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

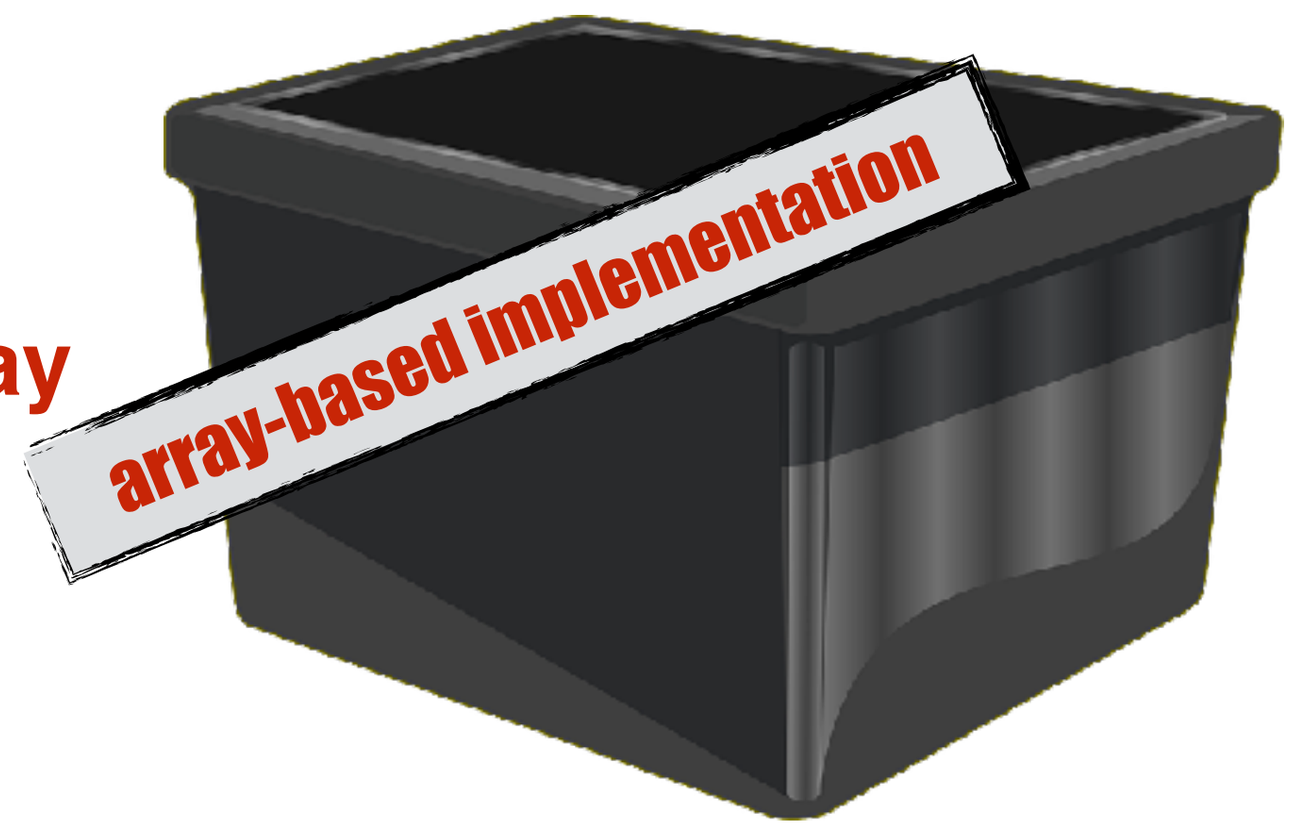
Lecture 22
Queues
(Array Implementation)

FIT 1008
Introduction to Computer Science

COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING
This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act).The material in this communication may be subject to copyright under the Act.Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.
Do not remove this notice.

Container ADTs

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



“Form an orderly queue to the left..”

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)

Queue Data Type

- Follows the **FIFO model**

- Its operations (interface) are:
  - **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)

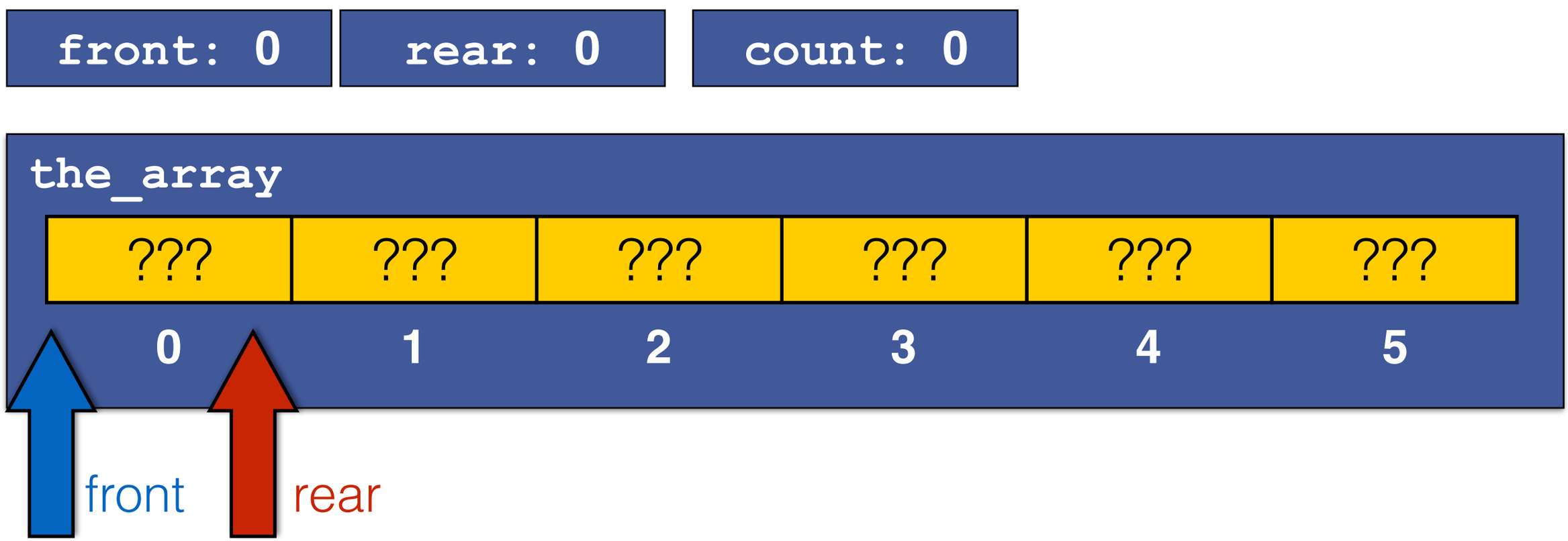
Possible implementation: linear queue

- We need to: **add items** at the **<u>rear</u>**. **take** items from the **<u>front.</u>**

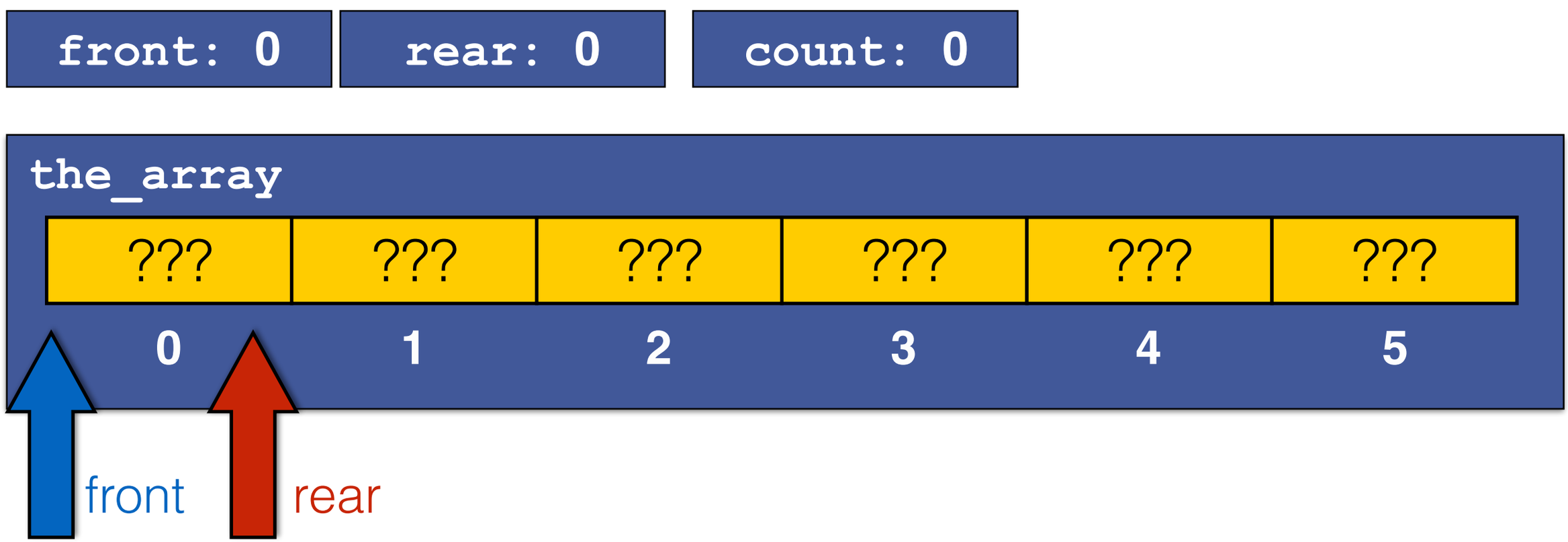
  A single marker is not going to be enough.

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

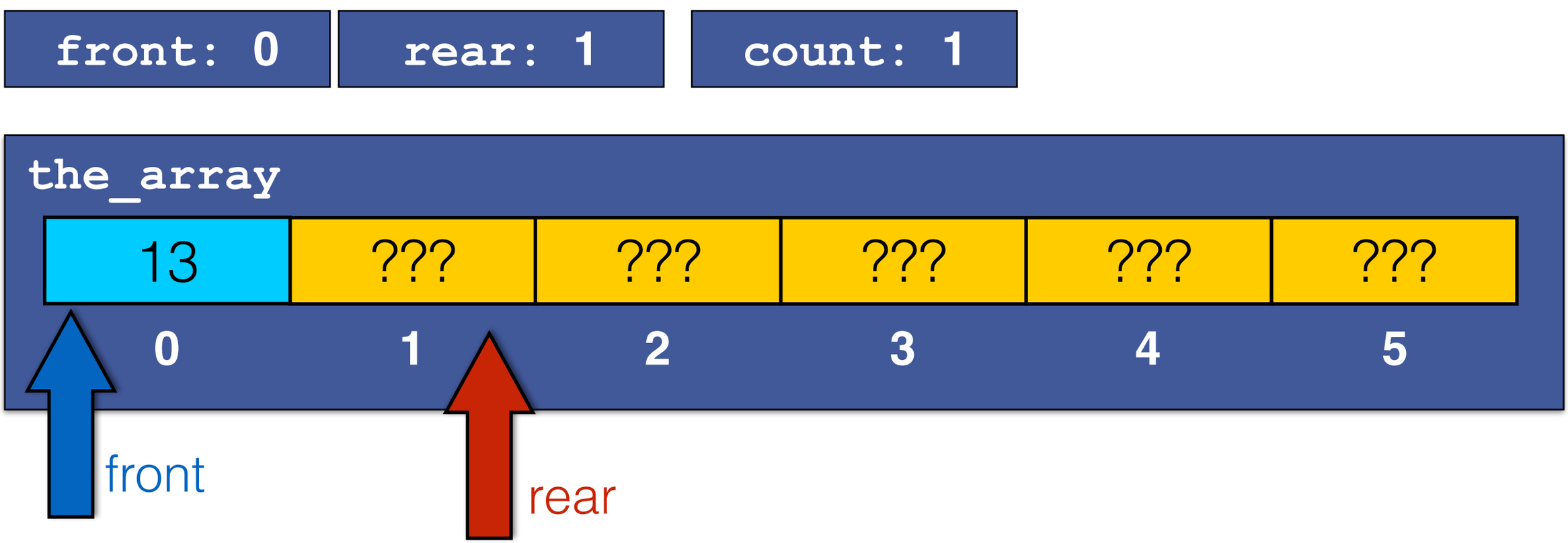
- **Invariant**: <u>valid data</u> appears in `front` … `rear-1` positions


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

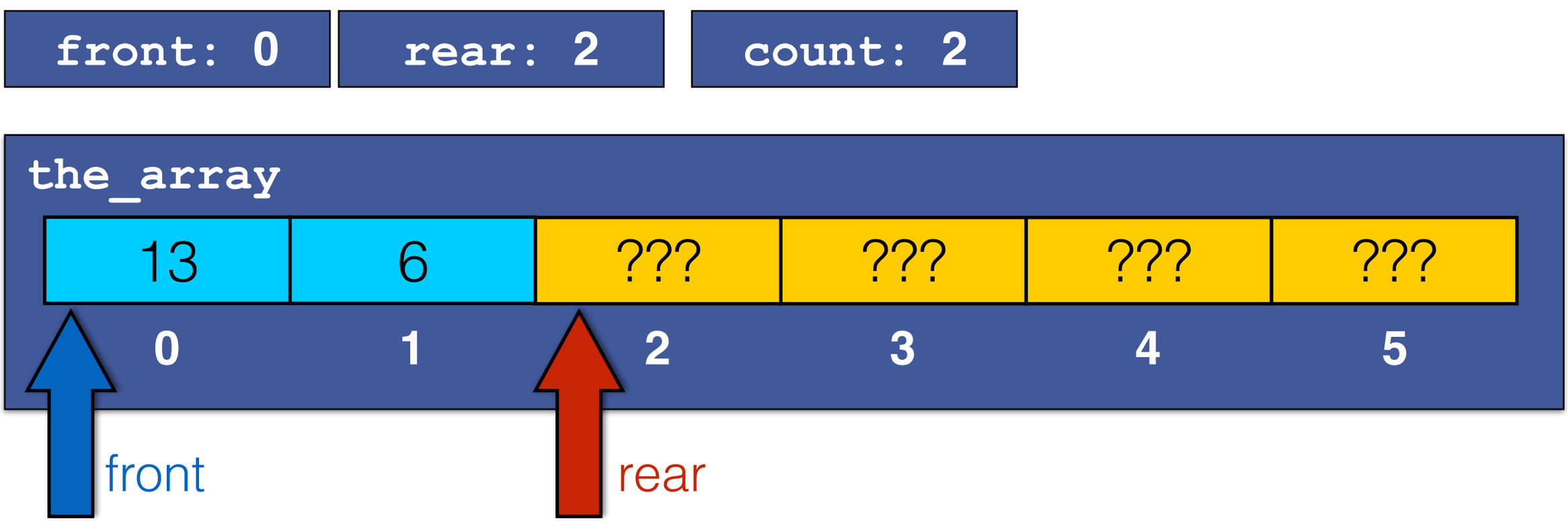
- Create a new queue: no items



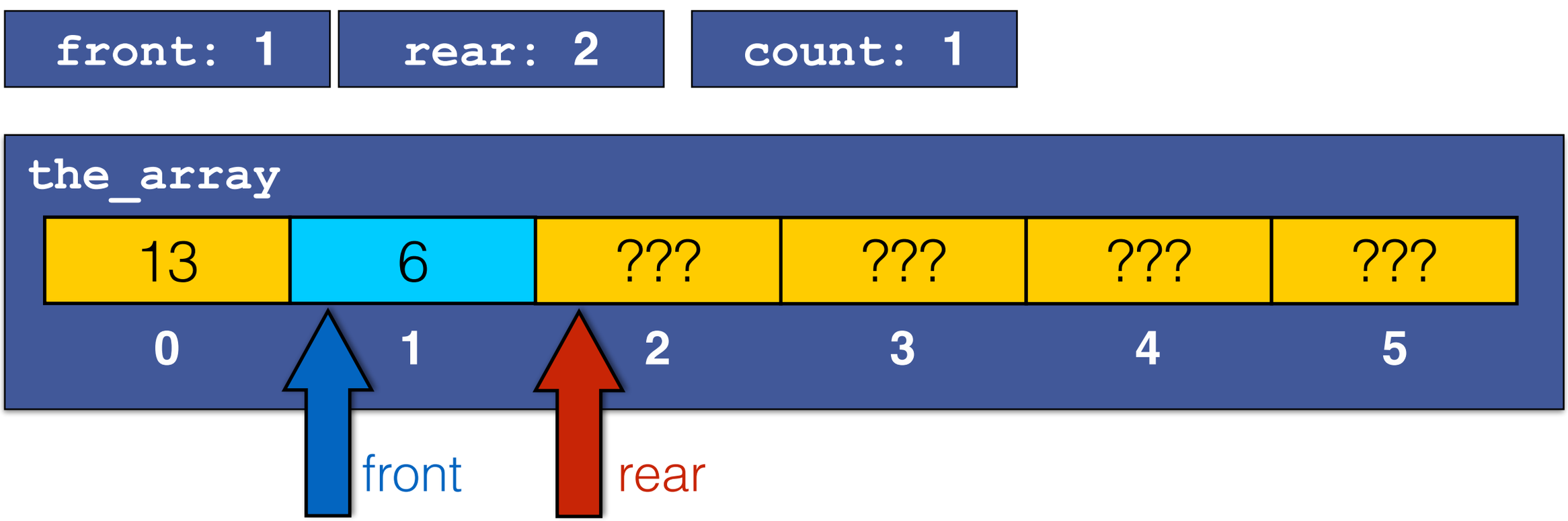
- Create a new queue: no items
- Append item 13



- 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 13



Creating a Linear Queue

```
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0
```

Creating a Linear Queue

```
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0
```

Creating a Linear Queue

```
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0
```

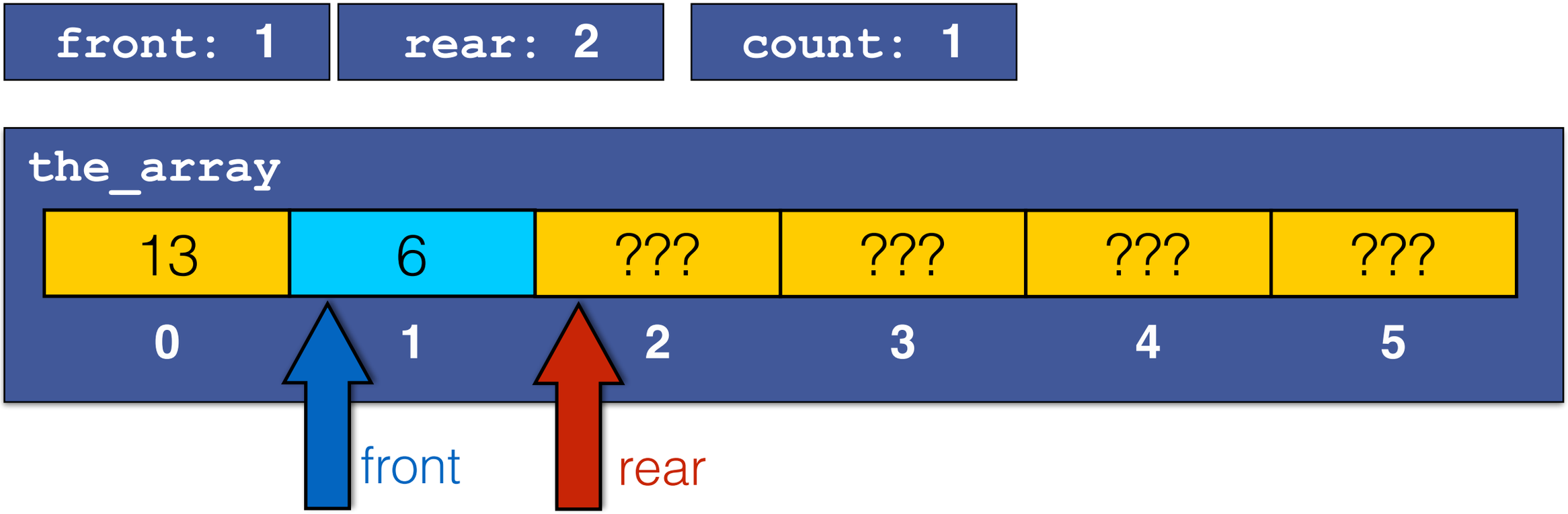
Instance variables

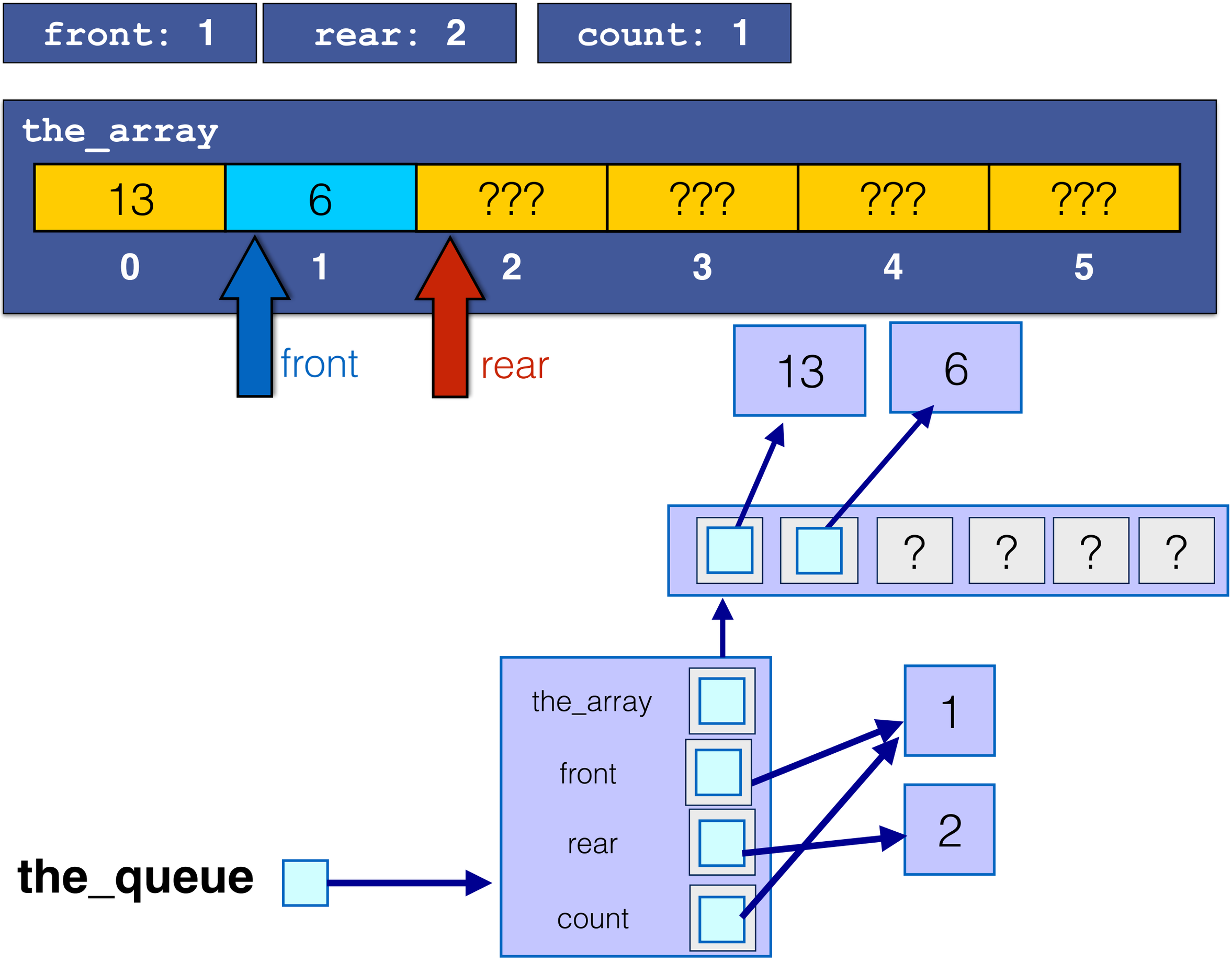
Creating a Linear Queue

```
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 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


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

Simple methods

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

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Simple methods

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

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Complexity is O(1)
for all of these methods.

Implementing Append

```
def append(self, new_item):
```

Implementing Append

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
```

Implementing Append

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

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
```

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
```

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):
```

Implementing Serve

```
def serve(self):
    assert not self.is_empty(), "Queue is empty"
```

Implementing Serve

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

Implementing Serve

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

Implementing Serve

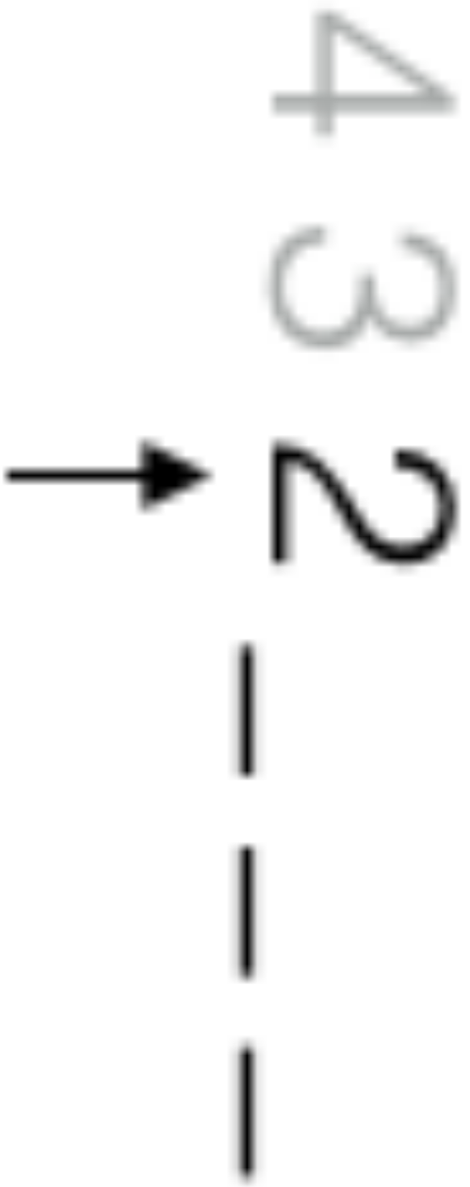
```
def serve(self):
    assert not self.is_empty(), "Queue is empty"
    item = self.the_array[self.front]
    self.front +=1
    self.count -=1
    return item
```

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)

```python
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0

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

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

    def reset(self):
        self.front = 0
        self.rear = 0
        self.count = 0

    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

    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
```


2
3
4

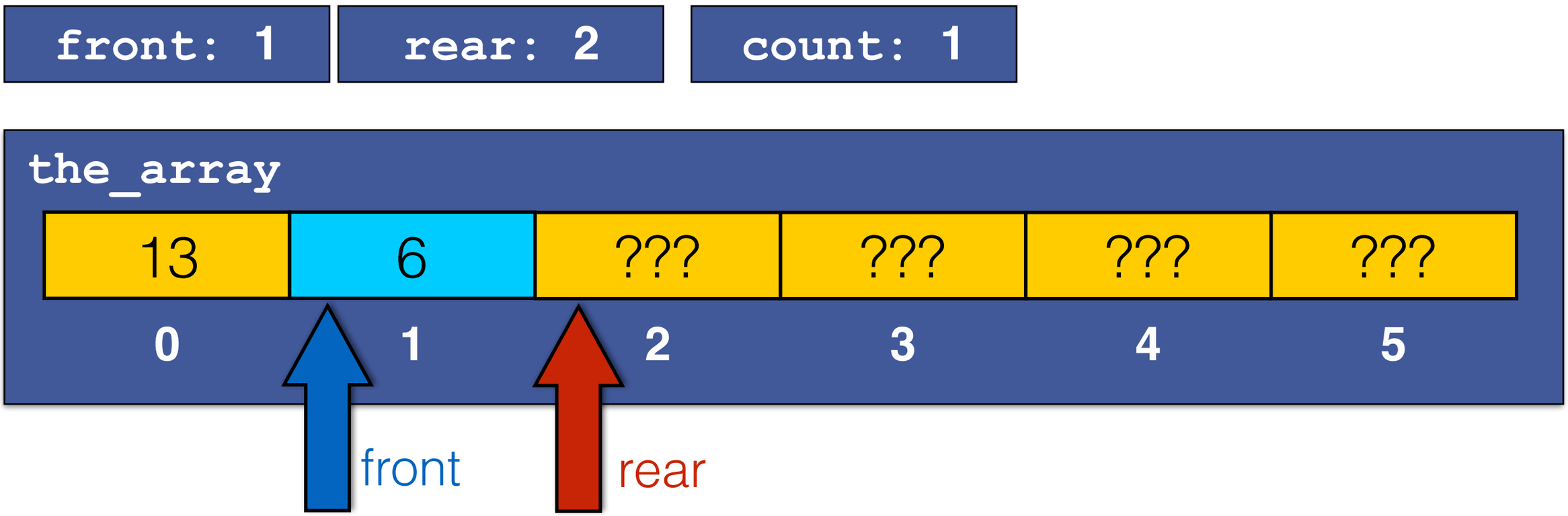
wasteful | ˈweɪstfʊl, -f(ə)l |

adjective

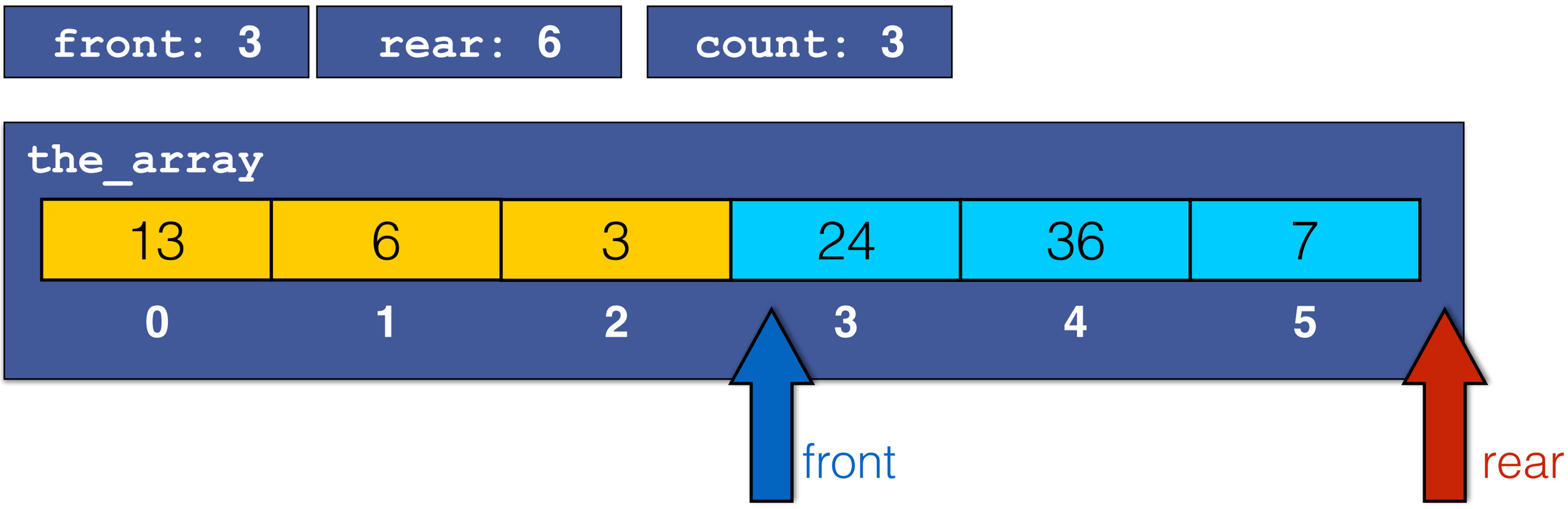
(of a person, action, or process) using or expending something of value carelessly, extravagantly, or to no purpose: *wasteful energy consumption*.

DERIVATIVES

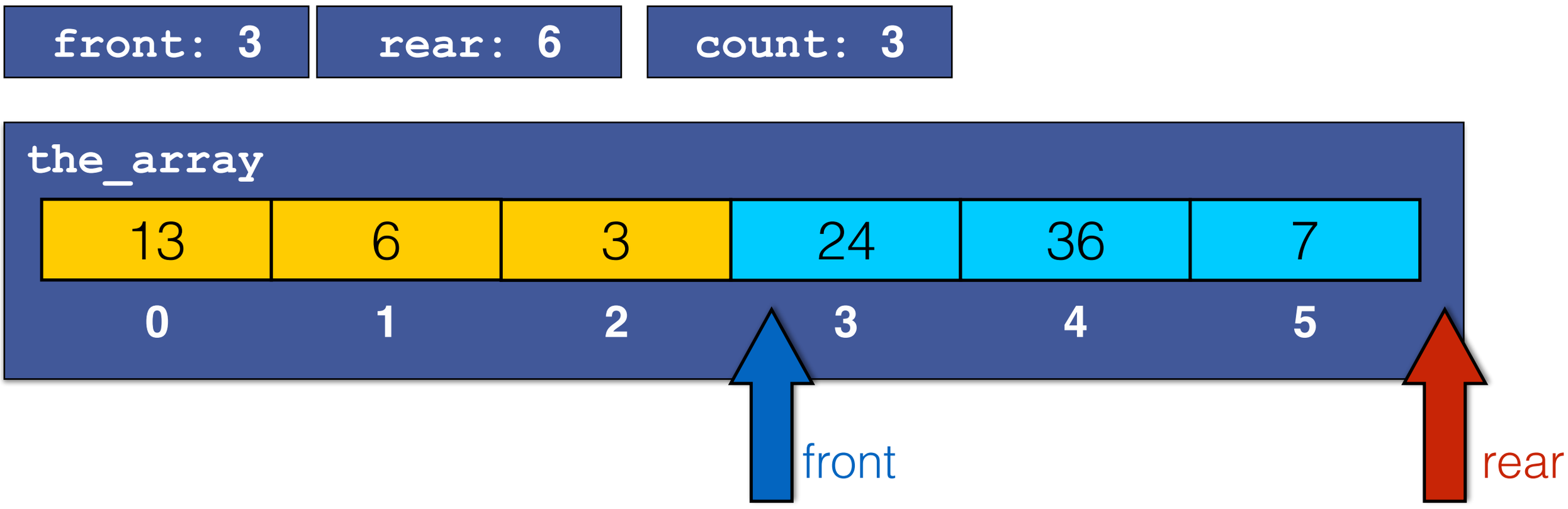
wastefully adverb ,
wastefulness noun


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

Implementation problem

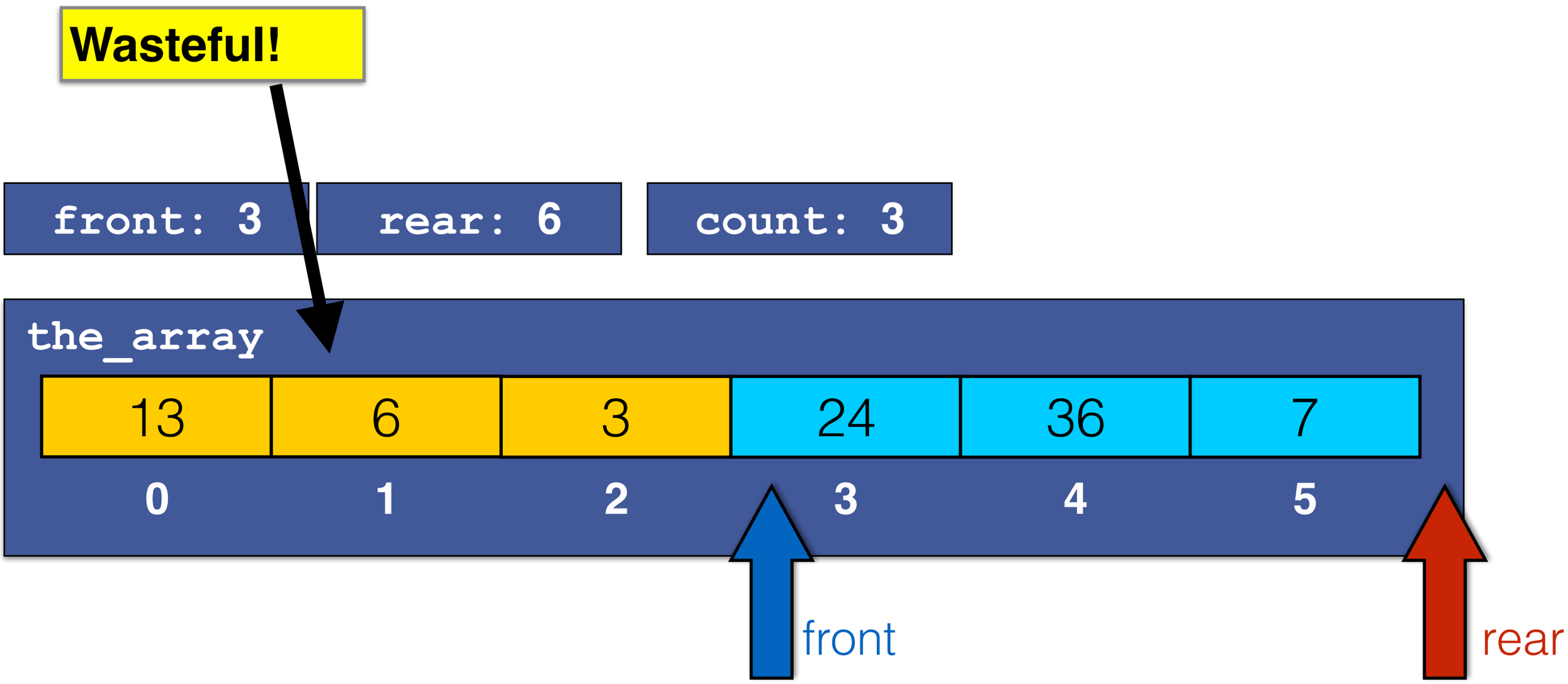


Implementation problem



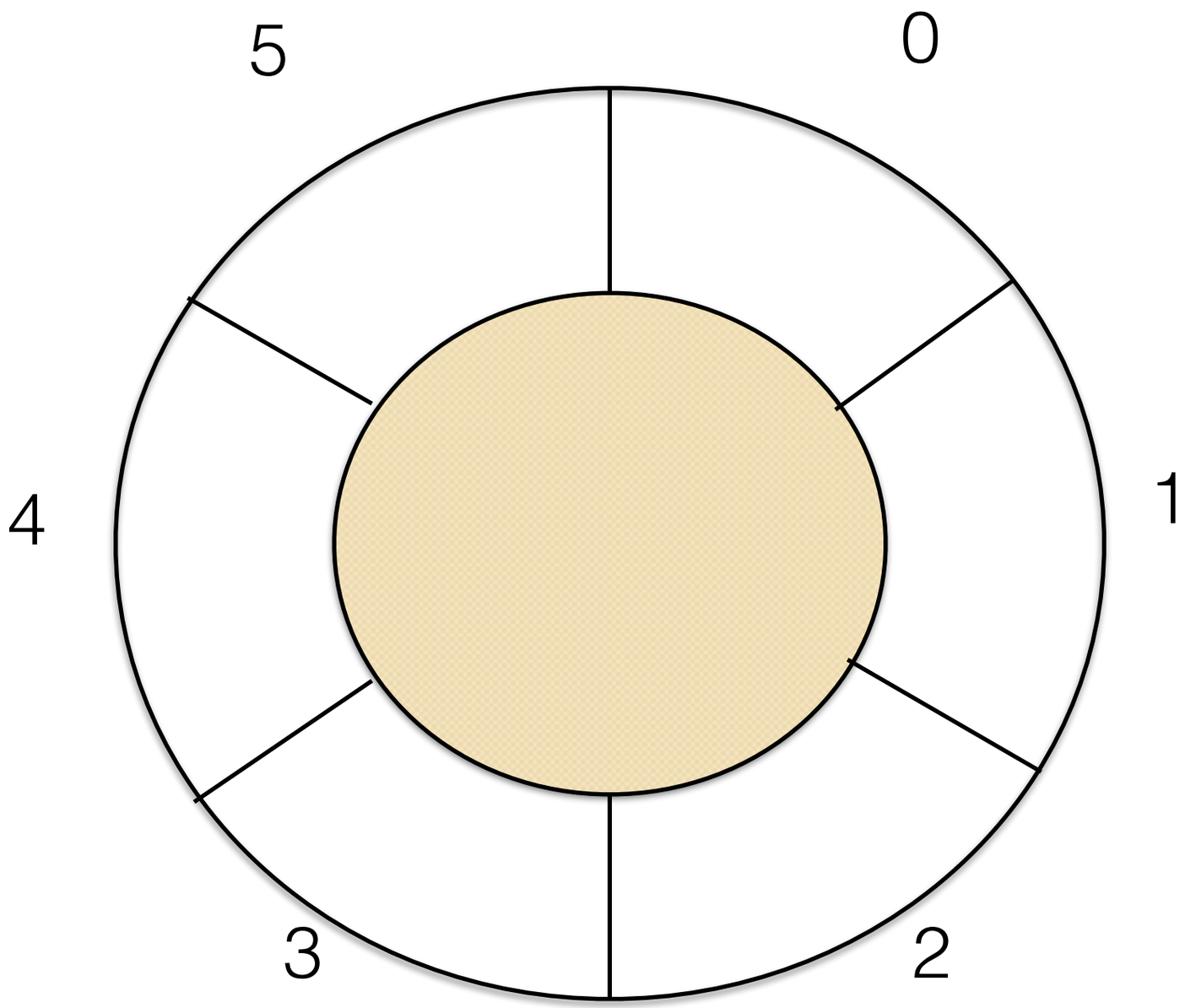

```
def is_full(self):
    return self.rear >= len(self.the_array)
```

Implementation problem

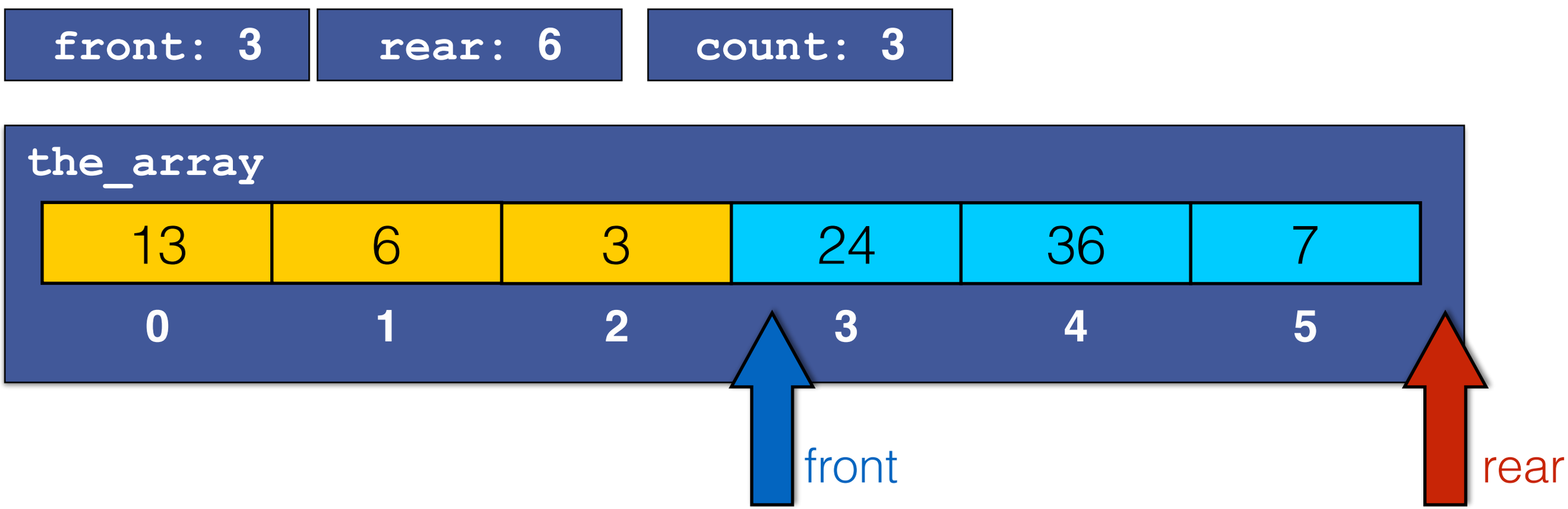


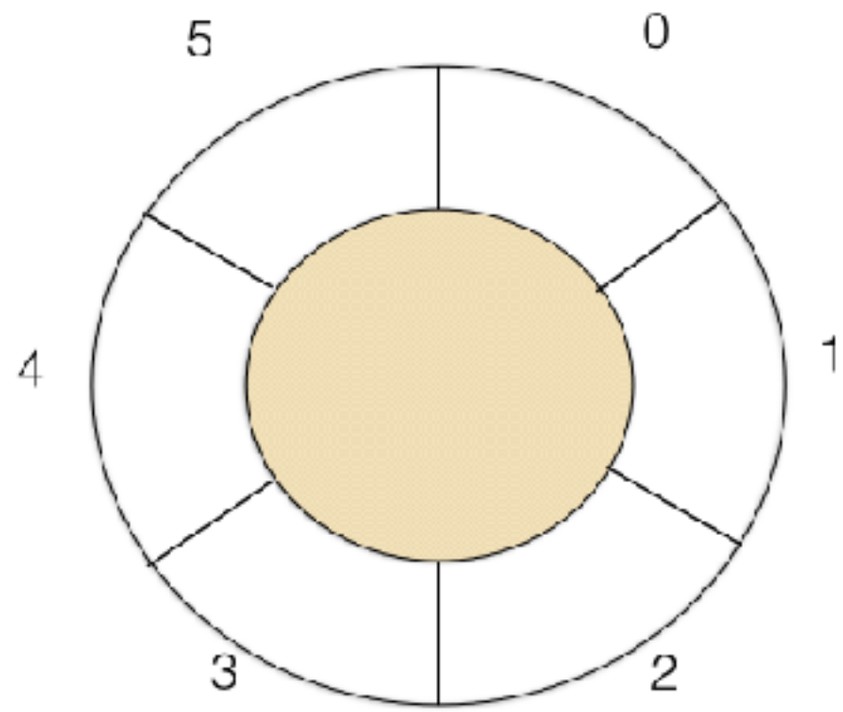

```
def is_full(self):
    return self.rear >= len(self.the_array)
```

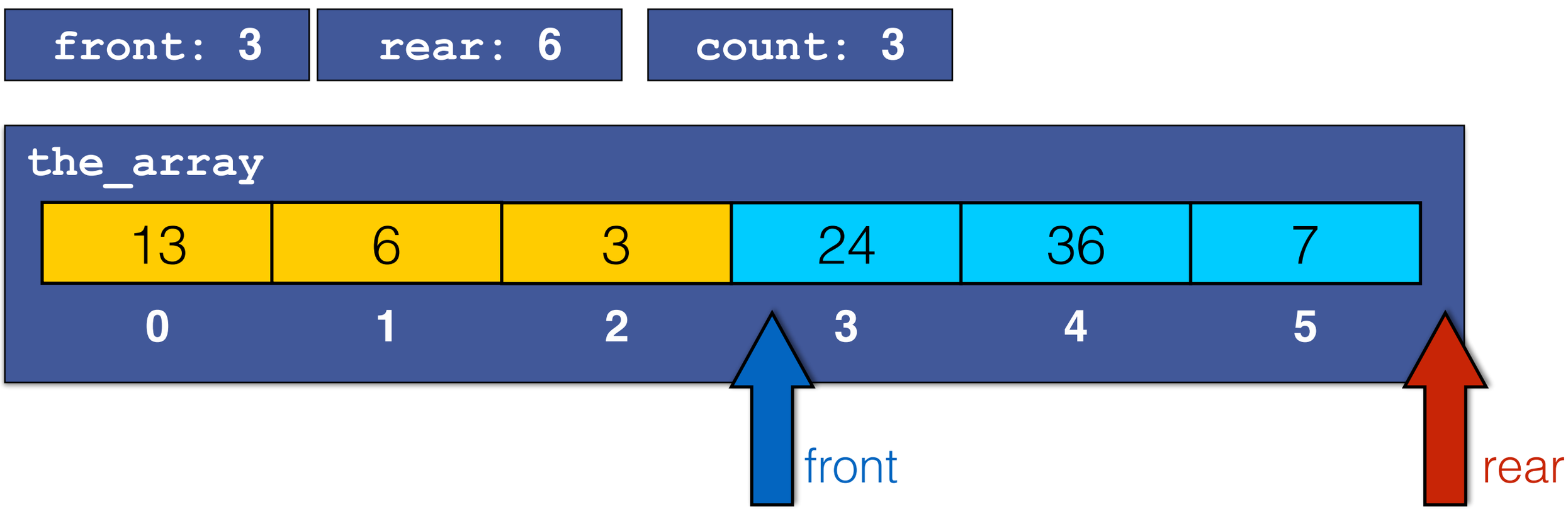
Solution: Circular Queues

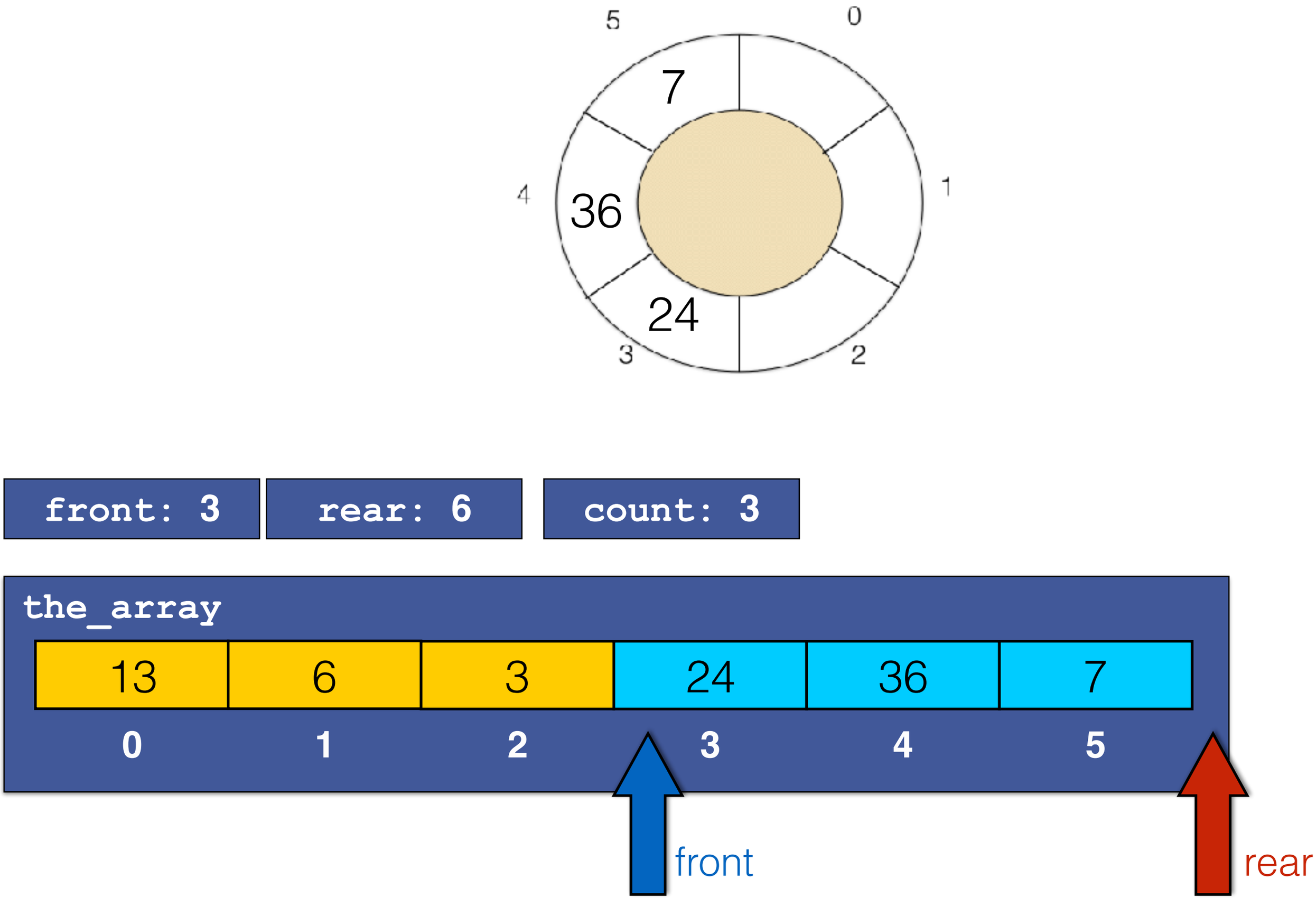


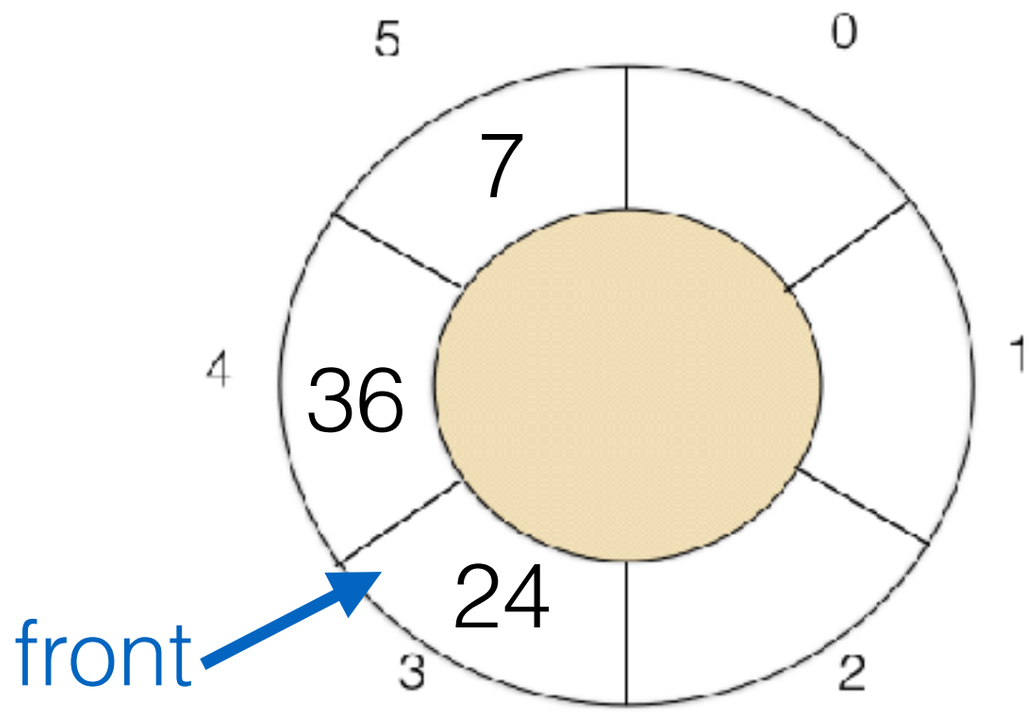
Simulated by allowing **rear** and **front** to wrap around each other

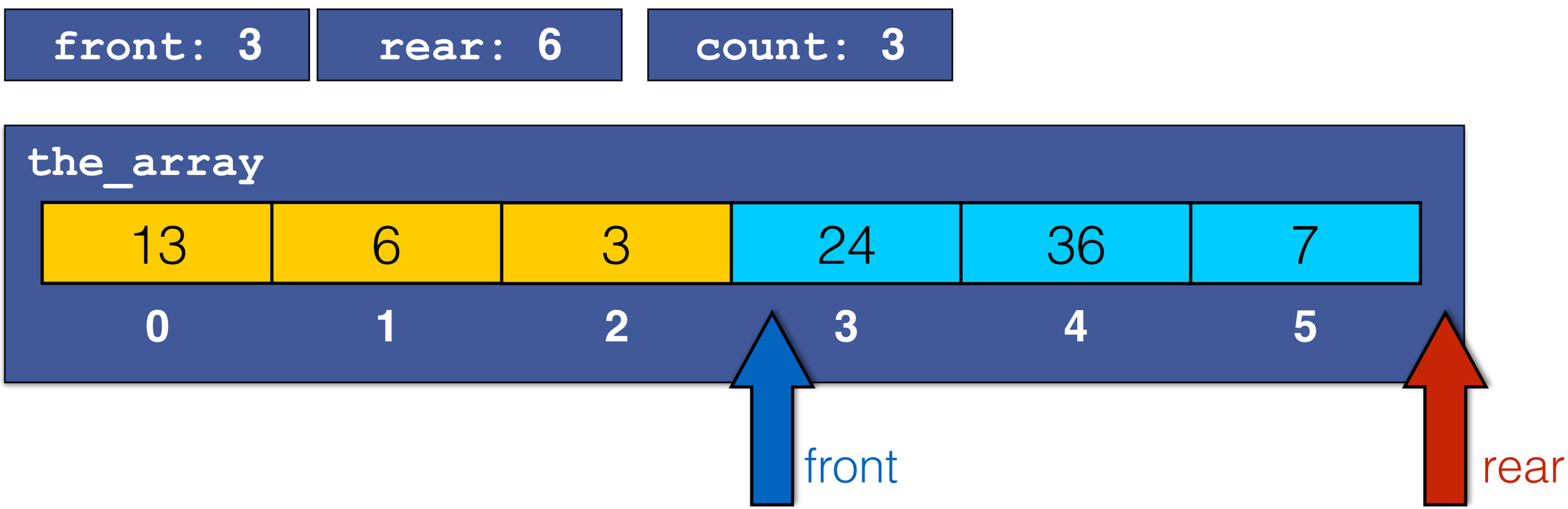

front: 3
rear: 6
count: 3
the_array
13
6
3
24
36
7
0
1
2
3
4
5
front
rear

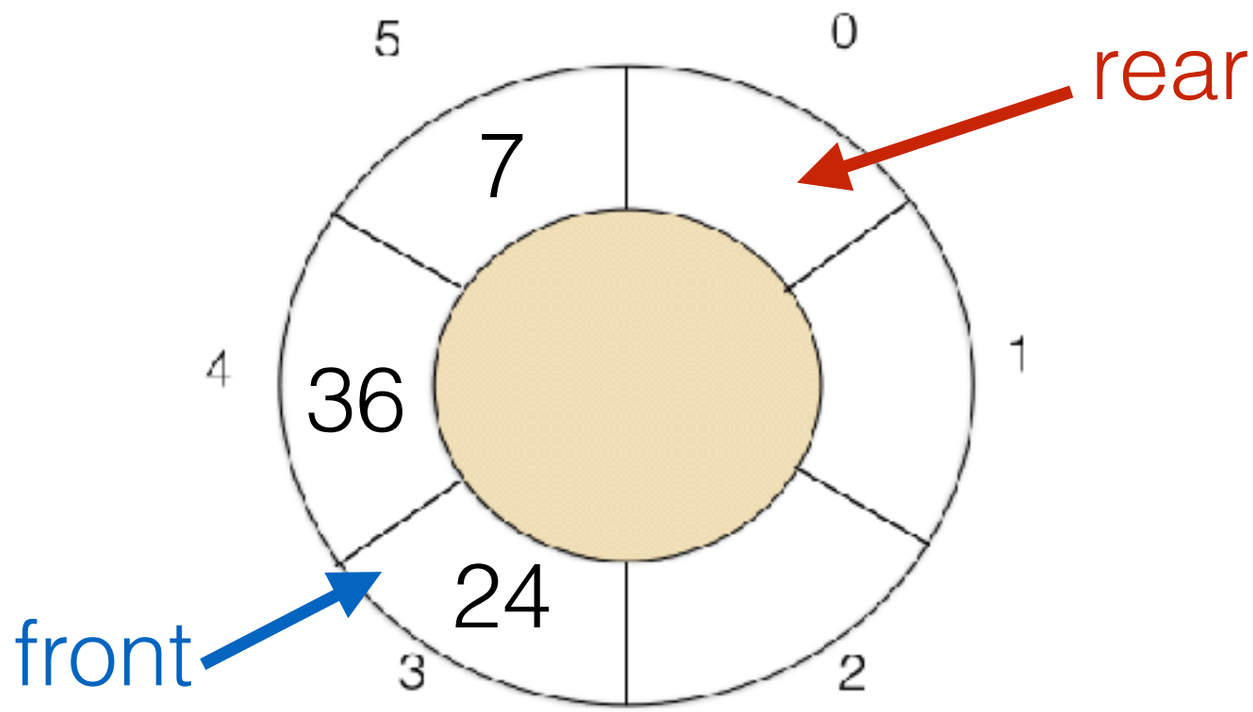

5
0
4
1
3
2

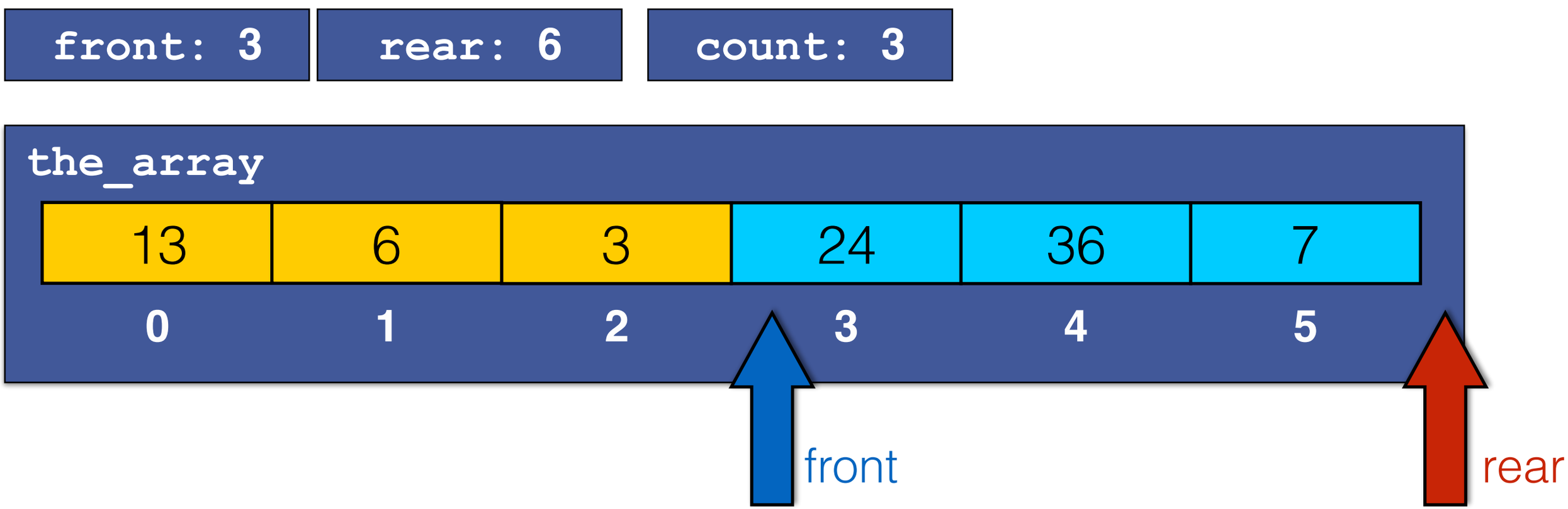

front: 3
rear: 6
count: 3
the_array
13
6
3
24
36
7
0
1
2
3
4
5
front
rear


5
0
7
4
36
1
24
3
2
front: 3
rear: 6
count: 3
the_array
13
6
3
24
36
7
0
1
2
3
4
5
front
rear

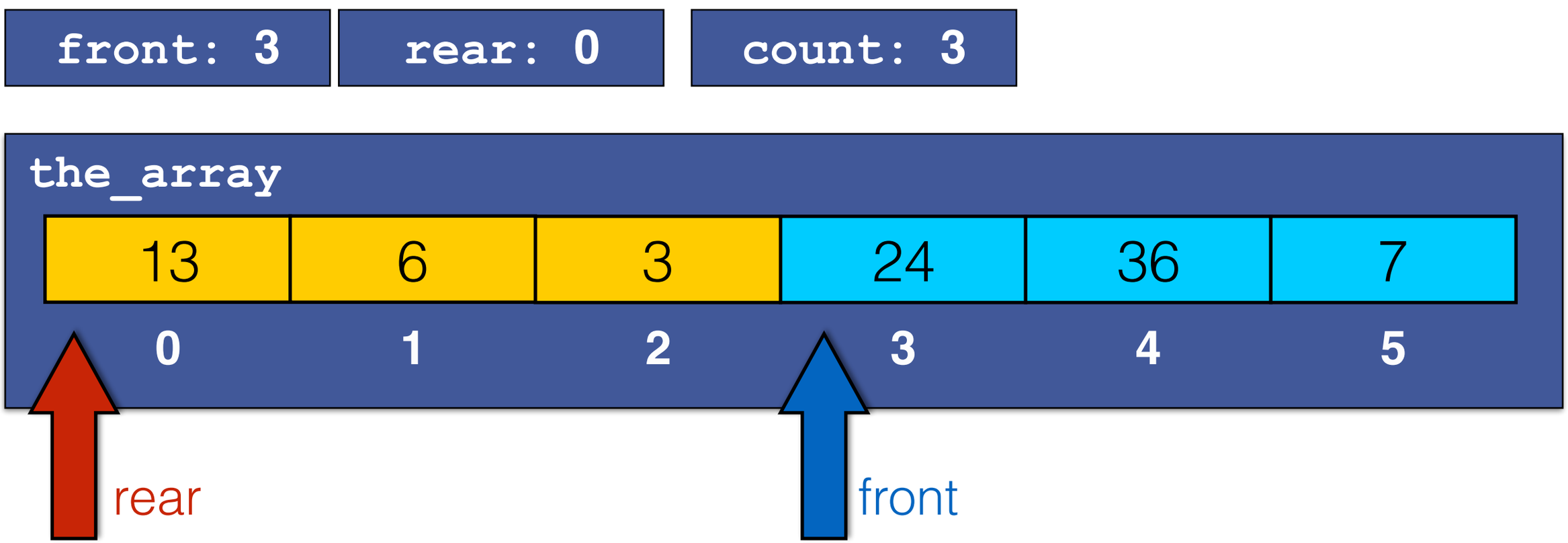

5
0
7
4
36
1
24
front
3
2


front: 3
rear: 6
count: 3
the_array
13
6
3
24
36
7
0
1
2
3
4
5
front
rear

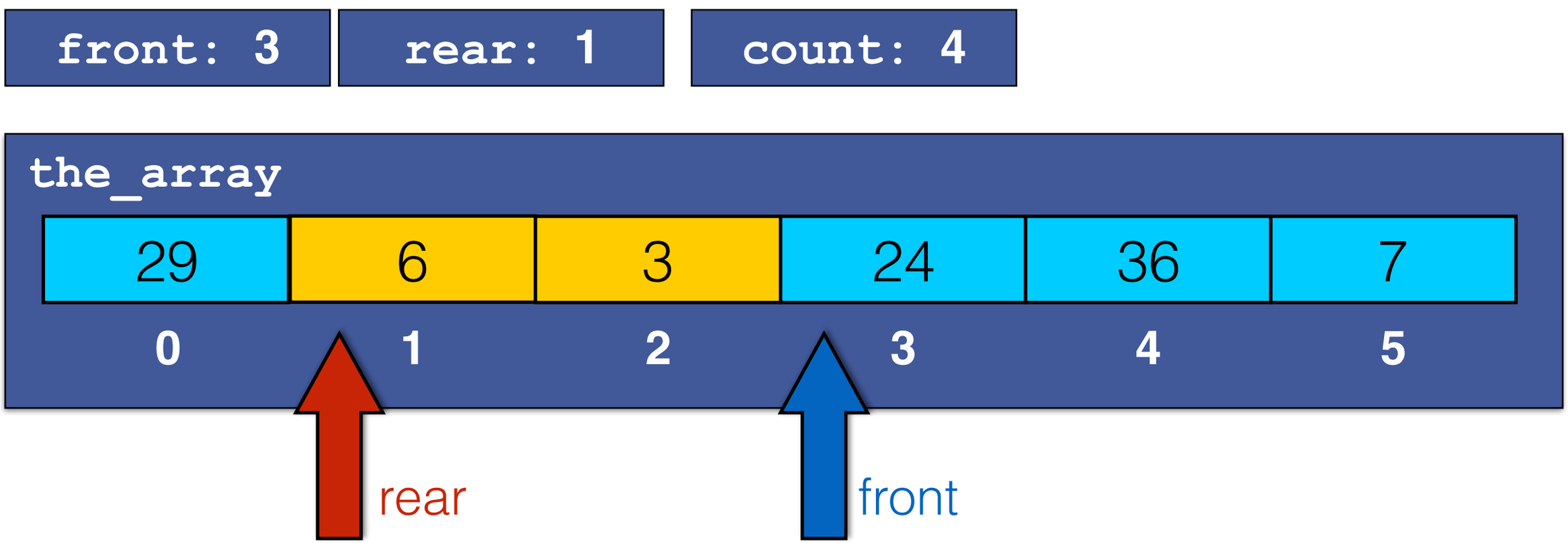

5
0
rear
7
4
36
1
24
front
3
2


front: 3
rear: 6
count: 3
the_array
13
6
3
24
36
7
0
1
2
3
4
5
front
rear

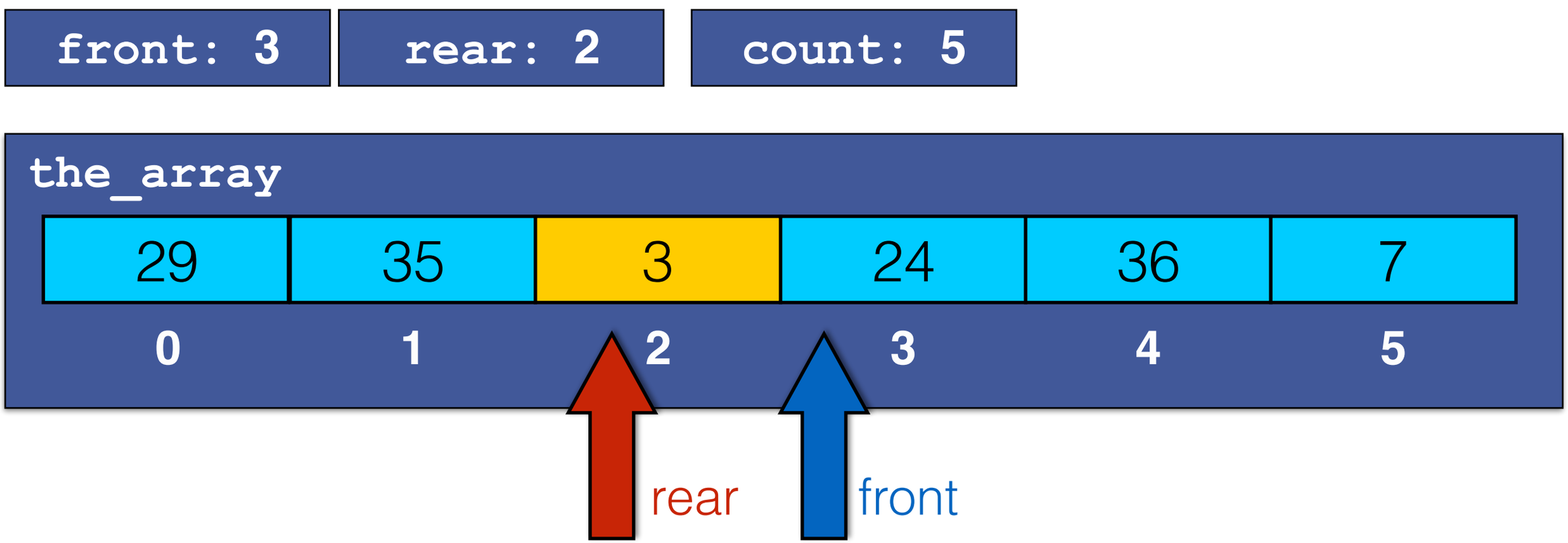
- After appending 7



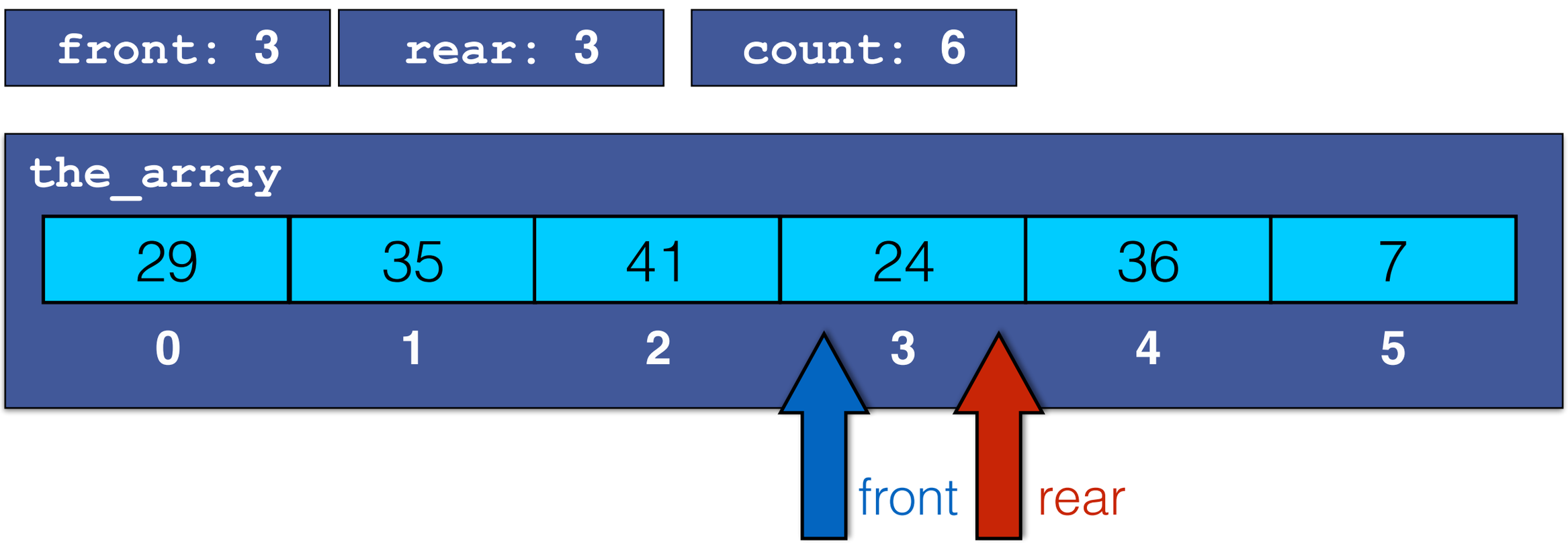
- After appending 7
- Append 29



- After appending 7
- Append 29
- Append 35

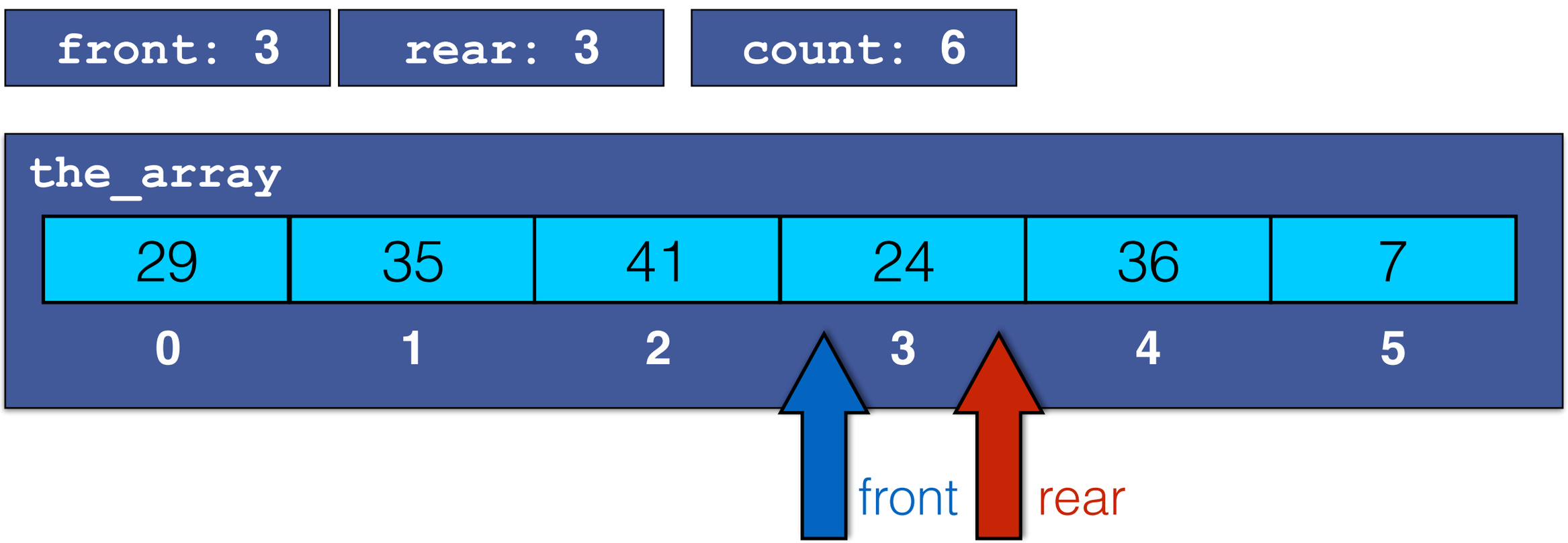


- After appending 7
- Append 29
- Append 35
- Append 41



- After appending 7
- Append 29
- Append 35
- Append 41

Important: Instead of using rear, to determine if the queue is full, we now need to use count.



Creating a circular Queue

```
def __init__(self, size):
    assert size > 0, "Size should be positive"
    self.the_array = size*[None]
    self.count = 0
    self.rear = 0
    self.front = 0
```

Creating a circular Queue

```
def __init__(self, size):
    assert size > 0, "Size should be positive"
    self.the_array = size*[None]
    self.count = 0
    self.rear = 0
    self.front = 0
```

Complexity is O(N)

Methods for Circular Queue

```
def is_empty(self):
    return self.count == 0
```

Methods for Circular Queue

```
def is_empty(self):
    return self.count == 0

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

Methods for Circular Queue

```
def is_empty(self):
    return self.count == 0

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Methods for Circular Queue

```
def is_empty(self):
    return self.count == 0

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Use **count,** not rear

Methods for Circular Queue

```
def is_empty(self):
    return self.count == 0

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Use **count,** not rear

Complexity is O(1)

Implementation of Append for a Circular Queue

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
```

Implementation of Append for a Circular Queue

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

Implementation of Append for a Circular Queue

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
```

Implementation of Append for a Circular Queue

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```

Implementation of Append for a Circular Queue

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```

Implementation of Append for a Circular Queue

```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```

If rear points outside of the_array but I know **the queue is not full**

You know that *len(self.the_array) = 6* and *self.rear = 5*

(self.rear + 1) **%** *len(self.the_array)* is equal to…

(self.rear + 1) **%** *len(self.the_array)*

(5 + 1) **%** *6*

6 **%** *6 = 0*

You know that *len(self.the_array) = 6* and *self.rear = 4*

(self.rear + 1) **%** *len(self.the_array)* is equal to…

(self.rear + 1) **%** *len(self.the_array)*

(4 + 1) **%** *6*

5 **%** *6 = 5*

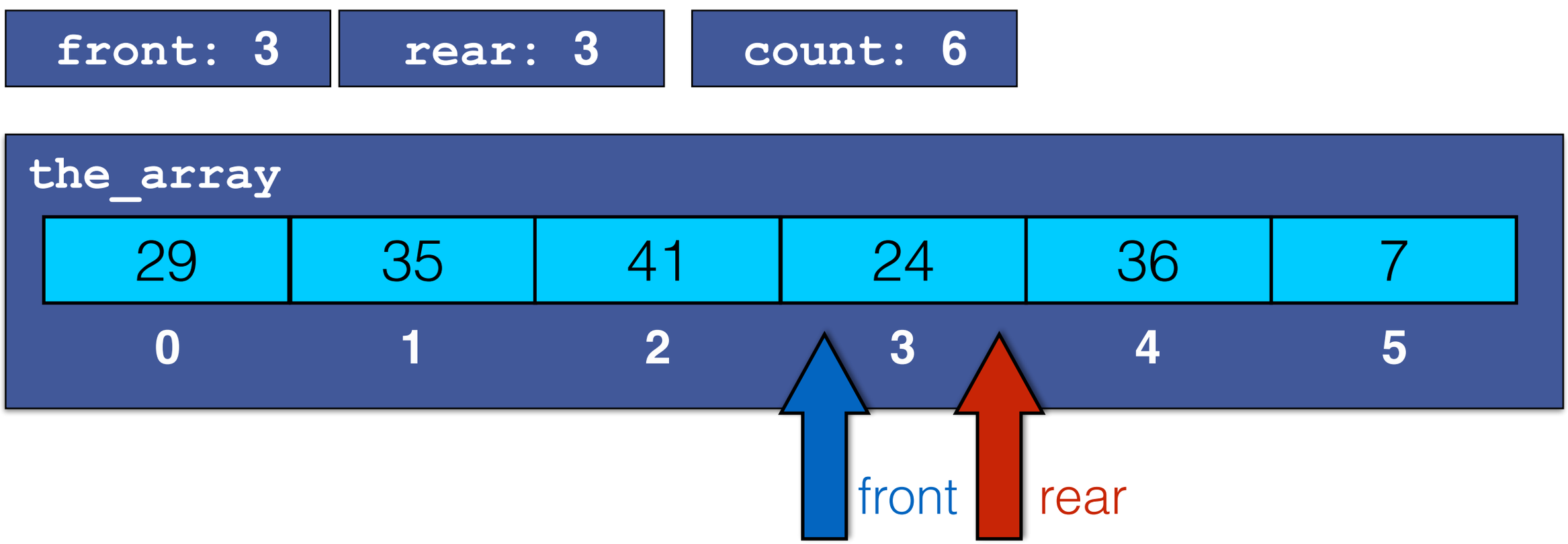
```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear = (self.rear+1)% len(self.the_array)
    self.count += 1
```

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

```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```

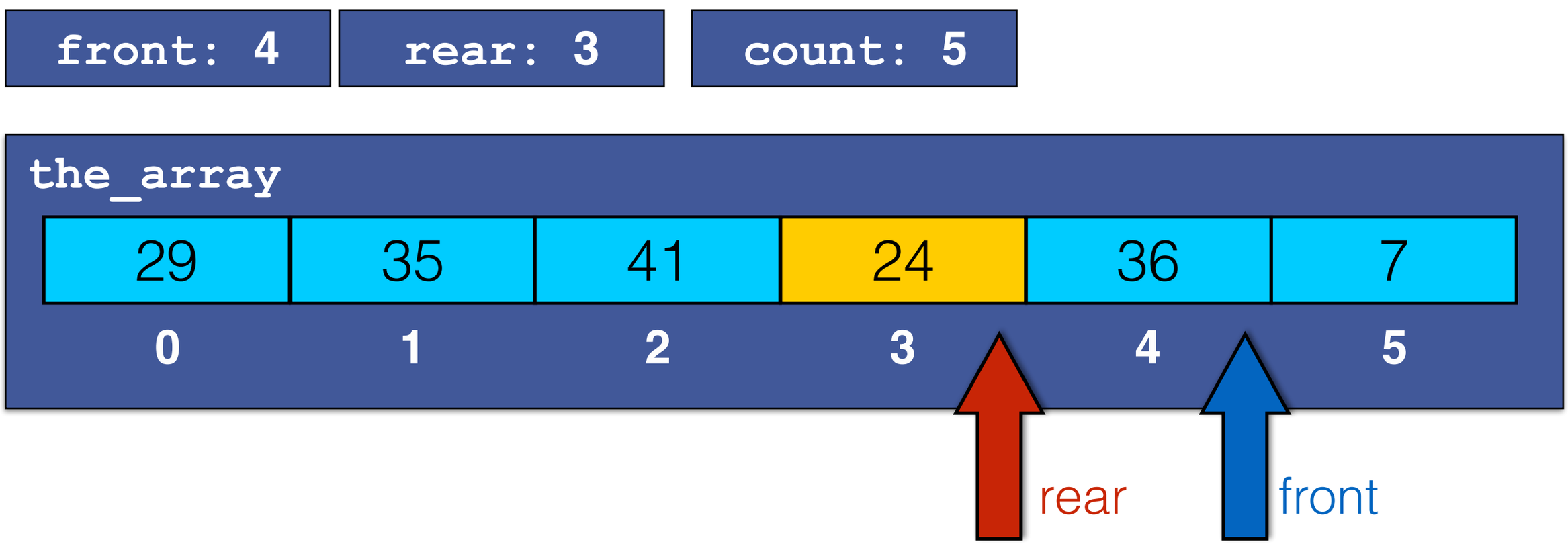
```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear = (self.rear+1)% len(self.the_array)
    self.count += 1
```

Circular queue: both front and rear wrap



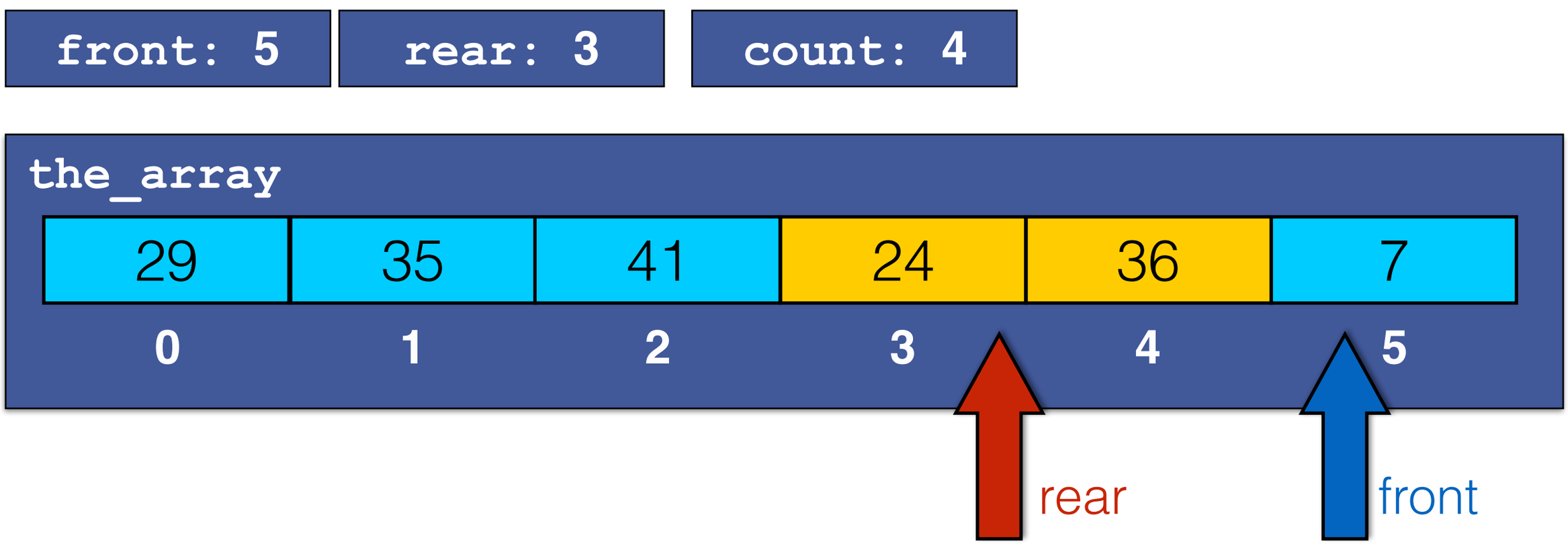
Circular queue: both front and rear wrap

- Serve item (returns 24)



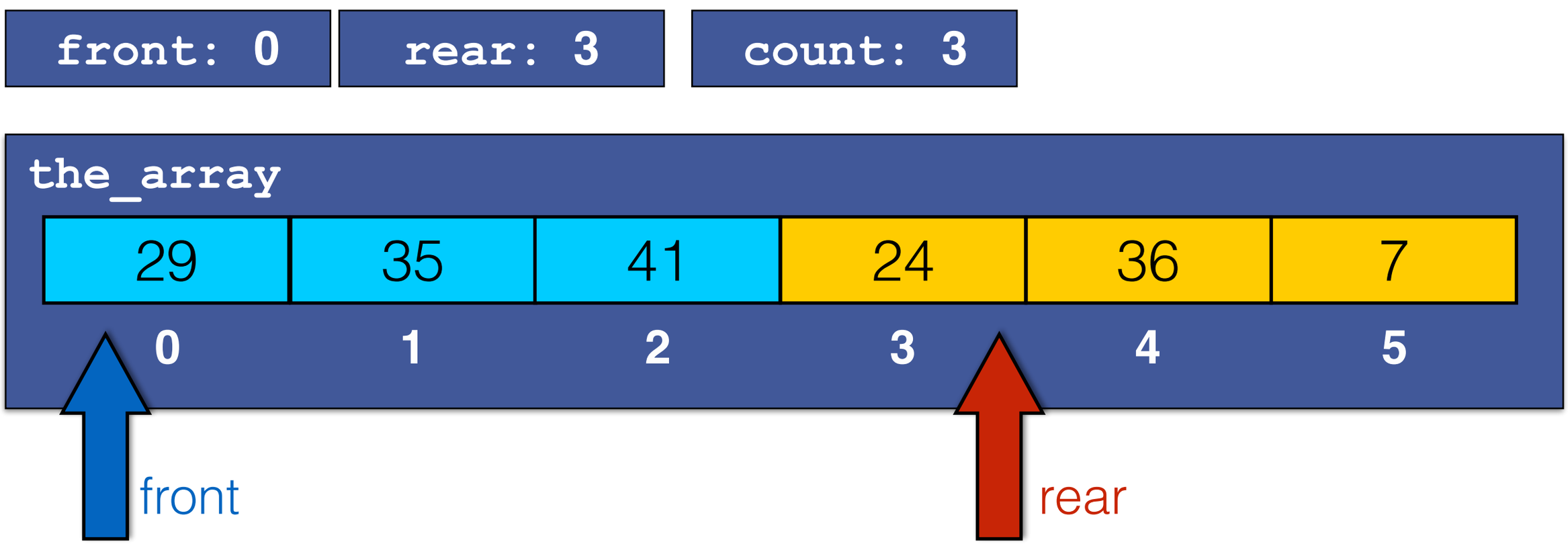
Circular queue: both front and rear wrap

- 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 of Serve for a Circular Queue

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

Implementation of Serve for a Circular Queue

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

Implementation of Serve for a Circular Queue

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

```
self.front +=1
if self.front == len(self.the_array):
    self.front = 0
```

Print Queue

- 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

Print Queue

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

Print Queue

Anonymous variable

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

Print Queue

Anonymous variable

print as many items as available in the queue

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

Print Queue

Anonymous variable

print as many items as available in the queue

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

Convert to string whatever is stored

Print Queue

Anonymous variable

print as many items as available in the queue

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

Convert to string whatever is stored

Increase index or make it zero if it points to outside of the_array

Some Queue Applications

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

Summary

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