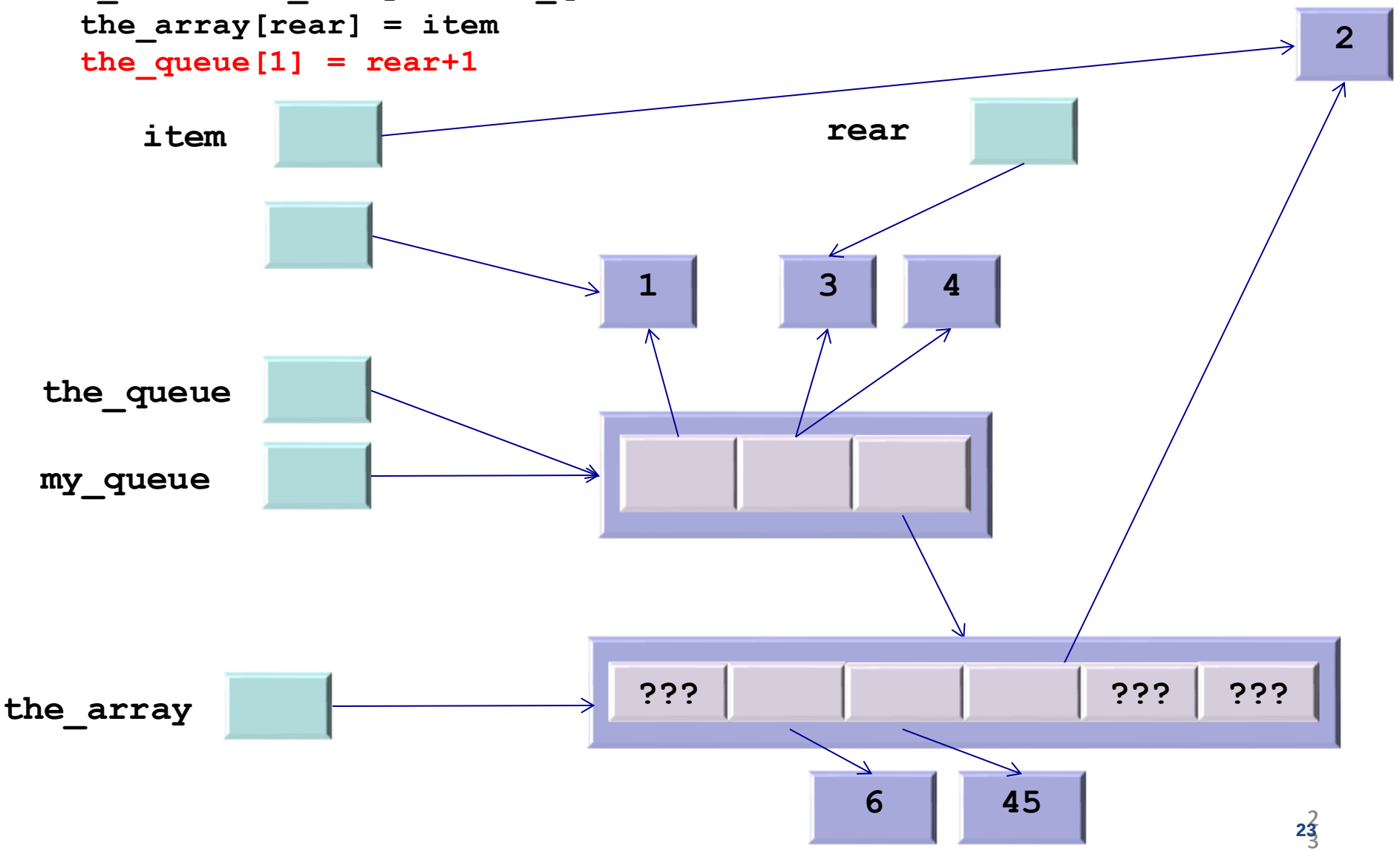
```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

append(my_queue,2)



```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array] = the_queue
    the_array[rear] = item
    the_queue[1] = rear+1
```

append(my_queue,2)



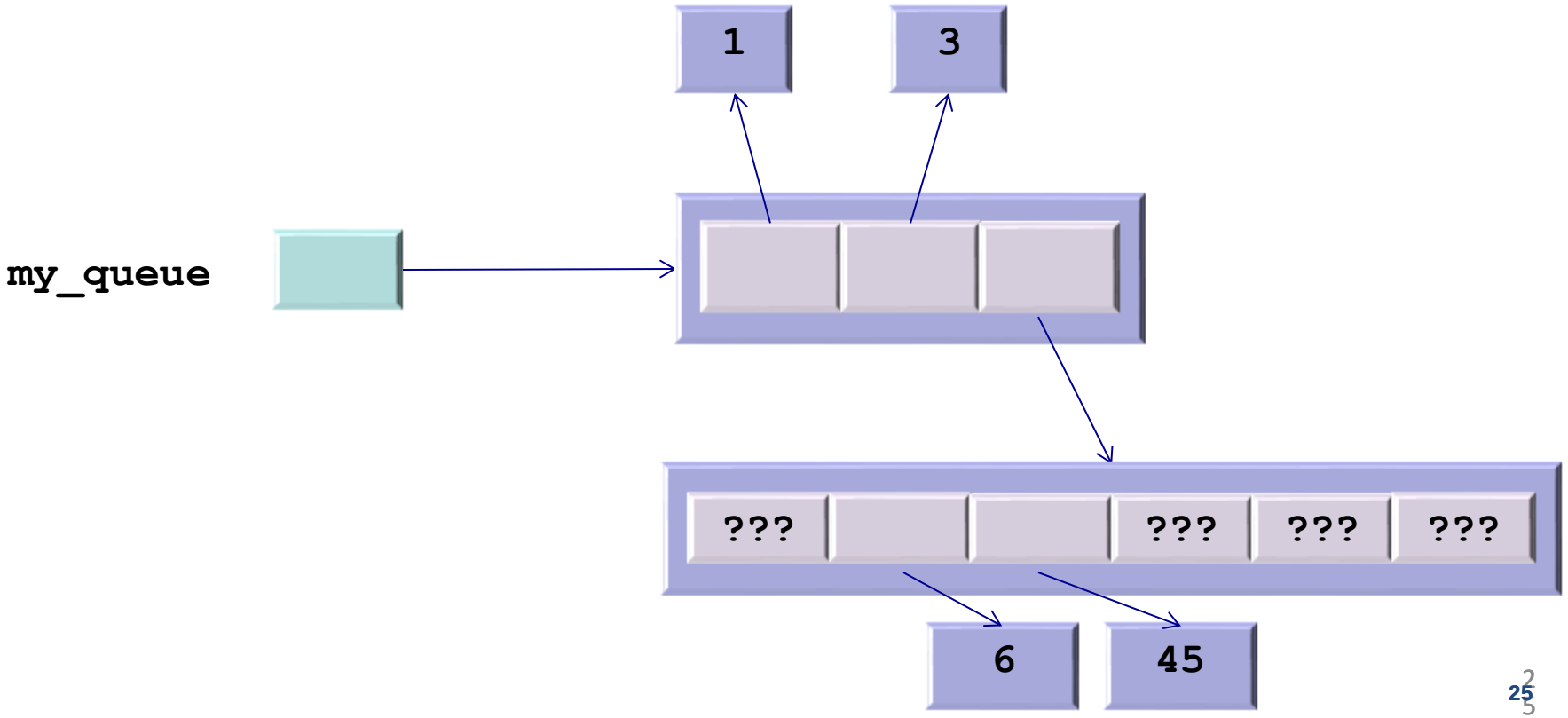
Implementation for a linear Queue

```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item

def reset(the_queue):
    the_queue[0] = 0
    the_queue[1] = 0
```

Again, **front** and **rear** must be equal

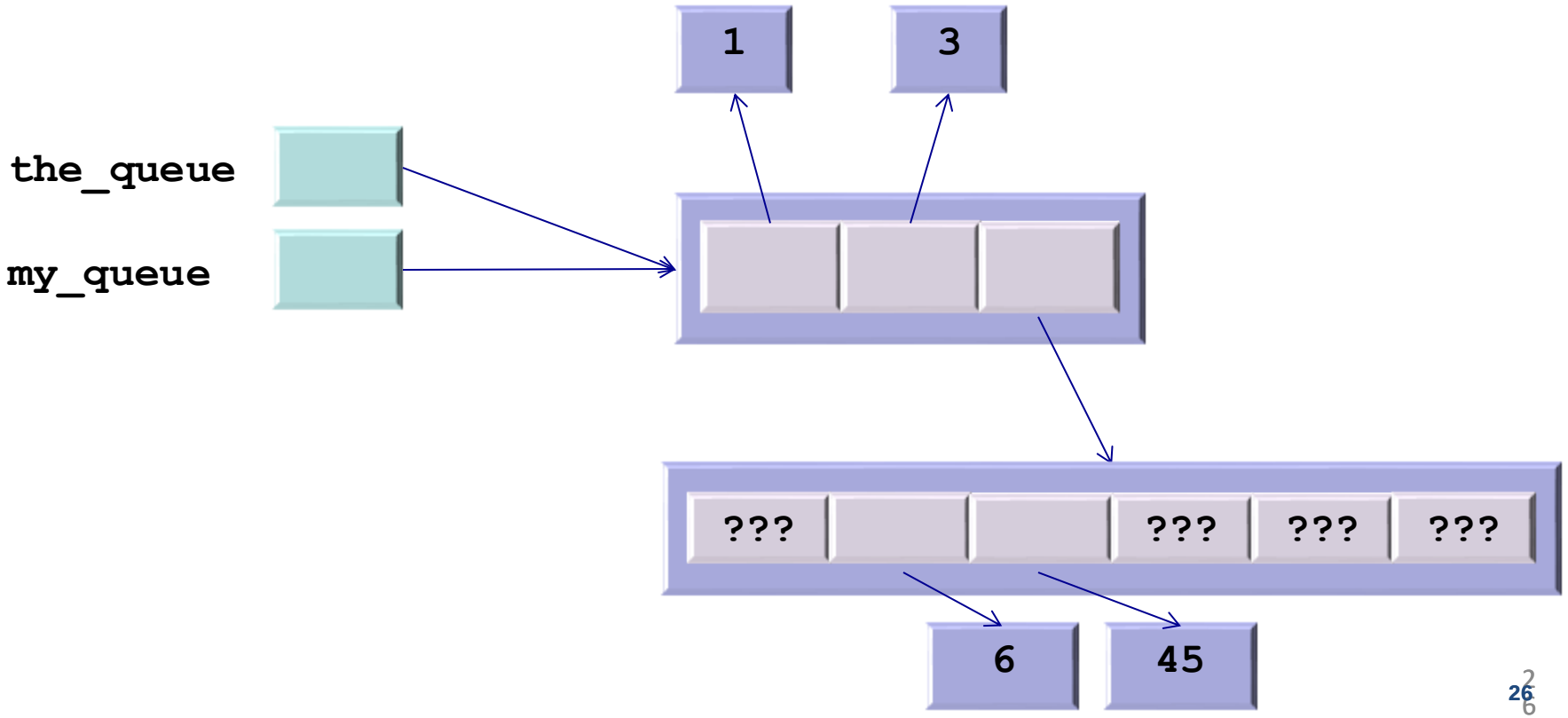
```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

serve(my_queue)



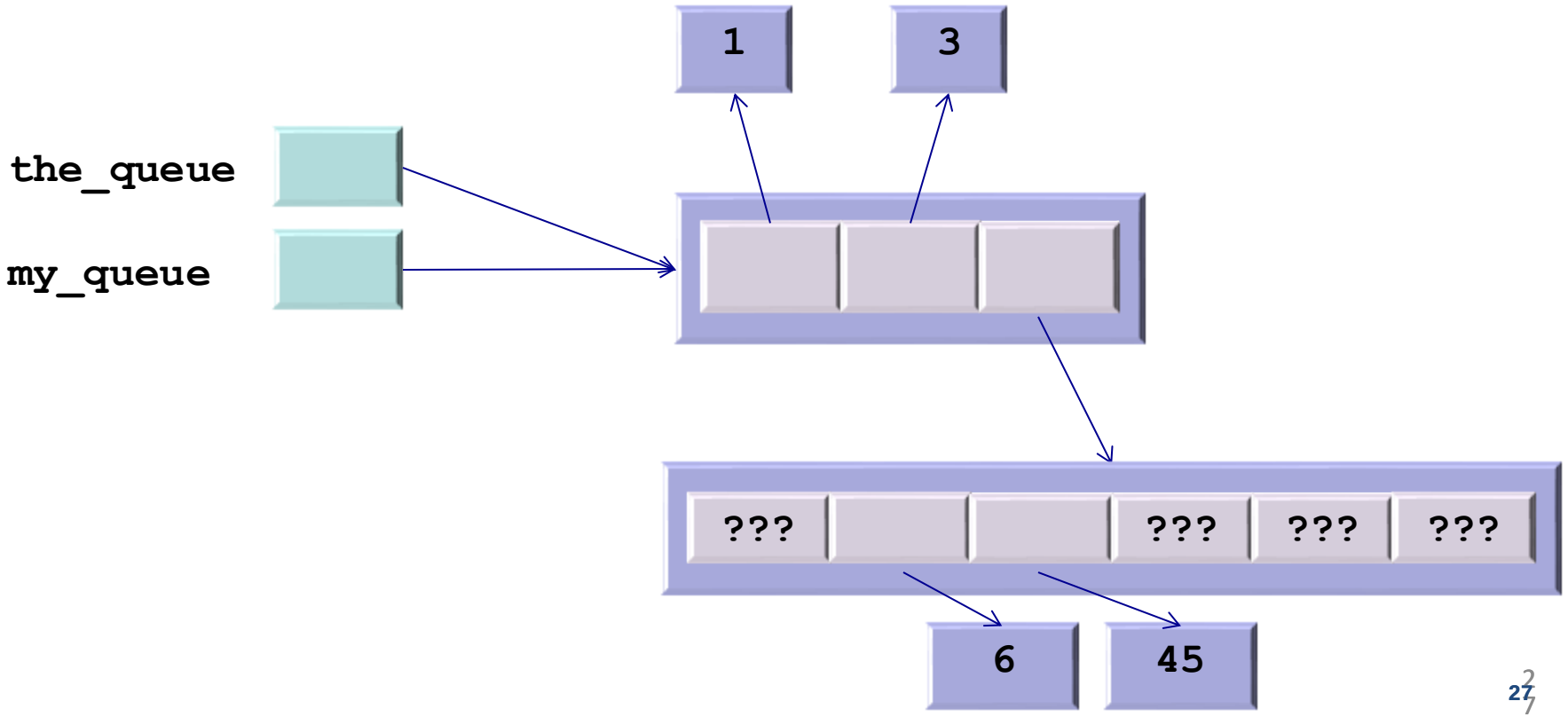
```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

serve(my_queue)



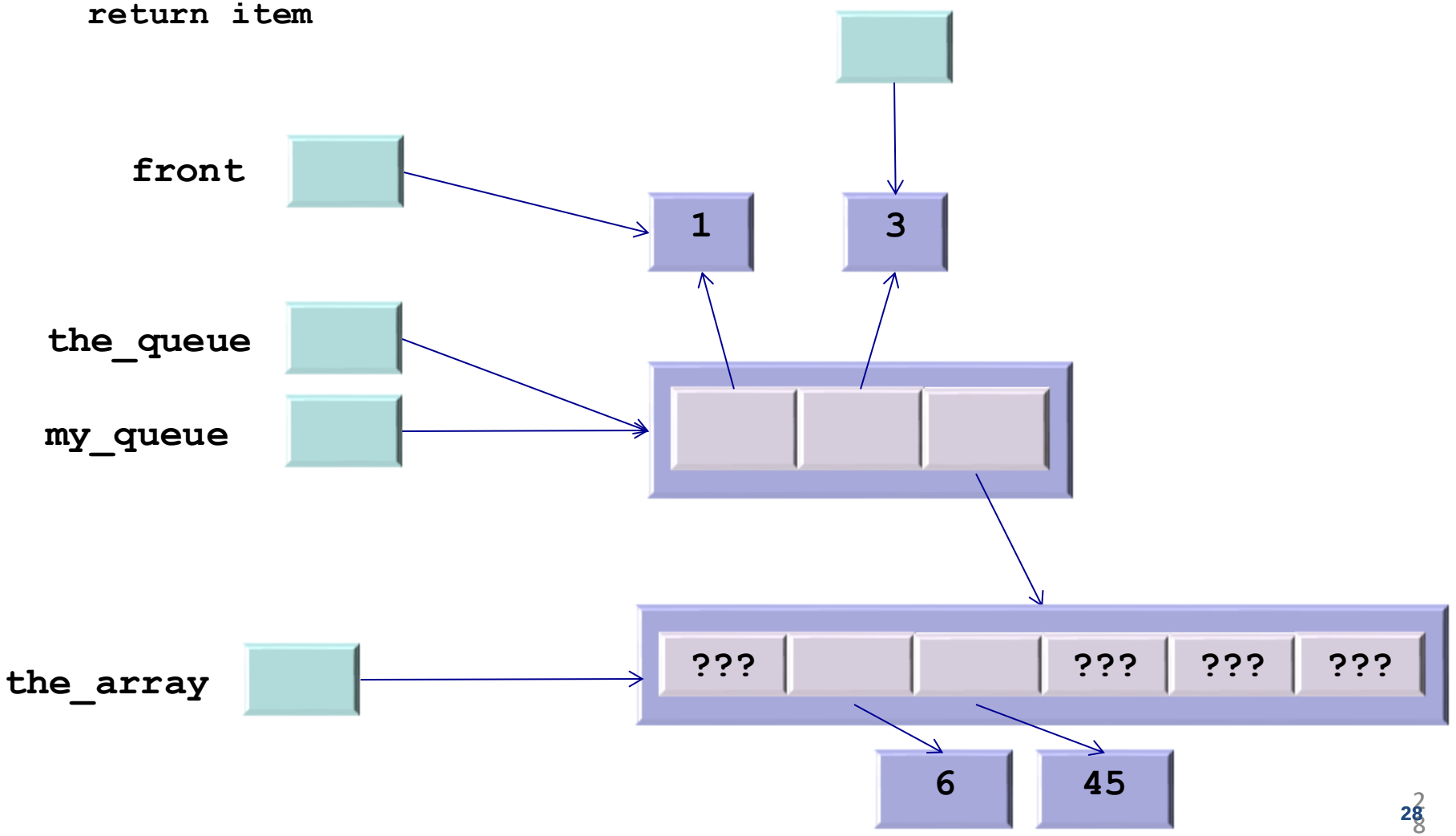
```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

serve(my_queue)



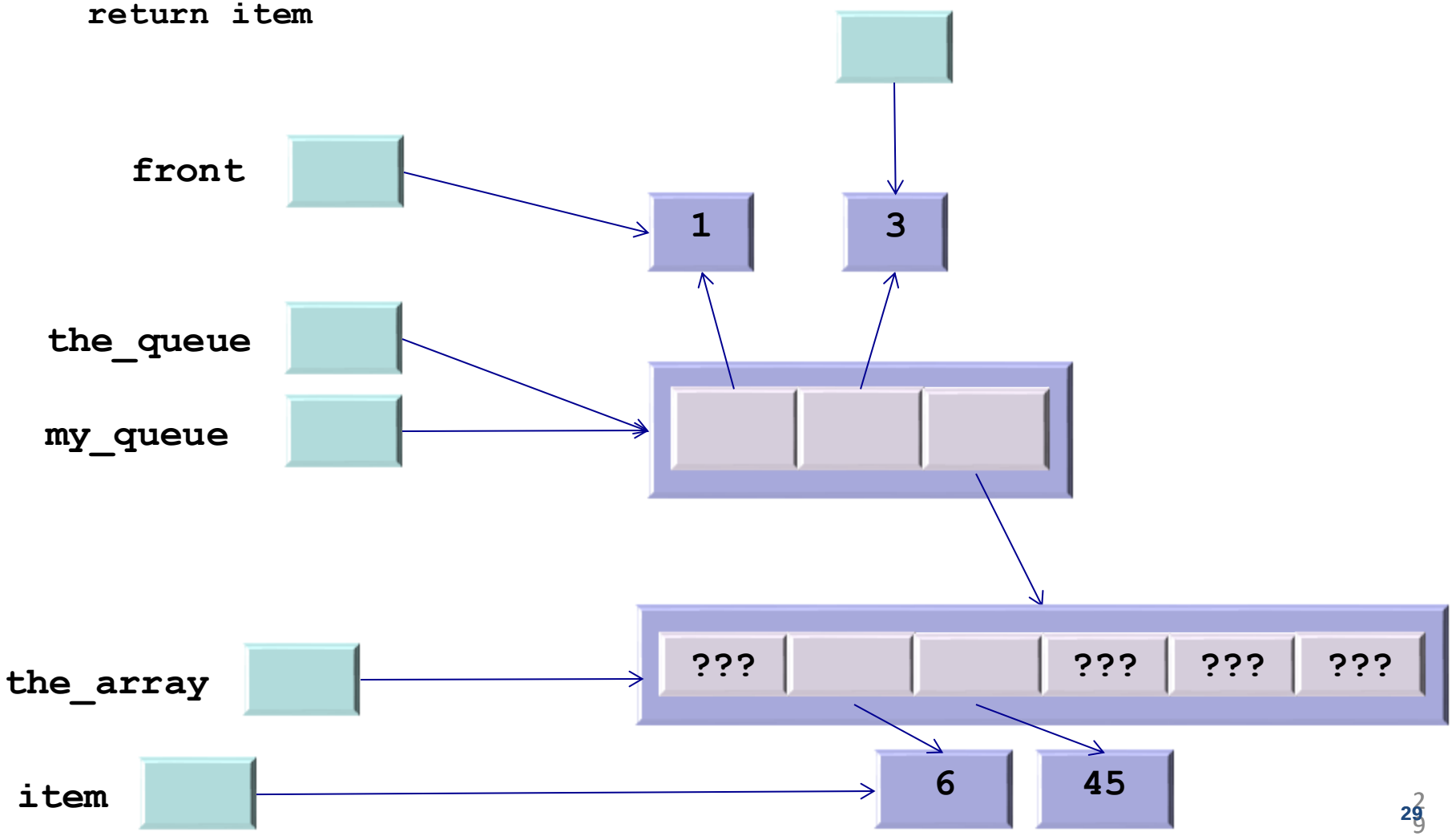
```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

serve(my_queue)



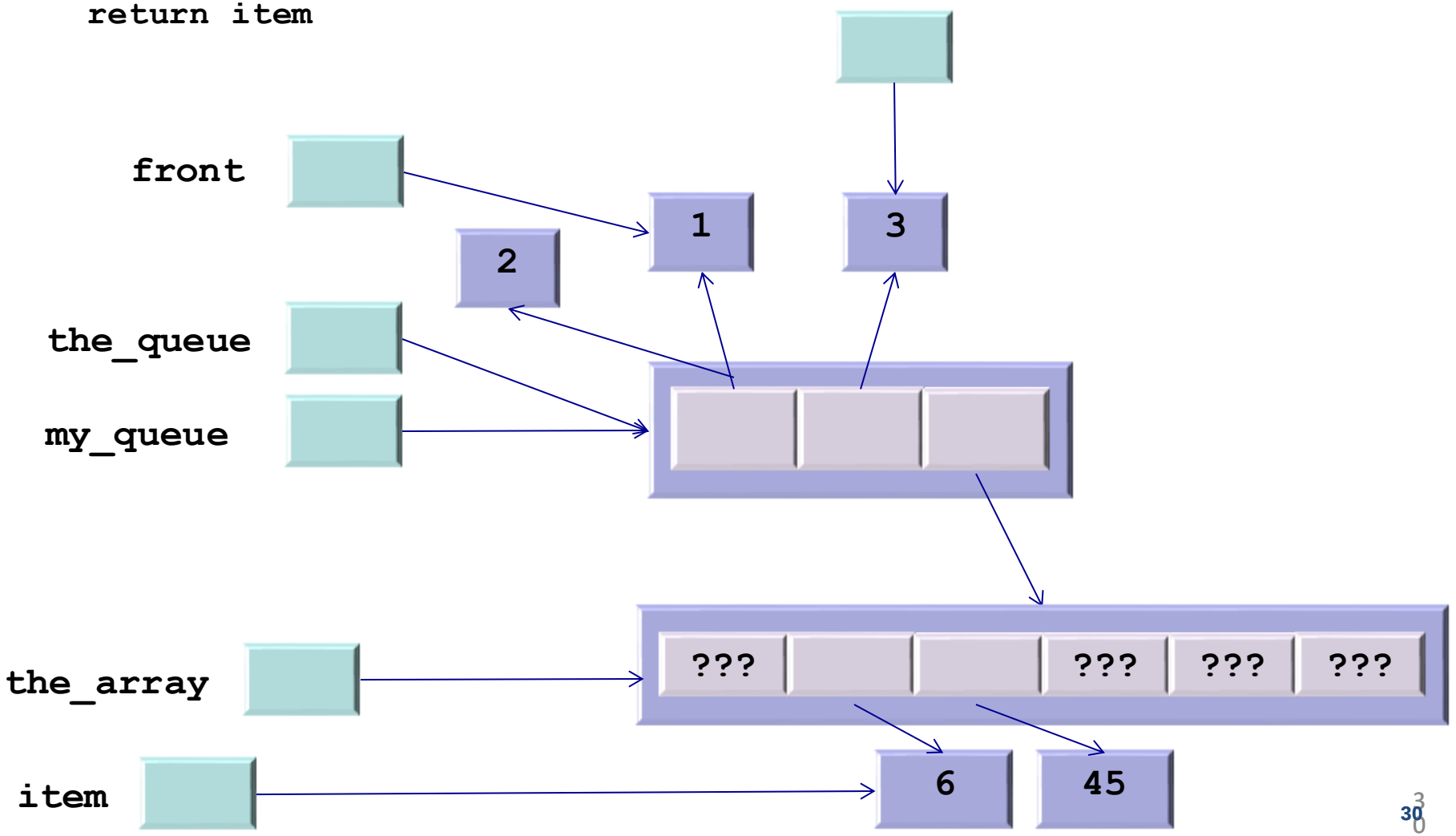
```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

serve(my_queue)



```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

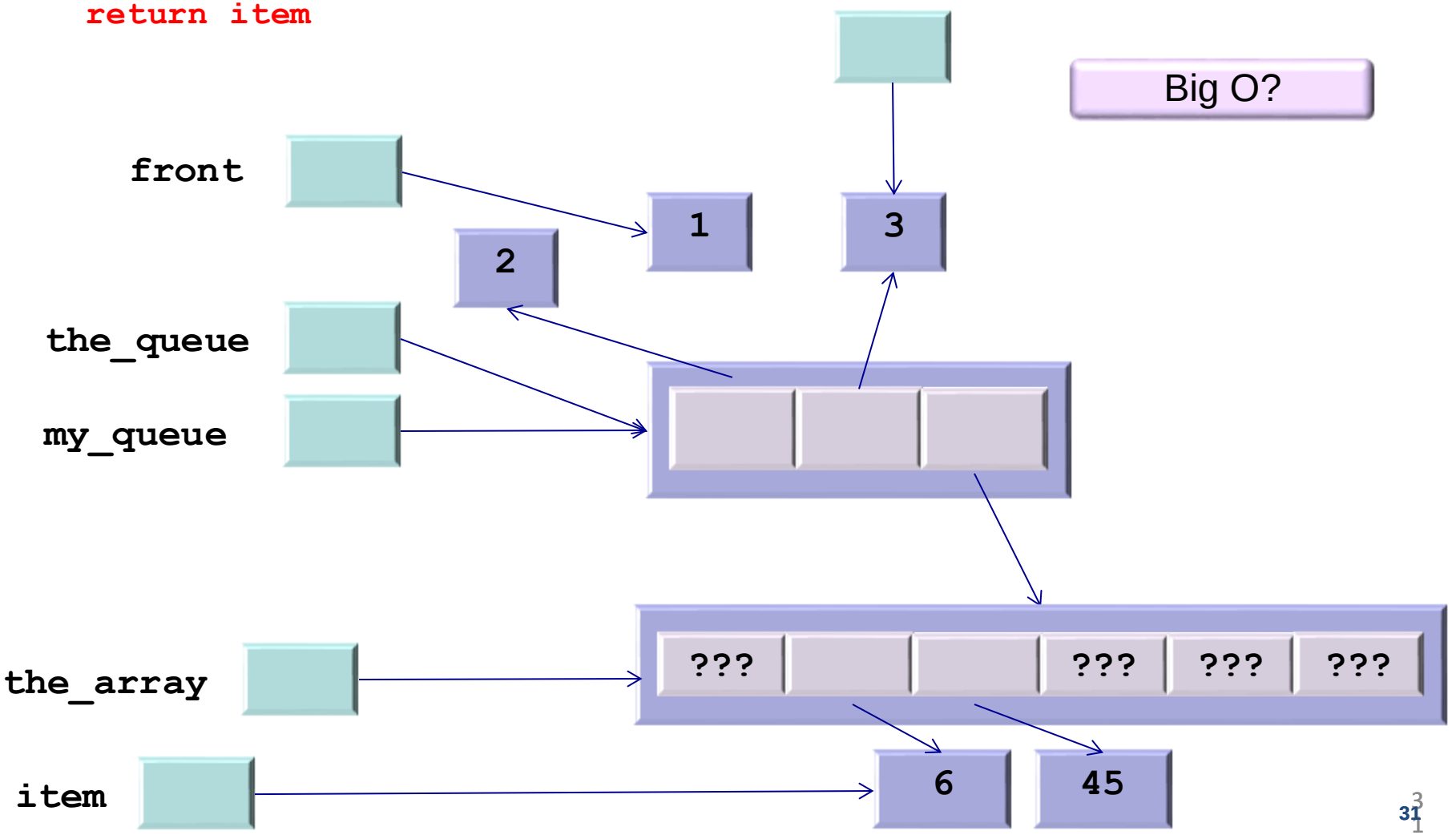
serve(my_queue)



```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array] = the_queue
    item = the_array[front]
    the_queue[0] = front+1
    return item
```

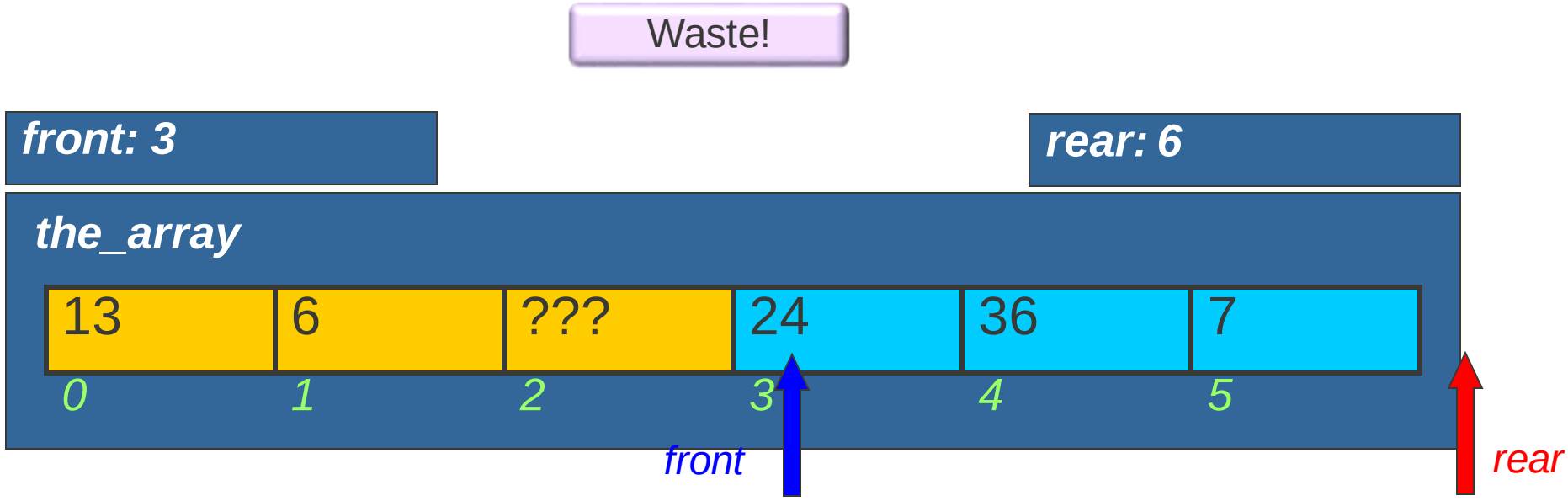
serve(my_queue)

Big O?

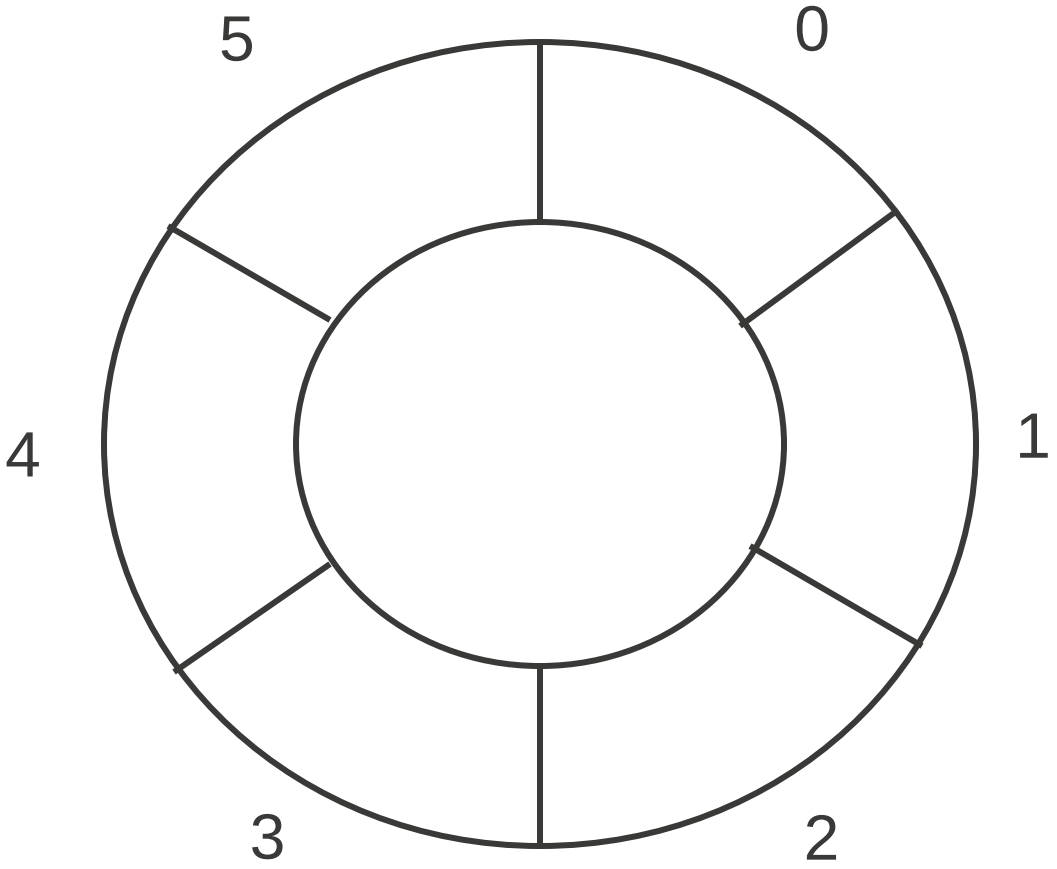


Implementation problem

- When rear reaches the end of the array
- No more elements can be added
- Even if there is space left!



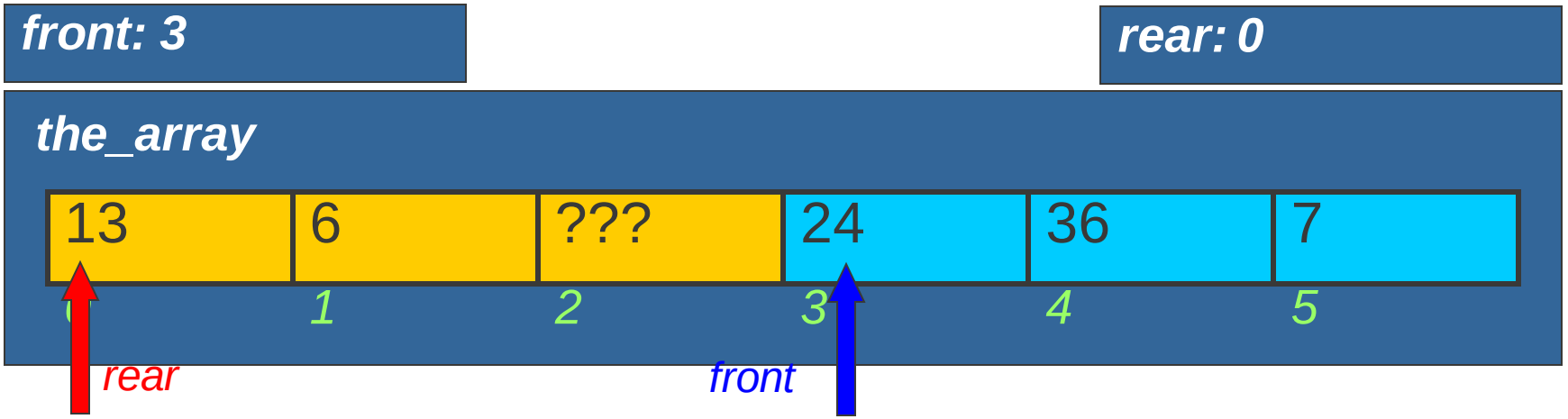
Solution: Circular Queues



Simulated by allowing rear and front to wrap around each other

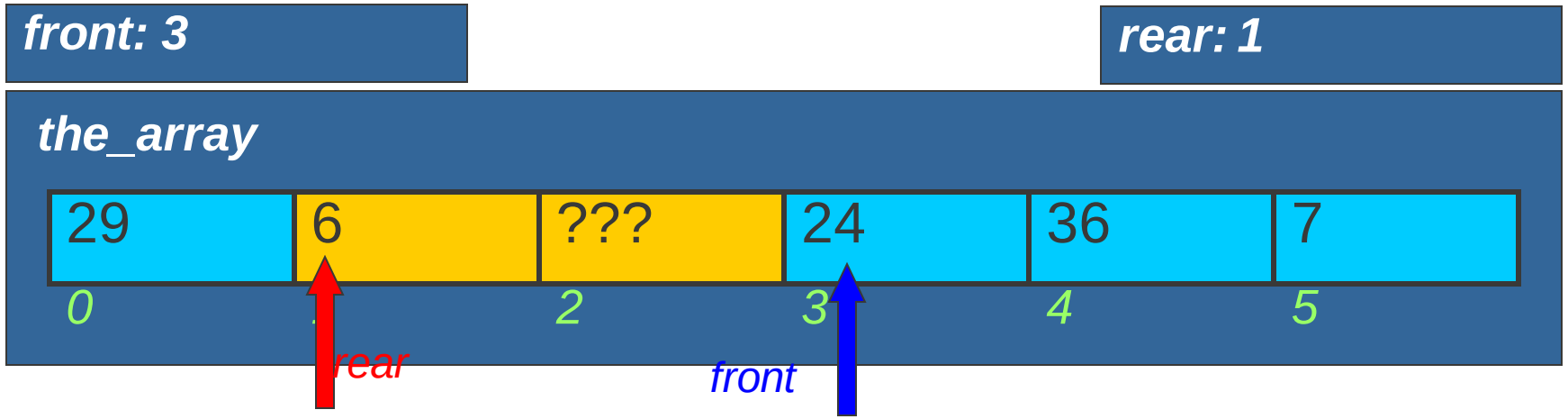
Circular queue

- after appending 7 the situation would be



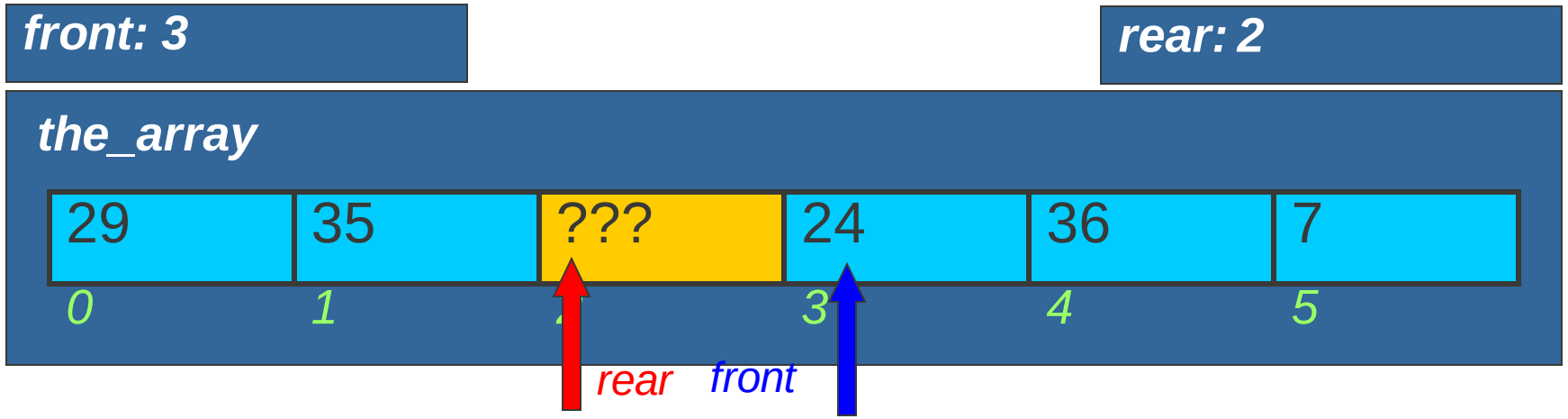
Circular queue

- after appending 7 the situation would be
- append item 29



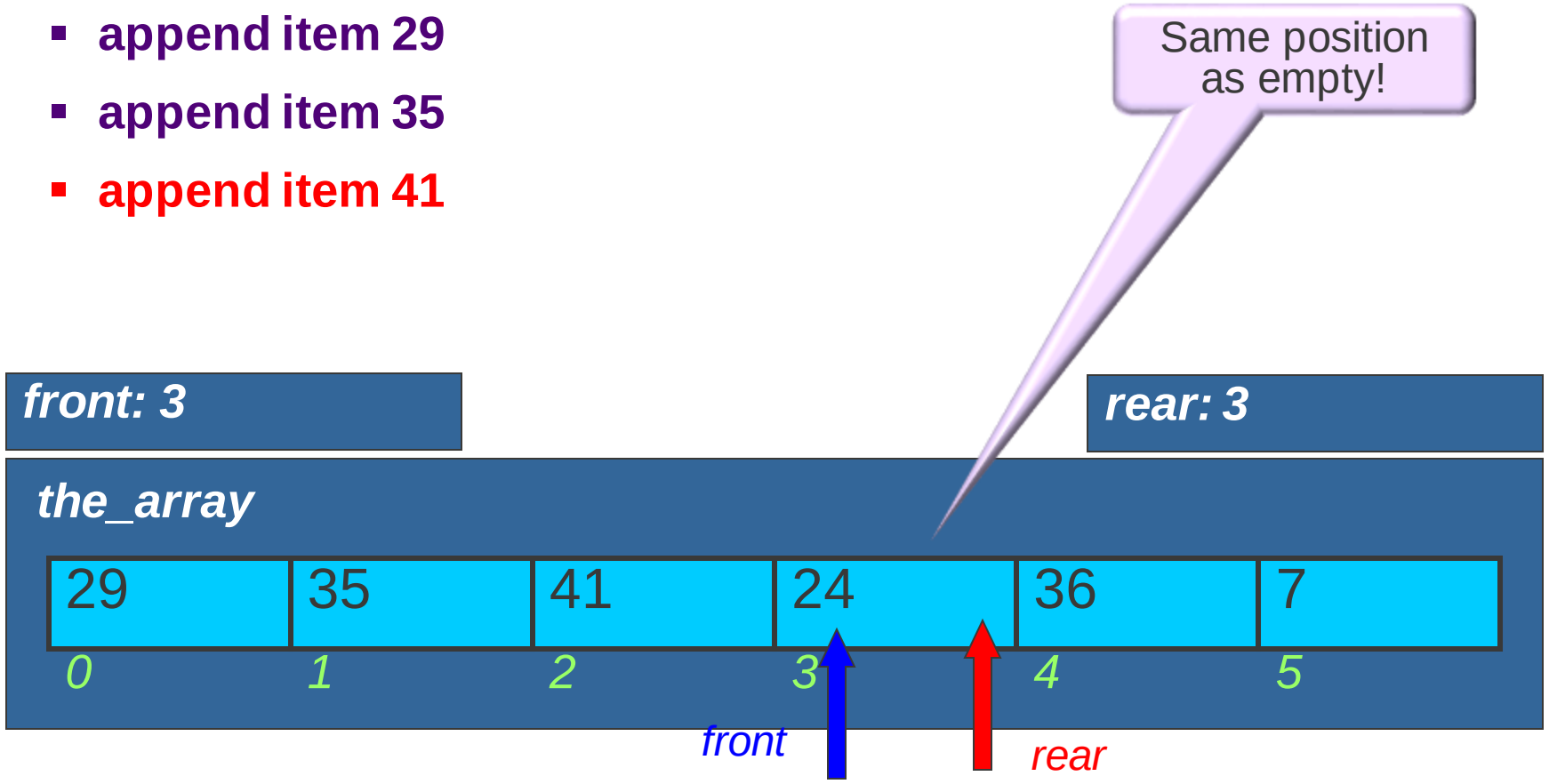
Circular queue

- after appending 7 the situation would be
- append item 29
- append item 35



Circular queue

- after appending 7 the situation would be
- append item 29
- append item 35
- append item 41



Circular queue

- **When front == rear, we need to know whether the queue is empty or full**
- **Some solutions:**
  - Add a counter for number of items
  - Add one cell which is never used
- **The former is simpler (to find the size of the queue…)**

Implementation for a circular Queue

```
def Queue(max_size):
    the_array = [None] * max_size
    front = 0
    rear = 0
    count = 0
    return [front, rear, the_array, count]

def size(the_queue):
    return the_queue[3]

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

def is_full(the_queue):
    [_,_,the_array,count] = the_queue
    return count == len(the_array)
```

Same as returning `count`

Or `count == 0`, that is, a very similar definition to the one used for stacks

Big O?

Implementation for a circular Queue

```
def append(the_queue, item):
    if is_full(the_queue):
        raise Exception("Queue is full")

    [_,rear,the_array,_] = the_queue
    the_array[rear] = item
    the_queue[1] = (rear+1) % len(the_array)
    the_queue[3] += 1
```

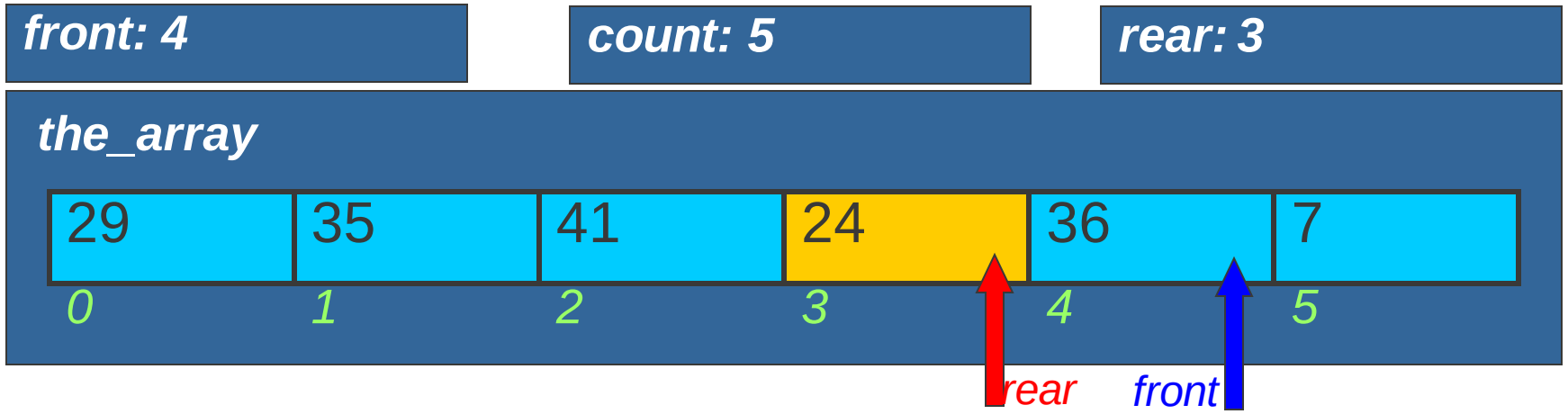
Wraps over the end of the array

Increases the count

Big O?

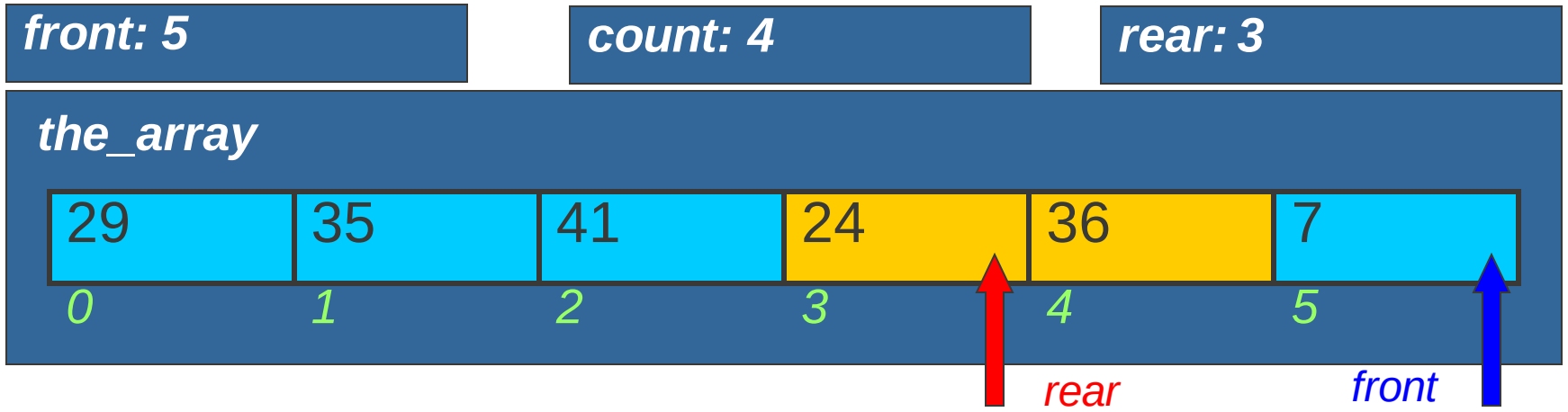
Circular queue: both front and rear wrap

- **serve item (returns 24)**



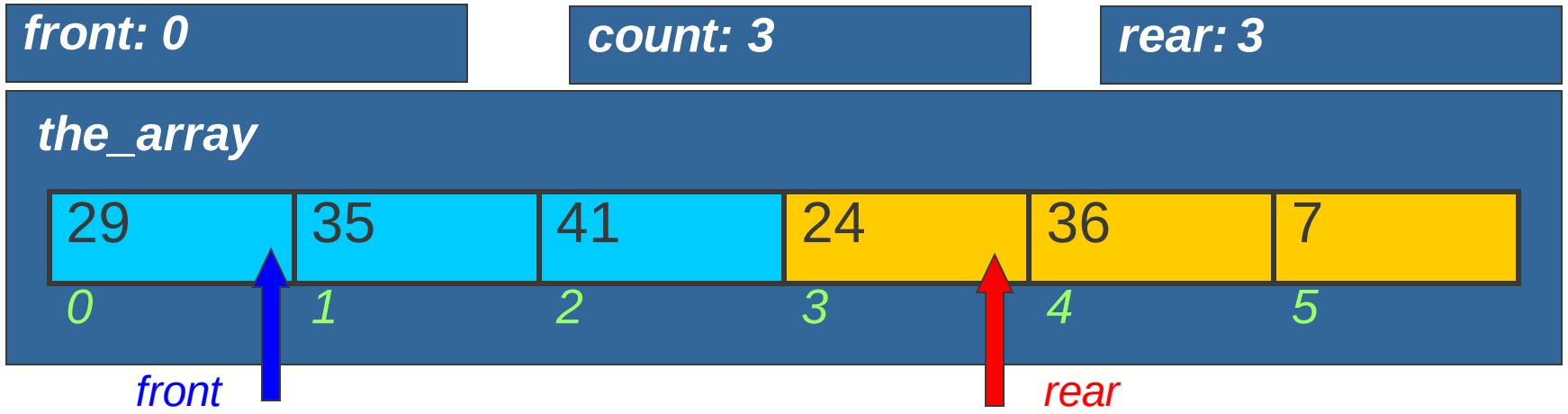
Circular queue: both front and rear wrap

- serve item (returns 24)
- serve item (returns 36)



Circular queue: both front and rear wrap

- serve item (returns 24)
- serve item (returns 36)
- serve item (returns 7)



Implementation for a circular Queue

```
def serve(the_queue):
    if is_empty(the_queue):
        raise Exception("Queue is empty")

    [front,_,the_array,_] = the_queue
    item = the_array[front]
    the_queue[0] = (front+1) % len(the_array)
    the_queue[3] -= 1
    return item
```

Wraps over the end of the array

Decreases the count

Big O?

Example: print elements from front to rear

- **Lets implement it as a function within the `Queue` ADT**
  - So, it has access to the implementation
- **Do not modify the queue, just print its elements**

```
def print_items(the_queue):
    [front,_,the_array,count] = the_queue
    index = front
    for _ in range(count):
        print(the_array[index])
        index = (index+1)%len(the_array)
```

Could you say:
`for item in the_array`?

Complexity of circular Queue operations

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

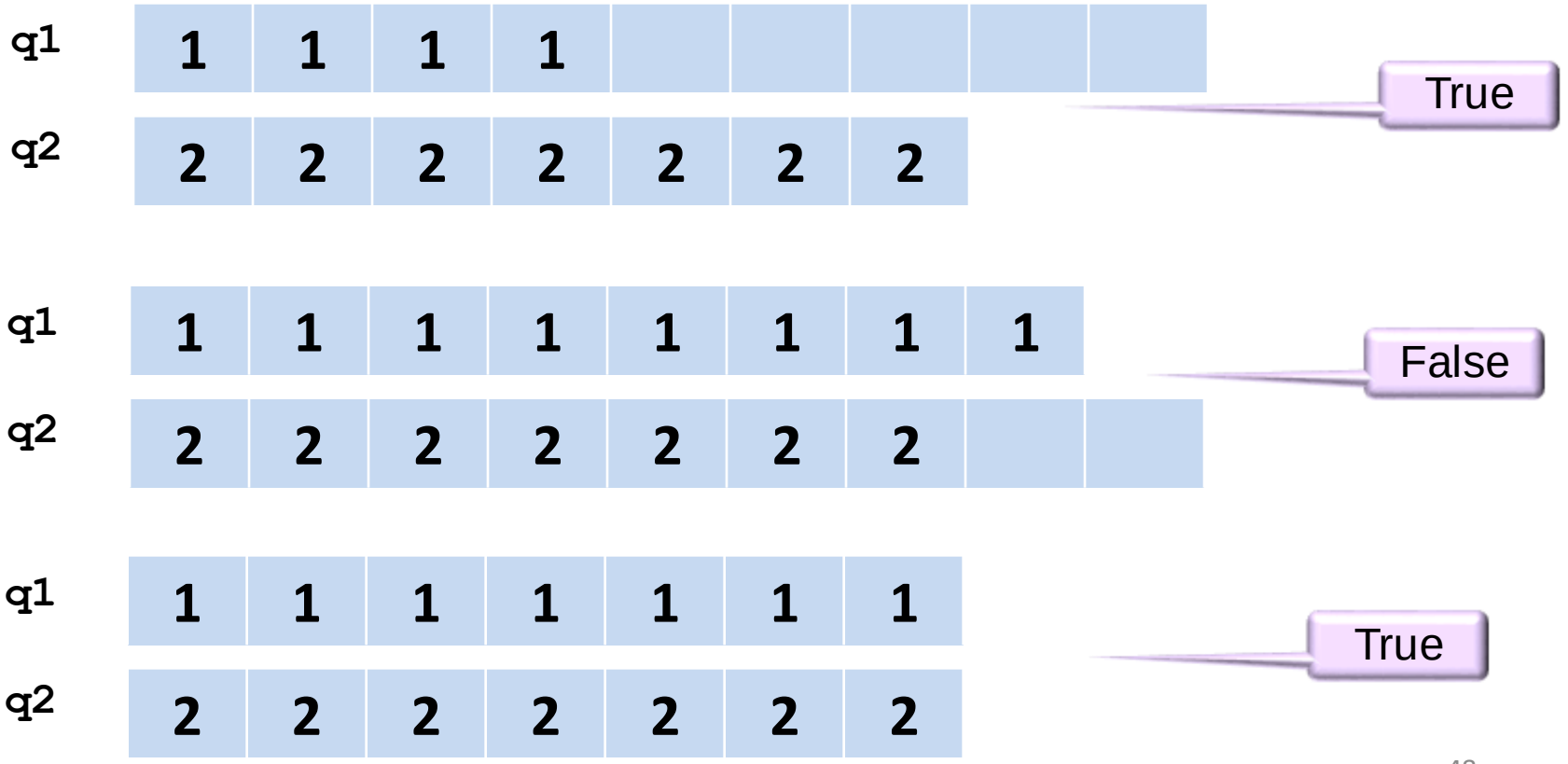
```
def print_items(the_queue):
    [front,_,the_array,count] = the_queue
    index = front
    for _ in range(count):
        print(the_array[index])
        index = (index+1)%len(the_array)
```

```
def greater(q1,q2)
```

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

Don't worry about modifying the input queues



Make sure you use it as an ADT (no access to the implementation)

Try first using only **`is_empty`** and **`serve`**

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

Try now using **`size`**

Some Queue Applications

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

Summary

- **Queues**
  - Array implementation
  - Basic operations
  - Their complexity