CSCI 132:
Basic Data Structures and Algorithms

Queues (Array implementation)

Reese Pearsall & Iliana Castillon

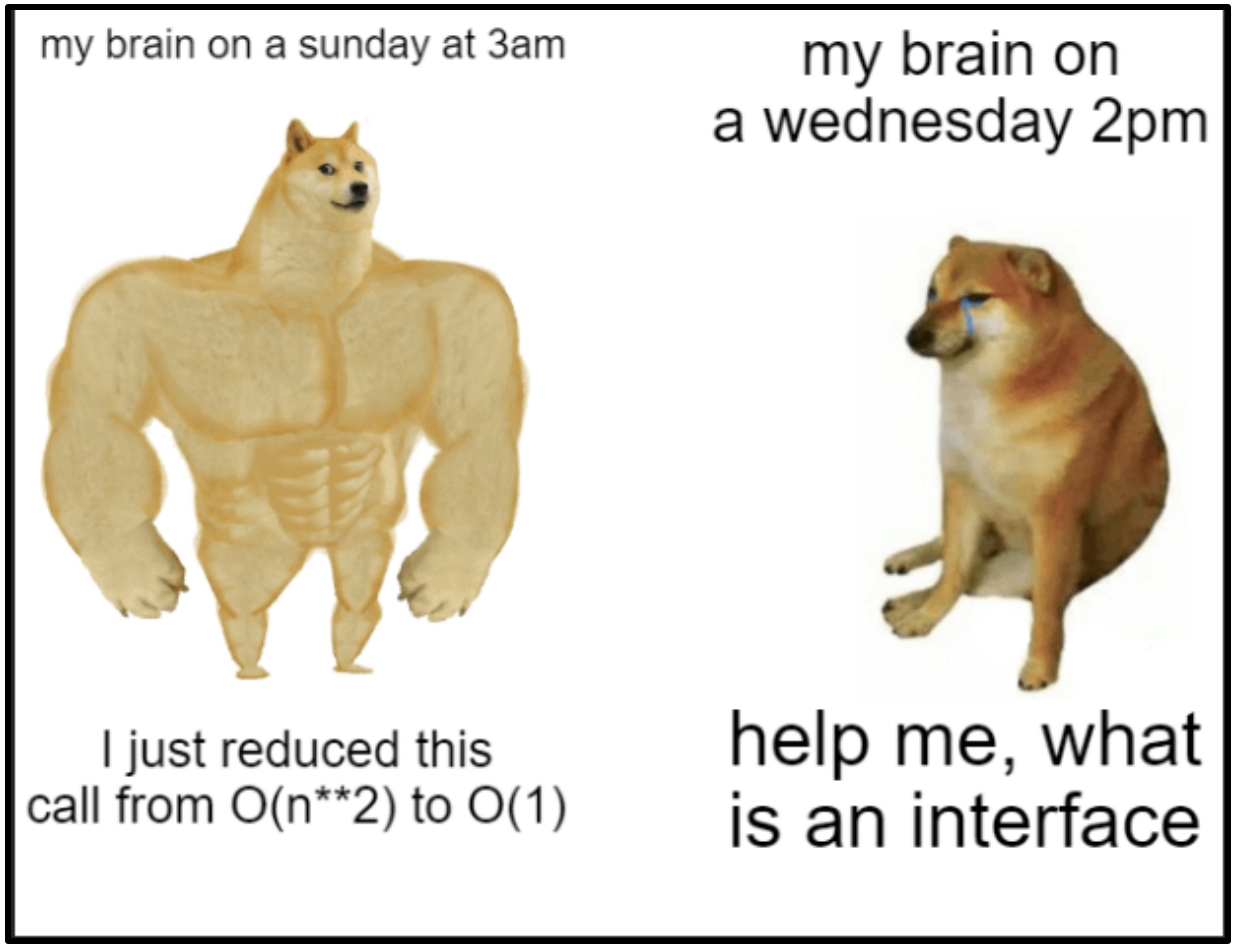
Fall 2024

Announcements

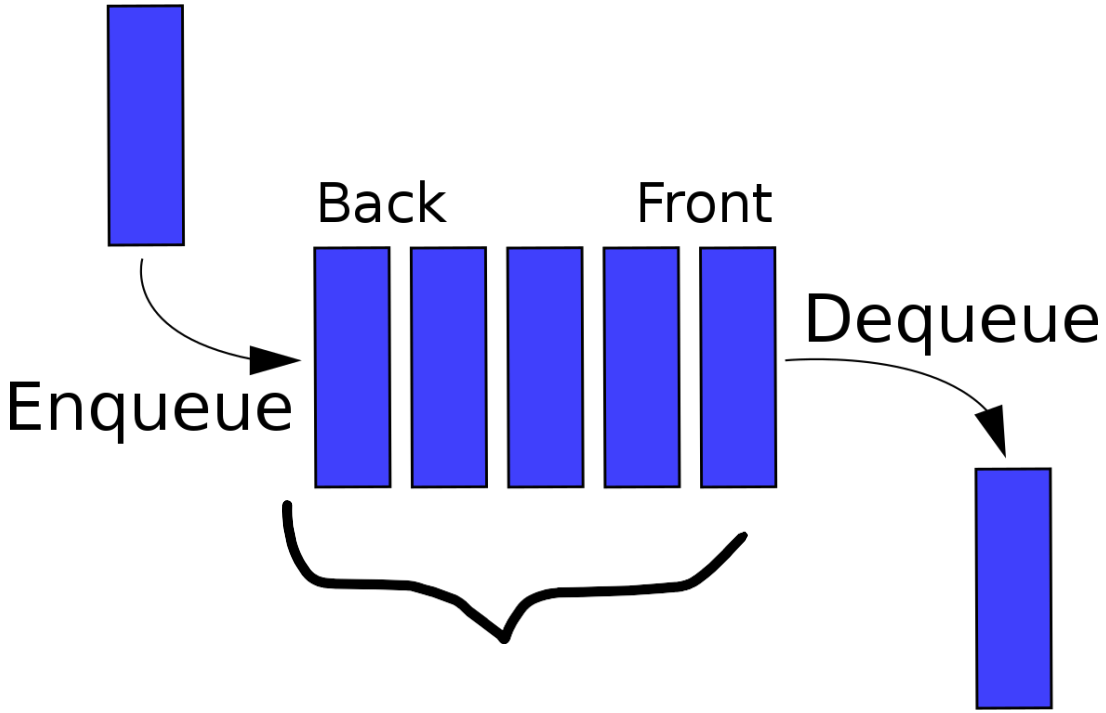
Lab 8 due **tomorrow** at 11:59pm

No lecture on Friday → Program 3 workday

Program 3 due next Friday (Nov. 1st)



A Queue is a data structure that holds data, but operates in a First-in First-out (FIFO) fashion



Once again, we need a data structure to hold the data of the queue

- Linked List
- Array

Elements get added to the **`Back`** of the Queue.

Elements get removed from the **`Front`** of the queue

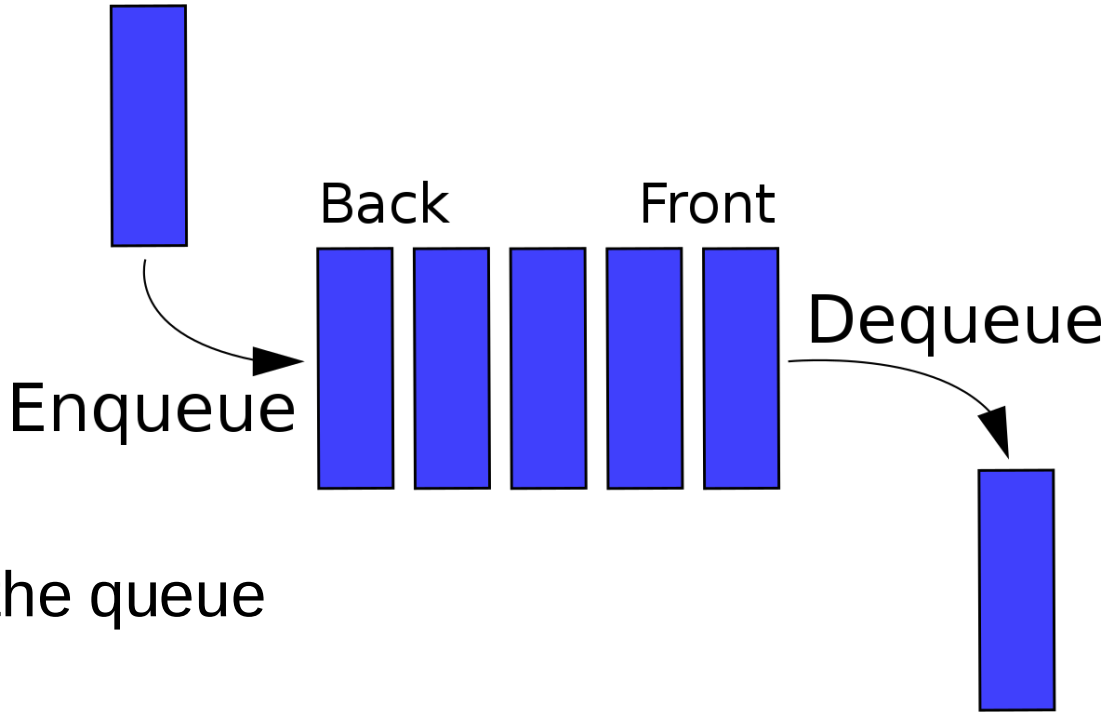
A Queue is a data structure that holds data, but operates in a First-in First-out (FIFO) fashion

The Queue ADT has the following methods:

Enqueue- Add new element to the queue

Dequeue- Remove element from the queue

** Always remove the front-most element

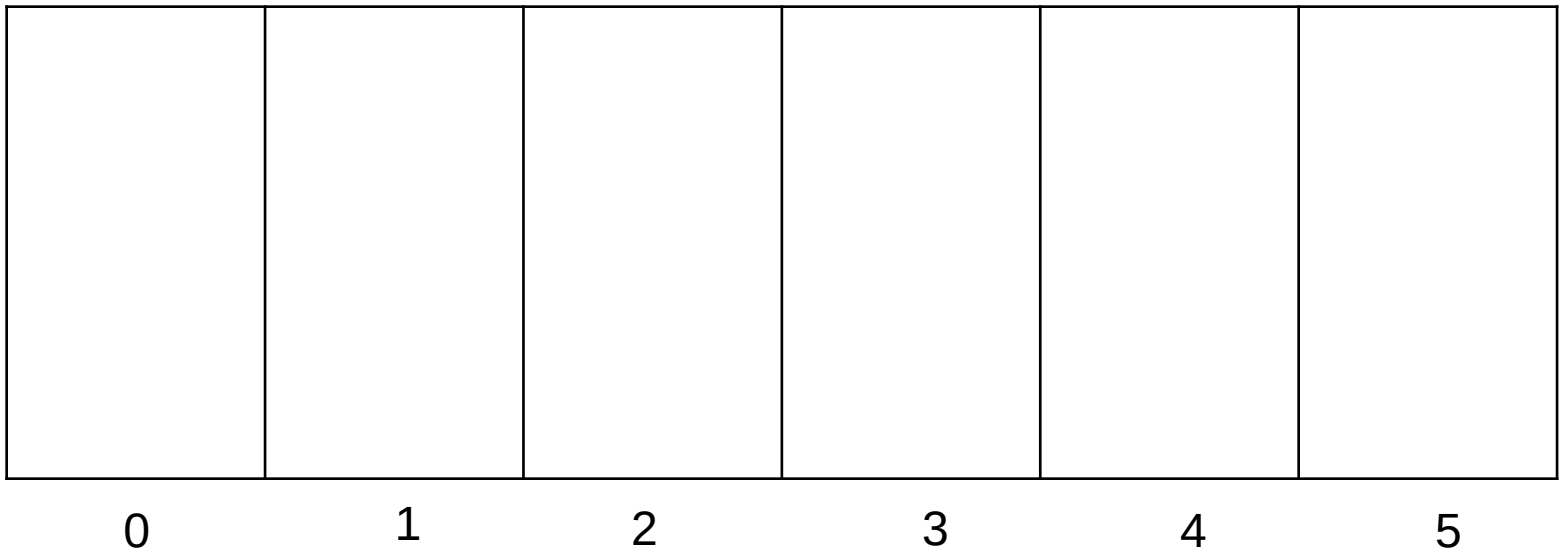


Peek()- Return the element that is at the front of the queue

IsEmpty() – Returns true if queue is empty, returns false is queue is not empty

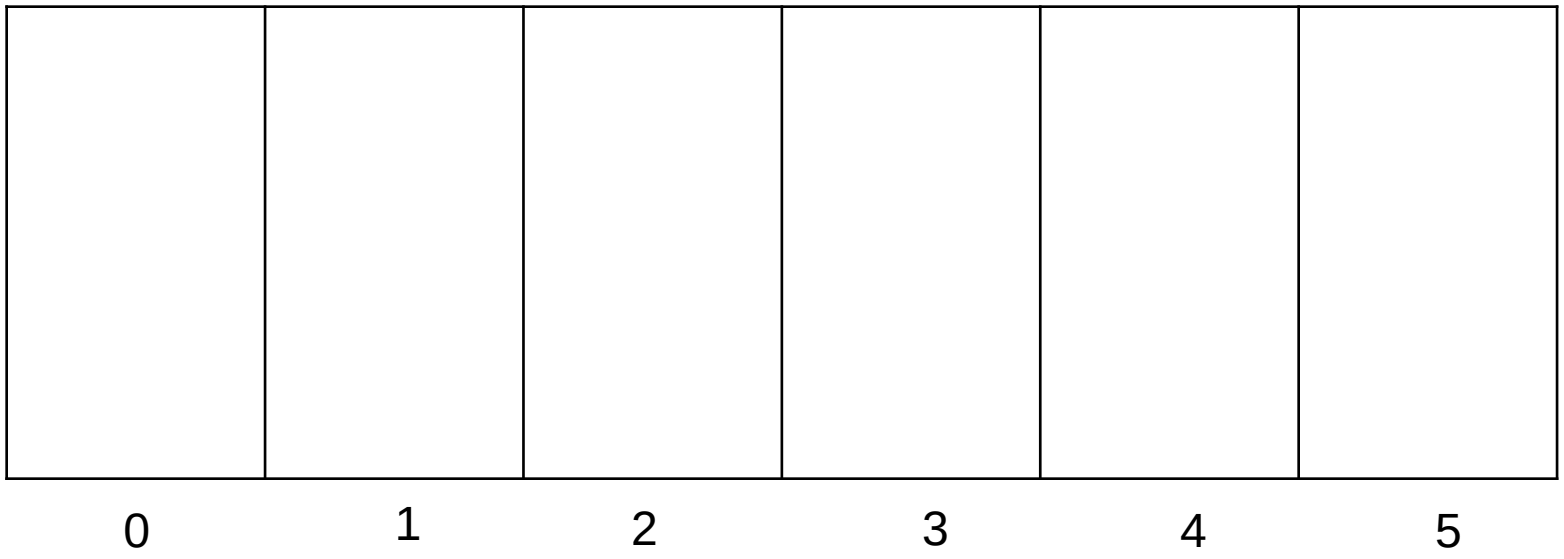
Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

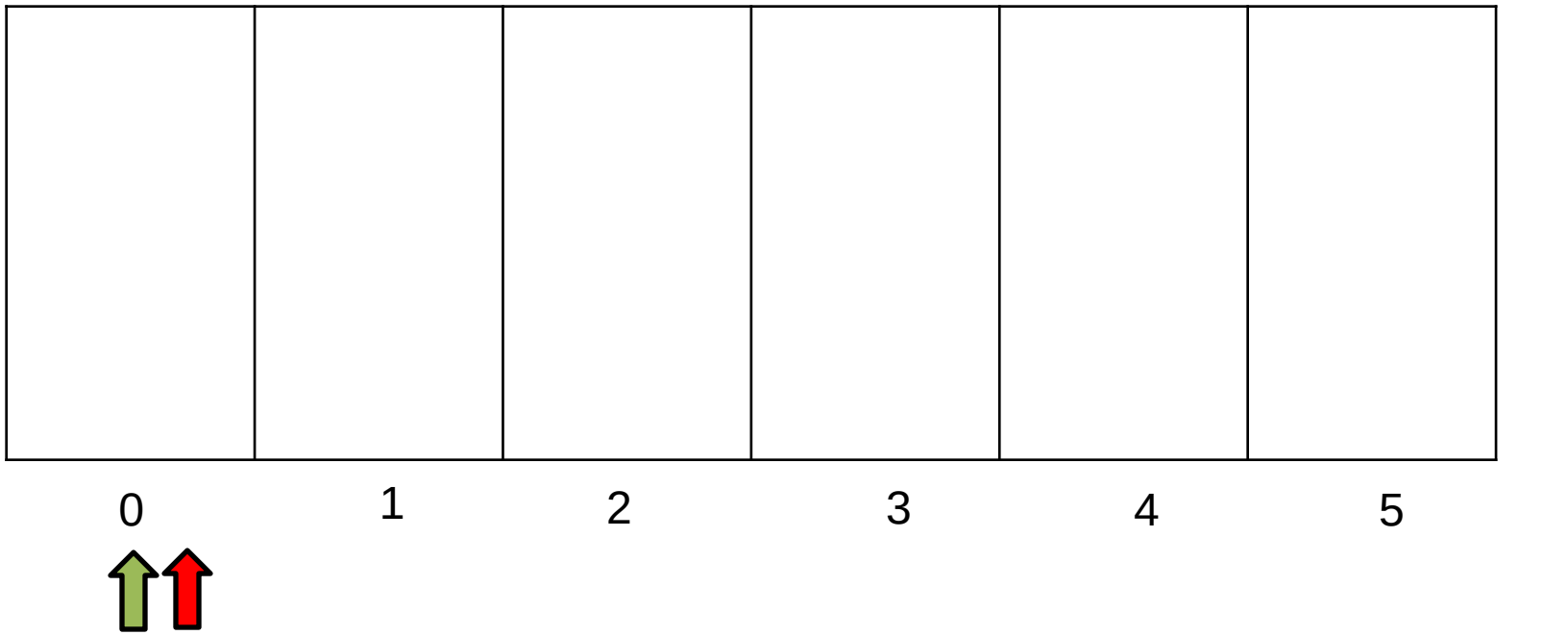


We need to keep track of a few things:

1. The index of the **front** of the queue
2. The index of the rear of the queue
3. The size of the queue
4. The capacity of the queue

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



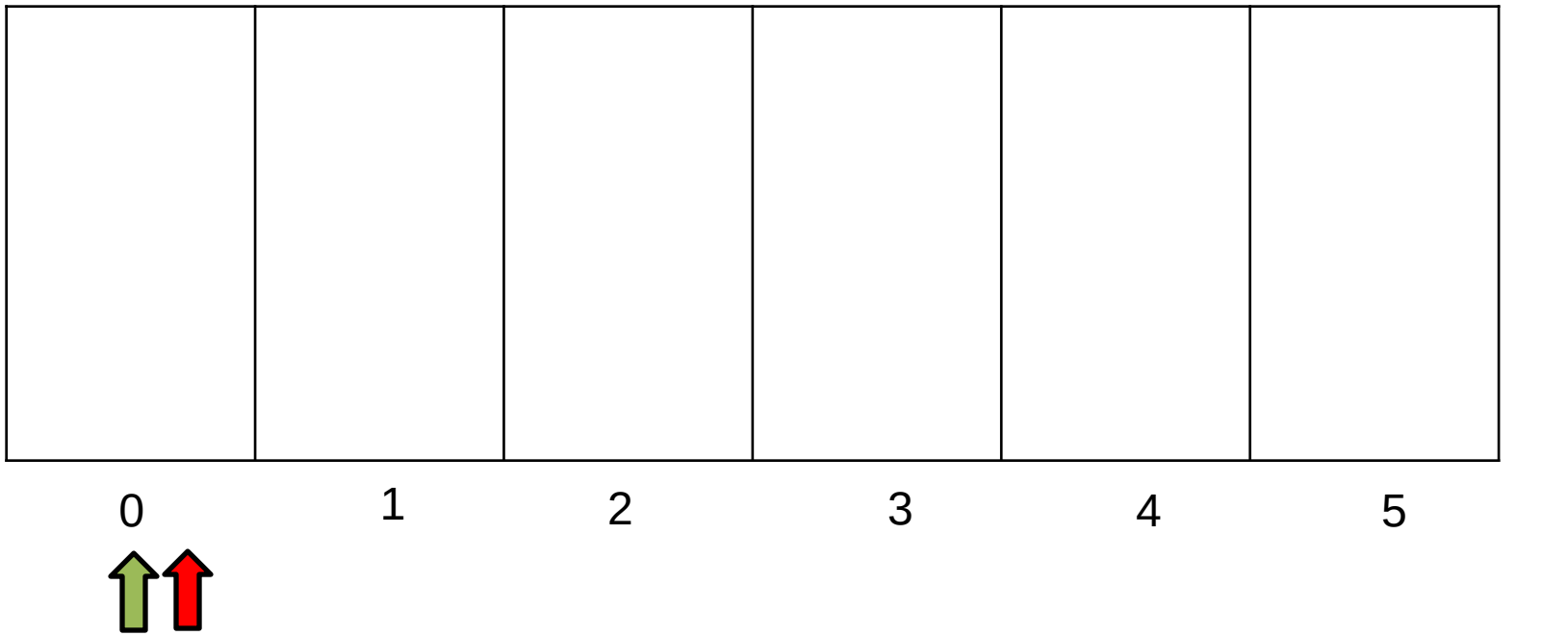

```
capacity = 6    front = 0
size = 0        rear = 0
```

We need to keep track of a few things:

1. The index of the **front** of the queue
2. The index of the rear of the queue
3. The size of the queue
4. The capacity of the queue

Today, we will be implementing a Queue with an Array.

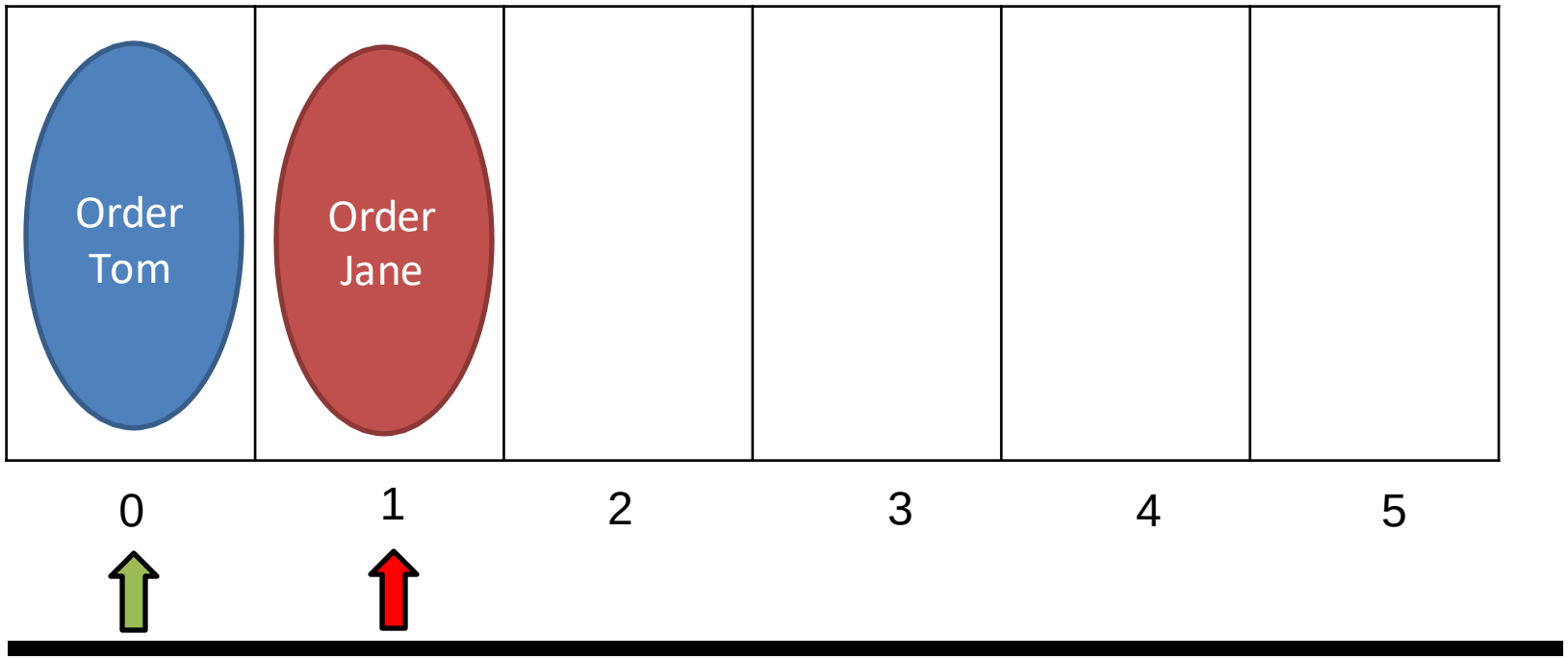
Suppose that we have a queue that can hold 6 elements




```
capacity = 6    front = 0
size = 0        rear = 0
```

Today, we will be implementing a Queue with an Array.

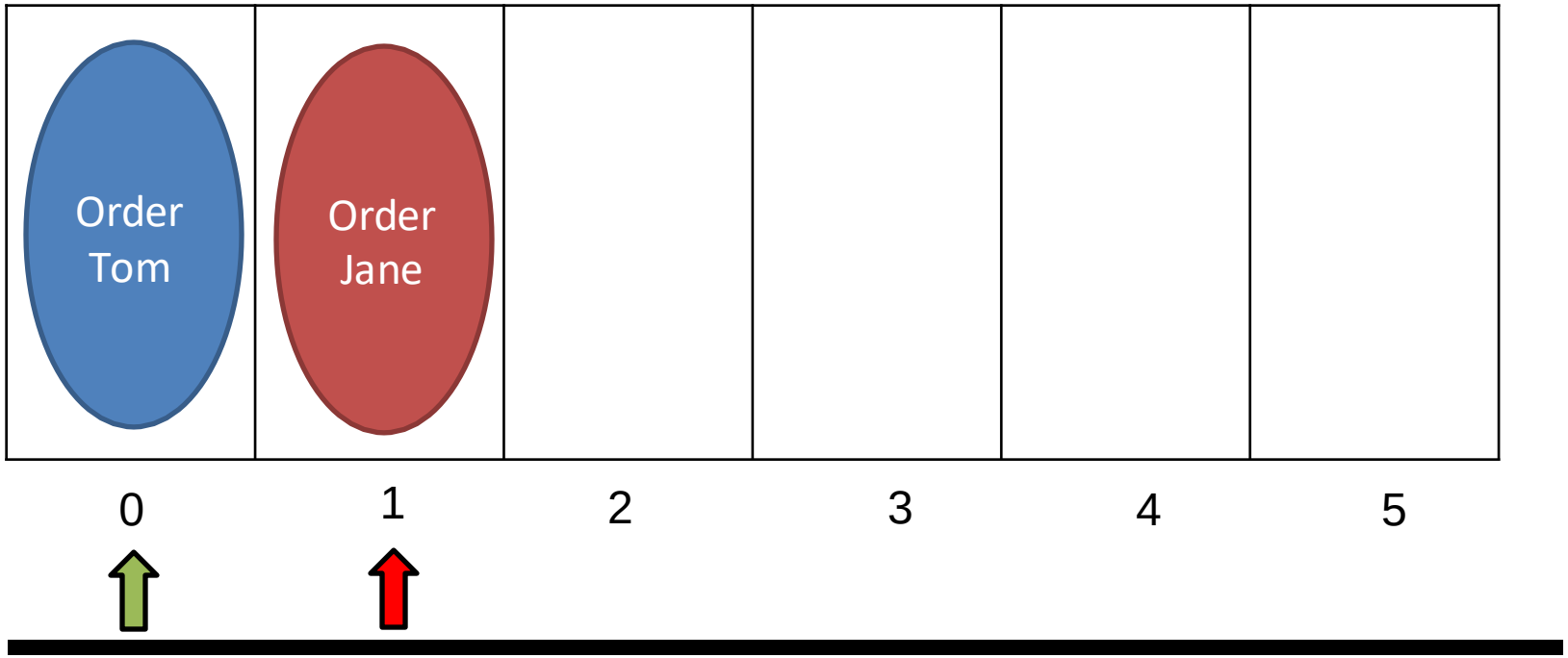
Suppose that we have a queue that can hold 6 elements




```
capacity = 6    front = 0
size = 2        rear = 1
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

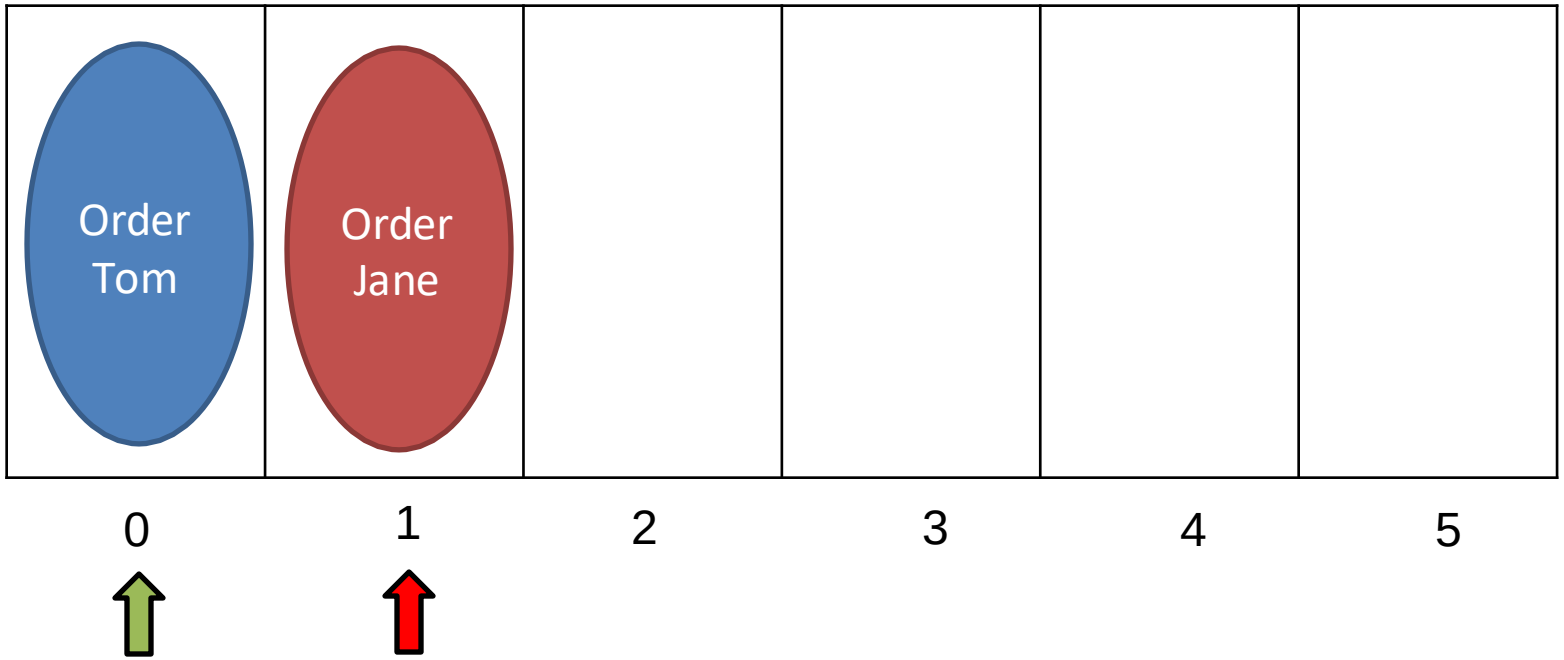


Enqueue?

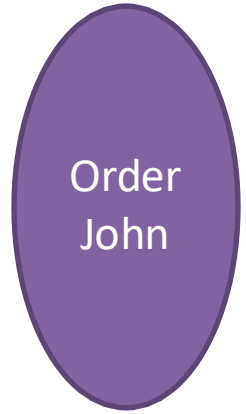
```
capacity = 6    front = 0
size = 2        rear = 1
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



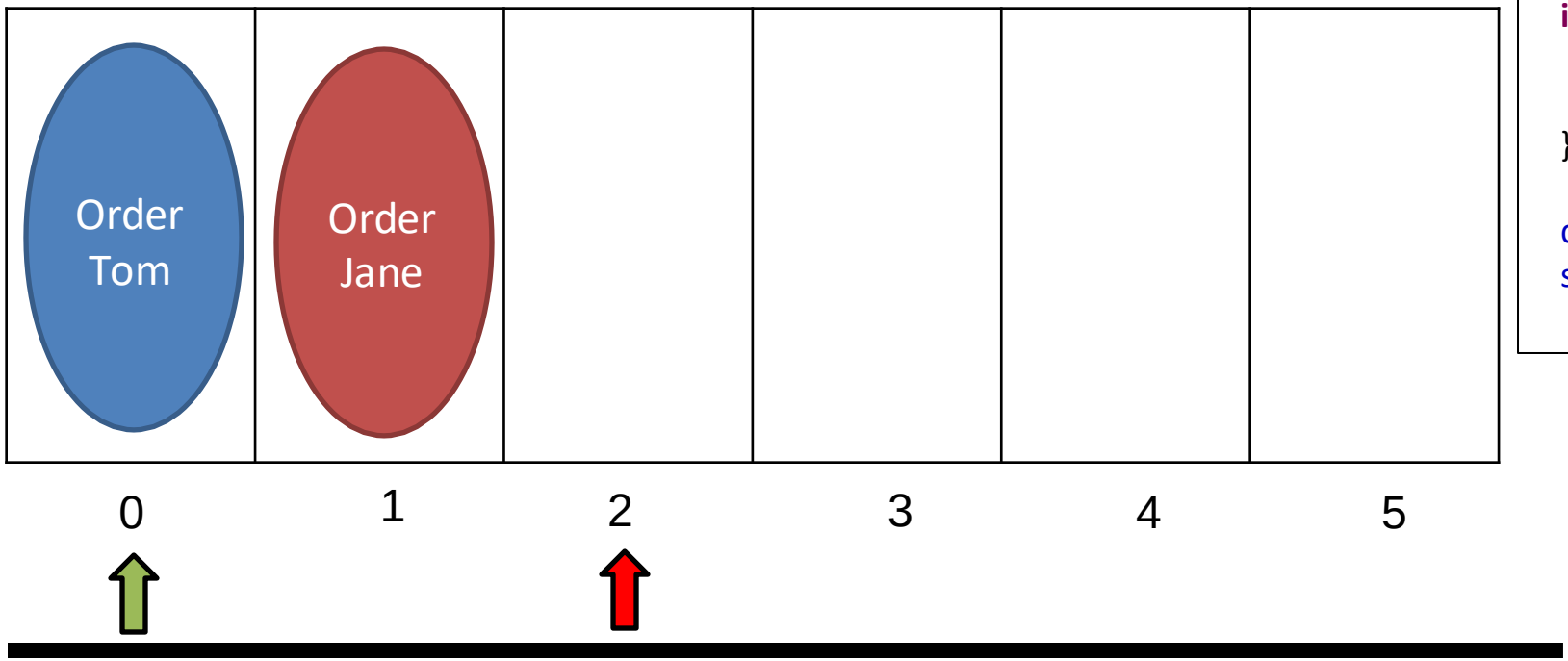

```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```



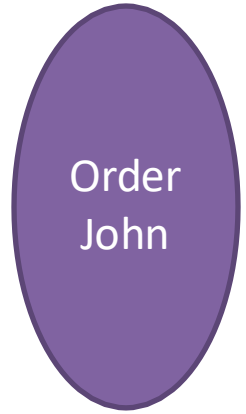

```
capacity = 6    front = 0
size = 2        rear = 1
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



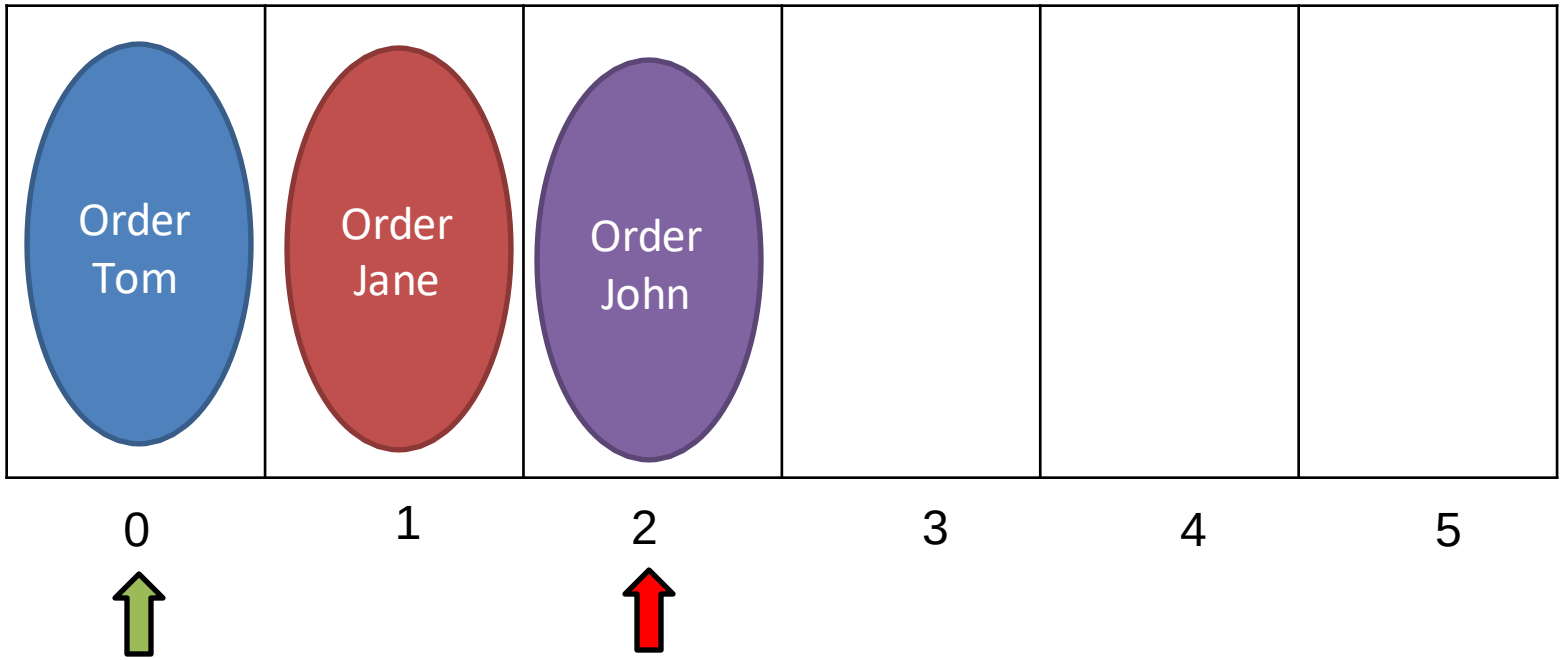

```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```




```
capacity = 6    front = 0
size = 2        rear = 2
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

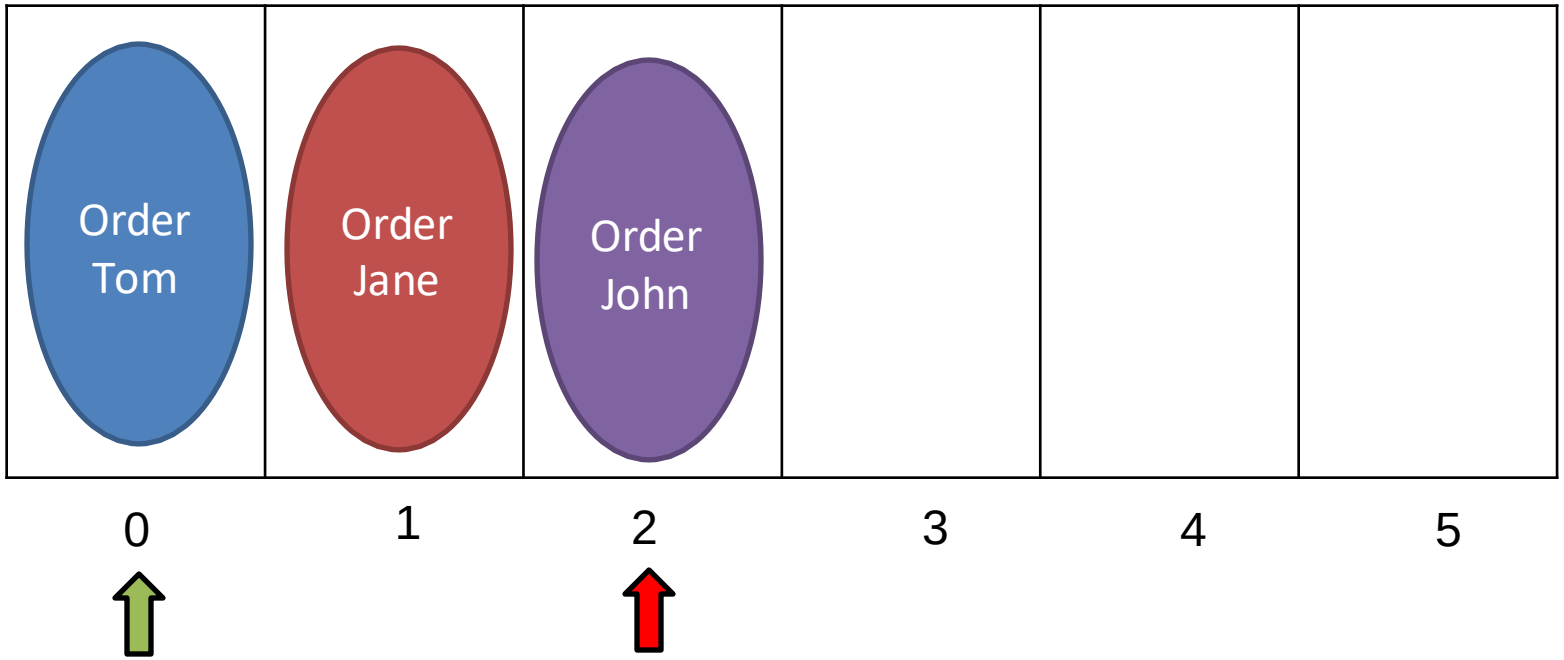



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

```
capacity = 6    front = 0
size = 2        rear = 2
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

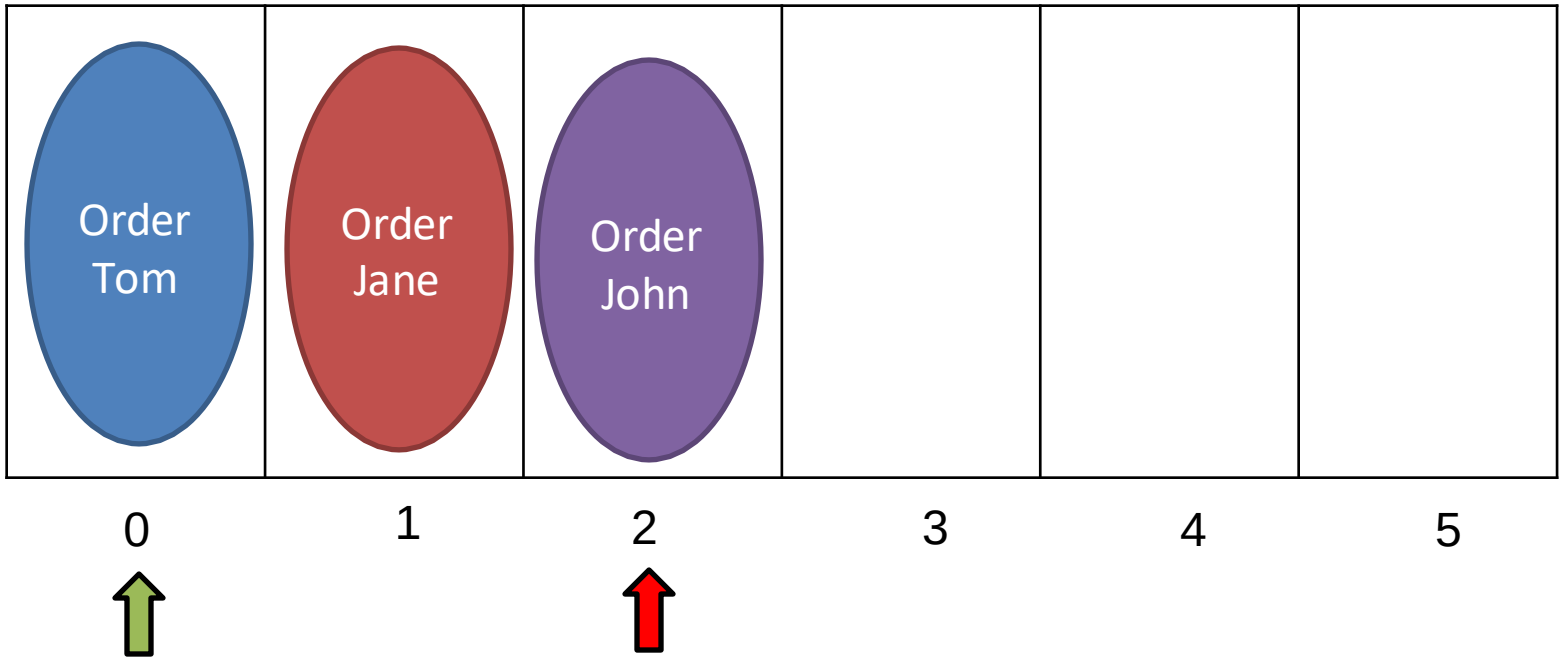



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

```
capacity = 6    front = 0
size = 3        rear = 2
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

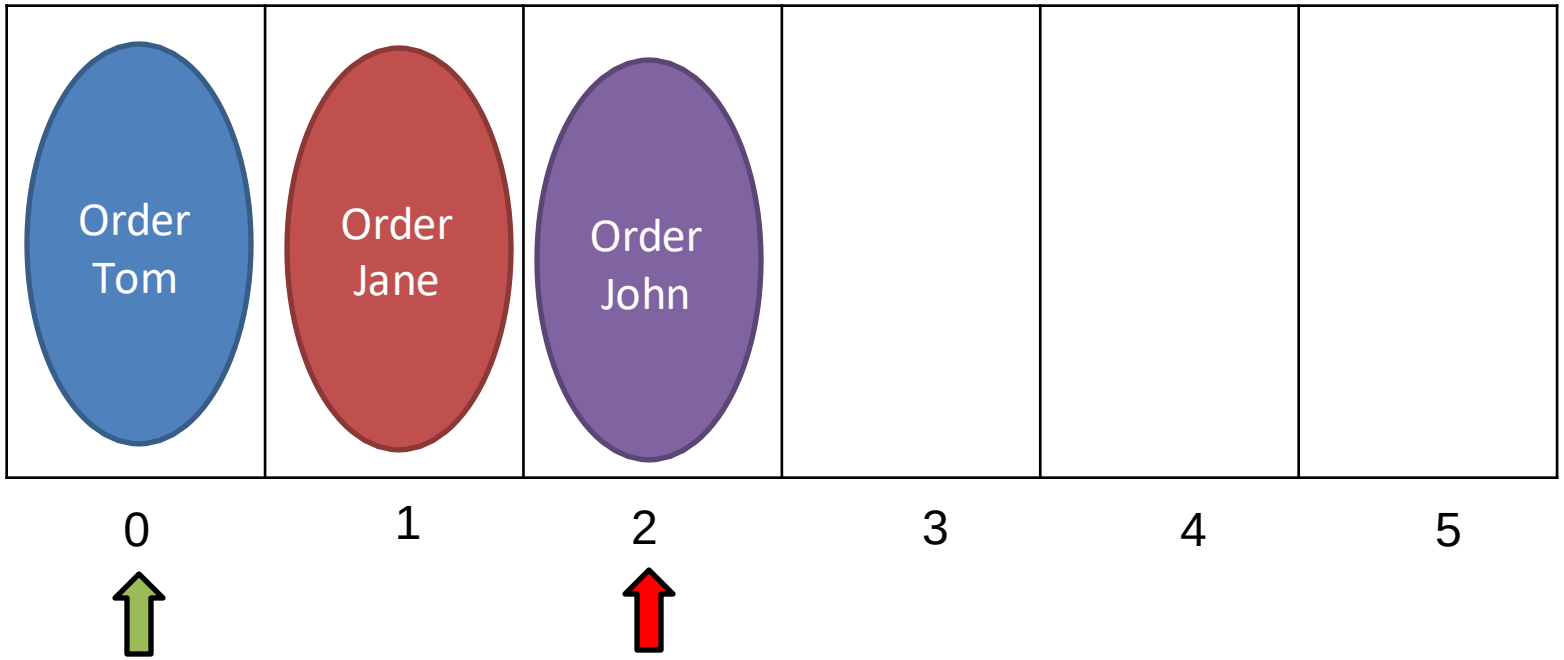



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

```
capacity = 6    front = 0
size = 3        rear = 2
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

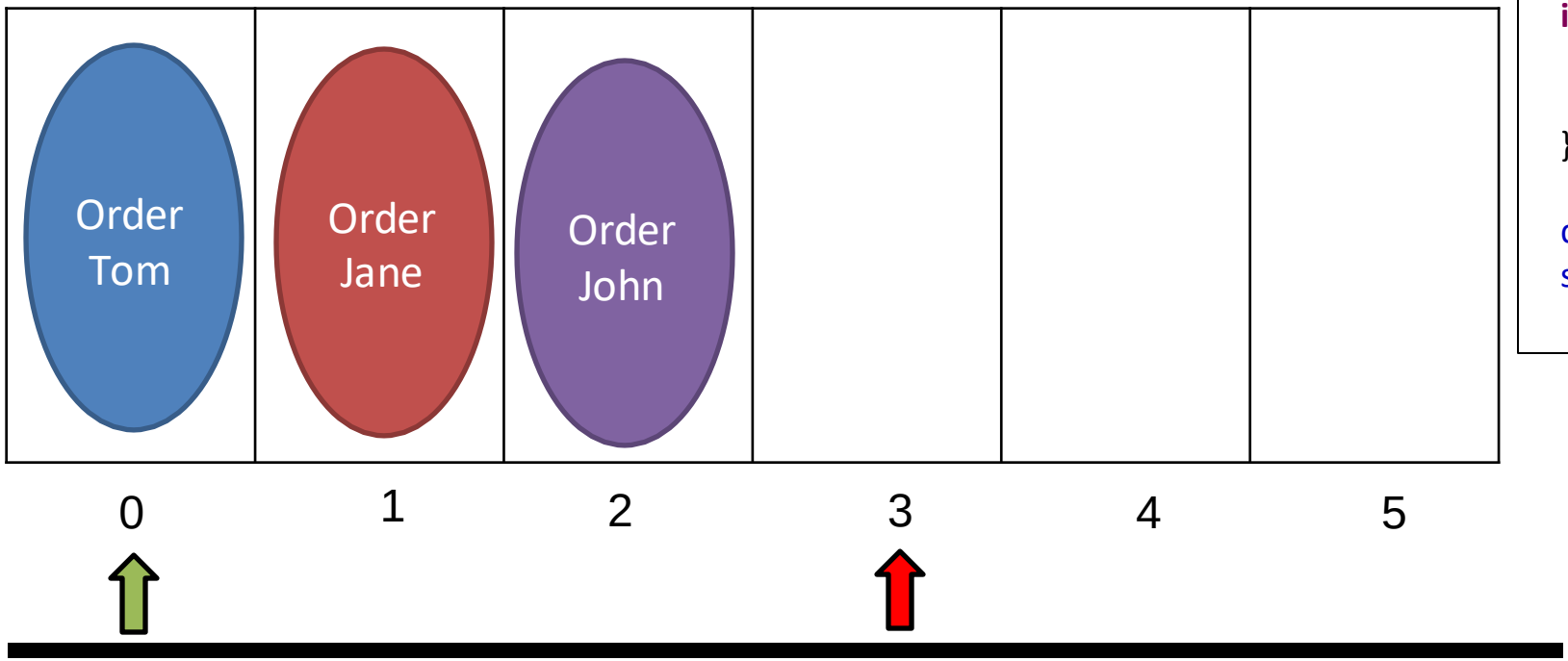



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```


```
capacity = 6    front = 0
size = 3        rear = 2
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

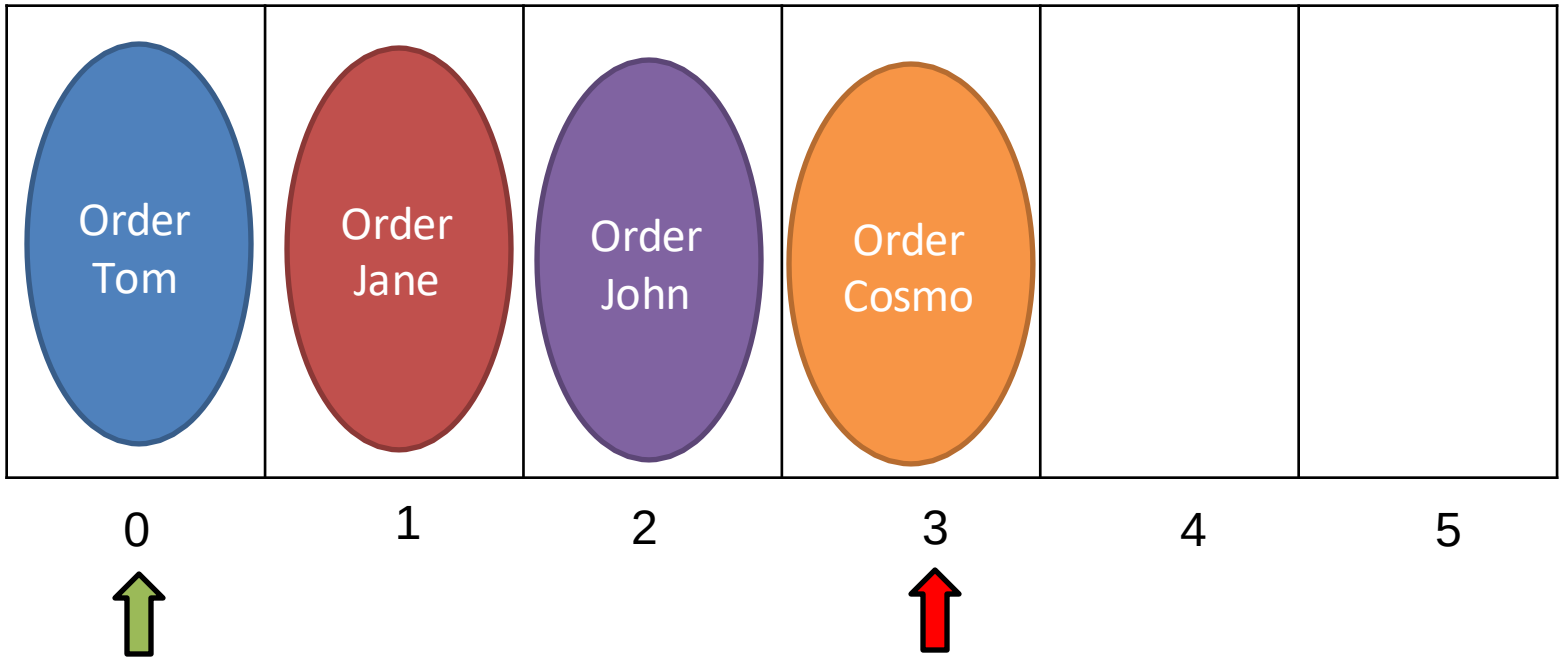



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```


```
capacity = 6    front = 0
size = 3        rear = 3
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

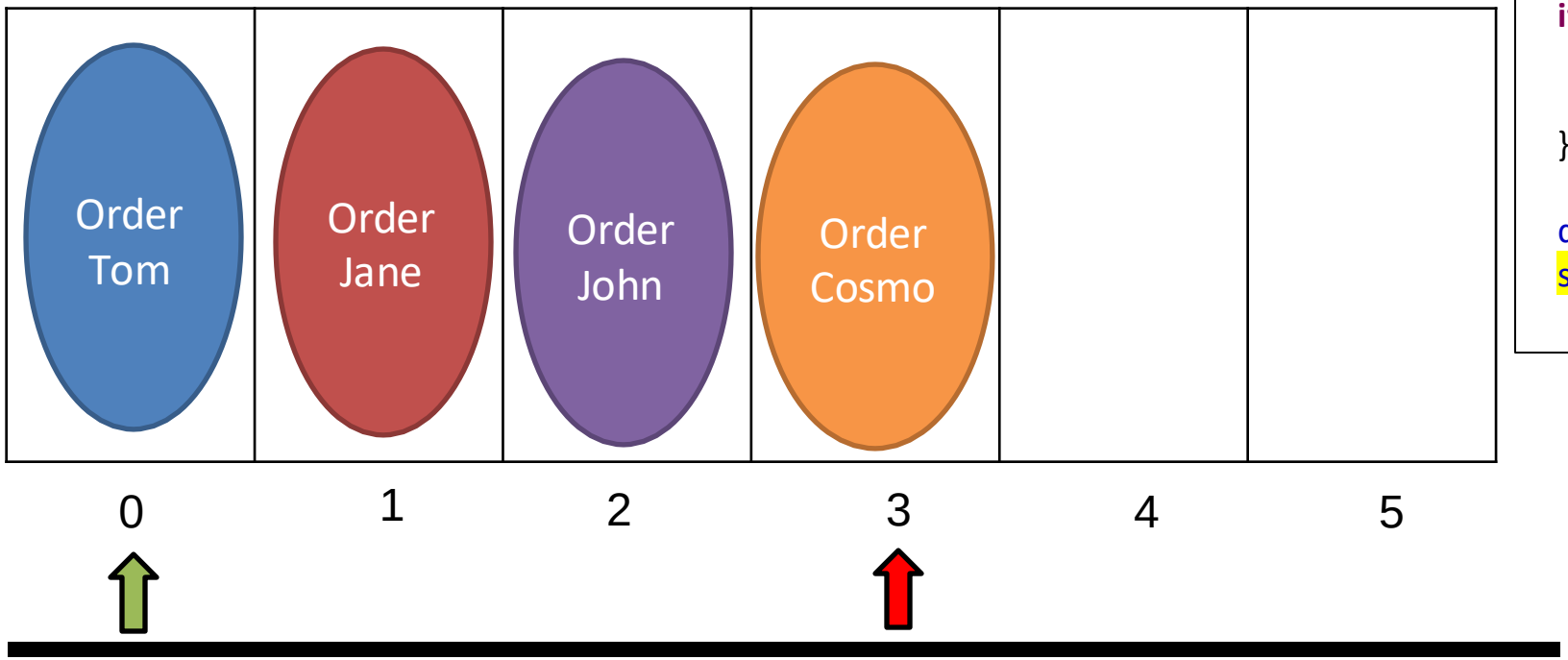



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

```
capacity = 6    front = 0
size = 3        rear = 3
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



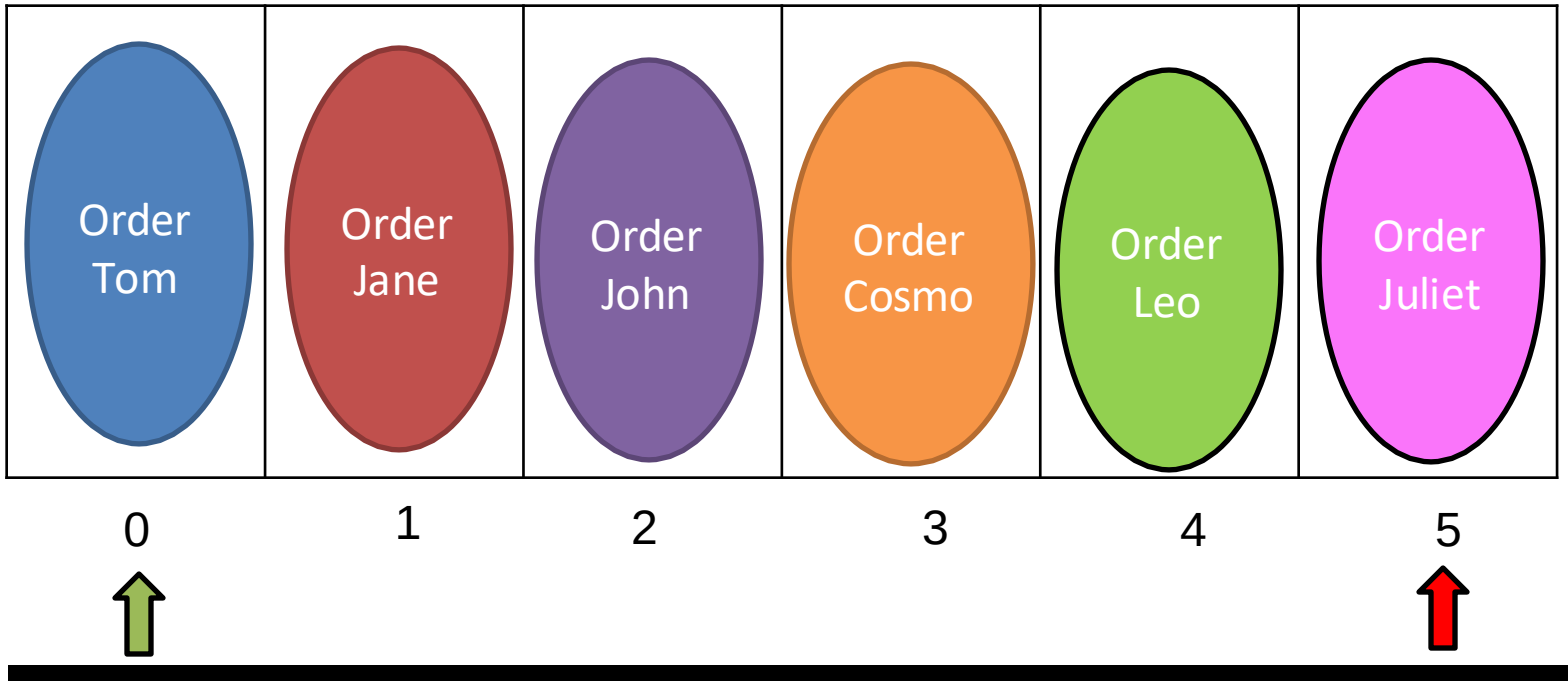

```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

capacity = 6 front = 0

size = 4 rear = 3

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

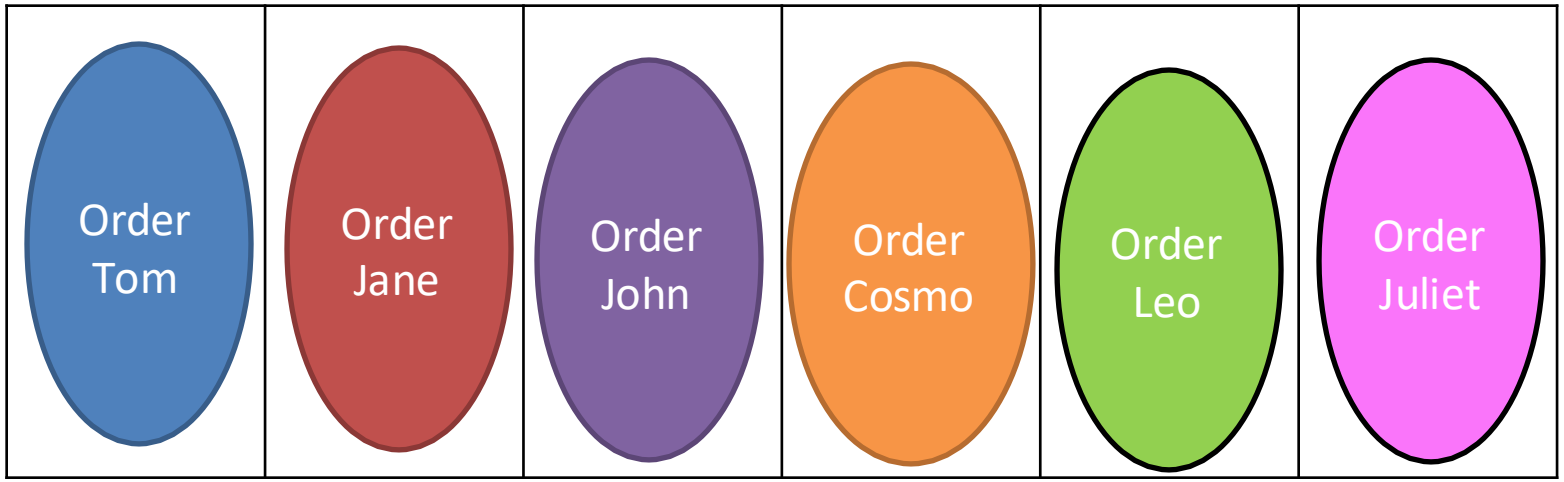



```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

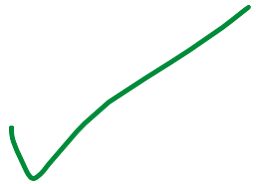
```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



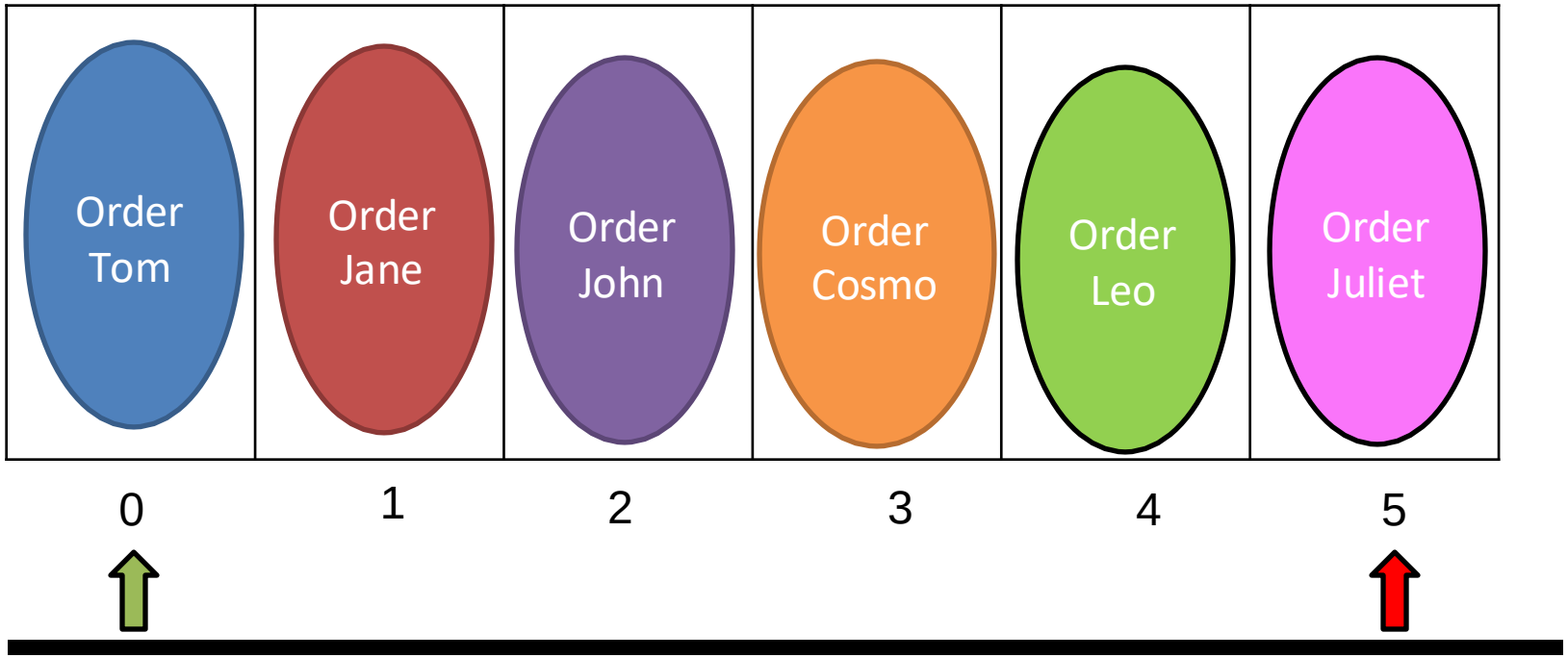

```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```



```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

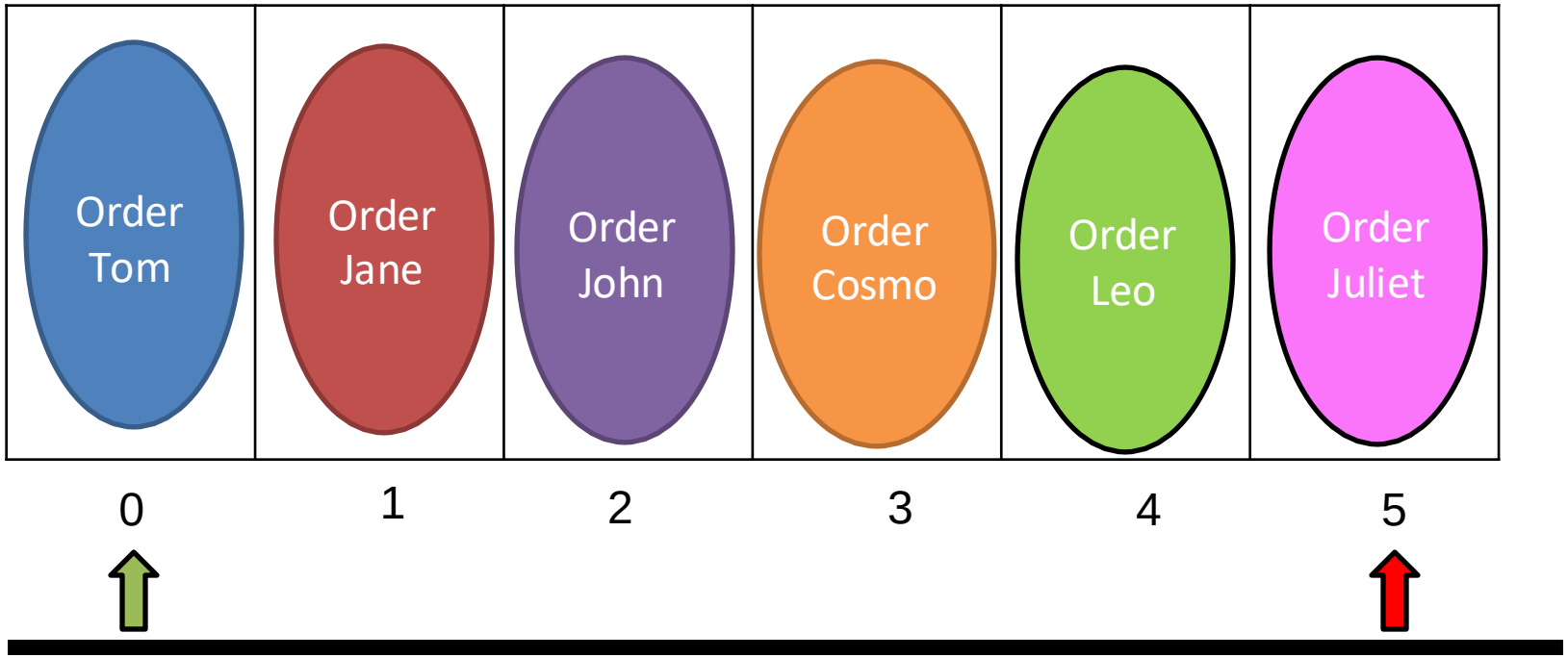


Dequeue?

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

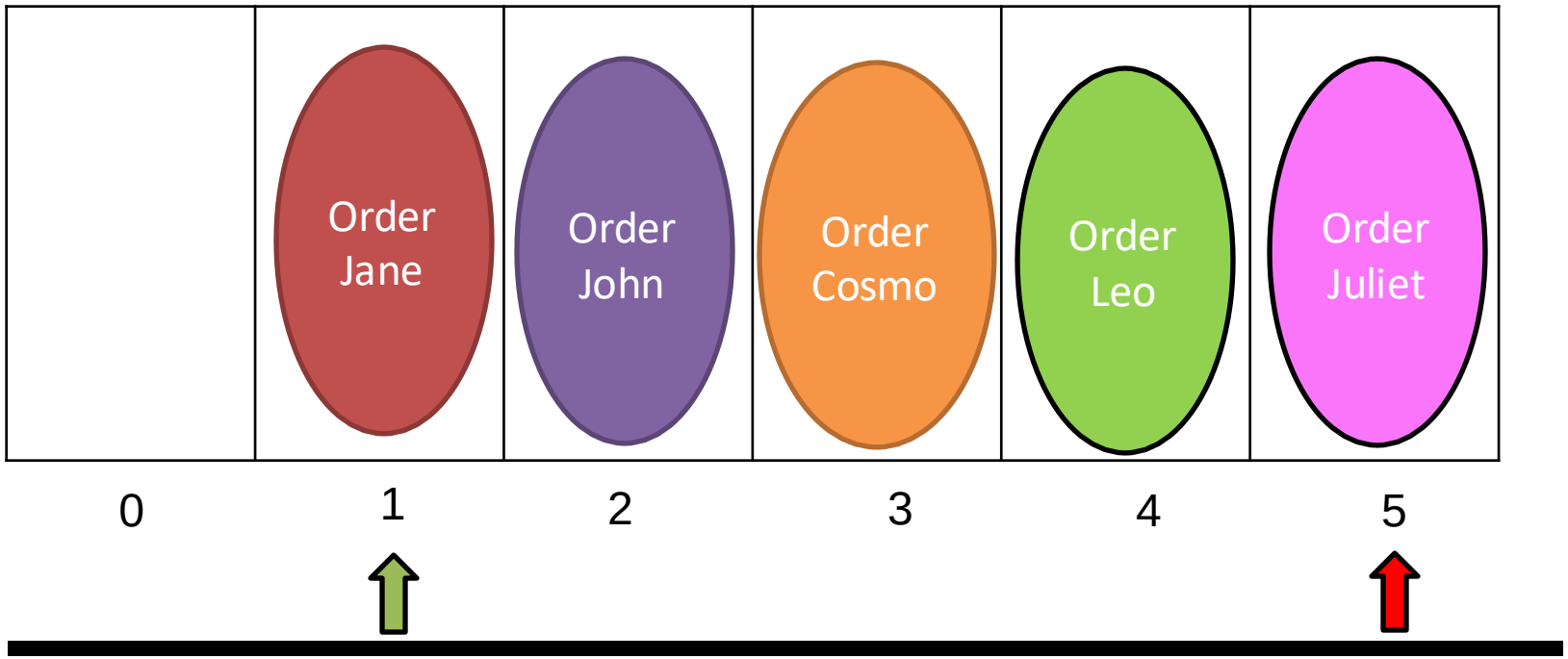


Remove the front element, move front pointer forward one spot

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

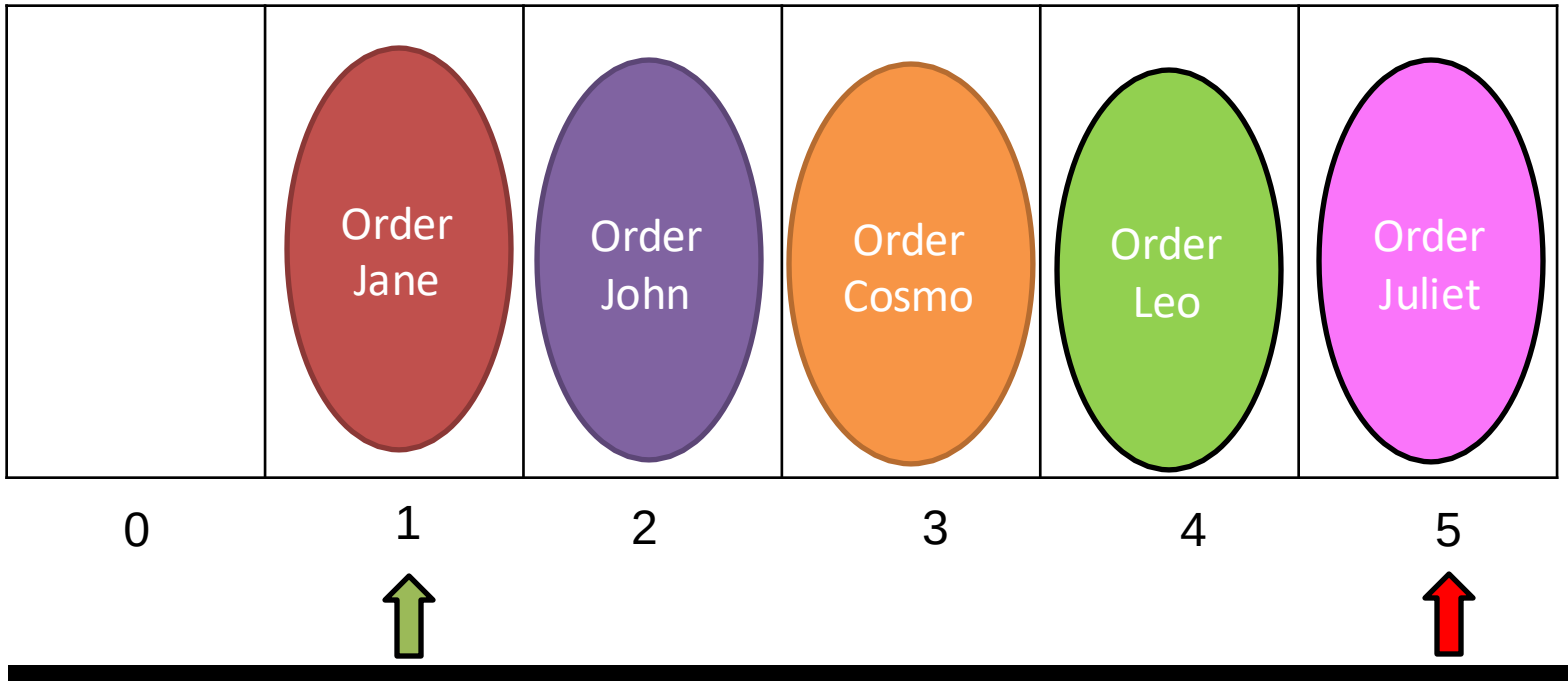


Remove the front element, move front pointer forward one spot

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



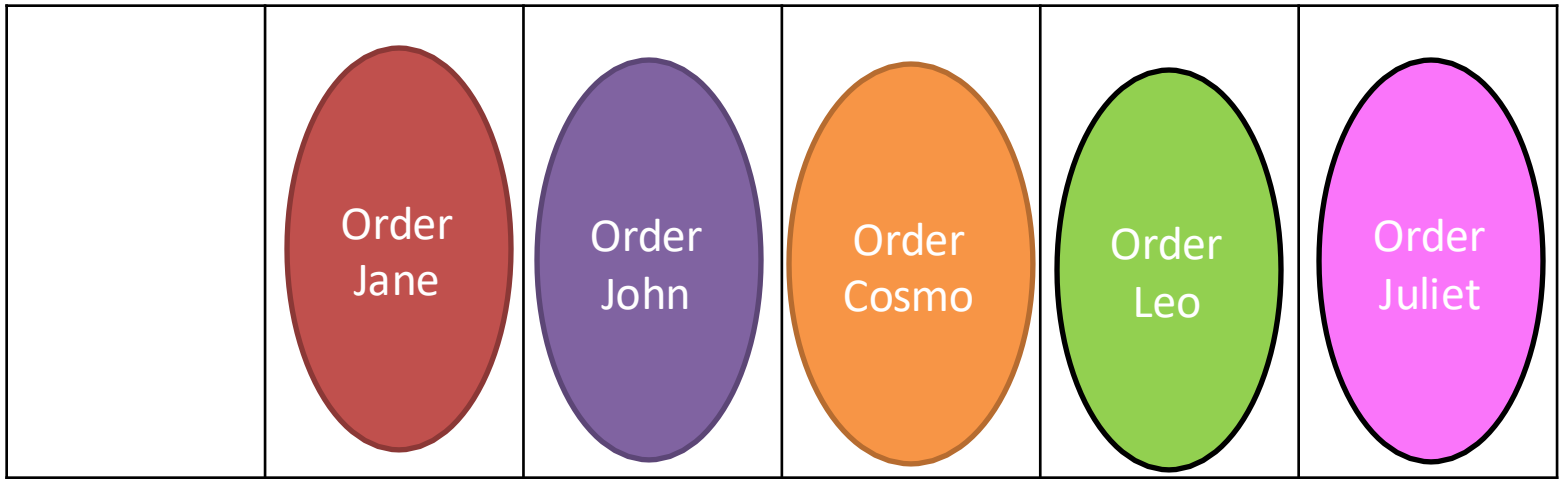
Enqueue again?

```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



Enqueue again?

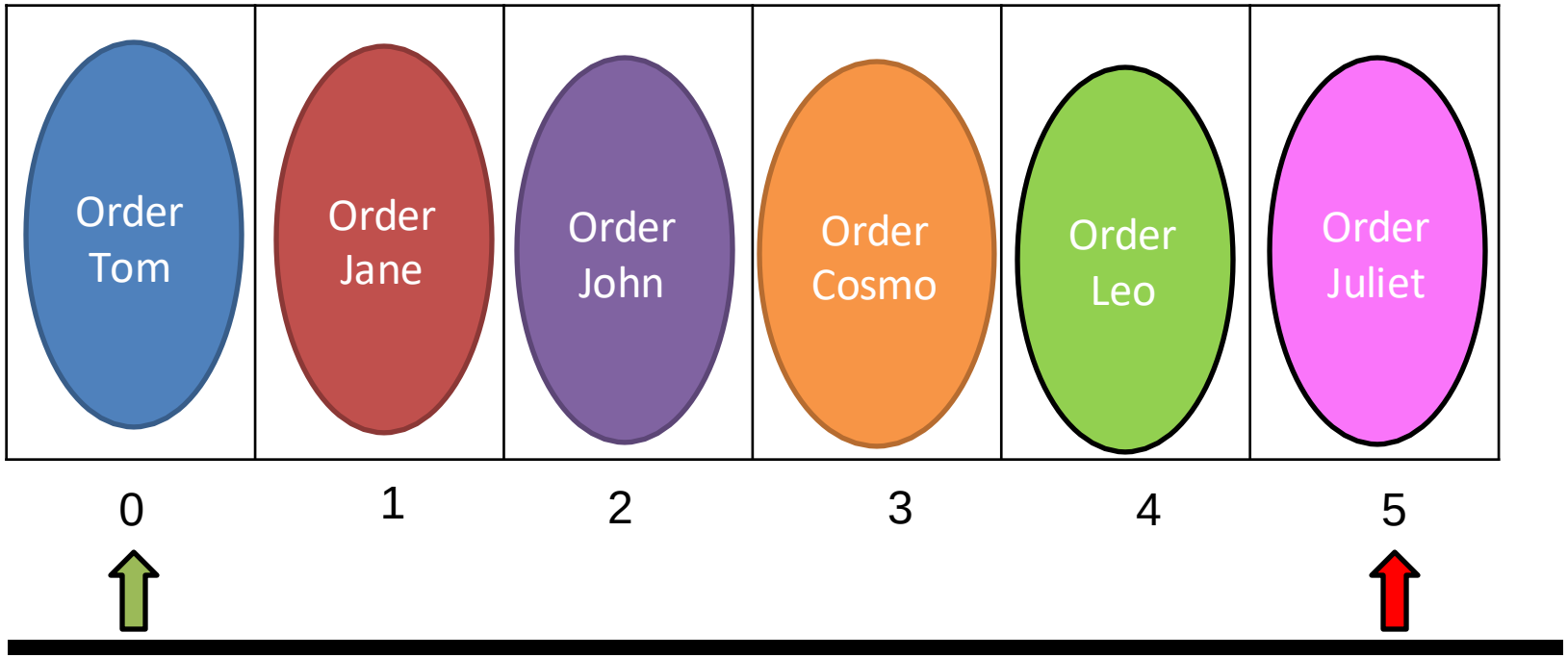
```
public void enqueue(Order newOrder) {

  if (size == capacity) {
    System.out.println("full");
    return;
  }

  data[size] = newOrder;
  size ++;
```

Array index out of bounds error!

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



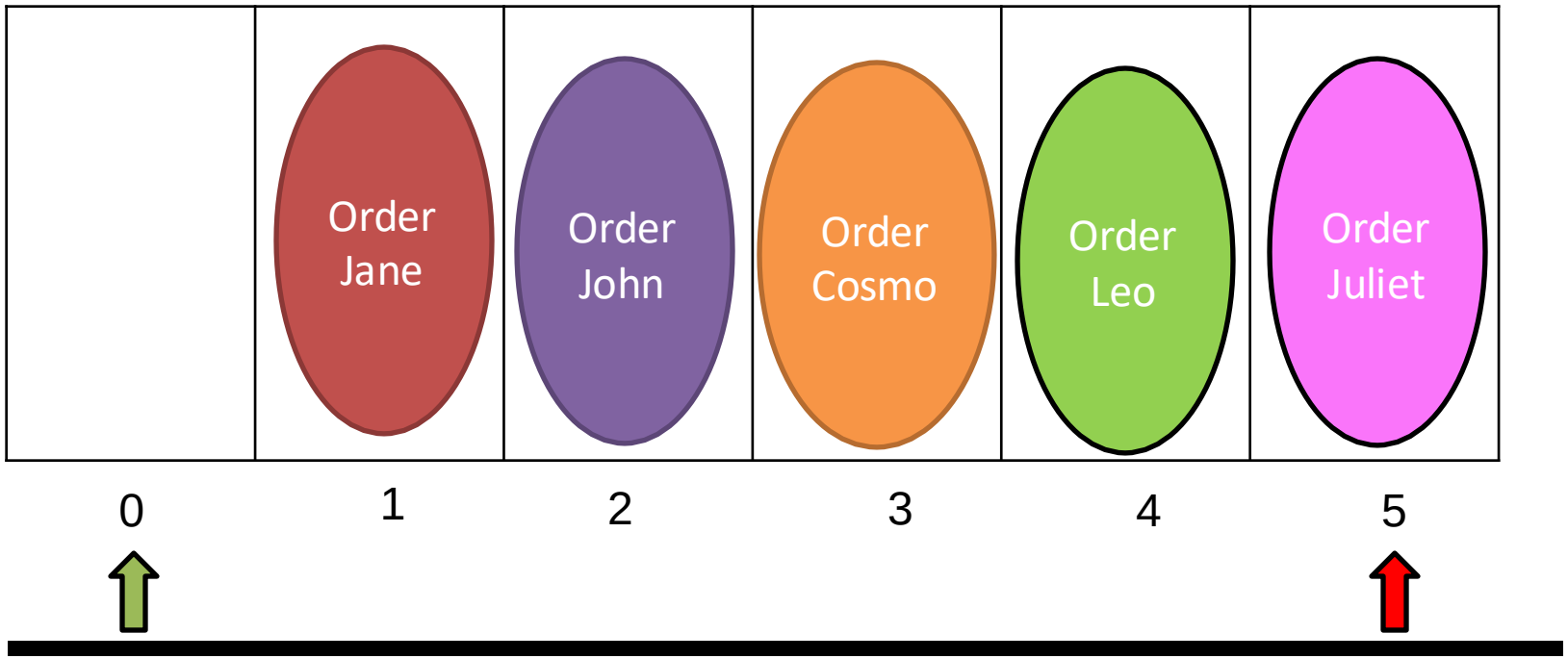
Dequeue?

1. Remove the front element
2. Make some room in the back

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



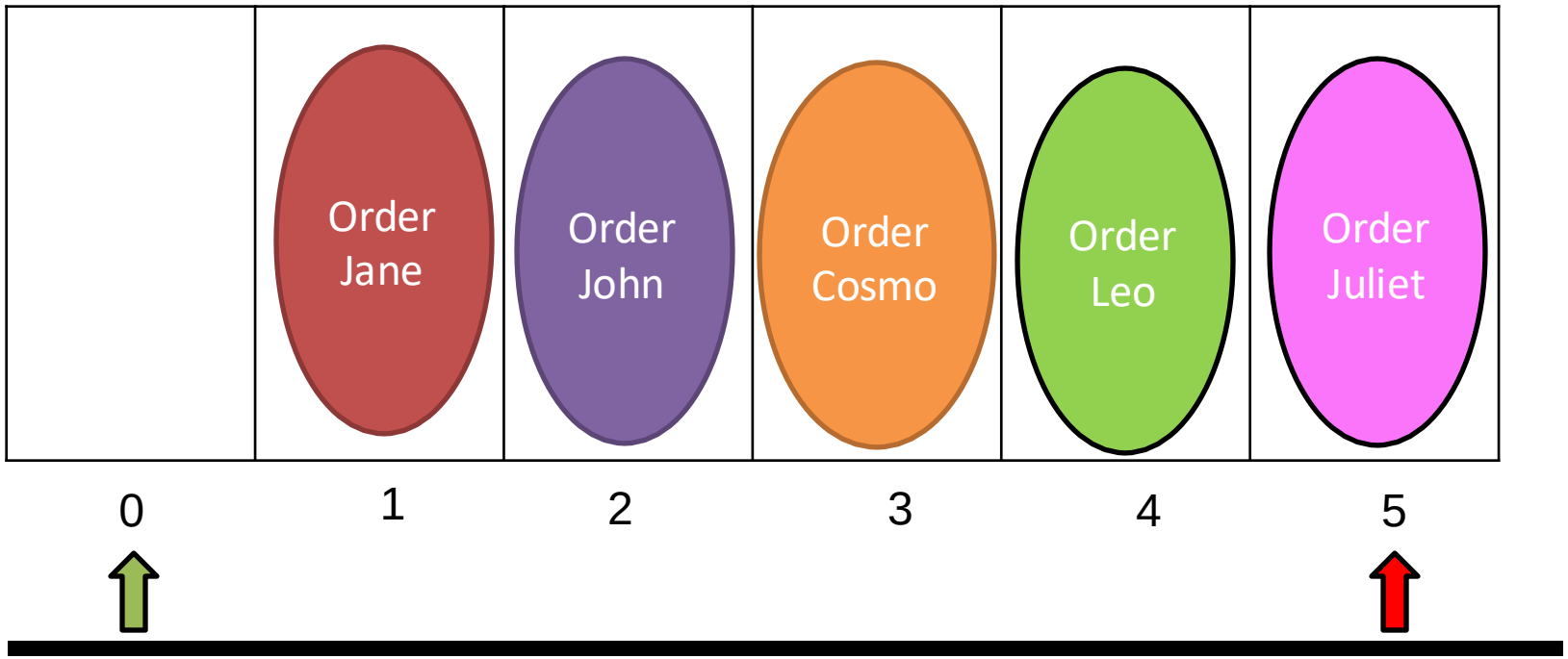
Dequeue?

1. Remove the front element
2. Make some room in the back

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



Dequeue?

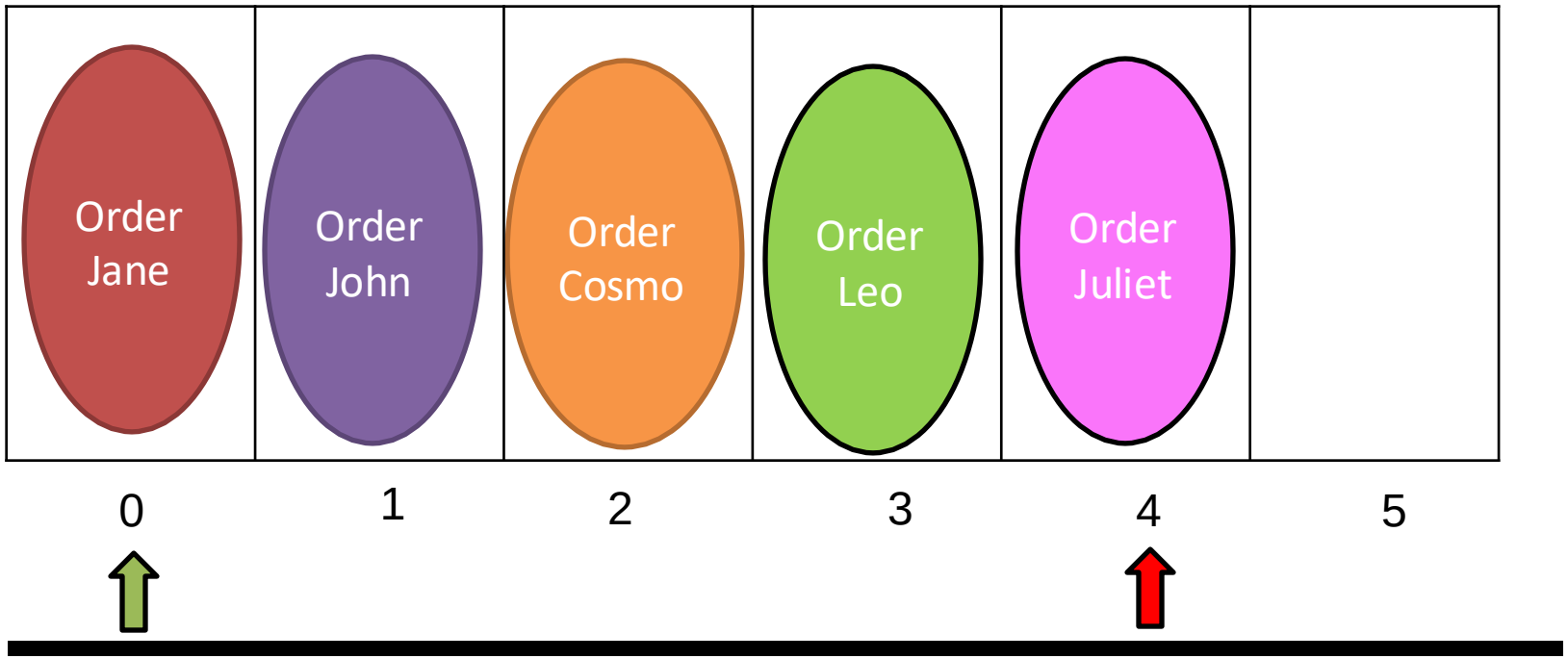
1. Remove the front element
2. Make some room in the back

Shift all of our data over one spot

```
capacity = 6    front = 0
size = 6        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



Dequeue?

1. Remove the front element
2. Make some room in the back

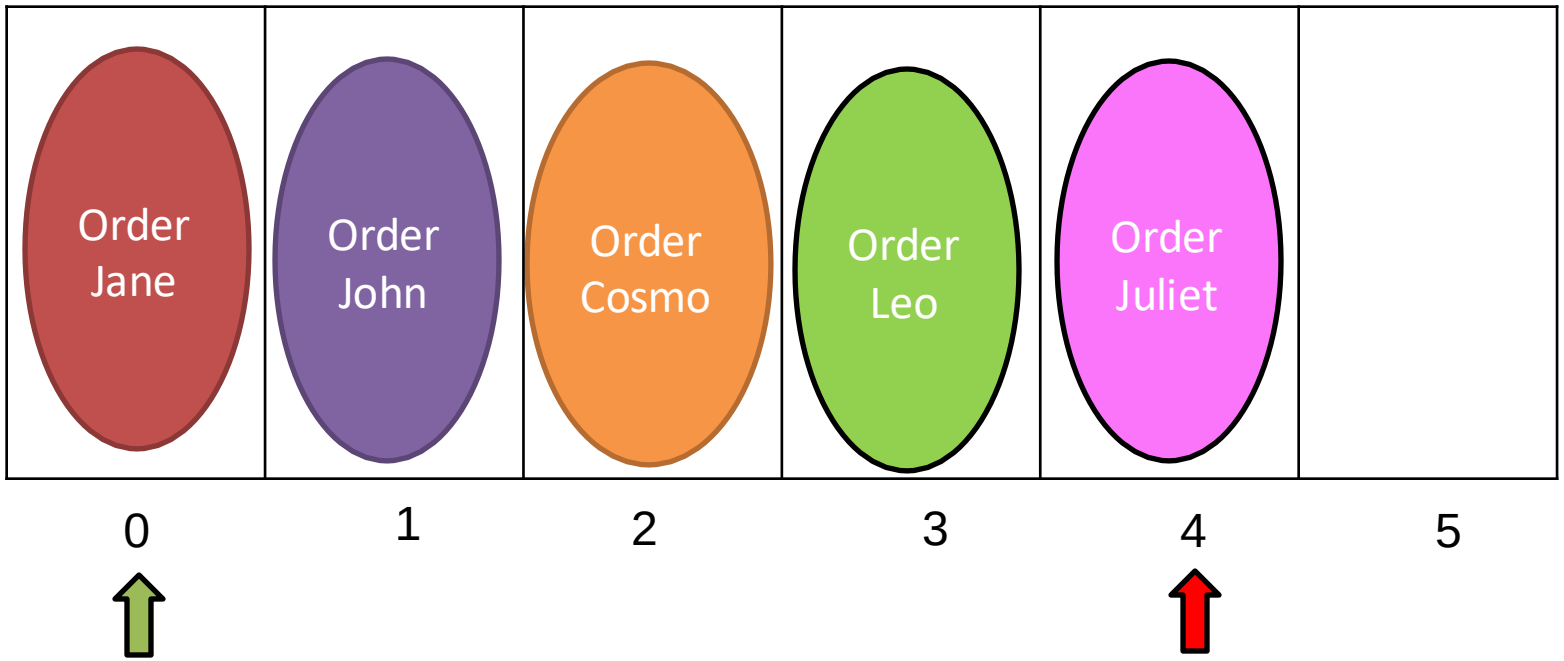
Shift all of our data over one spot

The `front` of our queue will **always** stay at zero

```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



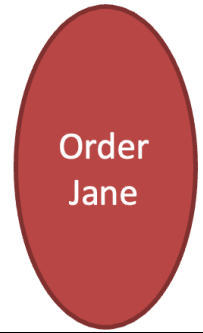

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

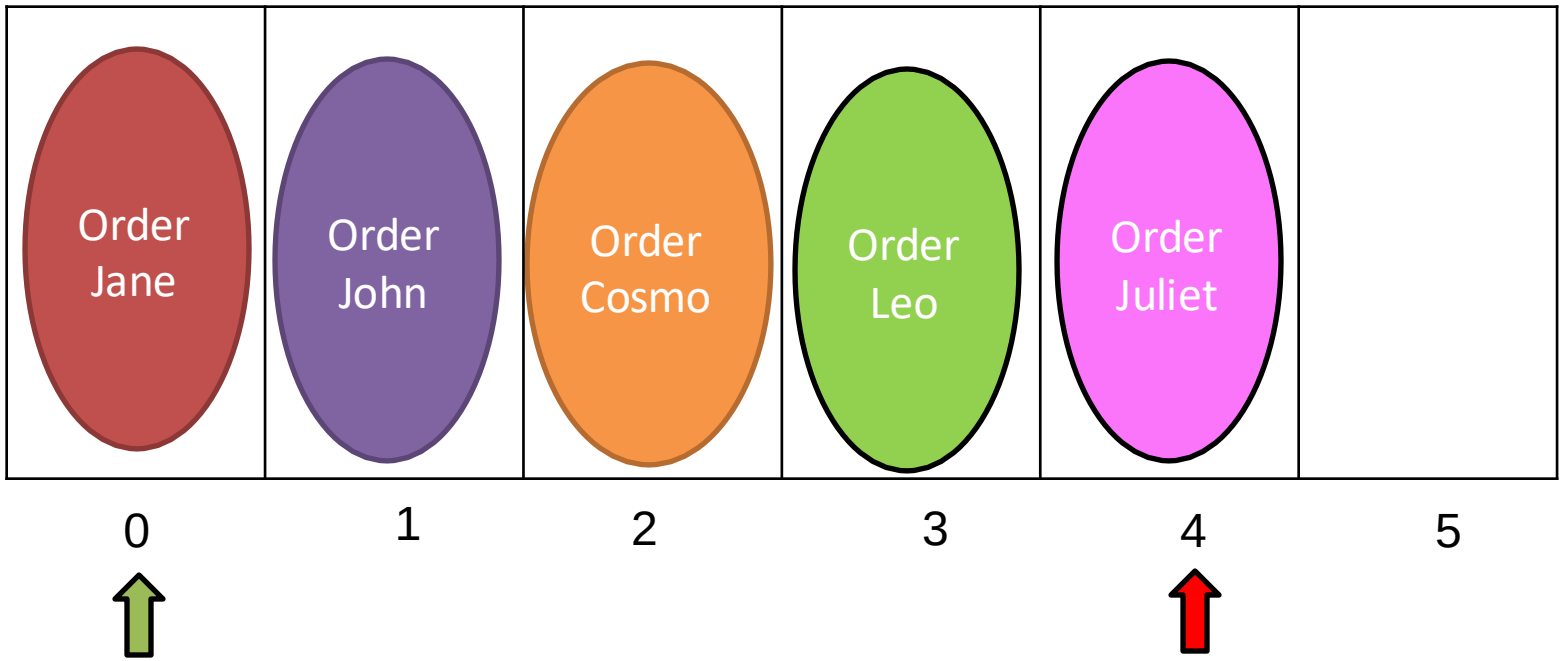
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =






```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

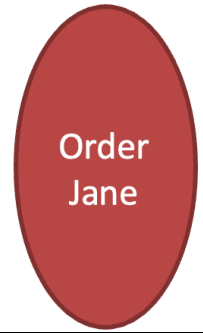
capacity = 6 front = 0

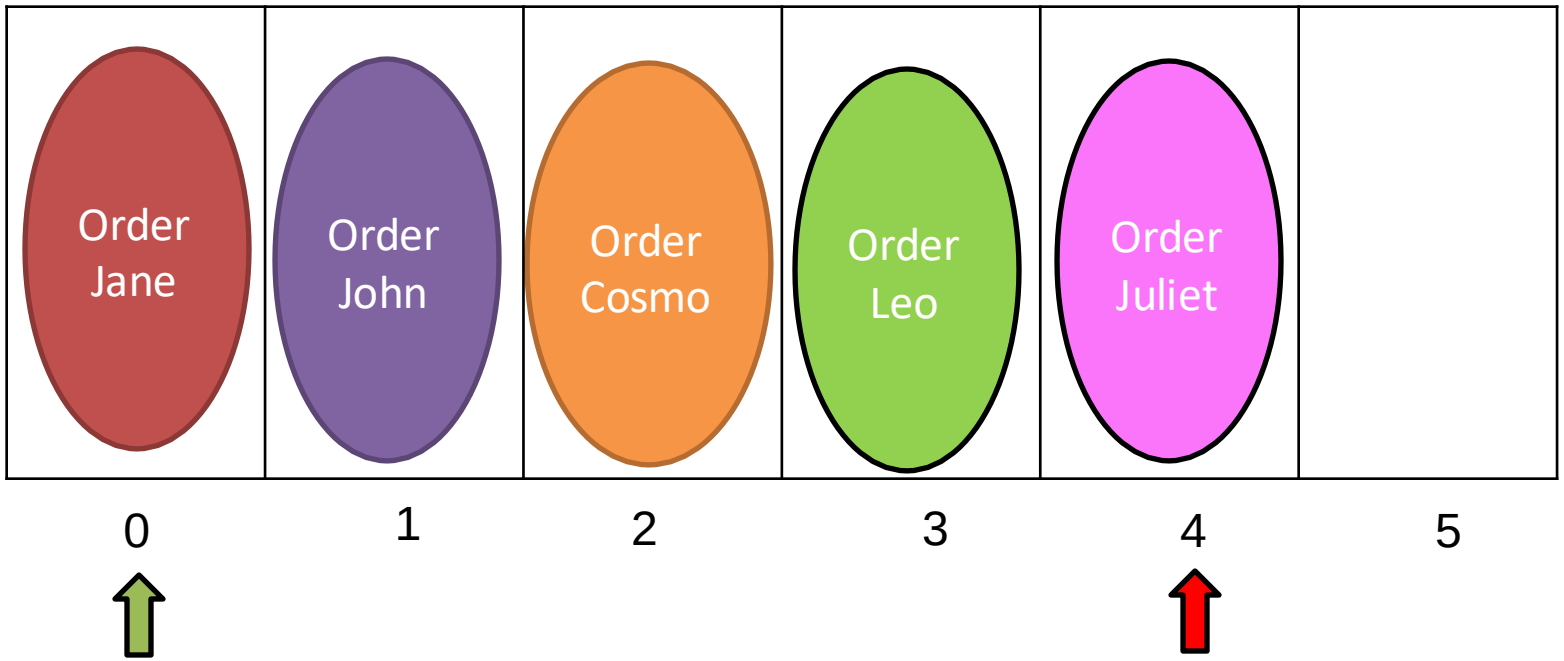
size = 5 rear = 4

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





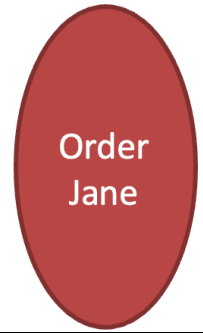

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

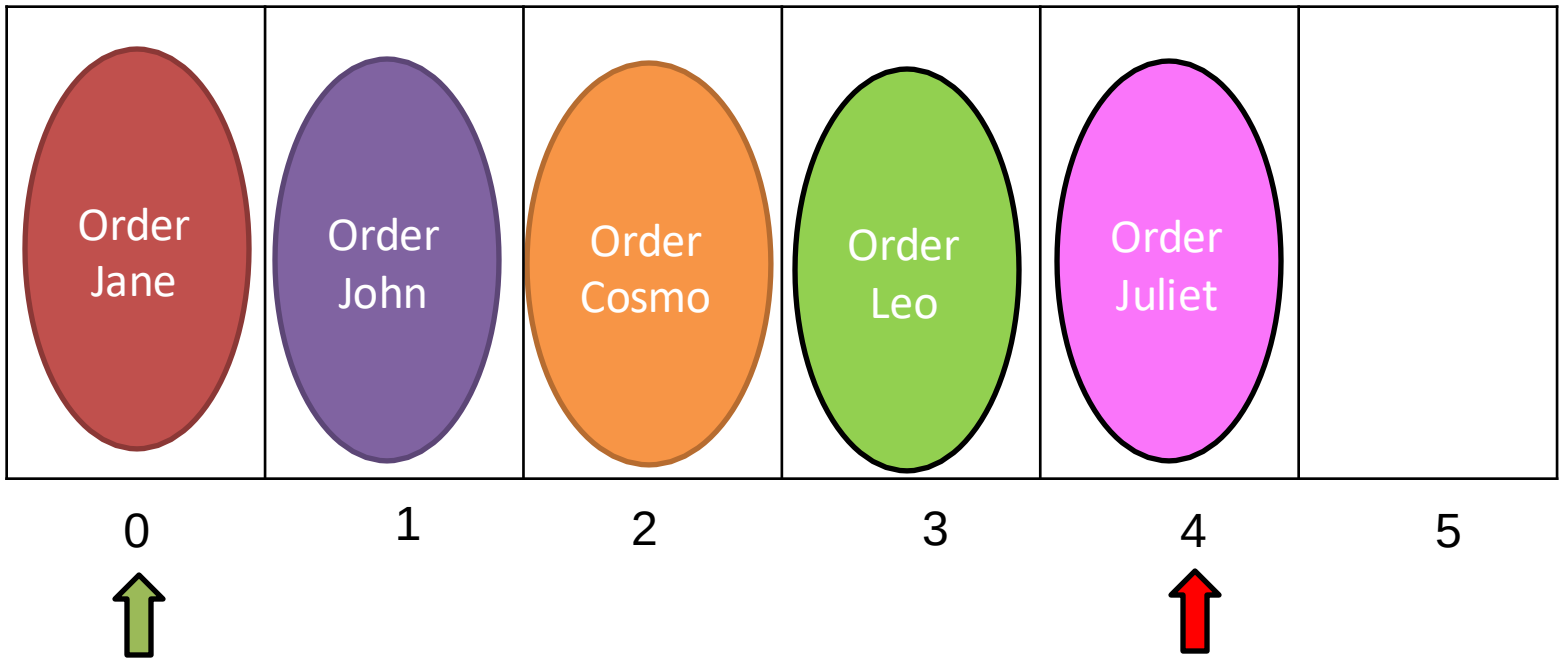
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =






```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

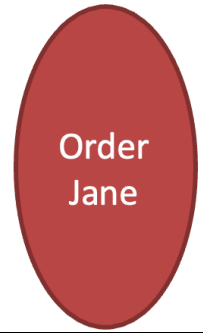
```
capacity = 6    front = 0
size = 5        rear = 4
```

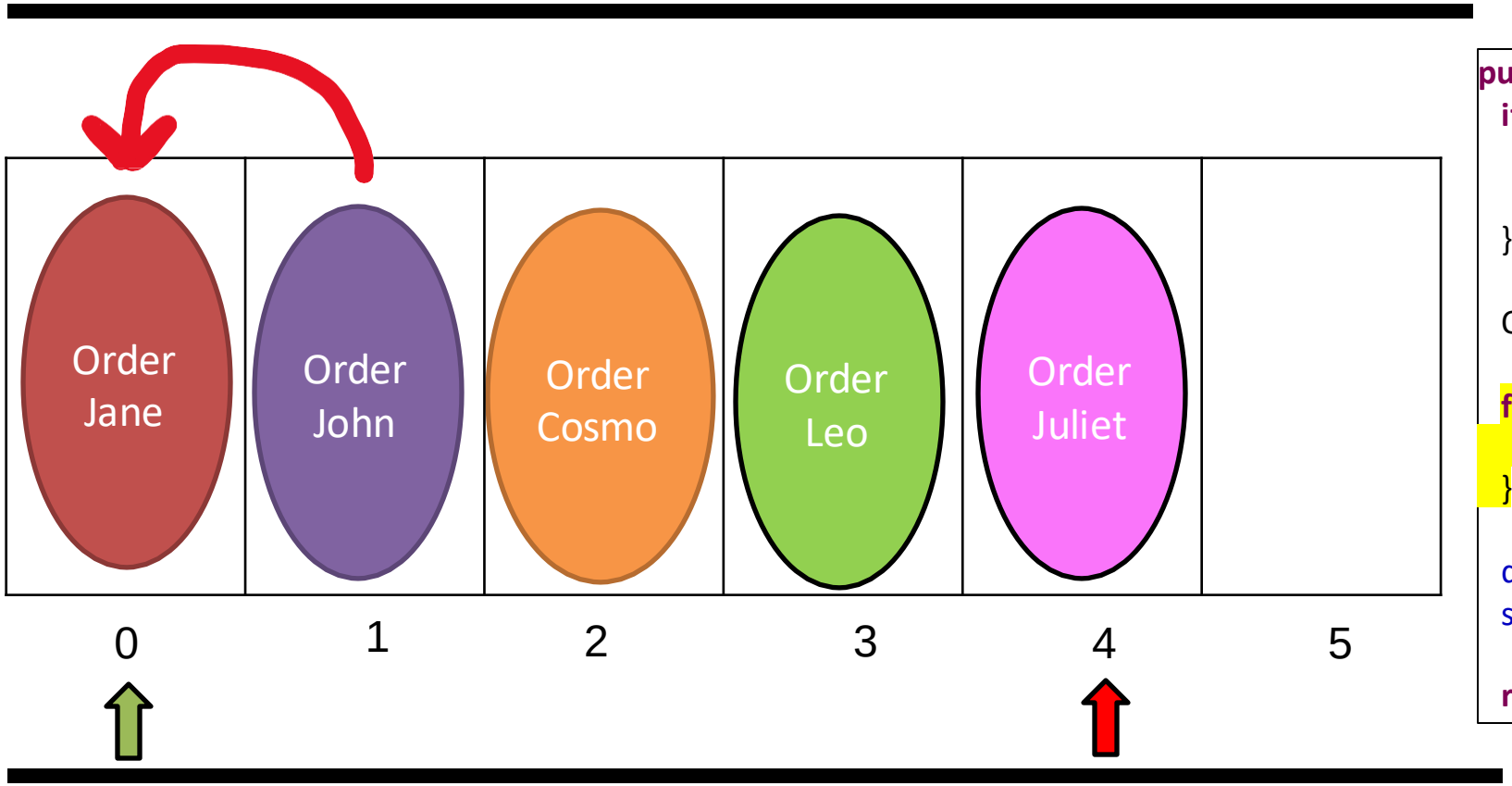
Shift everything over one spot

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





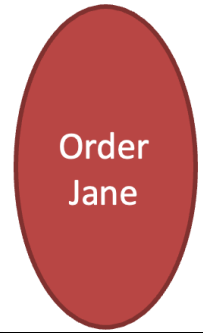

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

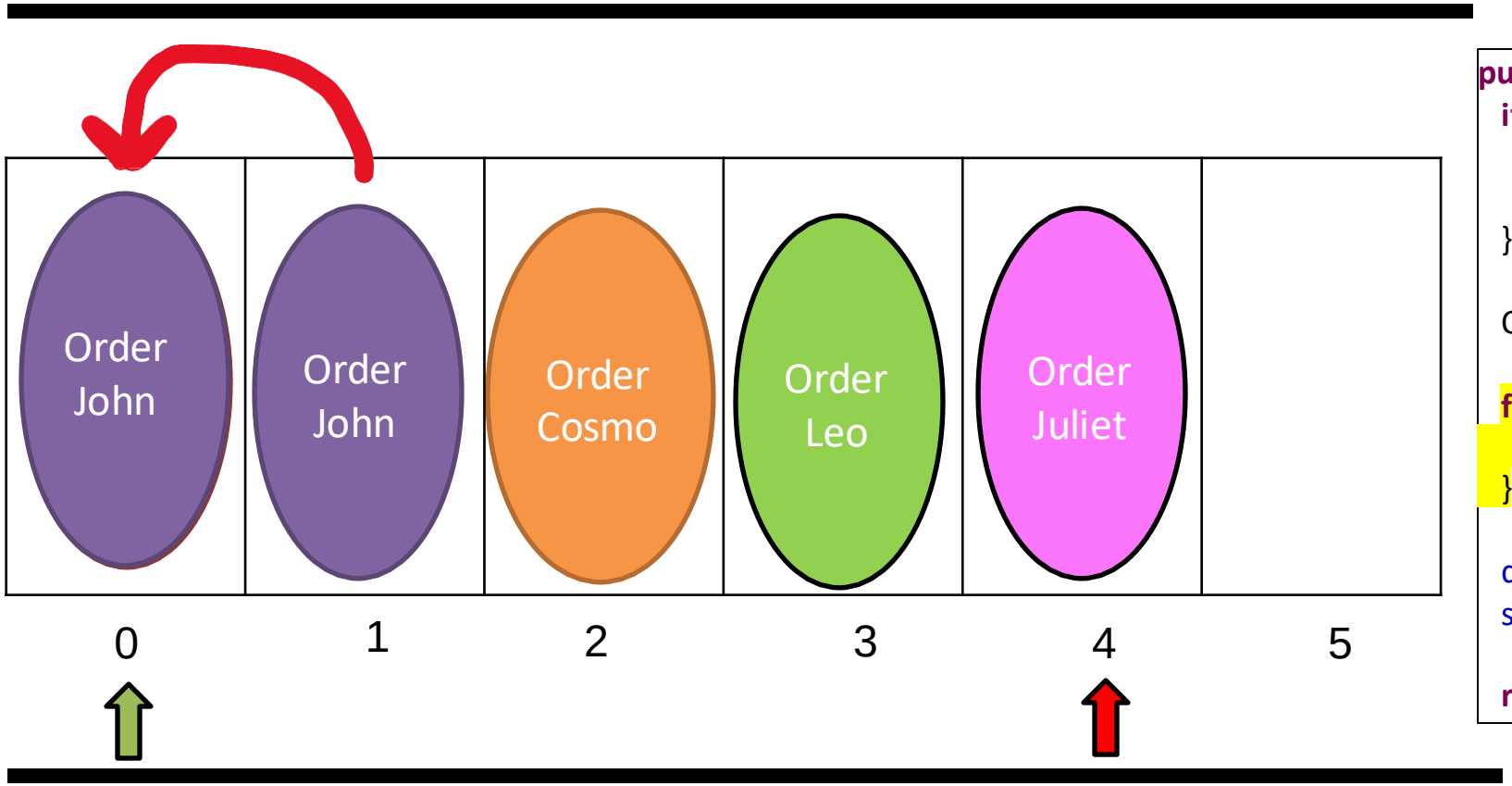
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





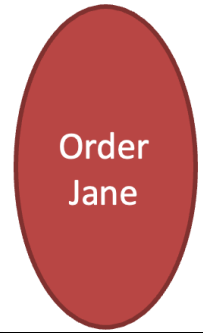

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

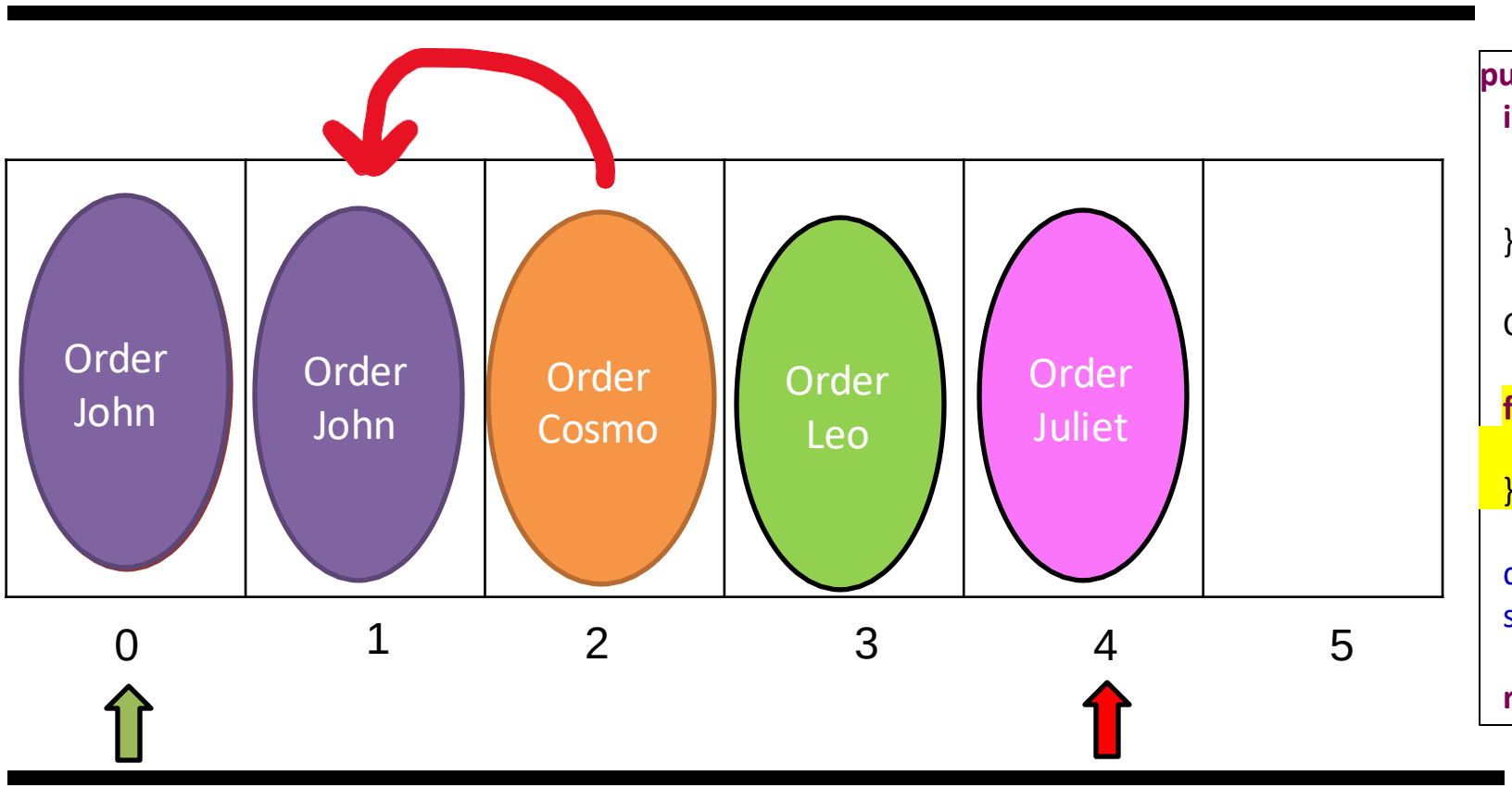
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





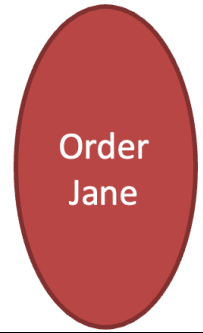

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

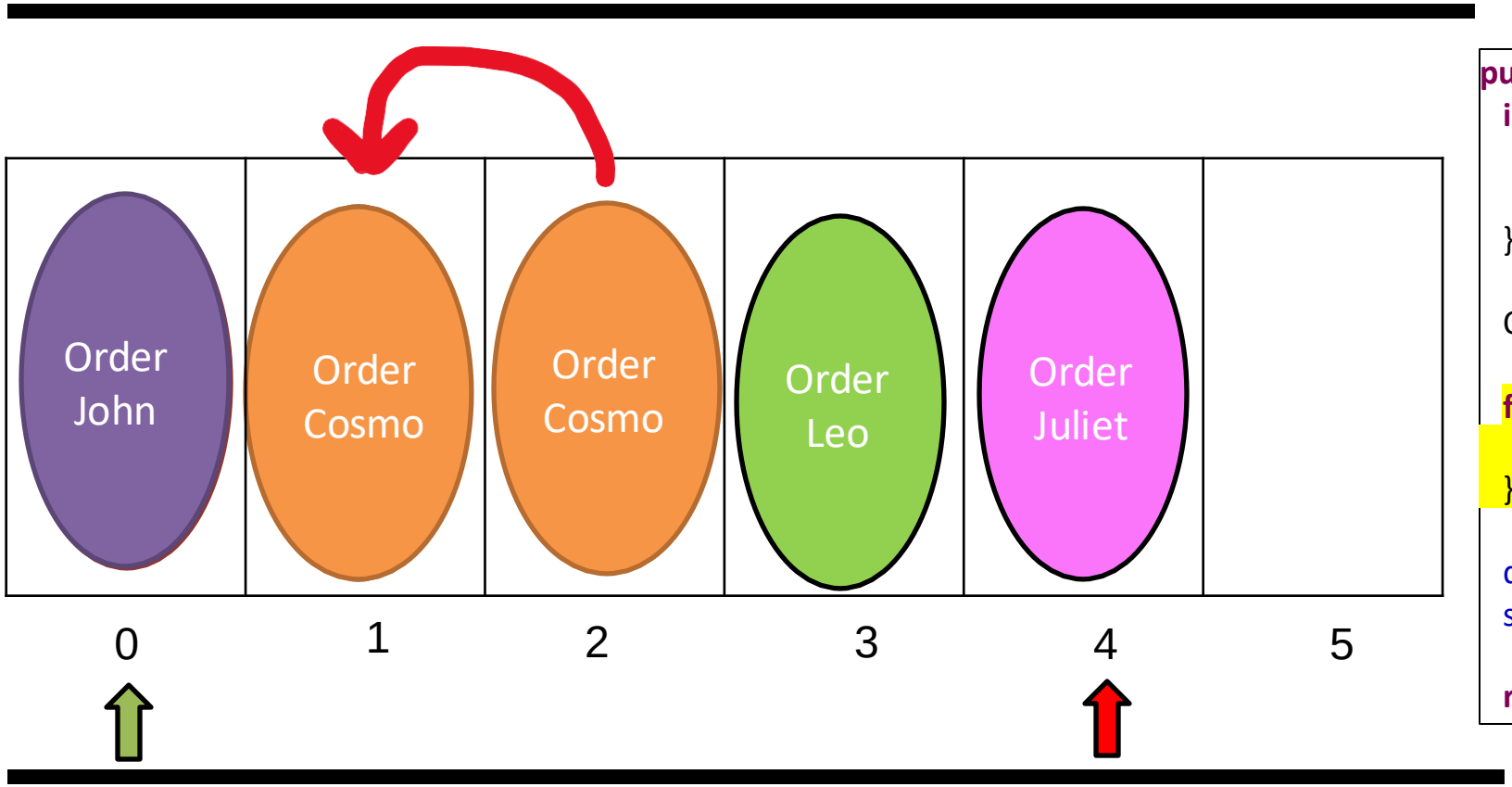
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





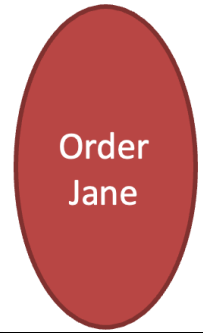

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

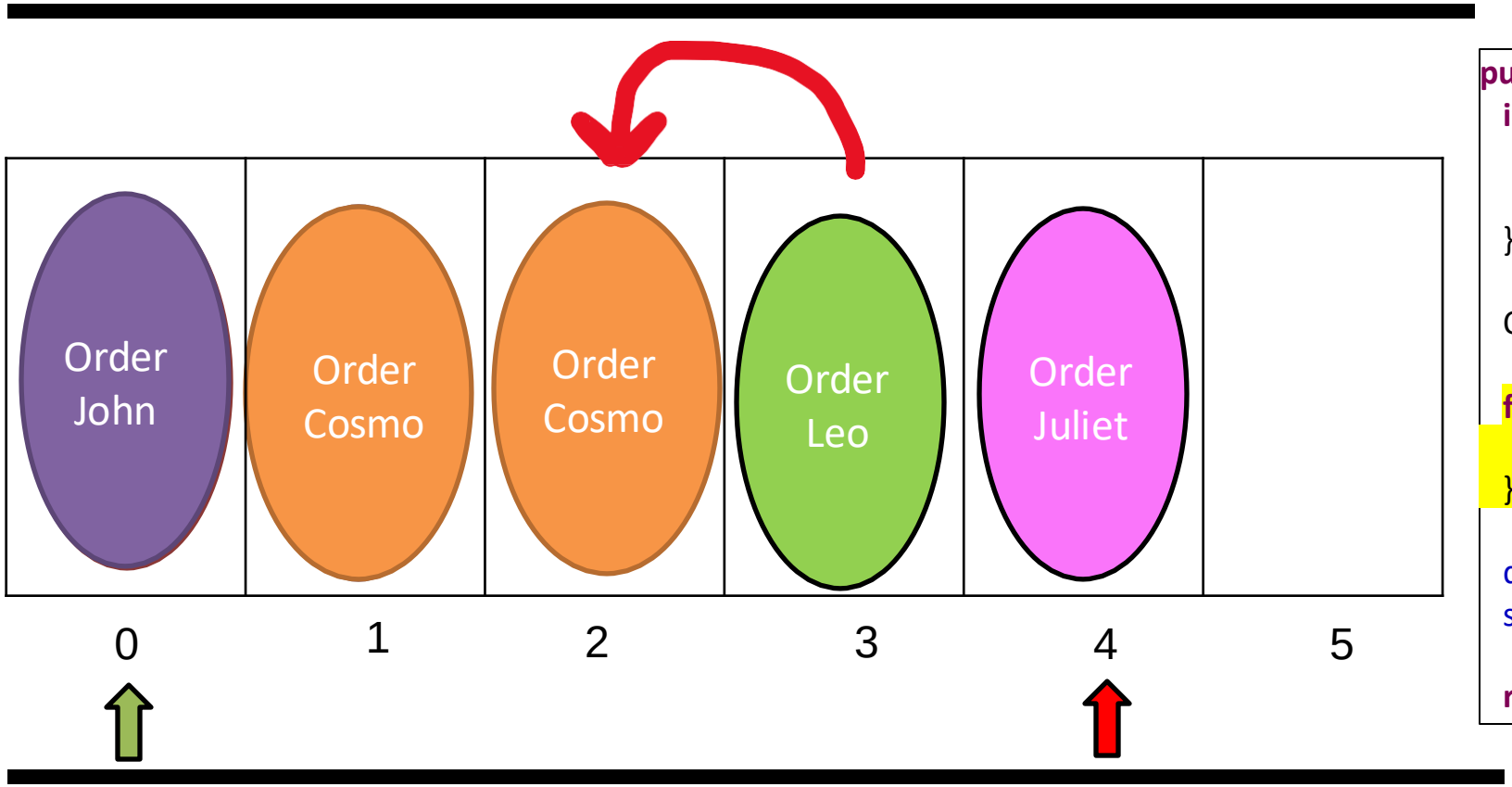
```
capacity = 6    front = 0
size = 5        rear = 5
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





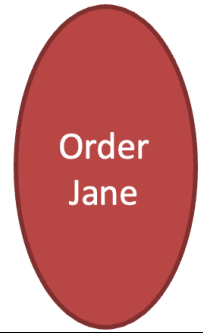

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

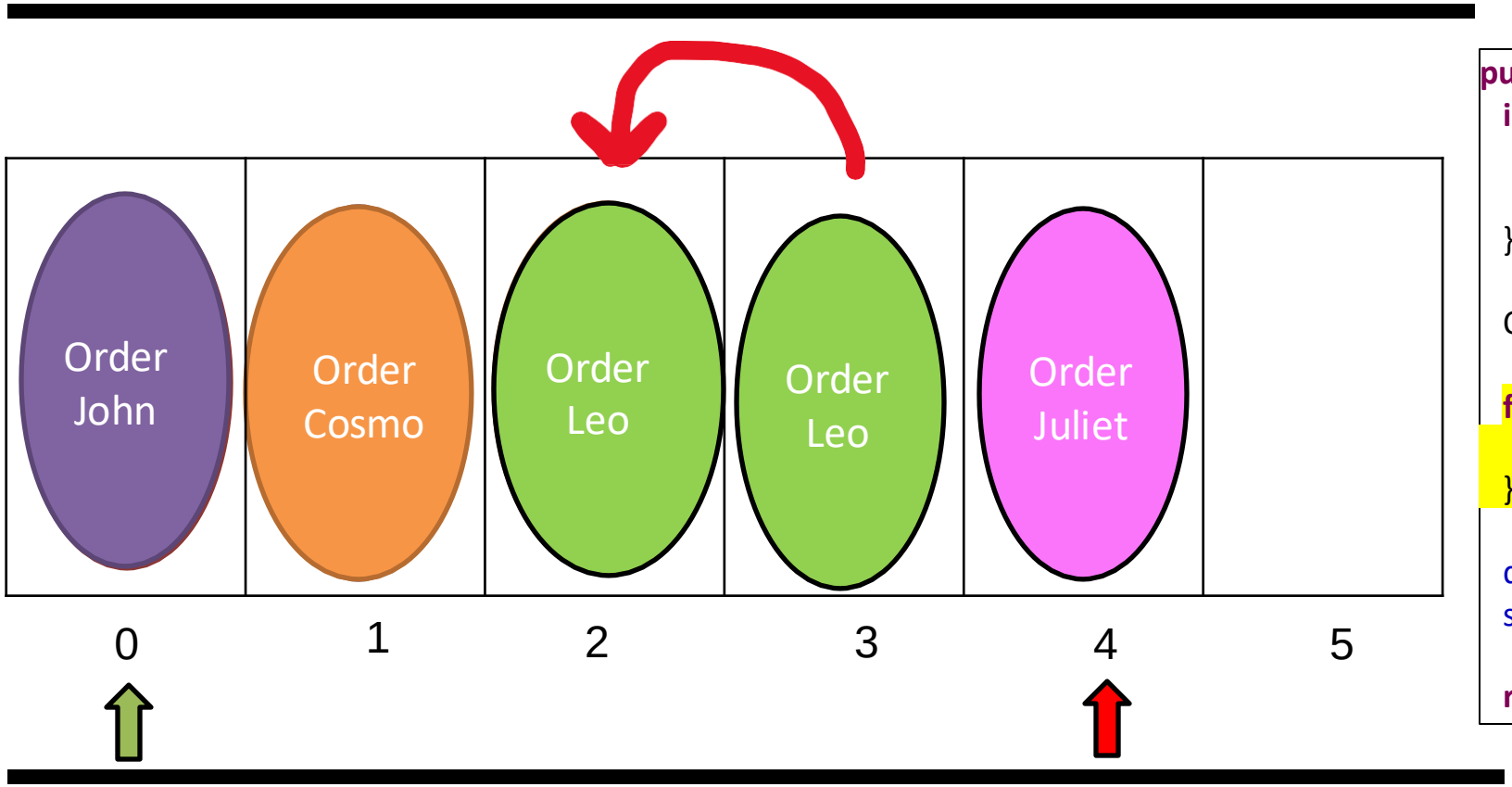
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





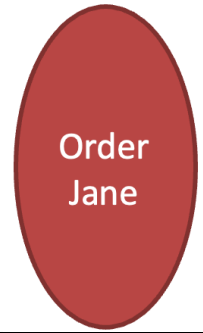

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

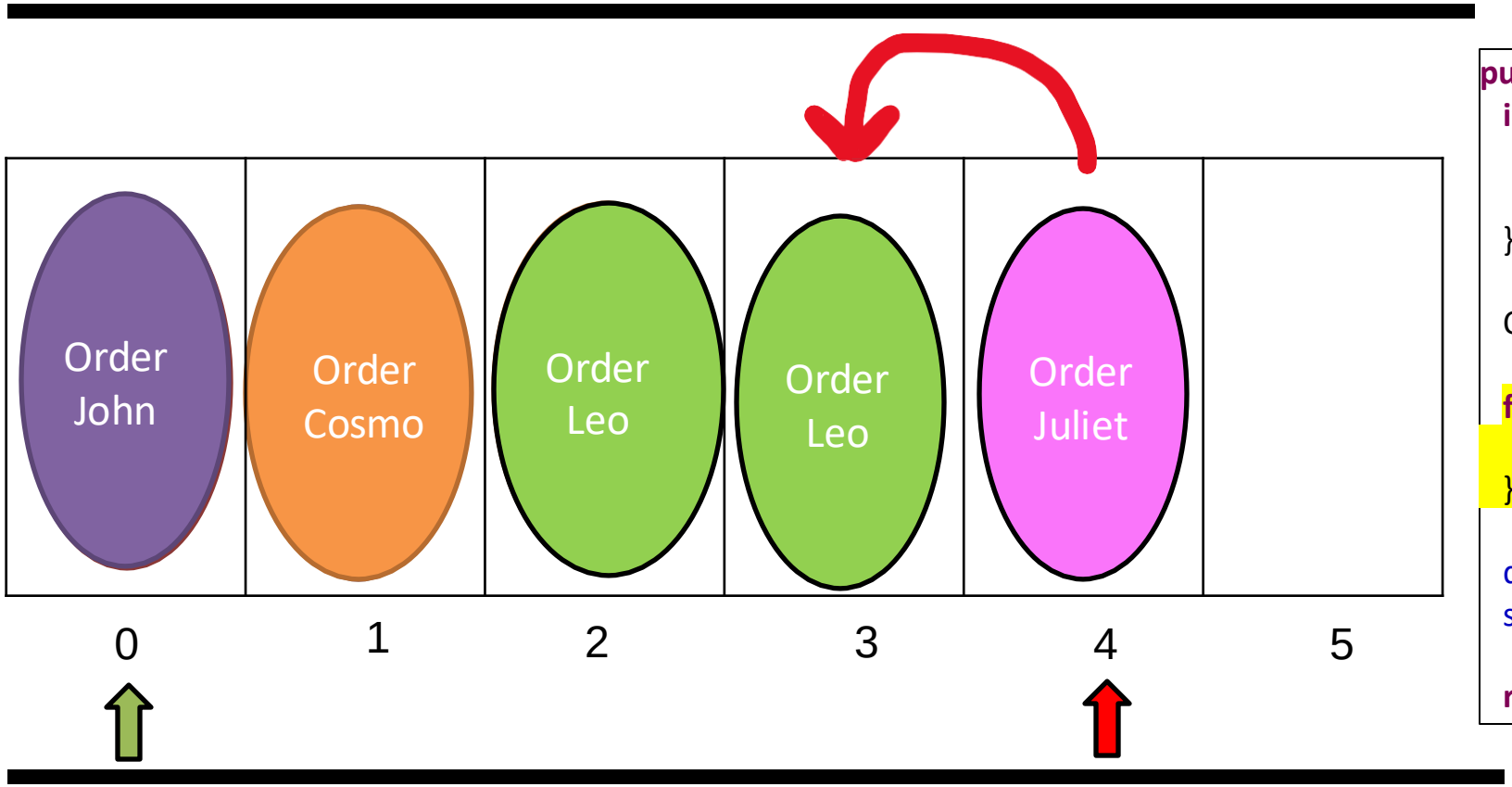
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





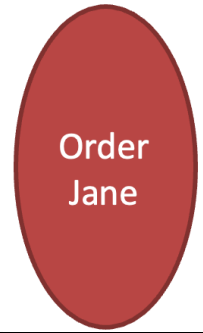

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

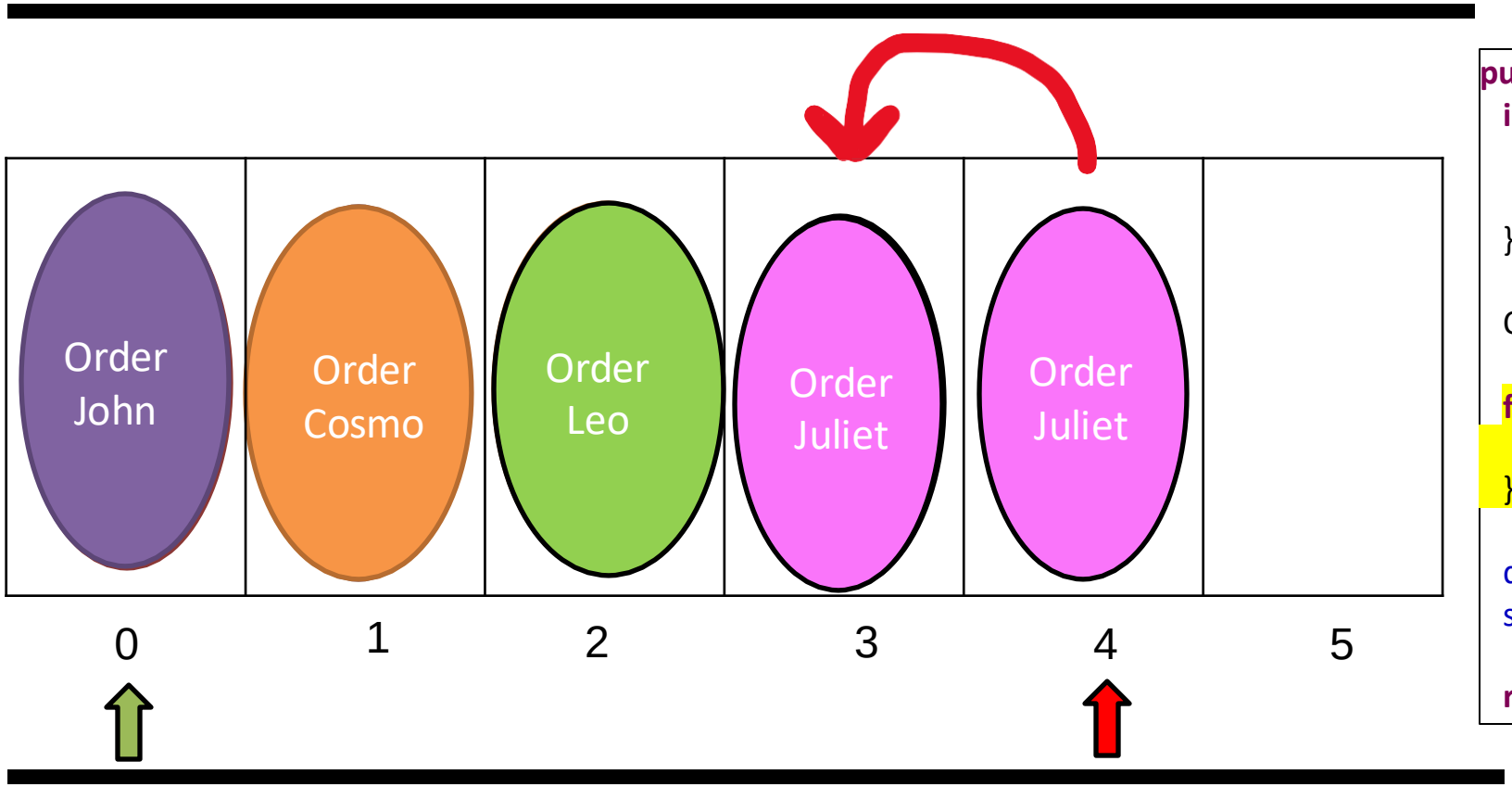
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





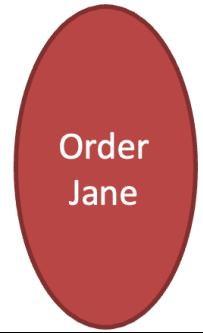

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

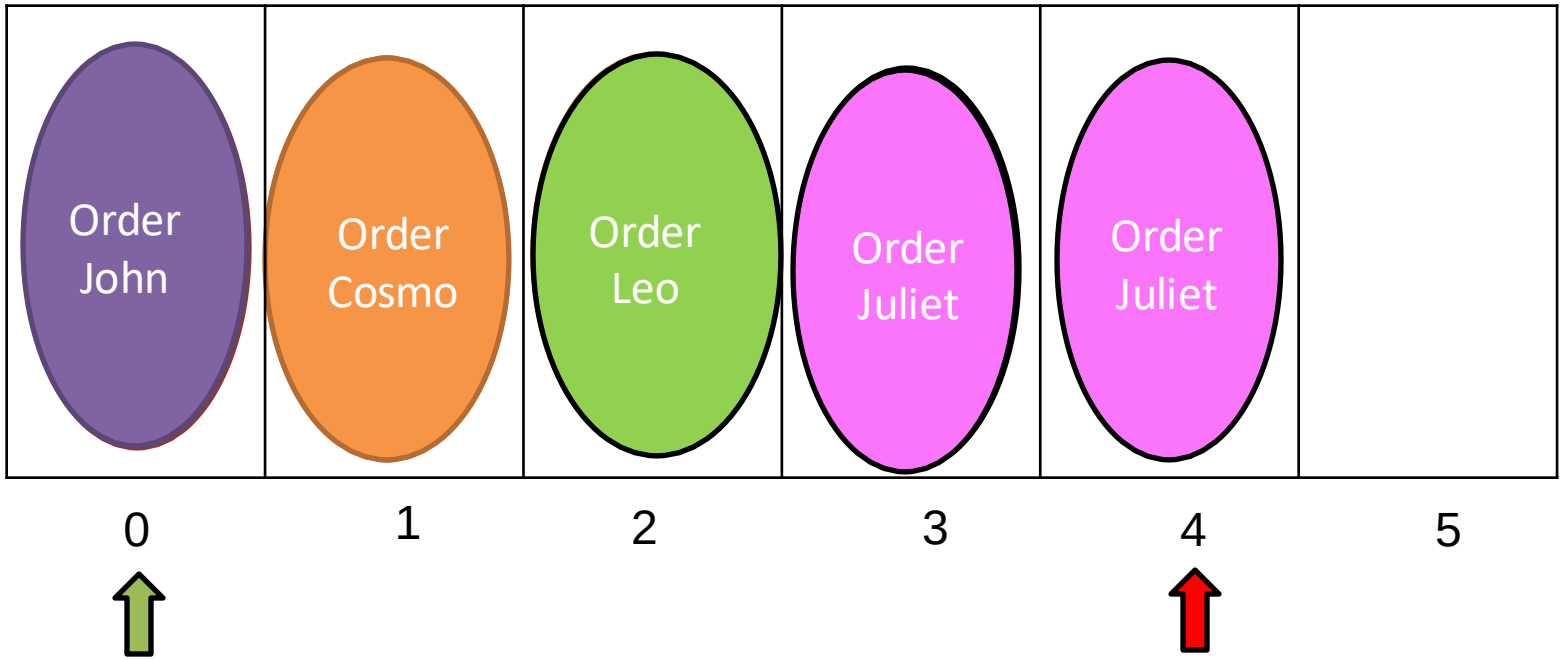
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





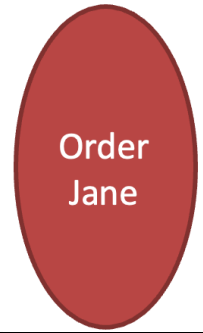

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

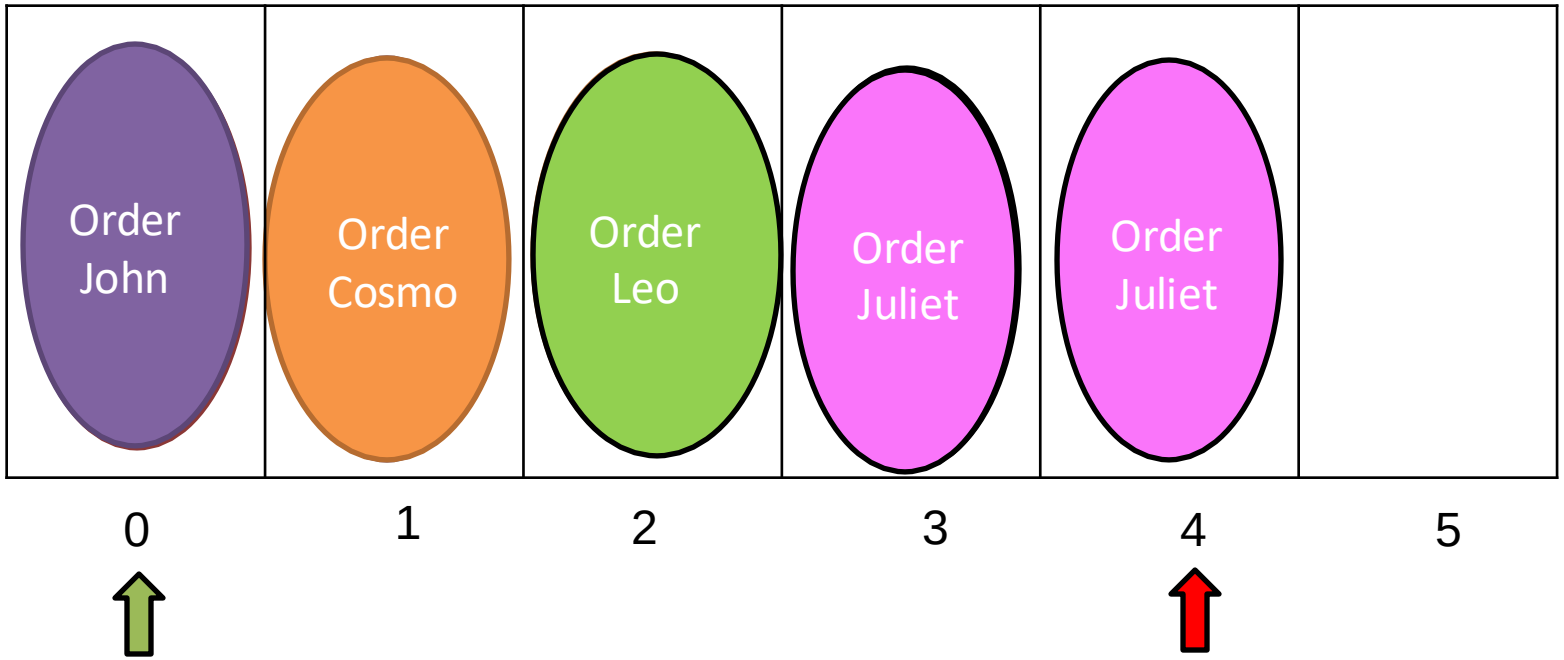
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





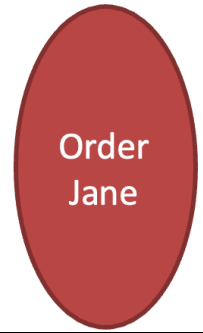

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

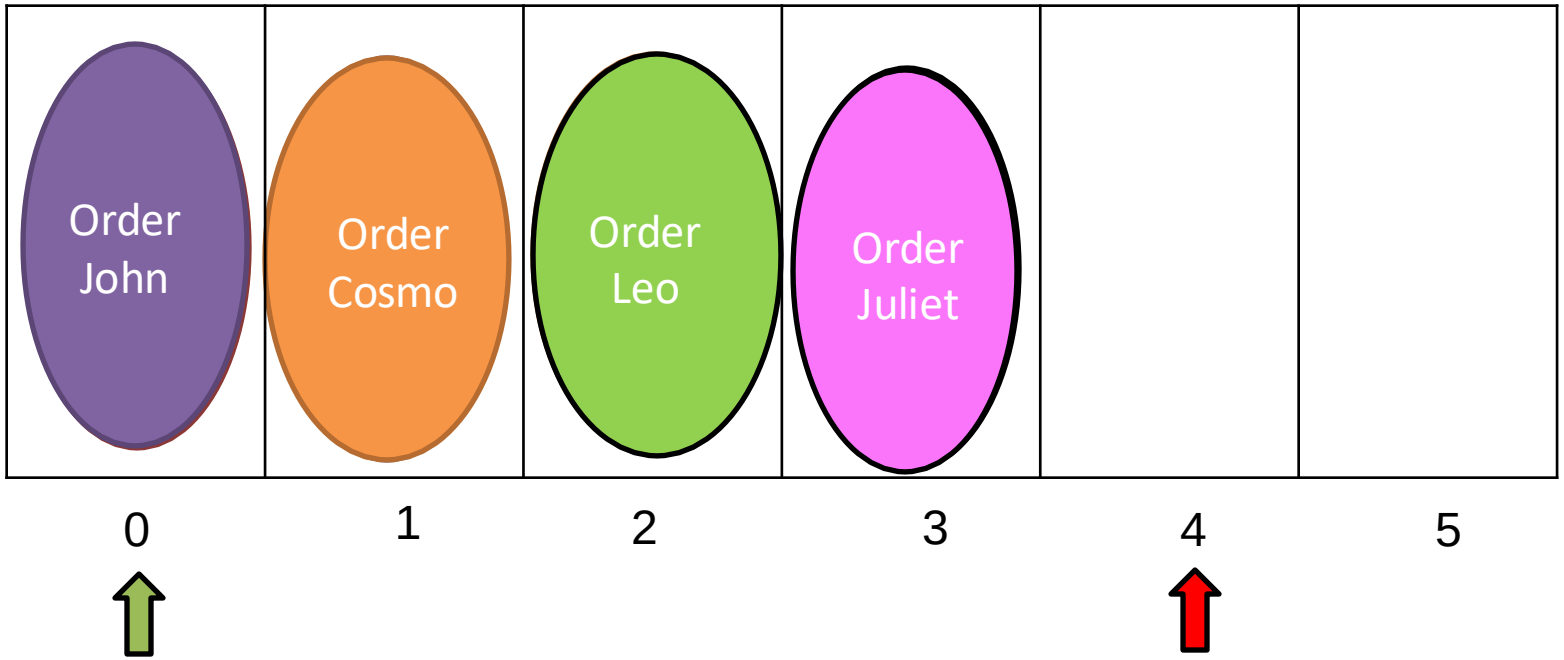
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =





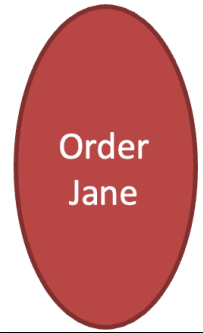

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

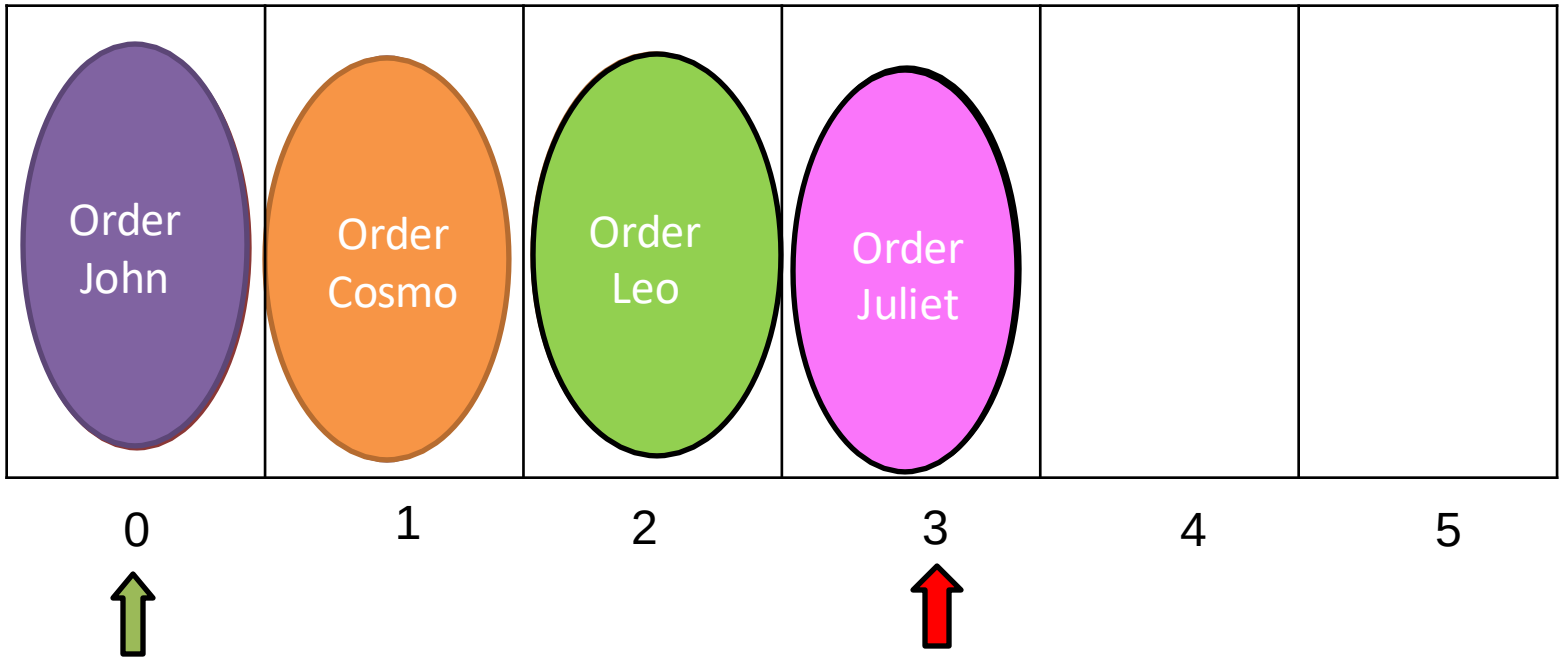
```
capacity = 6    front = 0
size = 5        rear = 4
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

Order temp =



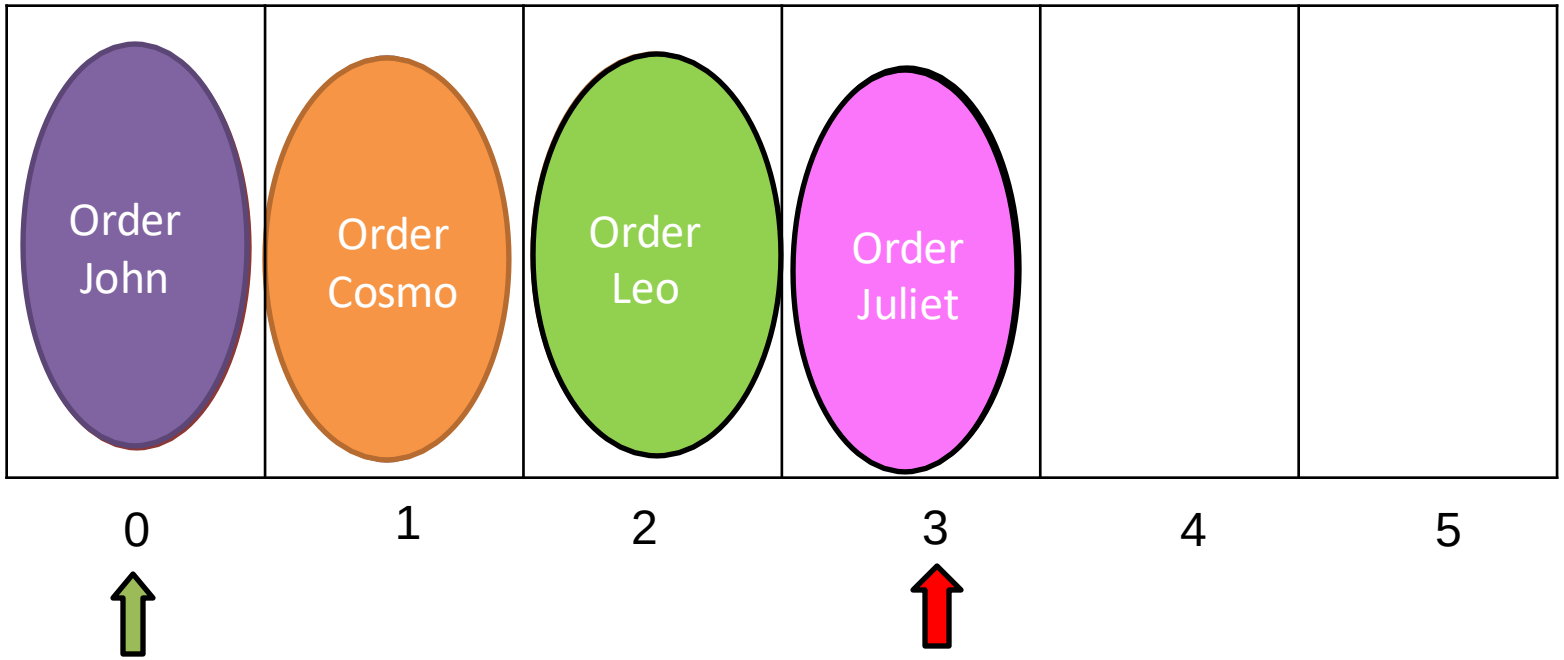



```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

```
capacity = 6    front = 0
size = 4        rear = 3
```

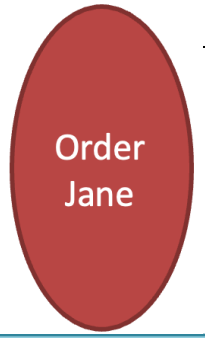
Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements




```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

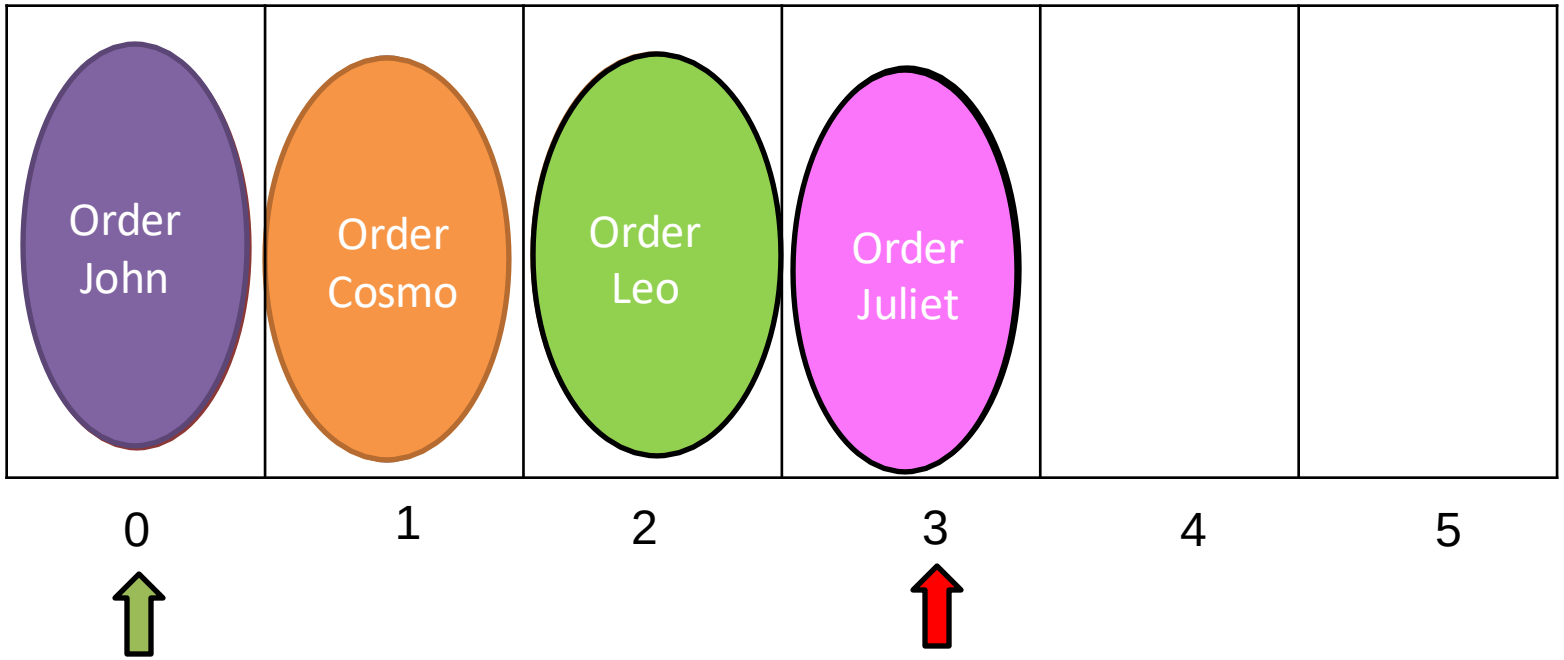
Order temp =



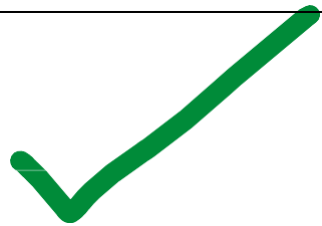

```
capacity = 6    front = 0
size = 4        rear = 3
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements



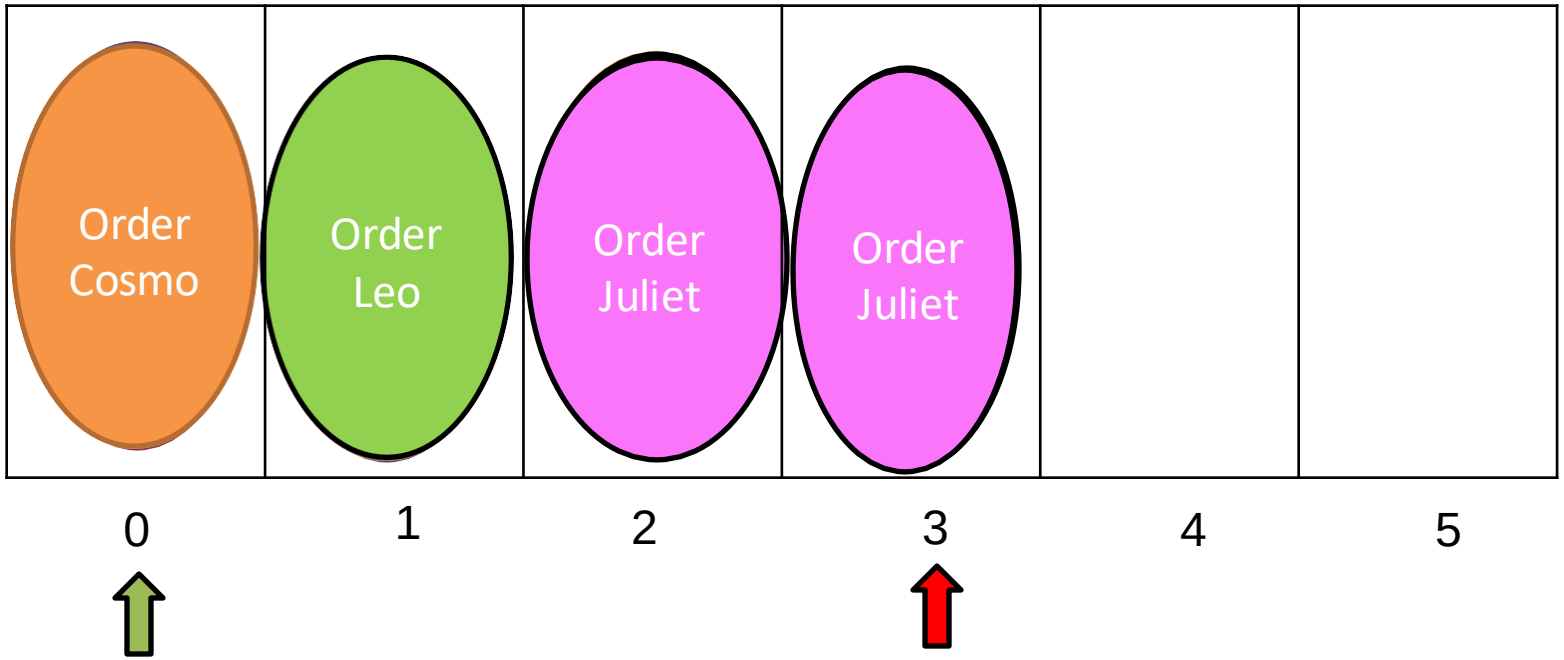

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```



```
capacity = 6    front = 0
size = 4        rear = 3
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

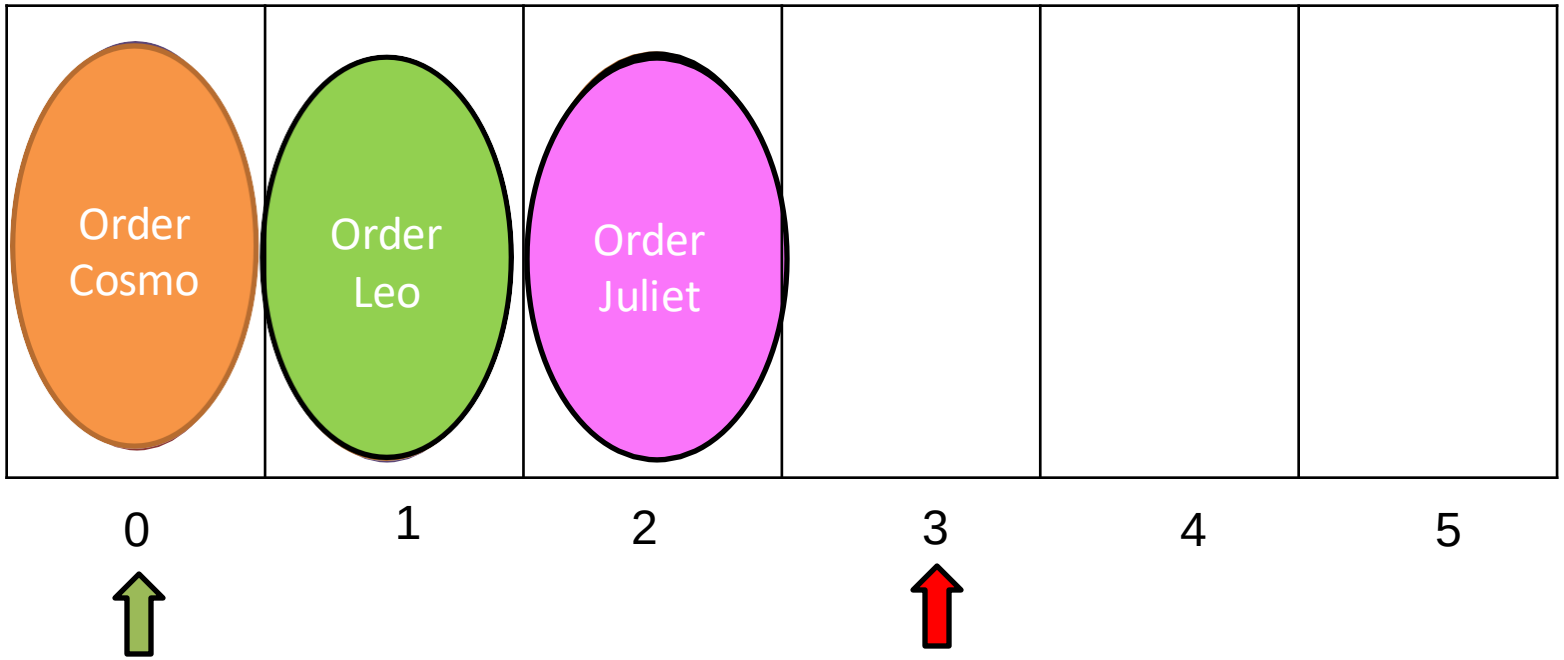



```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

```
capacity = 6    front = 0
size = 4        rear = 3
```

Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements

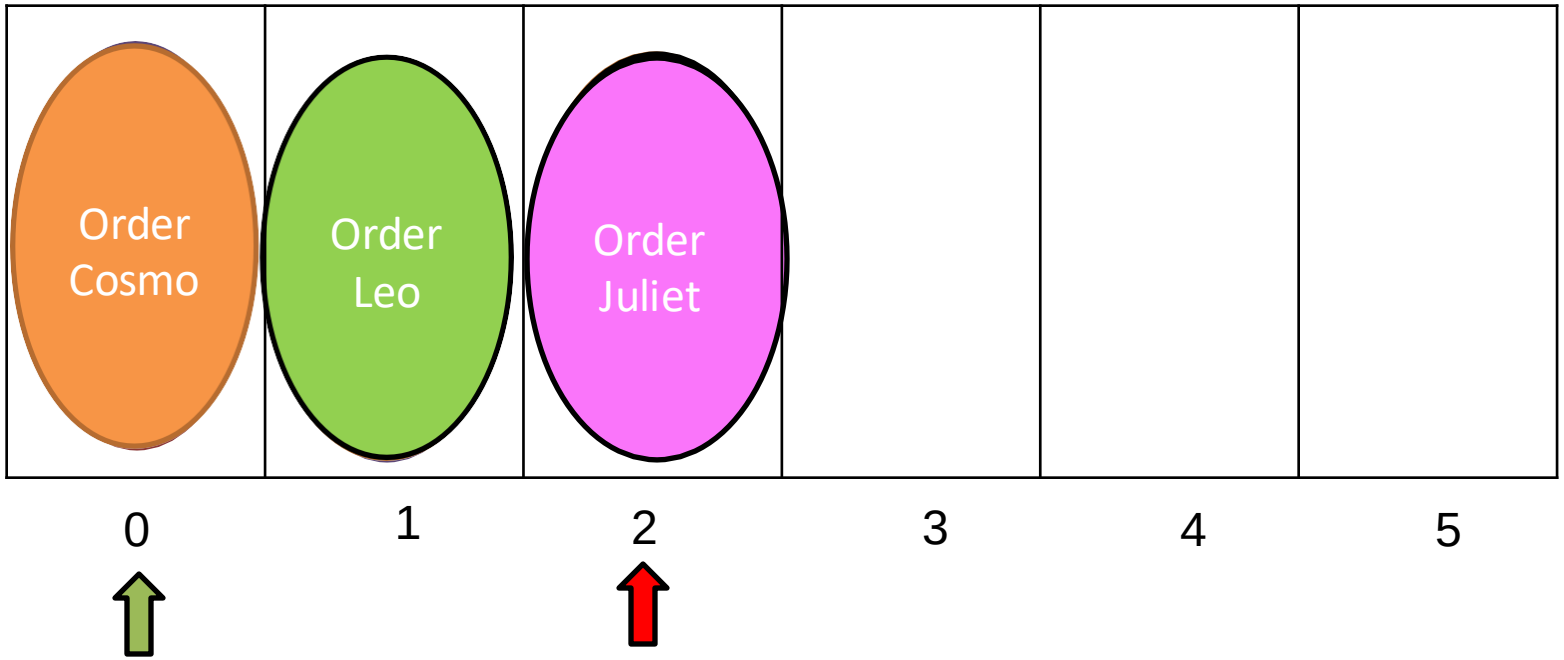



```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

```
capacity = 6    front = 0
size = 4        rear = 3
```

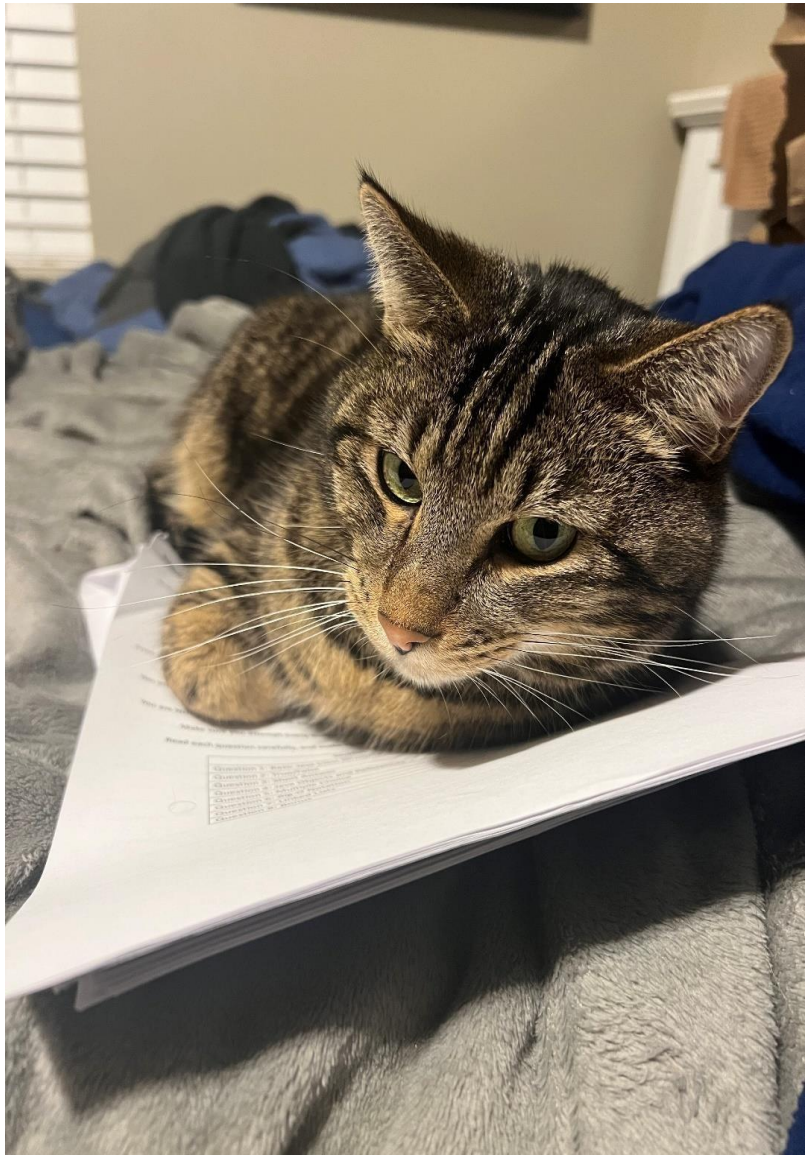
Today, we will be implementing a Queue with an Array.

Suppose that we have a queue that can hold 6 elements




```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Empty");
    return null;
  }

  Order temp = data[0];

  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }

  data[size-1] = null;
  size--;

  return temp;
```

```
capacity = 6    front = 0
size = 3        rear = 2
```



Runtime Analysis: Enqueue

```
public void enqueue(Order newOrder) {
  if (size == capacity) {
    System.out.println("Queue is full");
    return;
  }
  data[size] = newOrder;
  size ++;
```

Runtime Analysis: Enqueue

```
public void enqueue(Order newOrder) {
  if (size == capacity) {O(1)
    System.out.println("Queue is full"); O(1)
    return; O(1)
  }
  data[size] = newOrder; O(1)
  size ++; O(1)
```

Runtime Analysis: Enqueue

```
public void enqueue(Order newOrder) {
  if (size == capacity) {O(1)
    System.out.println("Queue is full"); O(1)
    return; O(1)
  }
  data[size] = newOrder; O(1)
  size ++; O(1)
```

Total running time:

O(1)

Runtime Analysis: Dequeue

```
public Order dequeue() {
  if (size == 0) {
          System.out.println(”Empty");
    return null;
  }
  Order temp = data[0];
  for (int i = 0; i < size - 1; i++) {
    data[i] = data[i + 1];
  }
  data[size-1] = null;
  size--;
  return temp;
```

Runtime Analysis: Dequeue

```
public Order dequeue() {
  if (size == 0) {O(1)
          System.out.println("Empty"); O(1)
    return null; O(1)
  }
  Order temp = data[0]; O(1)
  for (int i = 0; i < size - 1; i++) {O(N-1)
    data[i] = data[i + 1]; O(1)
  }
  data[size-1] = null; O(1)
  size--; O(1)
  return temp; O(1)
```

N = # elements in our queue

Runtime Analysis: Dequeue

```
public Order dequeue() {
  if (size == 0) {O(1)
          System.out.println("Empty"); O(1)
    return null; O(1)
  }
  Order temp = data[0]; O(1)
  for (int i = 0; i < size - 1; i++) {O(N-1)
    data[i] = data[i + 1]; O(1)
  }
  data[size-1] = null; O(1)
  size--; O(1)
  return temp; O(1)
```

N = # elements in our queue

Total running time:

O(N)

Runtime Analysis: Dequeue

Total running time: O(N)

```
public Order dequeue() {
  if (size == 0) {O(1)
        System.out.println("Empty"); O(1)
    return null; O(1)
  }
  Order temp = data[0]; O(1)
  for (int i = 0; i < size - 1; i++) {O(N-1)
    data[i] = data[i + 1]; O(1)
  }
  data[size-1] = null; O(1)
  size--; O(1)
  return temp; O(1)
```

This algorithm works fine, but the issue is that shifting data can be costly

(think about if this queue has 1,000,000 things in it→ we must shift 999,999 elements!)

Runtime Analysis: Dequeue

Total running time: O(N)

```
public Order dequeue() {
  if (size == 0) {O(1)
        System.out.println("Empty"); O(1)
    return null; O(1)
  }
  Order temp = data[0]; O(1)
  for (int i = 0; i < size - 1; i++) {O(N-1)
    data[i] = data[i + 1]; O(1)
  }
  data[size-1] = null; O(1)
  size--; O(1)
  return temp; O(1)
```

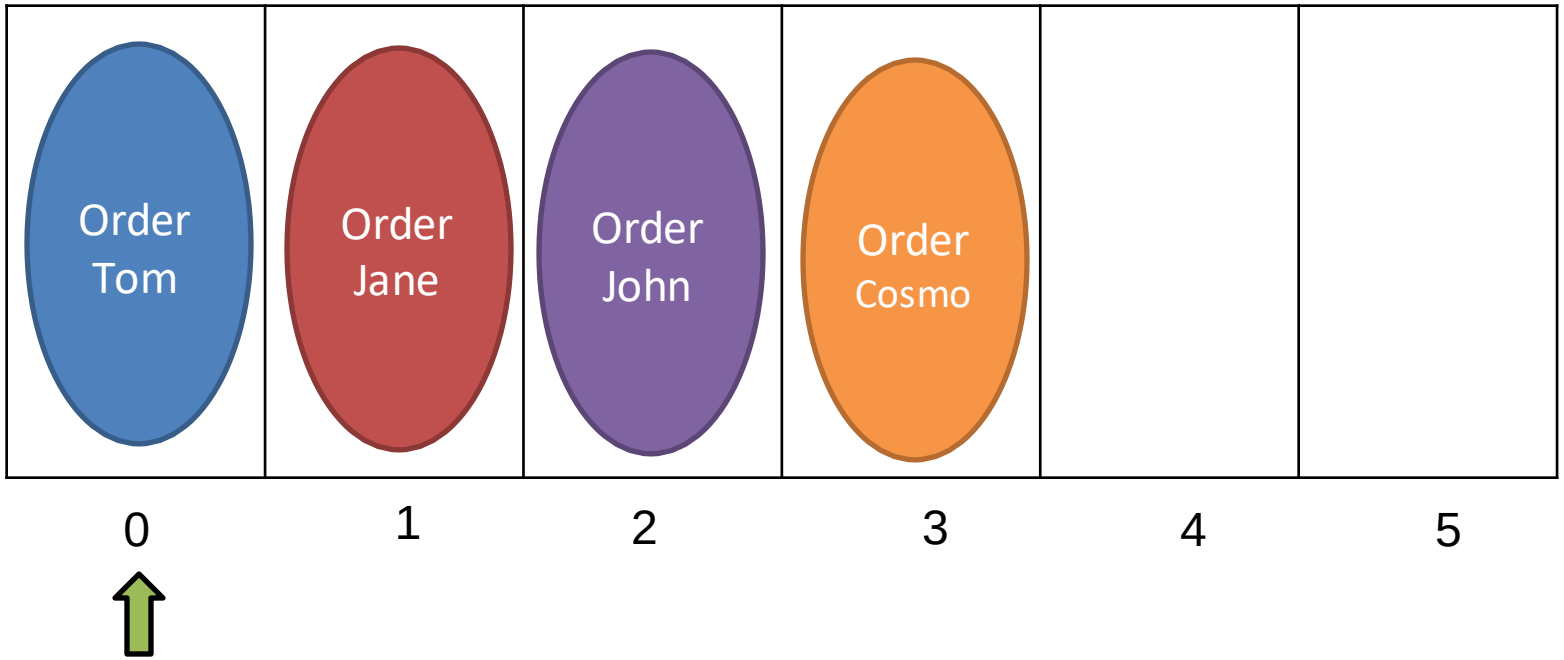
This algorithm works fine, but the issue is that shifting data can be costly

(think about if this queue has 1,000,000 things in it→ we must shift 999,999 elements!)

We prefer: Constant Time O(1) > Linear Time O(N)

How to improve our queue?

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



We are going to make use of the **modulus** (%) operator !

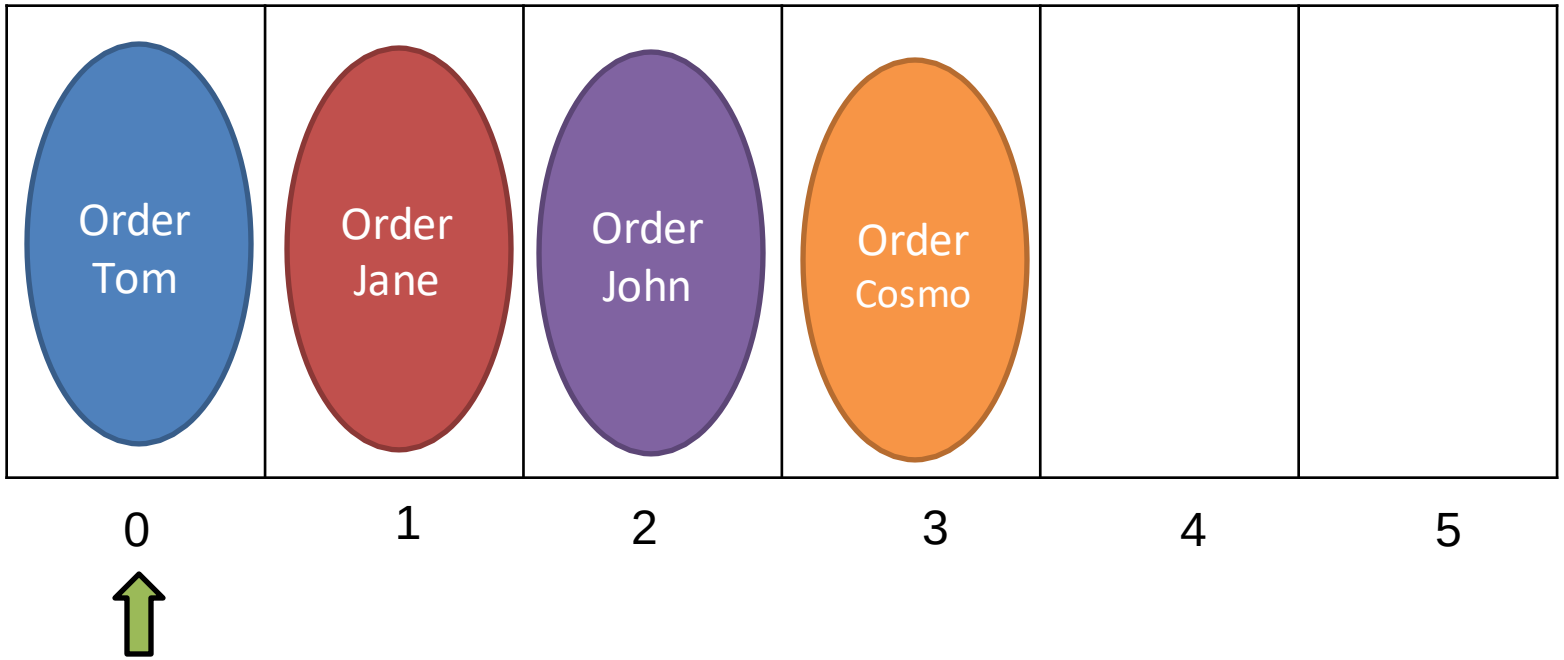
10 % 6 = 4

3 % 6 = 3

6 % 6 = 0

```
capacity = 6     front = 0
size = 4
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



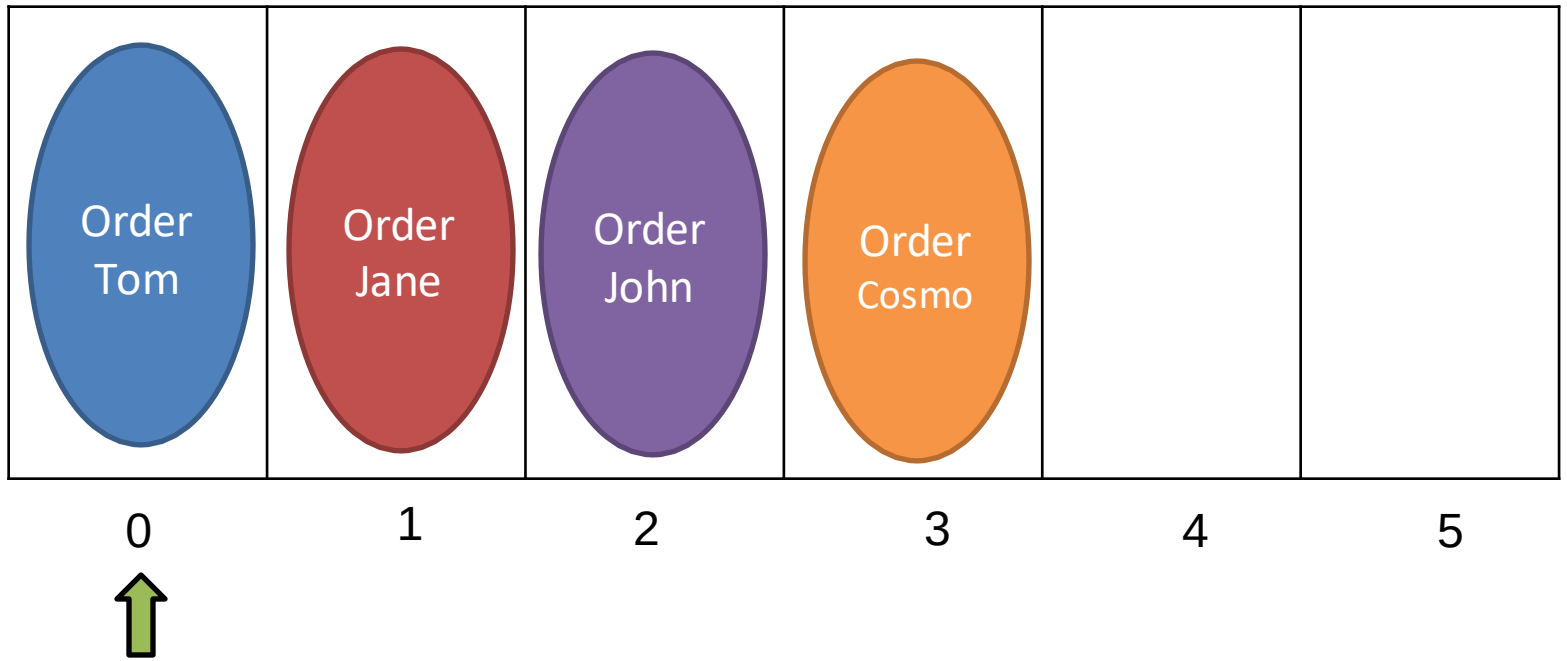

```
capacity = 6     front = 0
size = 4
```

Let's **enqueue**

Here is the formula for determining where to insert the new element

```
insert_spot = (front + size) % 6
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
capacity = 6      front = 0
size = 4          insert_spot = 4
```

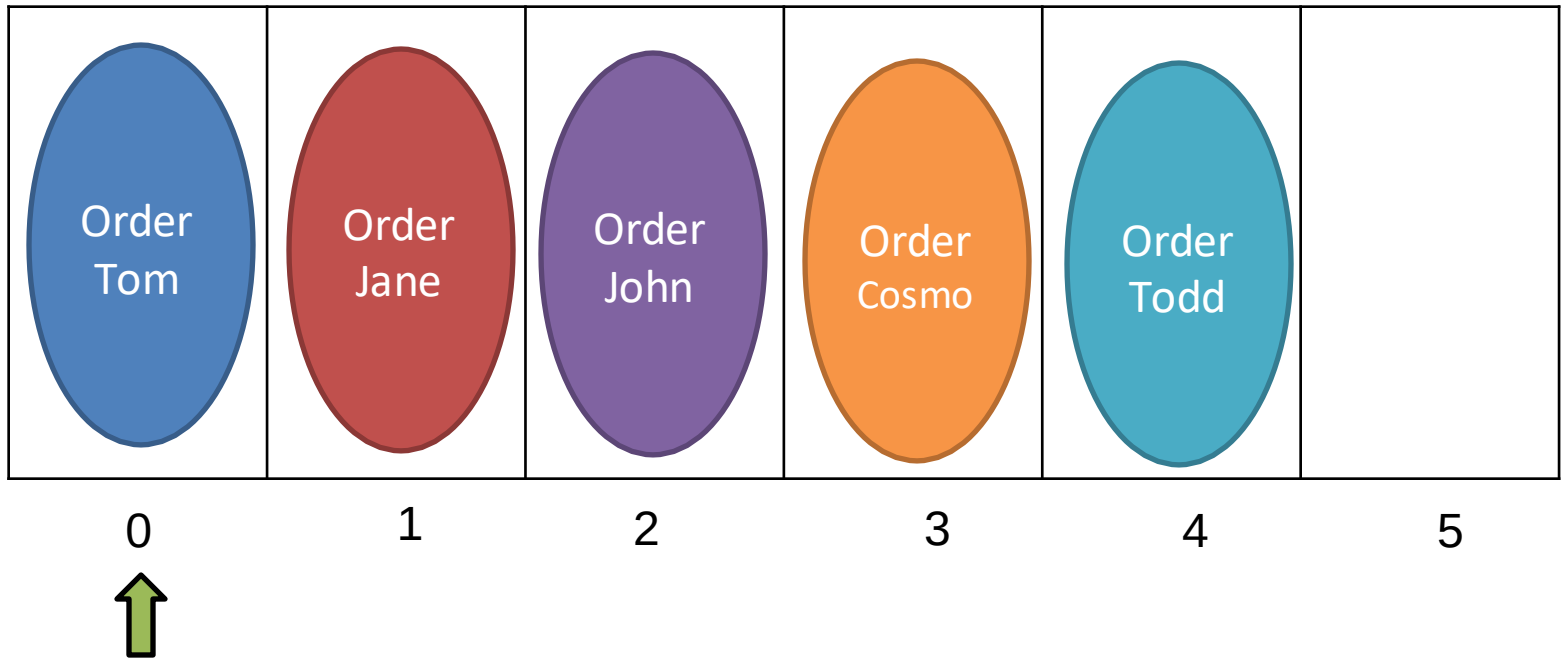
Let's **enqueue**

Here is the formula for determining where to insert the new element

```
insert_spot = (front + size) % 6
```

(0 + 4) % 6 = **Insert at spot 4**

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
capacity = 6      front = 0
size = 4          insert_spot = 4
```

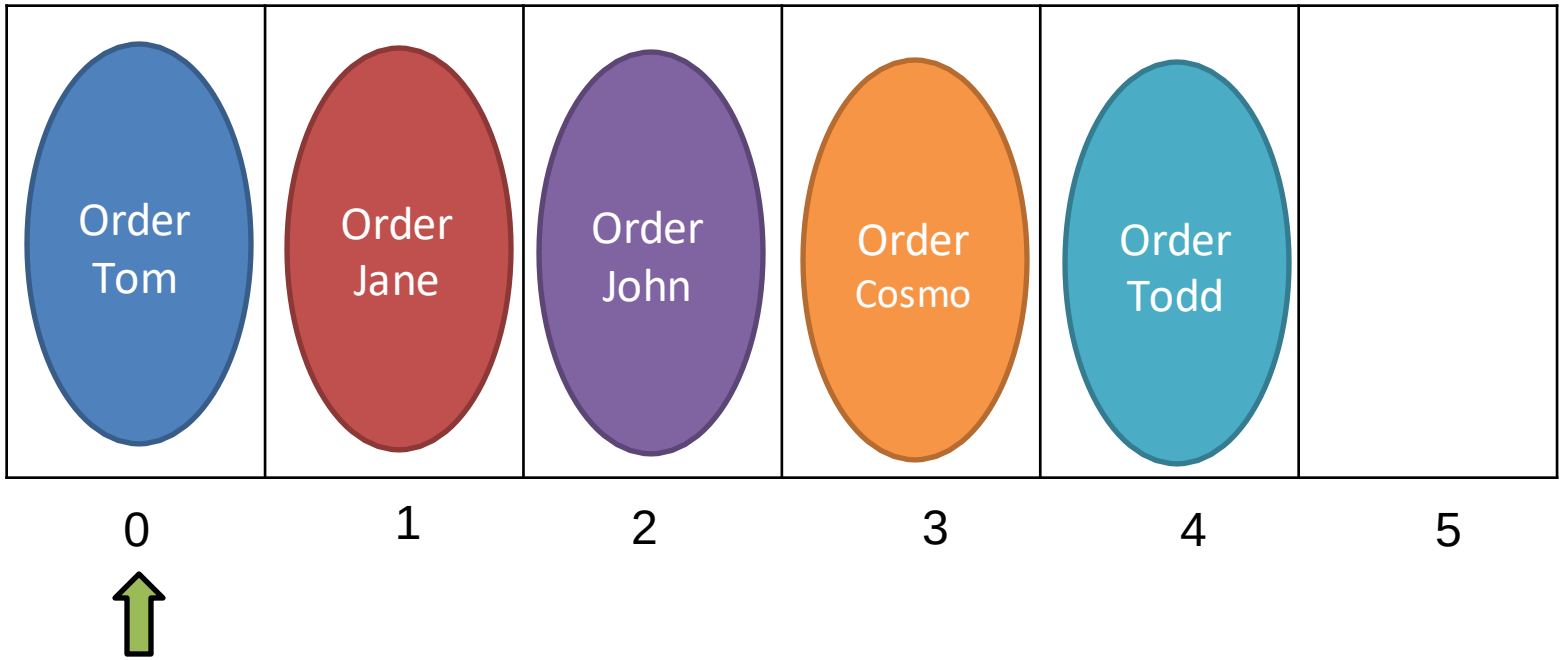
Let's **enqueue**

Here is the formula for determining where to insert the new element

```
insert_spot = (front + size) % 6
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

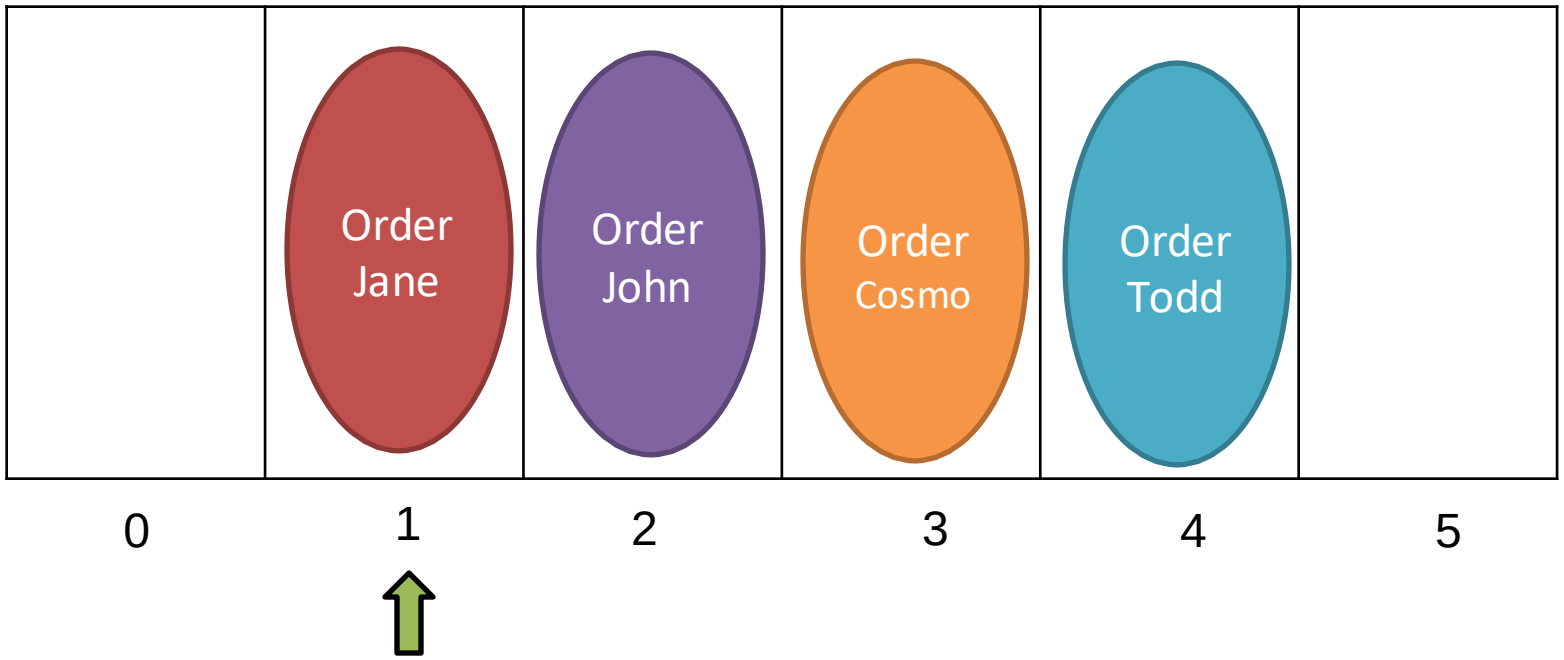
Let's **dequeue**




```
data[front] = null
```

```
capacity = 6      front = 0
size = 4          insert_spot = 4
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Let's **dequeue**

```
data[front] = null
```

```
front = (front + 1) % 6
```

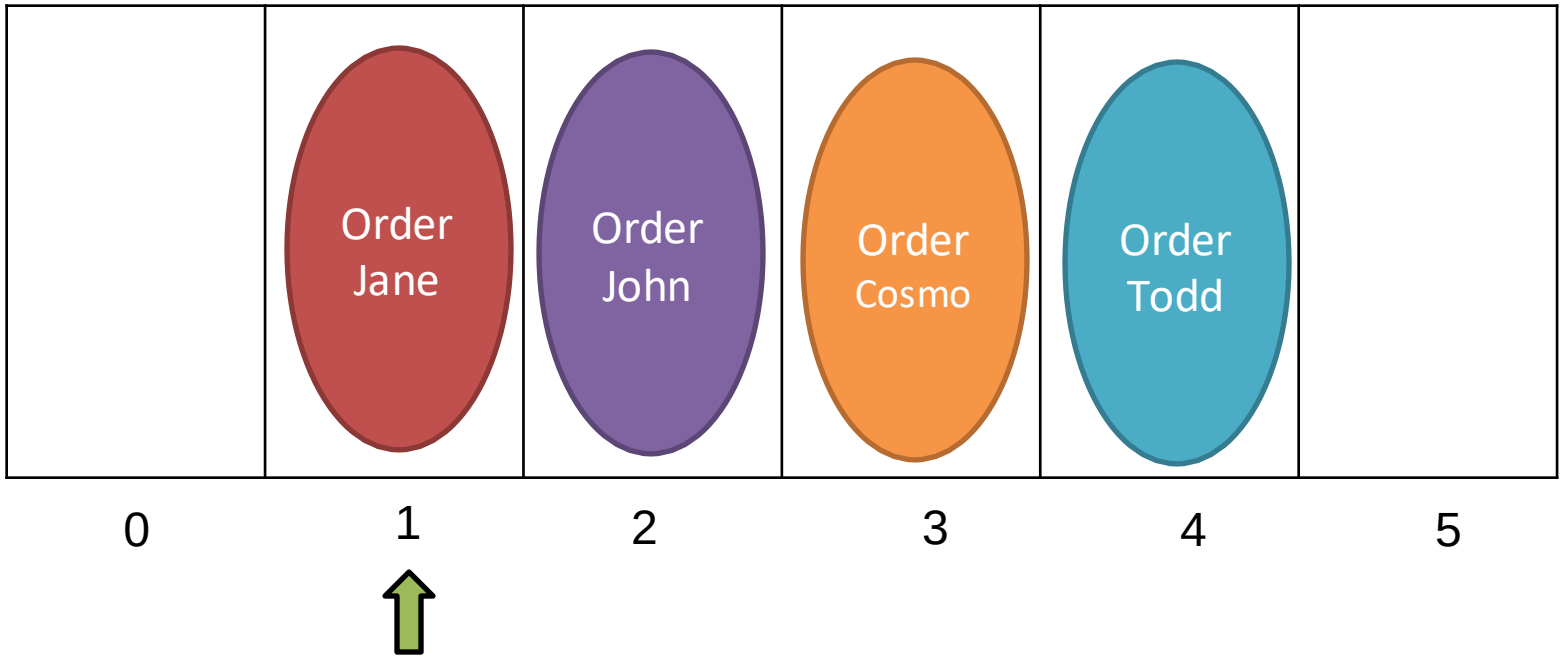
move the front pointer to the next element

= (0 + 1) % 6 = **1**

```
capacity = 6
size = 4
```

```
front = 1
insert_spot = 4
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Let's **dequeue (again)**

```
data[front] = null
```

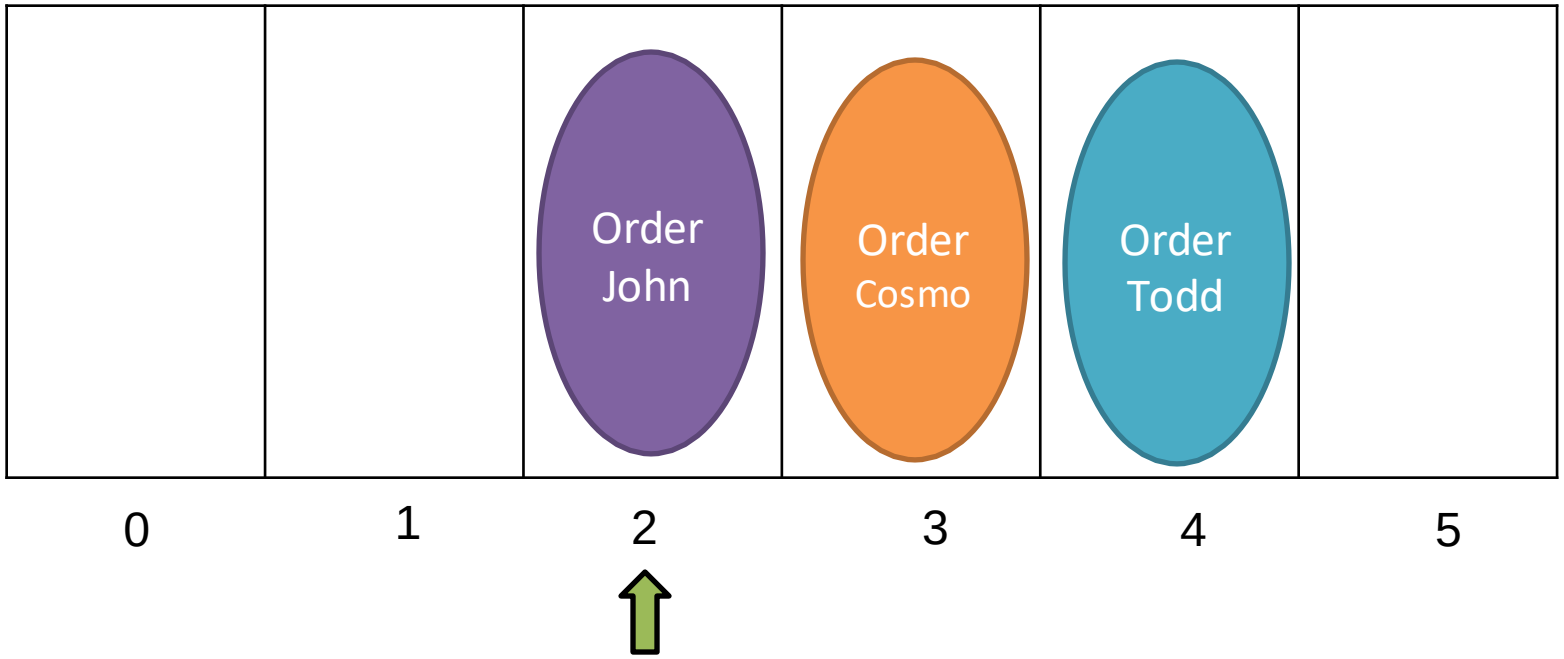
```
front = (front + 1) % 6
```

move the front pointer to the next element

= (0 + 1) % 6 = **1**

```
capacity = 6      front = 1
size = 4          insert_spot = 4
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Let's **dequeue (again)**

```
data[front] = null
```

```
front = (front + 1) % 6
```

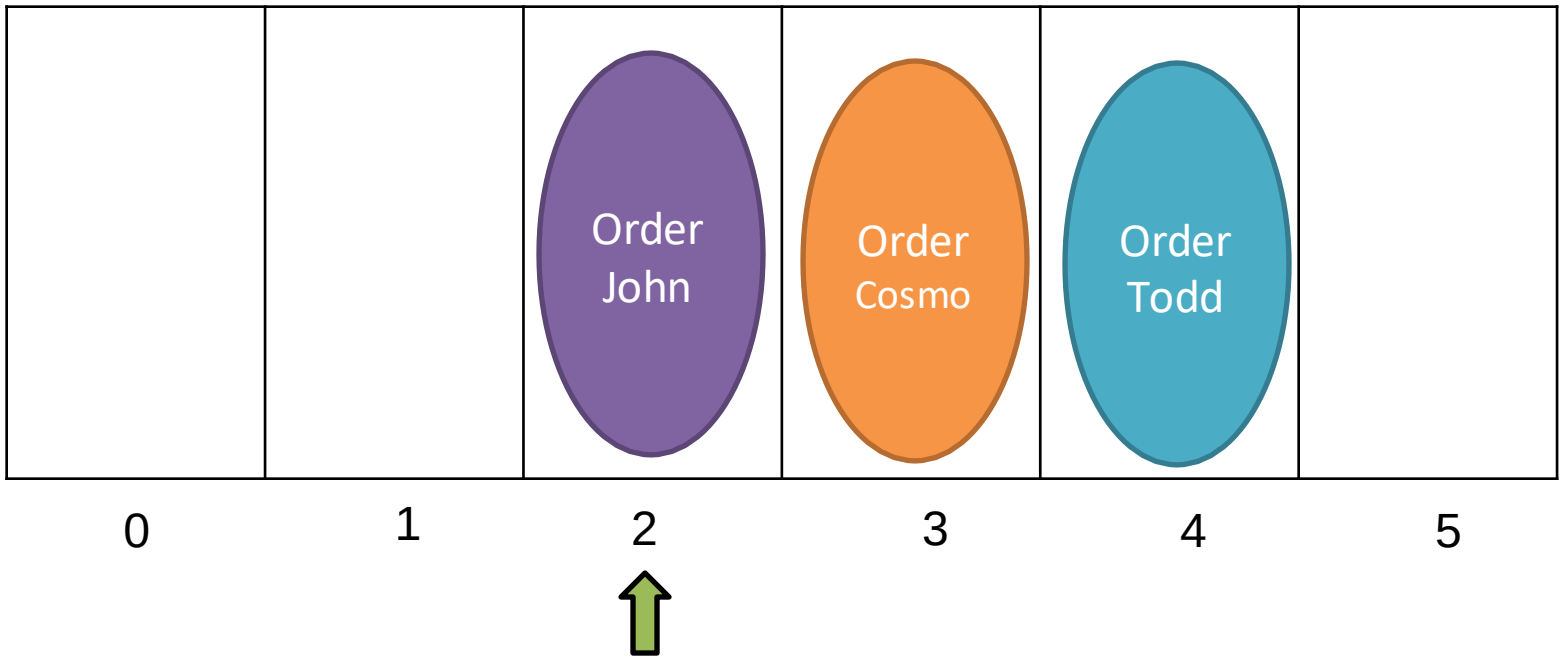
move the front pointer to the next element

= (1 + 1) % 6 = **2**

```
capacity = 6
size = 3
```

```
front = 2
insert_spot = 4
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
capacity = 6
size = 3
```

```
front = 2
insert_spot = 5
```

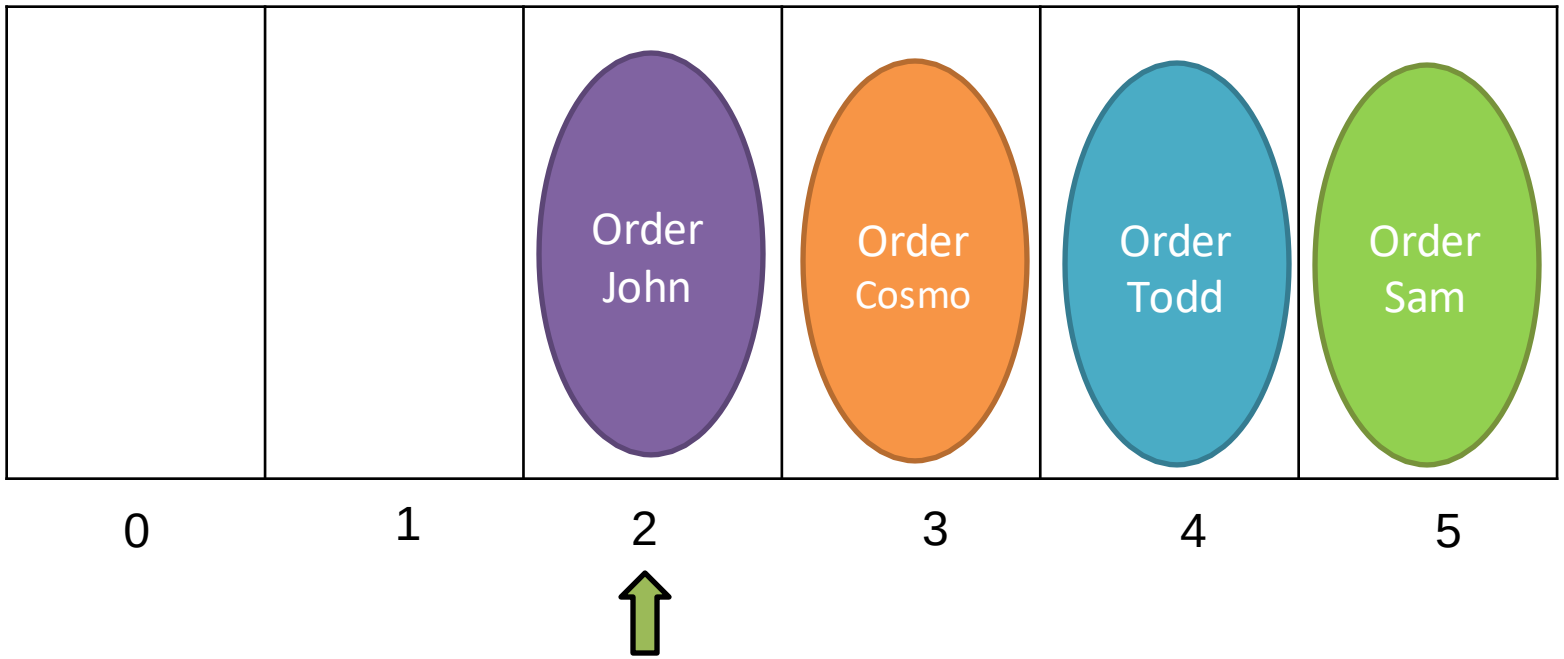
Let's **enqueue (again)**

```
insert_spot = (front + size) % 6
```

```
insert_spot = (2 + 3) % 6
```

5%6 = **5**

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



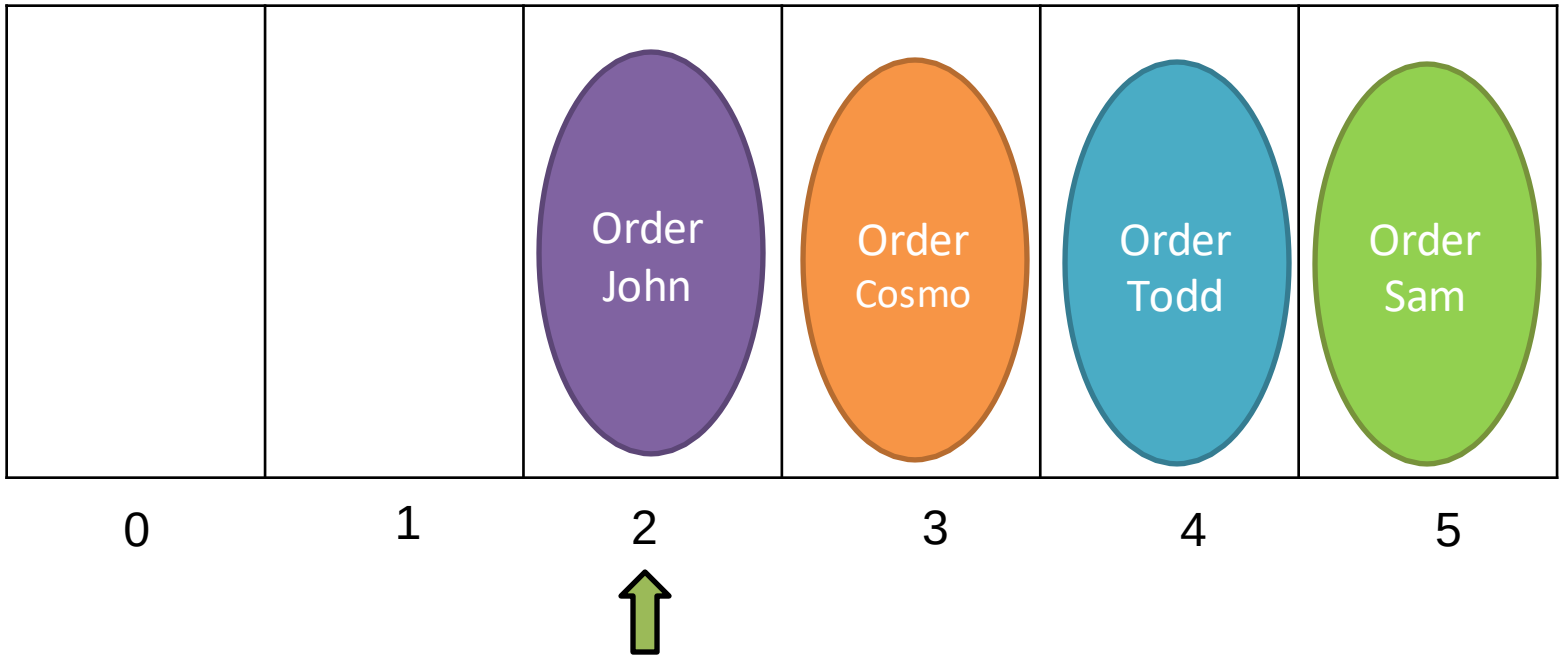
Let's **enqueue (again)**

```
insert_spot = (front + size) % 6

insert_spot = (2 + 3) % 6
```

5%6 = **5**

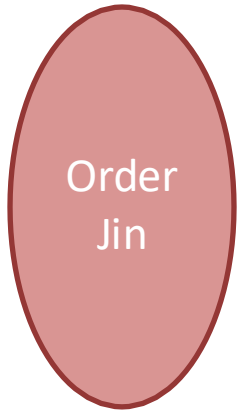
```
capacity = 6      front = 2
size = 4          insert_spot = 5
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



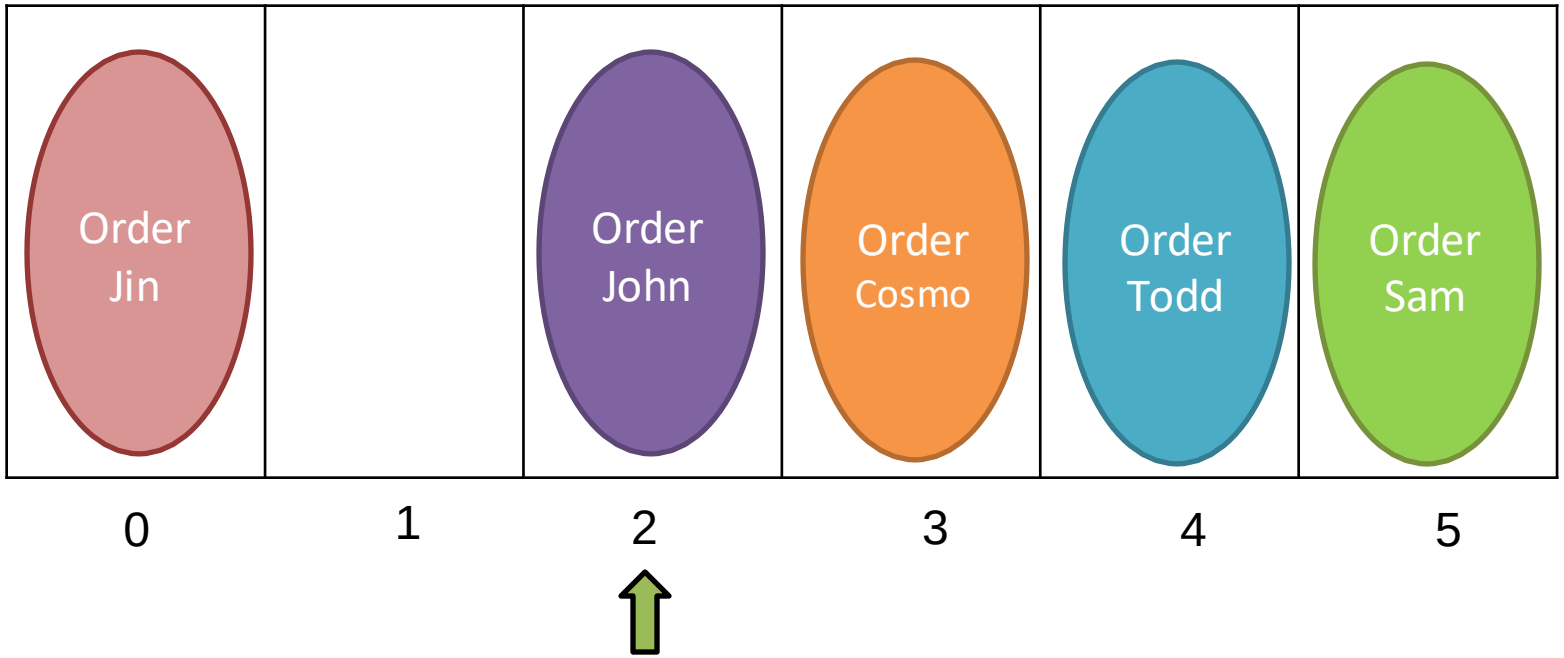
Let's **enqueue (<u>again</u>)**

```
insert_spot = (front + size) % 6

(2 + 4) % 6  = 0
```




```
capacity = 6      front = 2
size = 4          insert_spot = 0
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



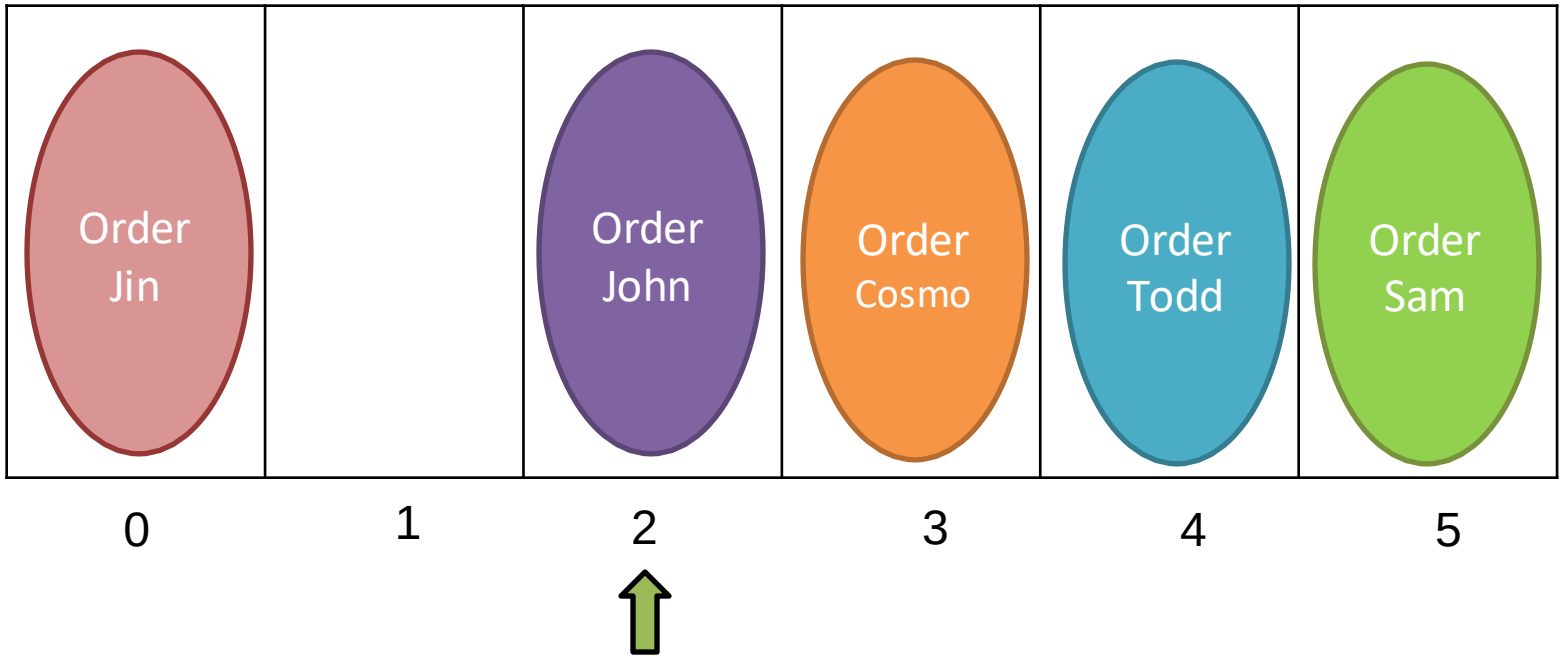

```
capacity = 6      front = 2
size = 4          insert_spot = 0
```

Let's **enqueue (<u>again</u>)**

```
insert_spot = (front + size) % 6

(2 + 4) % 6  = 0
```

The modulus operator allows us to "**wrap around**" in our array!

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
capacity = 6
size = 4
```

```
front = 2
insert_spot = 0
```

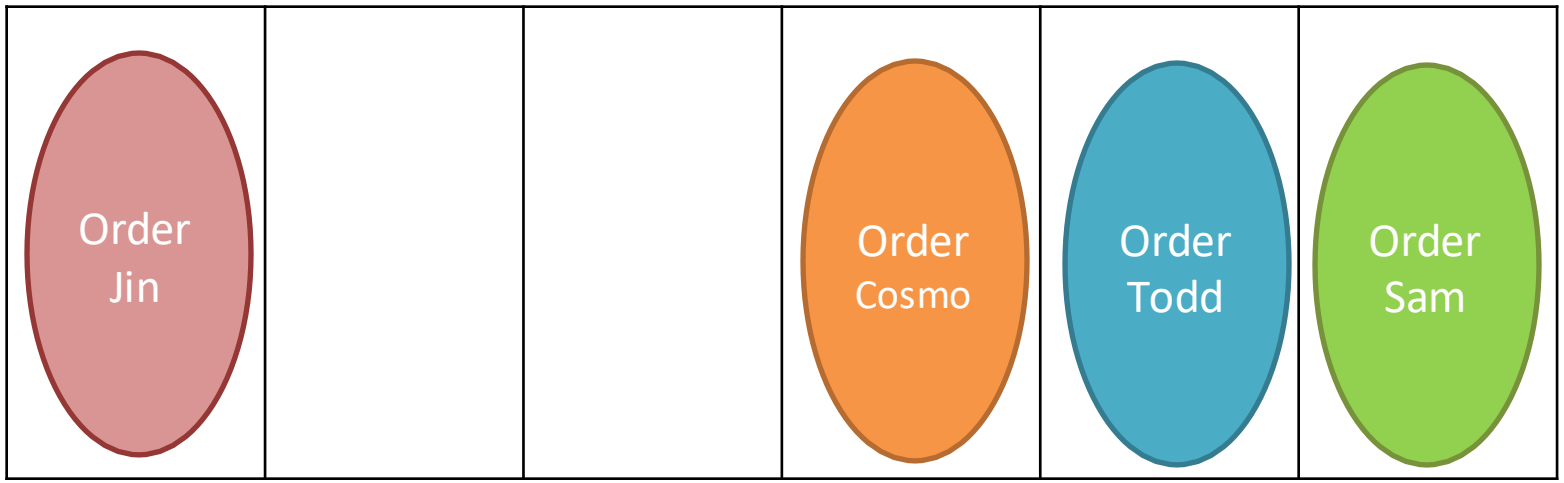
Let's **dequqe (again)**

```
data[front] = null
```

```
front = (front + 1) % 6
```

The modulus operator allows us to "**wrap around**" in our array!

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
capacity = 6      front = 3
size = 4          insert_spot = 0
```

Let's **dequqe (<u>again</u>)**

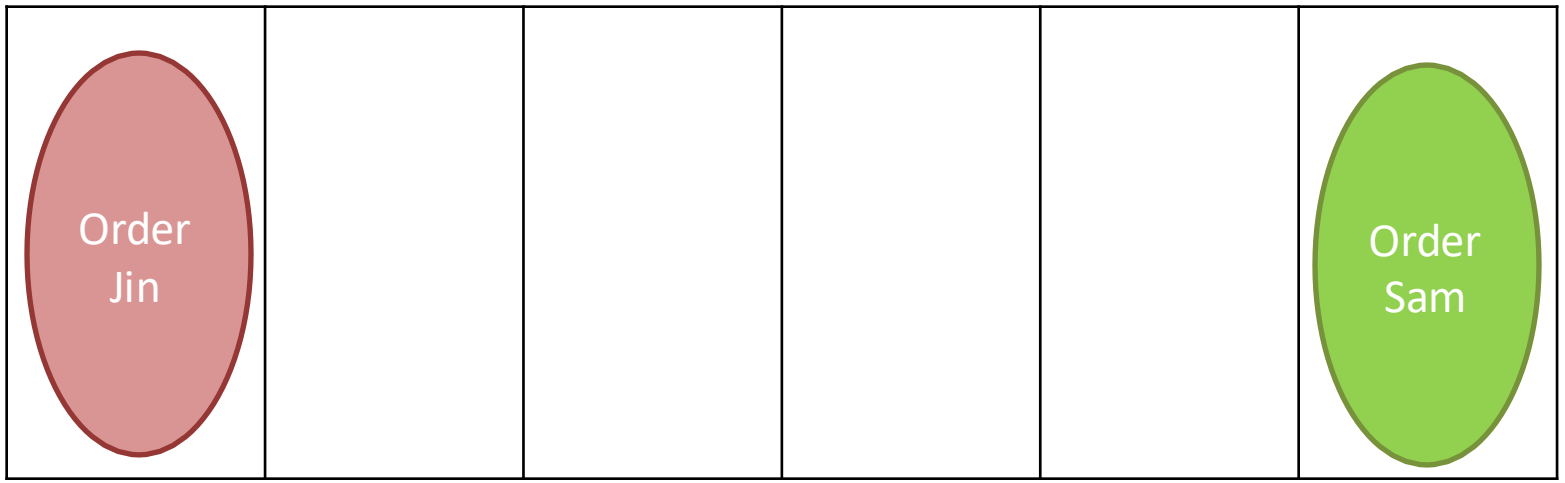
```
data[front] = null
```

```
front = (front + 1) % 6
```

(2+1) % 6 **= 3**

The modulus operator allows us to "**wrap around**" in our array!

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

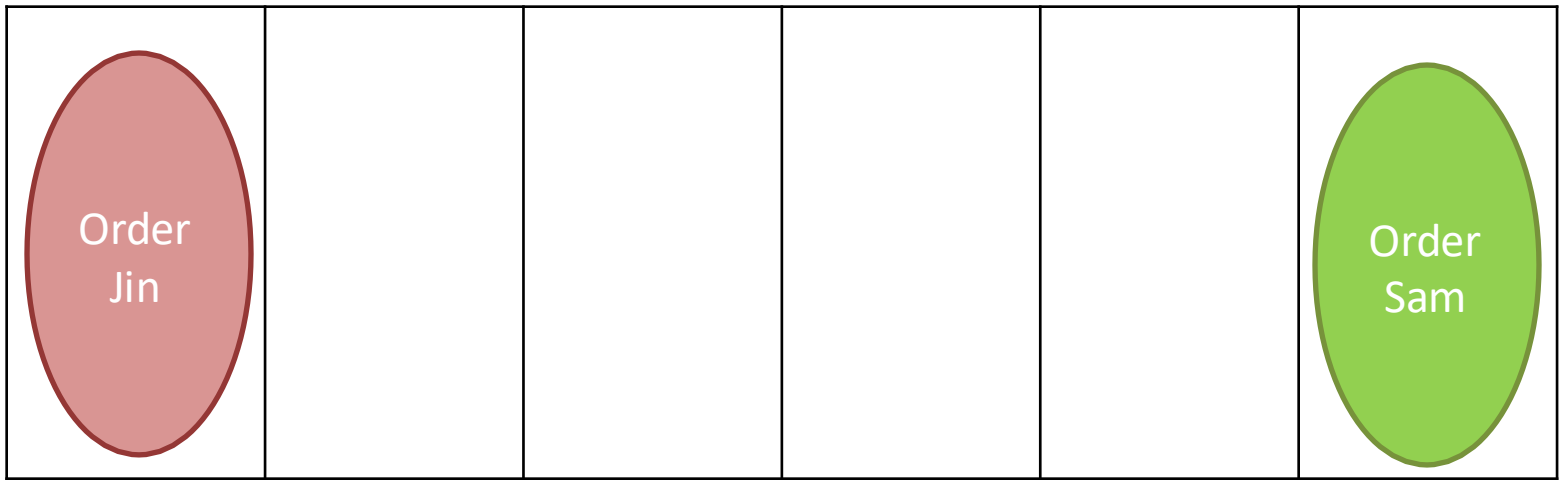


Let's **dequqe (again)**

```
data[front] = null

front = (front + 1) % 6
```

```
capacity = 6
size = 2
front = 5
insert_spot = 0
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Let's **dequqe (again)**

```
data[front] = null
```

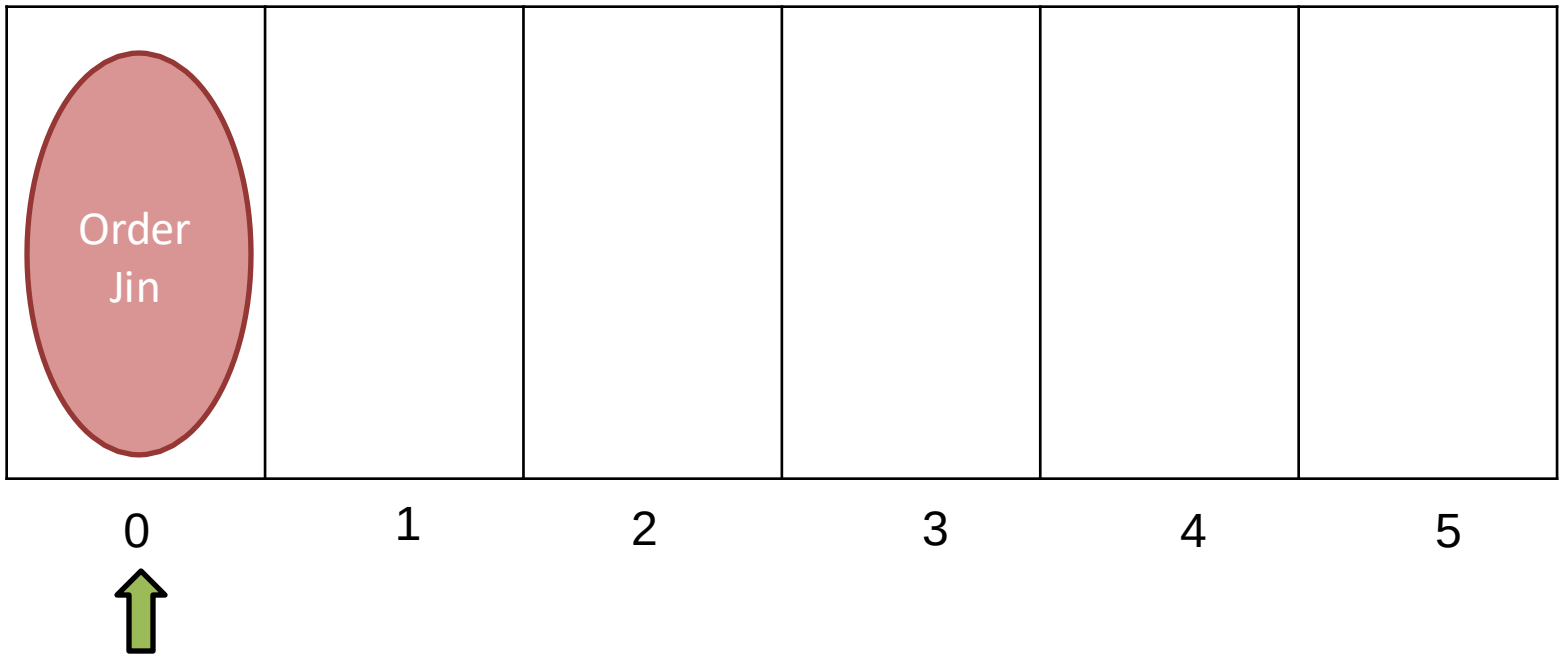
```
front = (front + 1) % 6
```

Front = (5 + 1) % 6 **= 0**

```
capacity = 6
size = 2
```

```
front = 5
insert_spot = 0
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



Let's **dequqe (again)**

```
data[front] = null
```

```
front = (front + 1) % 6
```

Front = (5 + 1) % 6 **= 0**

```
capacity = 6
size = 1
```

```
front = 0
insert_spot = 0
```

```
public void enqueue(Order newOrder) {
  if (size == capacity) {
    System.out.println("Queue is full");
    return;
}
else {
  int insert_spot = (front + size) % capacity;
  data[insert_spot] = newOrder;
  size++;
  System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot + "\n");
  }
}
```

```
public Order dequeue() {
  if (size == 0) {
    System.out.println("Queue is empty");
    return null;
  }
  Order temp = data[front];


  data[front] = null;
  front = (front + 1) % capacity;
  size--;

  return temp;
}
```

Queue Runtime Analysis

```
public QueueLinkedList() {
        this.orders = new LinkedList<Order>();
        this.size = 0;
}
```

```
public QueueArray2() {
        this.orders = new Order[6];
        this.size = 0;
        this.front = 0;
        this.capacity = this.orders.length; //6
}
```

	Linked List	Array
Creation	O(1)	O(n)
Enqueue		
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public QueueLinkedList() {
        this.orders = new LinkedList<Order>();
        this.size = 0;
}
```

O(1)

```
public QueueArray2() {
        this.orders = new Order[6];
        this.size = 0;
        this.front = 0;
        this.capacity = this.orders.length; //6
}
```

O(n), n = | array |

	Linked List	Array
Creation	O(1)	O(n)
Enqueue		
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public void enqueue(Order newOrder) {

        this.orders.addLast(newOrder);
        this.size++;

}
```

```
public void enqueue(Order newOrder) {

        if(this.size == this.capacity) {
                System.out.println("Error... queue is full");
                return;
        }

        int insert_spot = (front + size) % capacity;
        this.orders[insert_spot] = newOrder;

        this.size++;
        System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot);
}
```

	Linked List	Array
Creation	O(1)	O(n)
Enqueue		
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public void enqueue(Order newOrder) {

        this.orders.addLast(newOrder);
        this.size++;

}
```

O(1)
O(1)
O(1)

```
public void enqueue(Order newOrder) {

        if(this.size == this.capacity) {
                System.out.println("Error... queue is full");
                return;
        }

        int insert_spot = (front + size) % capacity;
        this.orders[insert_spot] = newOrder;

        this.size++;
        System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot);
}
```

O(1)
O(1)
O(1)
O(1)
O(1)

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public Order dequeue() {
        if(this.size != 0) {
                Order removed = this.orders.removeFirst();
                System.out.println(removed.getName() + "'s order
                size--;
                return removed;
        }
        else {
                return null;
        }
}
```

```
public void dequeue() {

        if(this.size == 0) {
                System.out.println("Error... queue is empty");
                return;
        }
        else {
                Order o = this.orders[front];
                this.orders[front] = null;
                front = (front + 1) % capacity;
                this.size--;
                System.out.println(o.getName() + "'s order was removed");

        }

}
```

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public Order dequeue() {
        if(this.size != 0) {
                Order removed = this.orders.removeFirst();
                System.out.println(removed.getName() + "'s order
                size--;
                return removed;
        }
        else {
                return null;
        }
}
```

O(1)

O(1)

```
public void dequeue() {

        if(this.size == 0) {
                System.out.println("Error... queue is empty");
                return;
        }
        else {
                Order o = this.orders[front];
                this.orders[front] = null;
                front = (front + 1) % capacity;
                this.size--;
                System.out.println(o.getName() + "'s order was removed");

        }

}
```

O(1)

O(1)

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek		
Print Queue		

Queue Runtime Analysis

```
return this.orders.getFirst()
```

```
return this.orders[front]
```

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek		
Print Queue		

Queue Runtime Analysis

```
return this.orders.getFirst()
```
O(1)

```
return this.orders[front]
```
O(1)

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue		

Queue Runtime Analysis

```
public void printQueue() {
        int counter = 1;
        for(Order each_order: this.orders) {
                each_order.printOrder(counter);
                counter++;
        }
}
```

```
public void printQueue() {

        int start = front;
        int counter = 1;
        int n = 0;
        while(n != this.size) {
                System.out.println(counter + ". " + this.orders[start].getName());
                start = (start + 1) % capacity;
                counter++;
                n++;
        }


}
```

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue		

Queue Runtime Analysis

```
public void printQueue() {
        int counter = 1;
        for(Order each_order: this.orders) {
            each_order.printOrder(counter);
            counter++;
        }
}
```

O(1) O(n) O(1) O(1)

n = # of elements in queue

```
public void printQueue() {

        int start = front;
        int counter = 1;
        int n = 0;
        while(n != this.size) {
                System.out.println(counter + ". " + this.orders[start].getName());
                start = (start + 1) % capacity;
                counter++;
                n++;
        }


}
```

O(1) O(1) O(1) O(n) O(1)

n = # of elements in queue

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Takeaway: Adding and removing elements from a **queue** runs in constant time (`O(1)`)

(FIFO)

Takeaway: Adding and removing elements from a **stack** runs in constant time (`O(1)`)

(LIFO)

Queue Runtime Analysis

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Stack Runtime Analysis

	Linked List	Array
Creation	O(1)	O(n)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)