CSCI 132:
Basic Data Structures and Algorithms

Queues (Array Implementation)

Reese Pearsall

Spring 2023

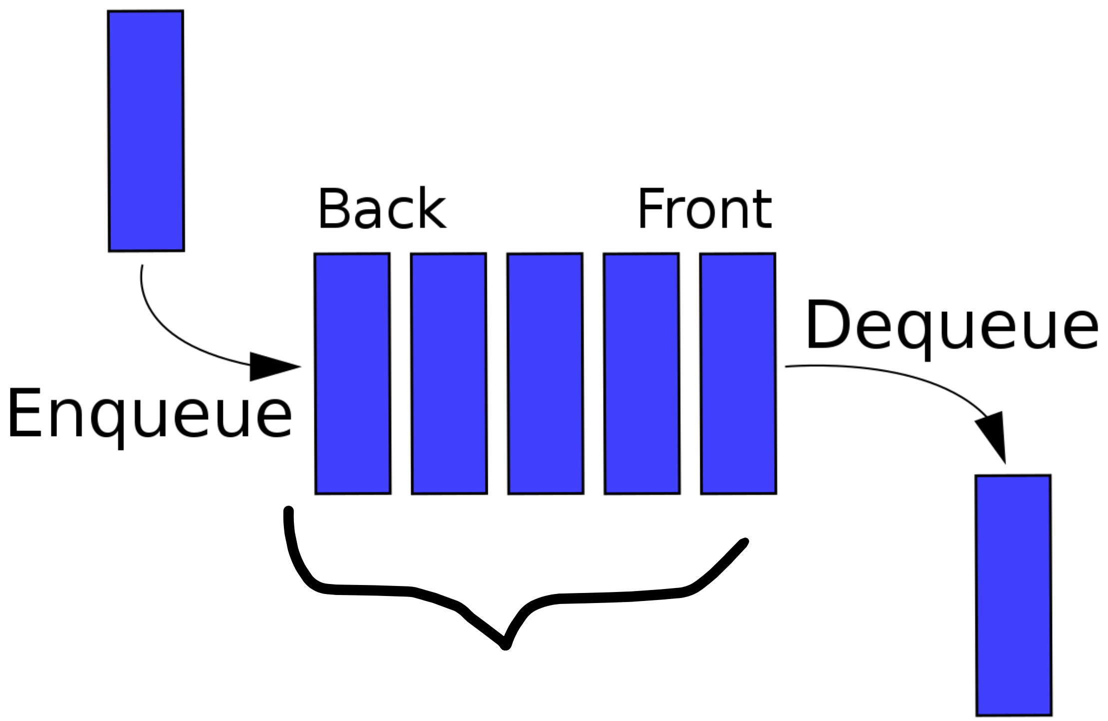
Announcements

Lab 9 due tomorrow @ 11:59 PM

Program 4 due Sunday 4/16*

Next Monday's lecture (4/10) will be asynchronous (don't come to class)

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

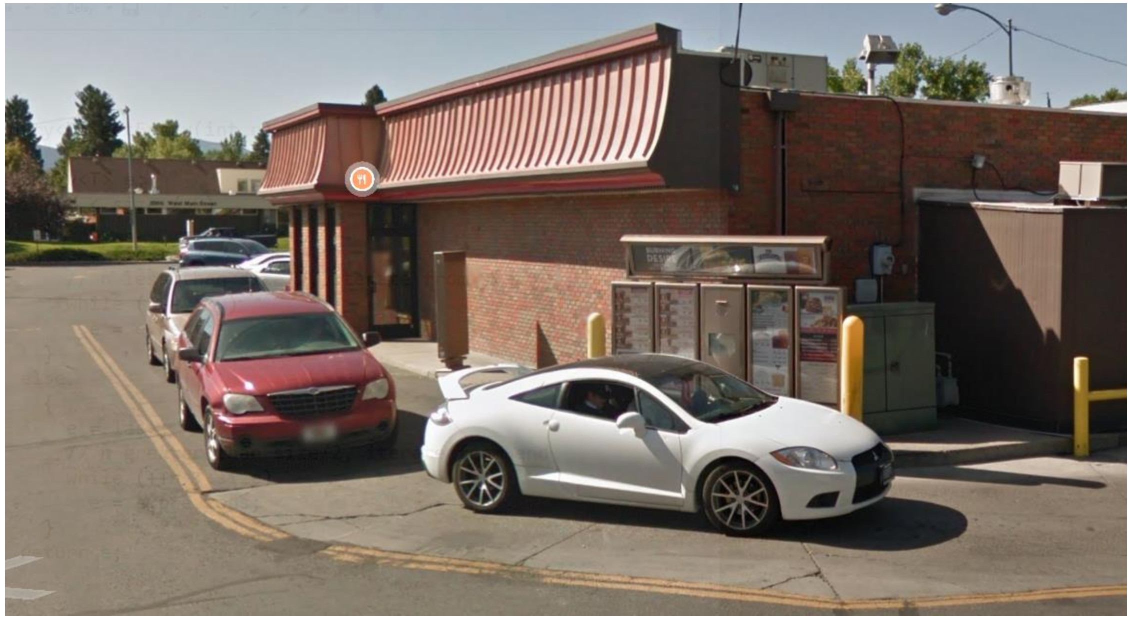


Once again, we need a data structure to hold the data of the queue

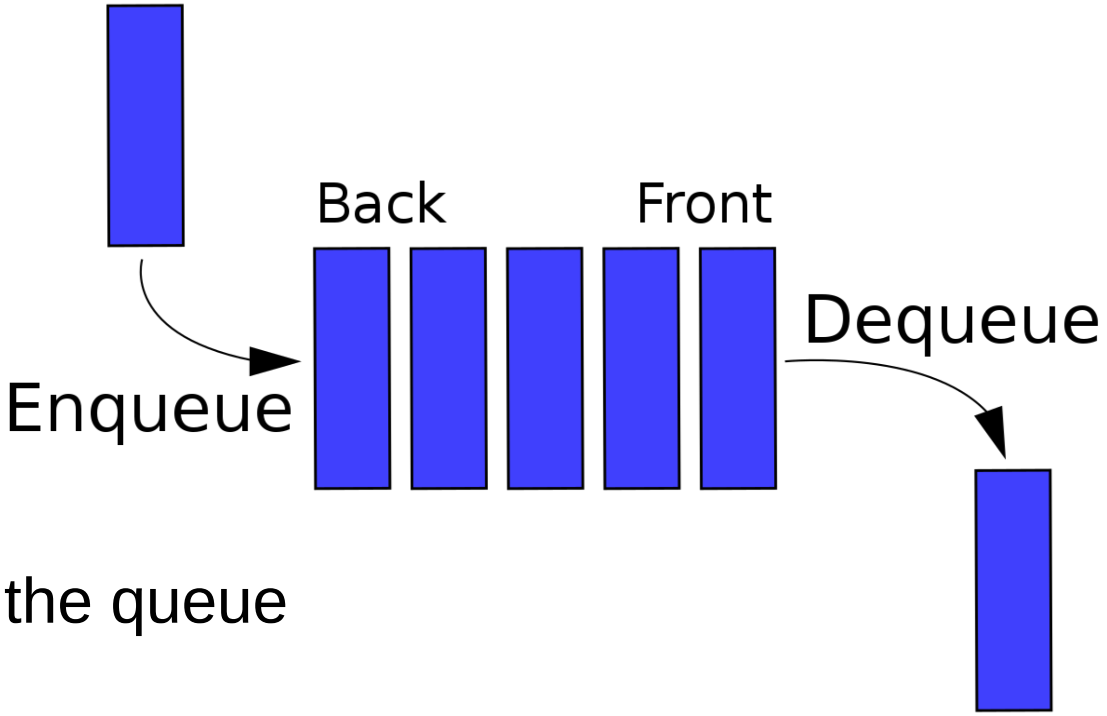
- Linked List (today)
- Array (tomorrow)

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

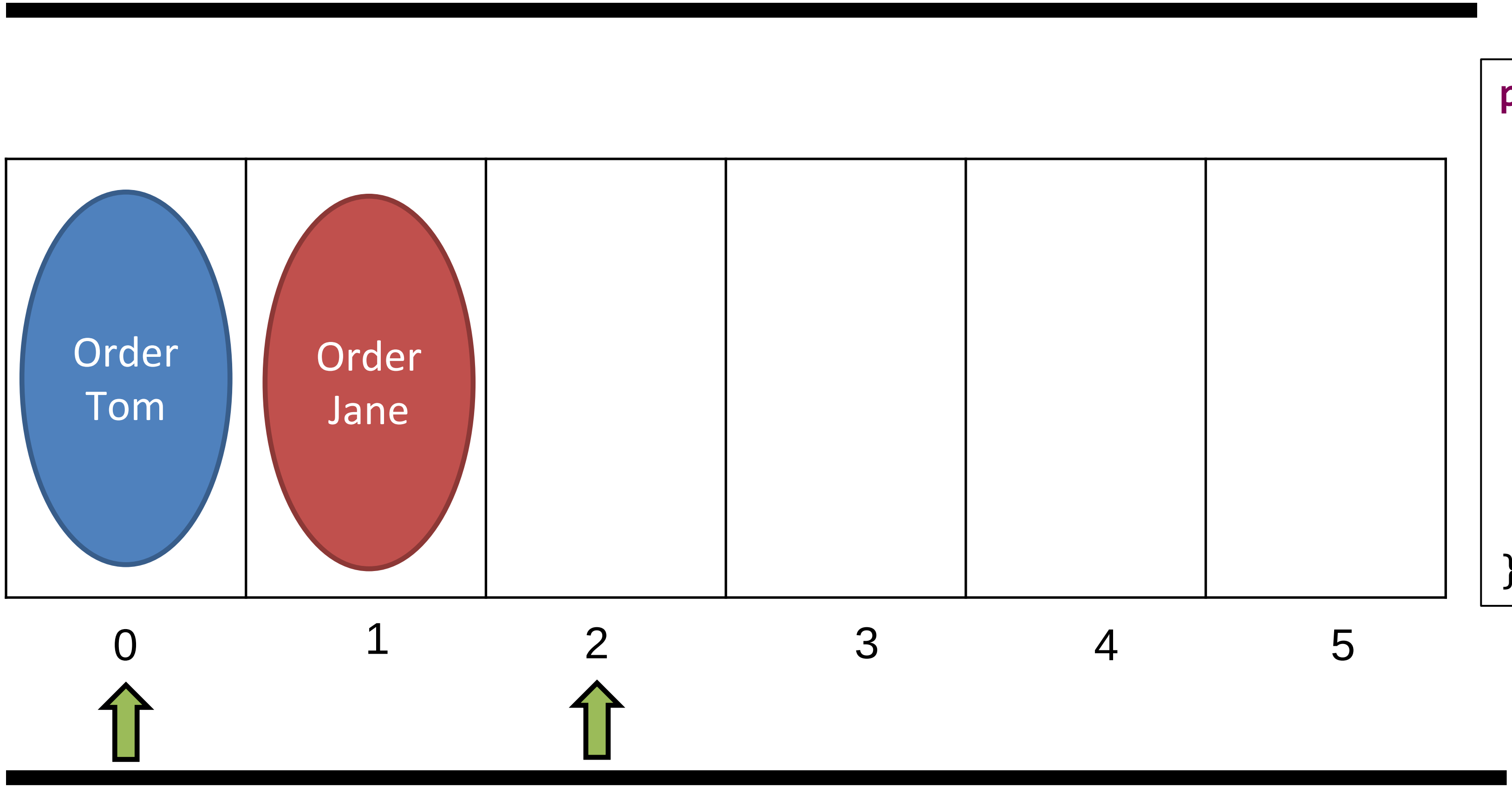
Dequeue- Remove element from the queue

** 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

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

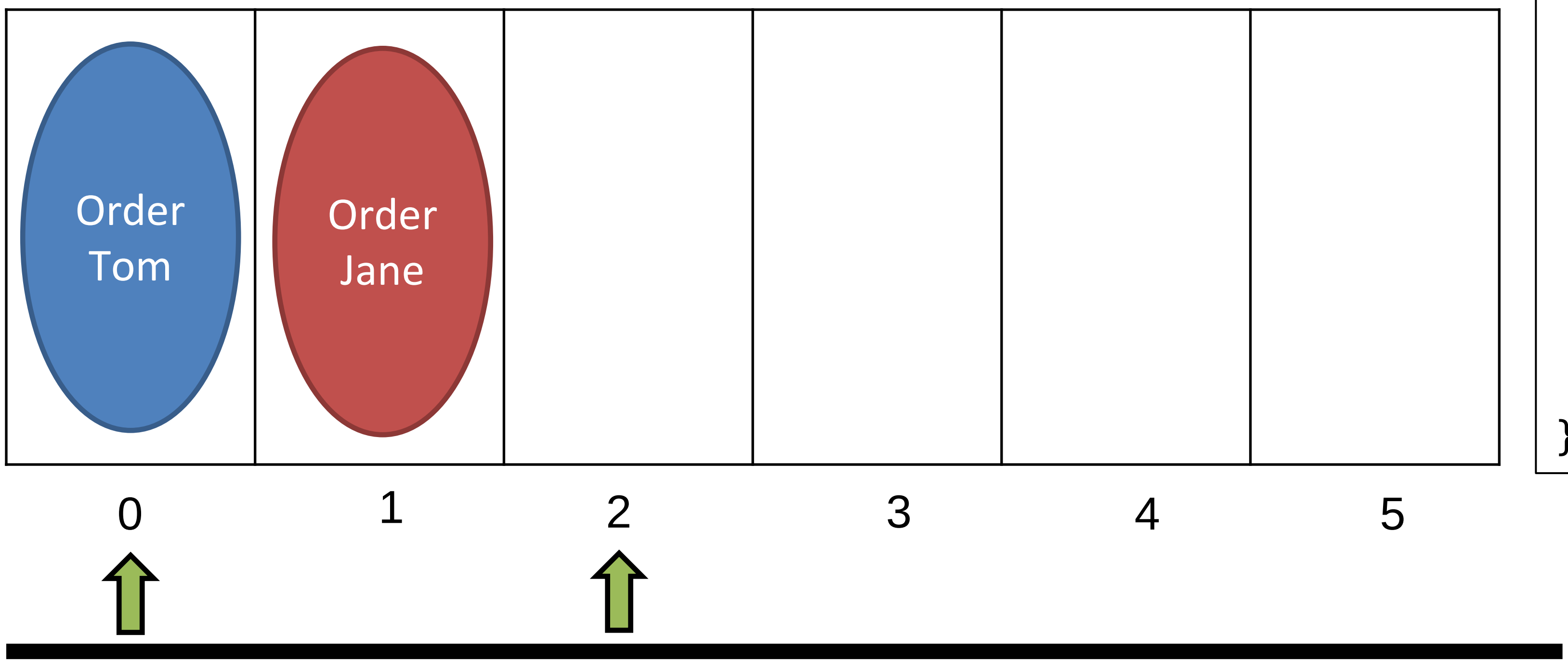



```
public void enqueue(Order newOrder) {
  if(rear == capacity) {
    System.out.println("full...");
    return;
  }
  else {
    this.data[rear] = newOrder;
    rear++;
    this.size++;
  }
}
```

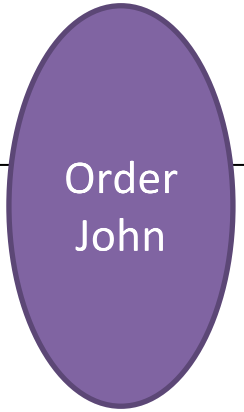
```
capacity = 6    front = 0
size = 2        rear = 2
```

In our previous array implementation, `rear` pointed to the next <u>empty spot</u> in the array

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

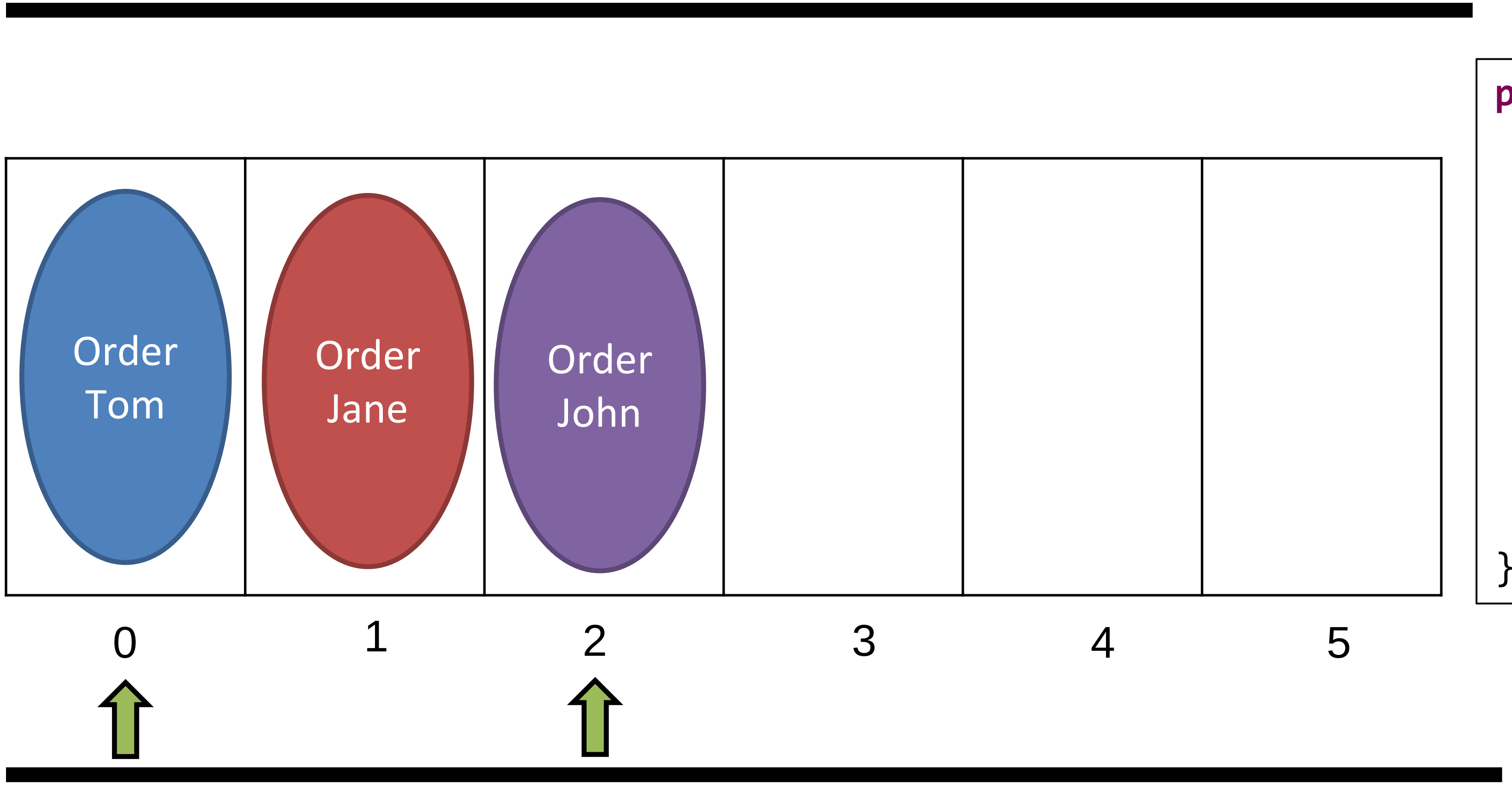



```java
public void enqueue(Order newOrder) {
  if(rear == capacity) {
    System.out.println("full...");
    return;
  }
  else {
    this.data[rear] = newOrder;
    rear++;
    this.size++;
  }
}
```




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

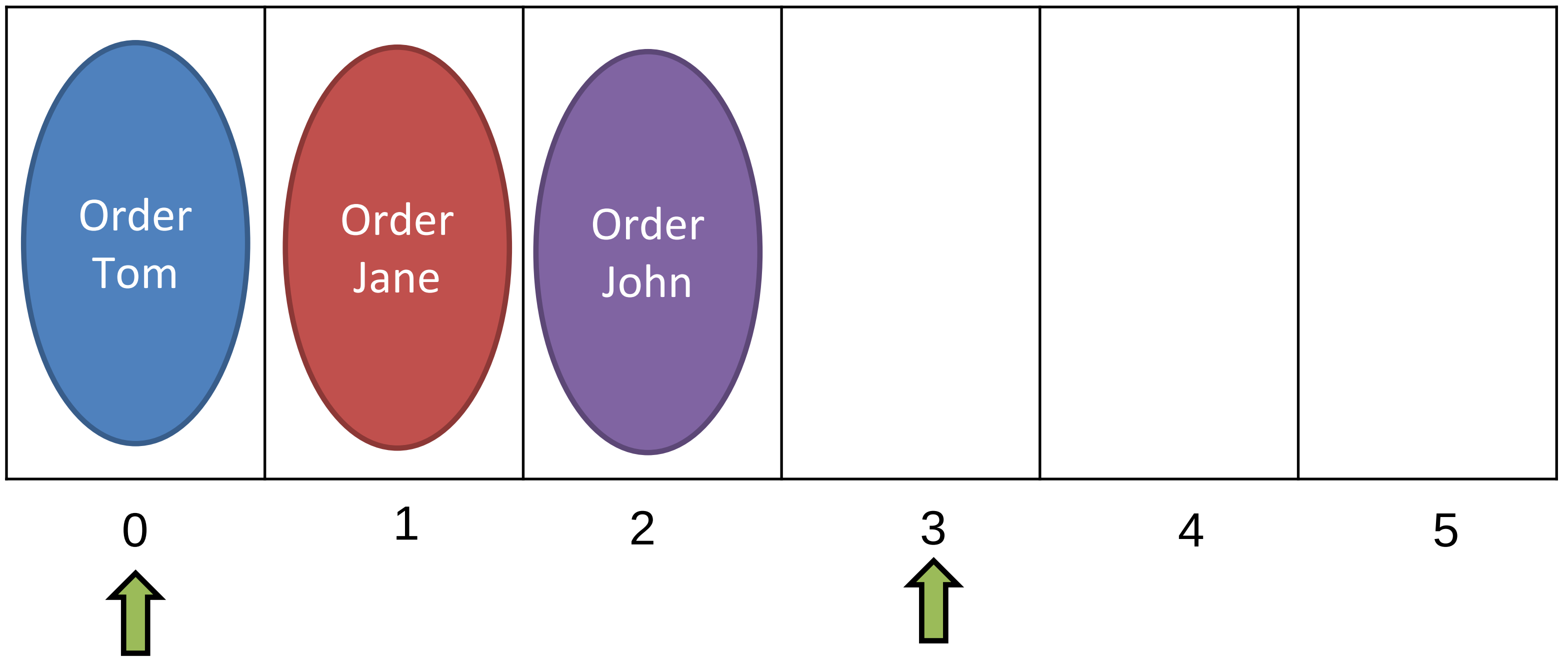
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




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

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

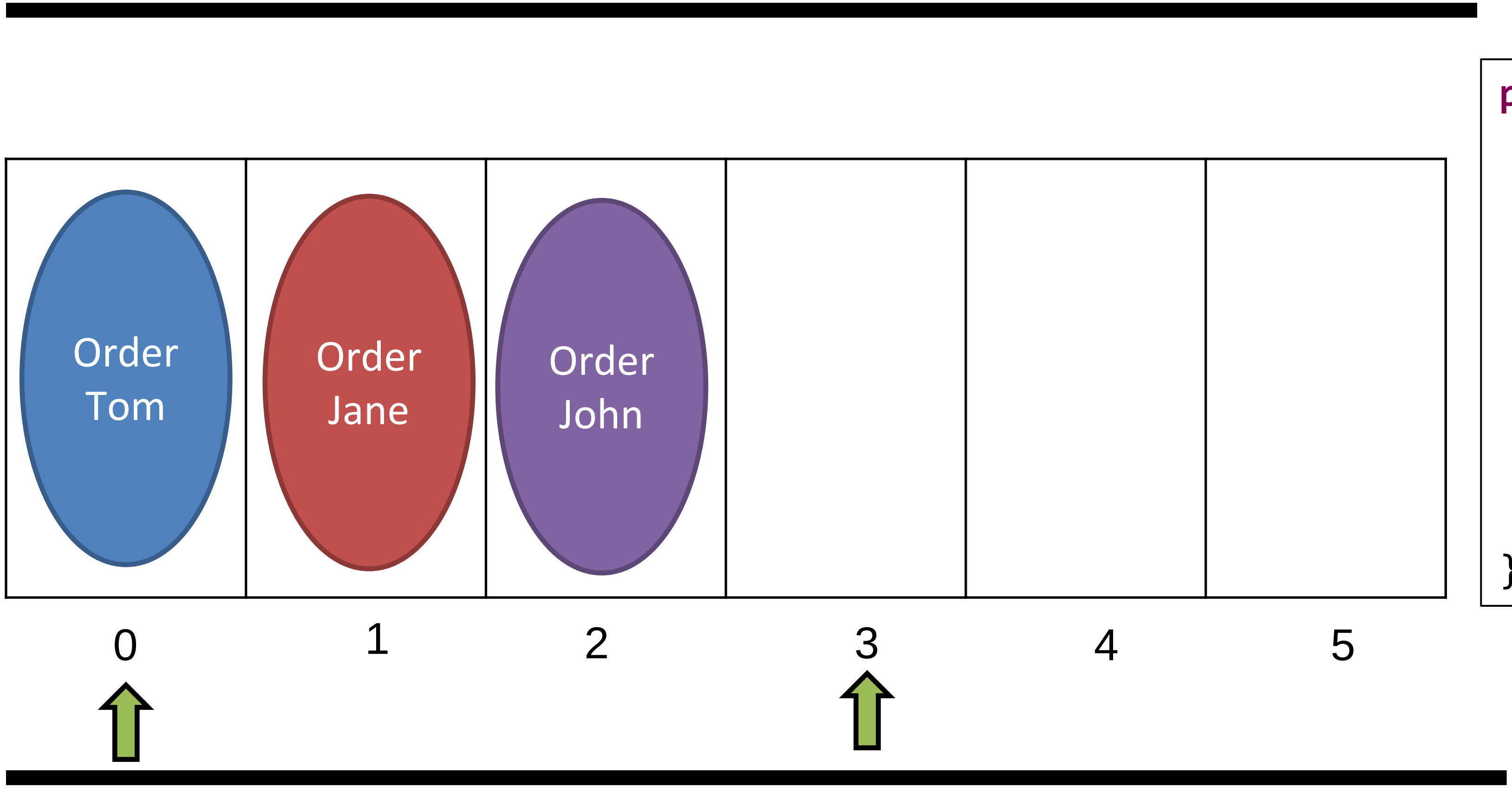
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



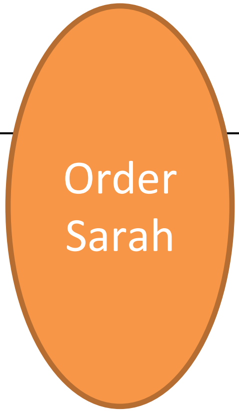

```
public void enqueue(Order newOrder) {
  if(rear == capacity) {
    System.out.println("full...");
    return;
  }
  else {
    this.data[rear] = newOrder;
    rear++;
    this.size++;
  }
}
```

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

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

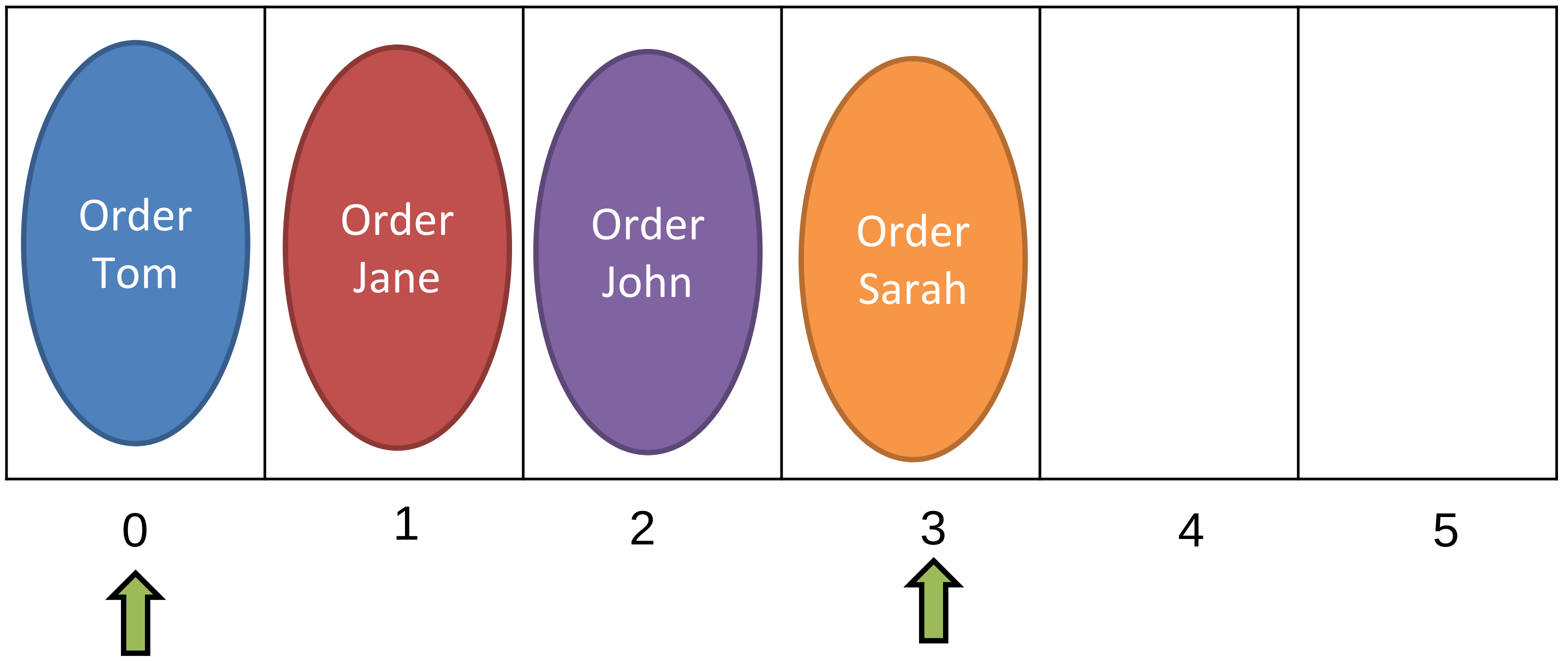



```java
public void enqueue(Order newOrder) {
  if(rear == capacity) {
    System.out.println("full...");
    return;
  }
  else {
    this.data[rear] = newOrder;
    rear++;
    this.size++;
  }
}
```




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

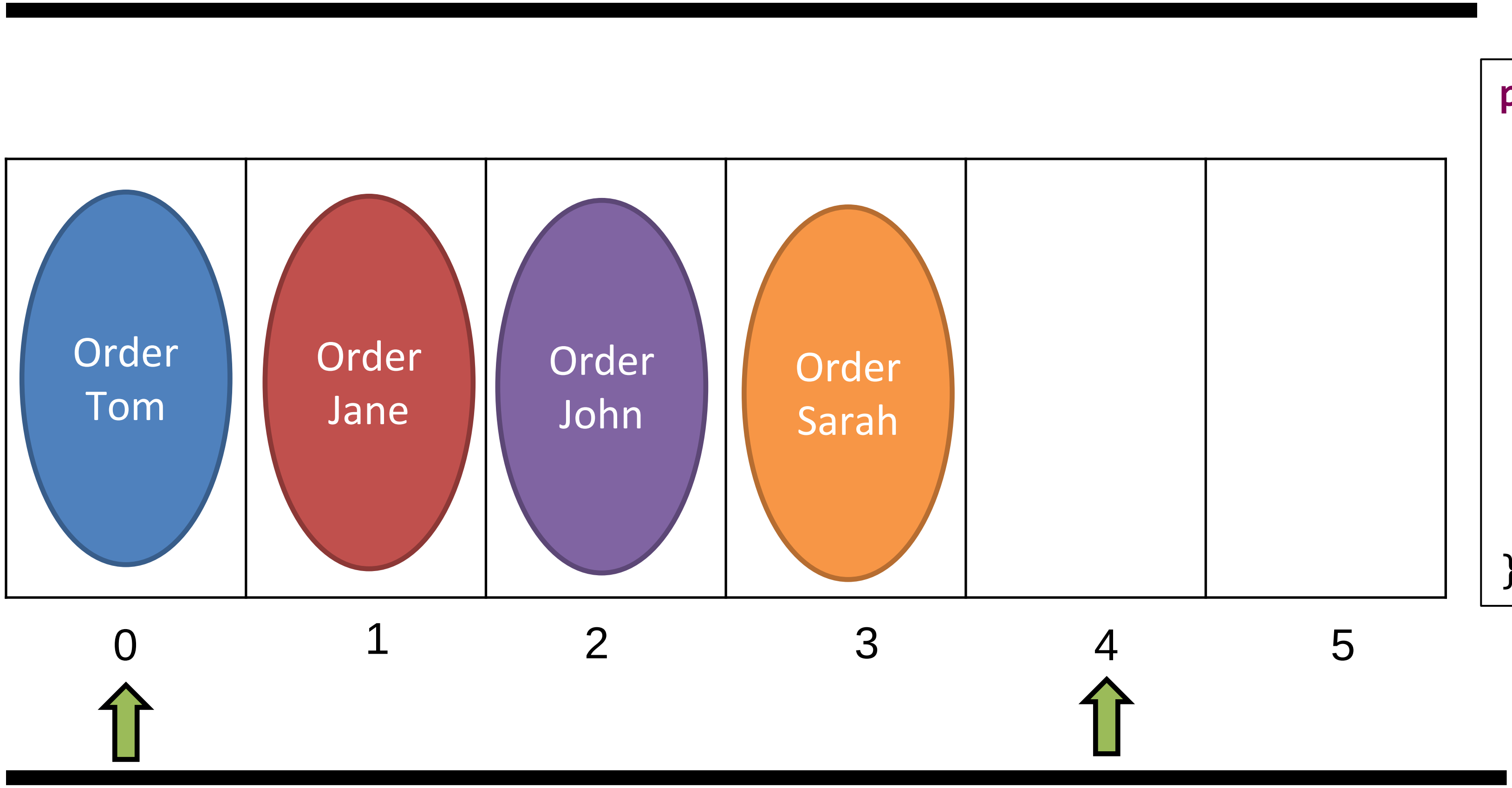
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




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

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

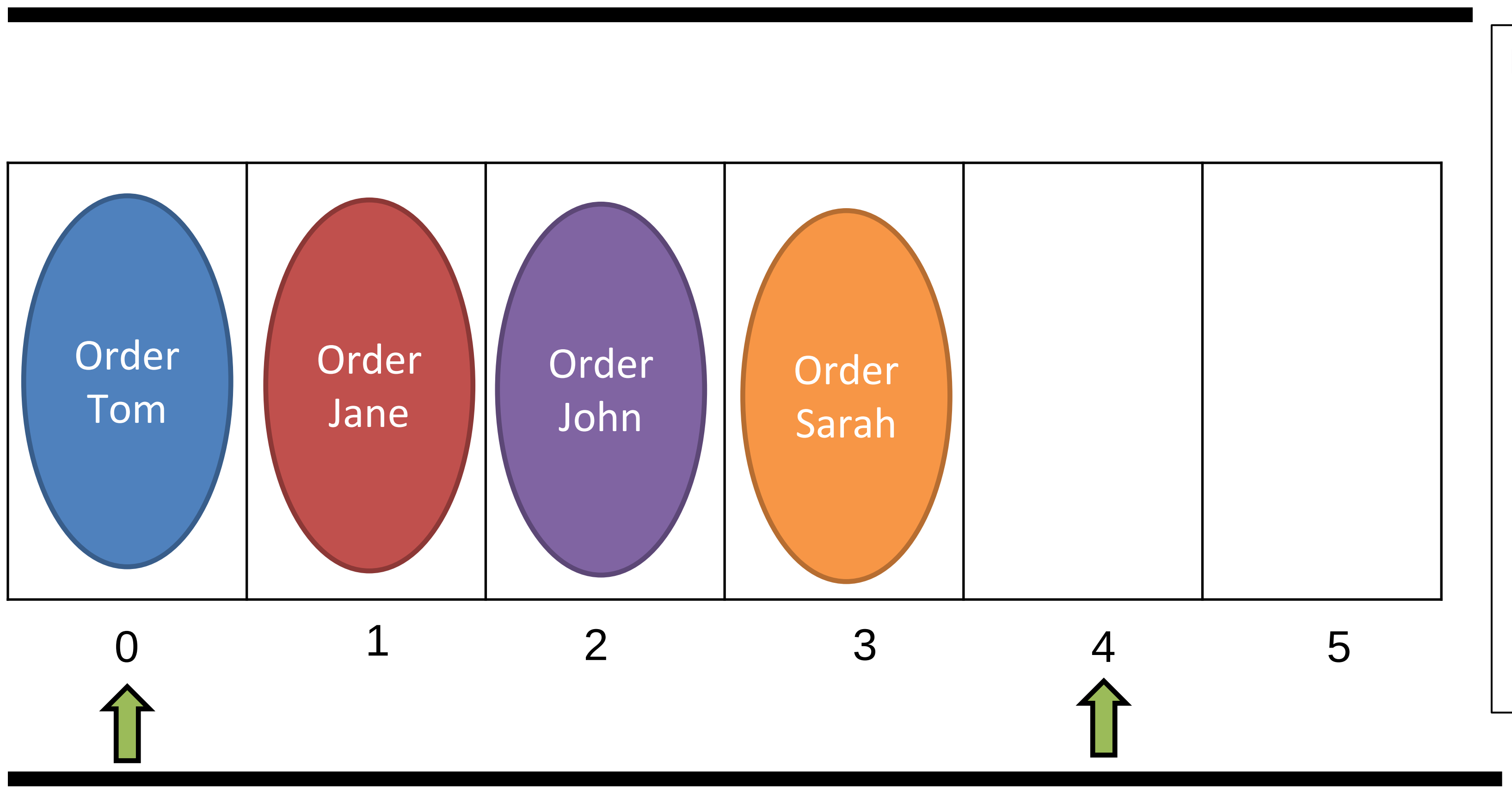
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




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

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

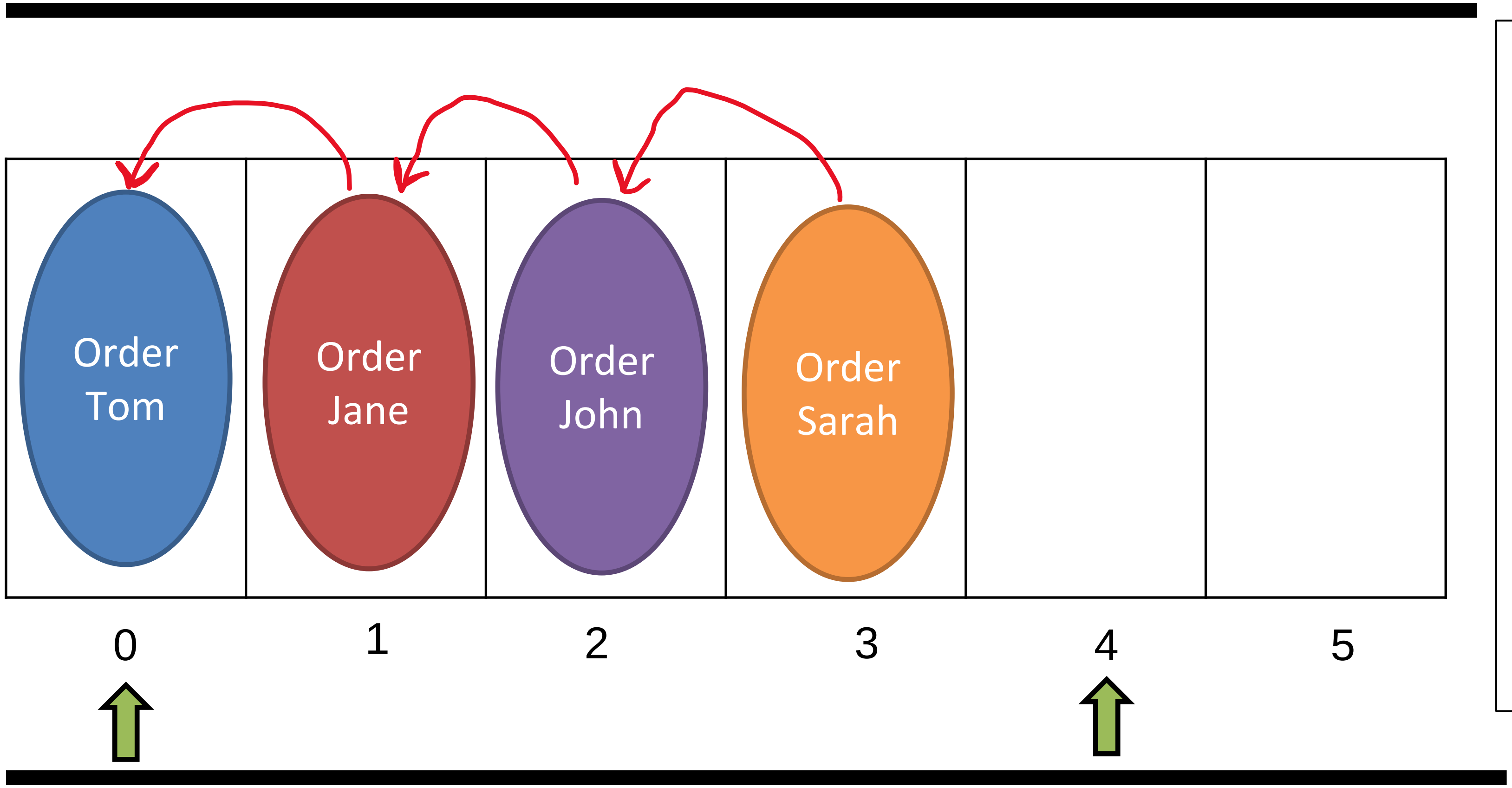
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

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

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

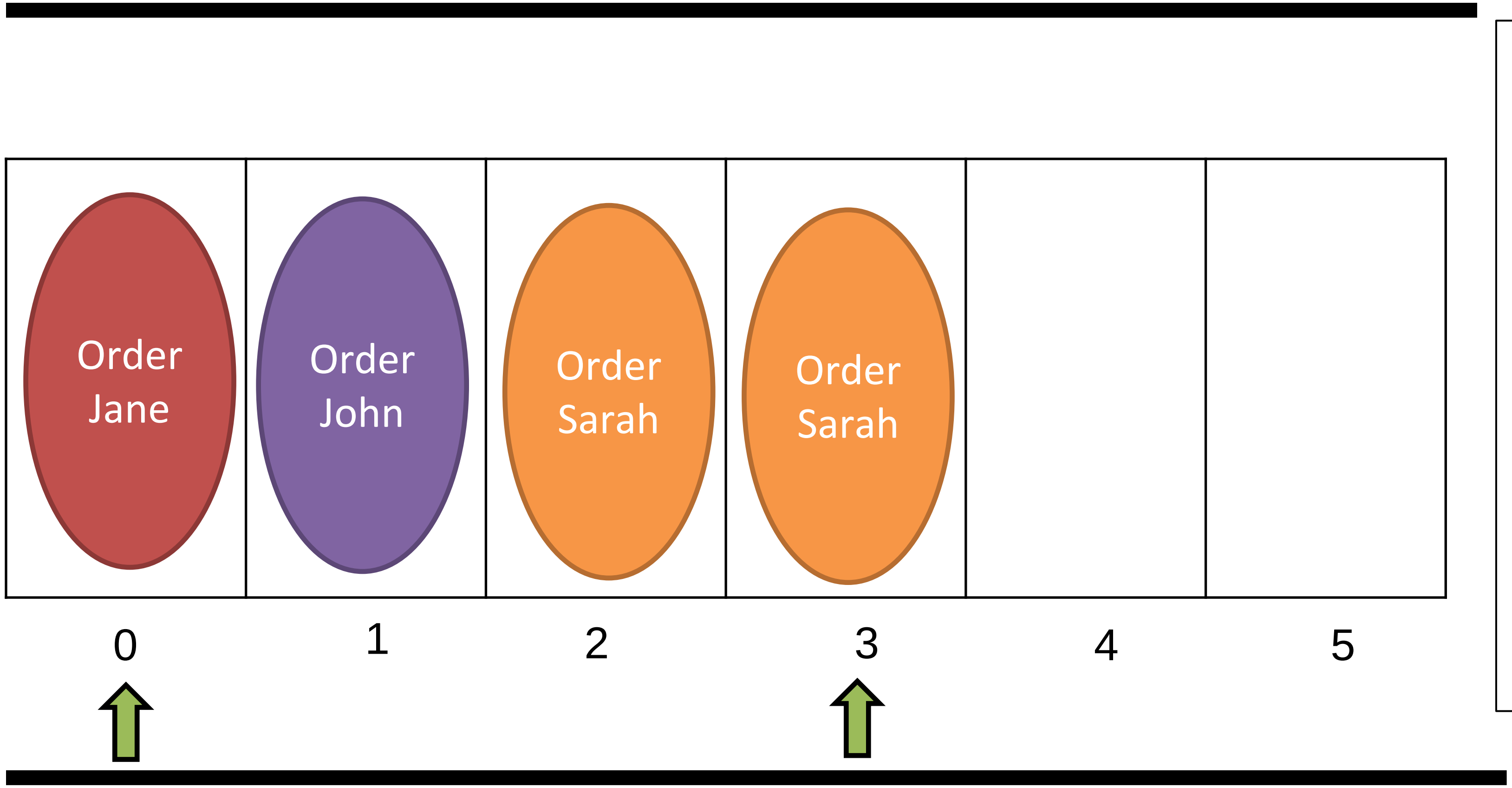



```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
        this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
        this.data[rear] = null;
    }
  }
}
```

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

Shift everything over one spot!

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

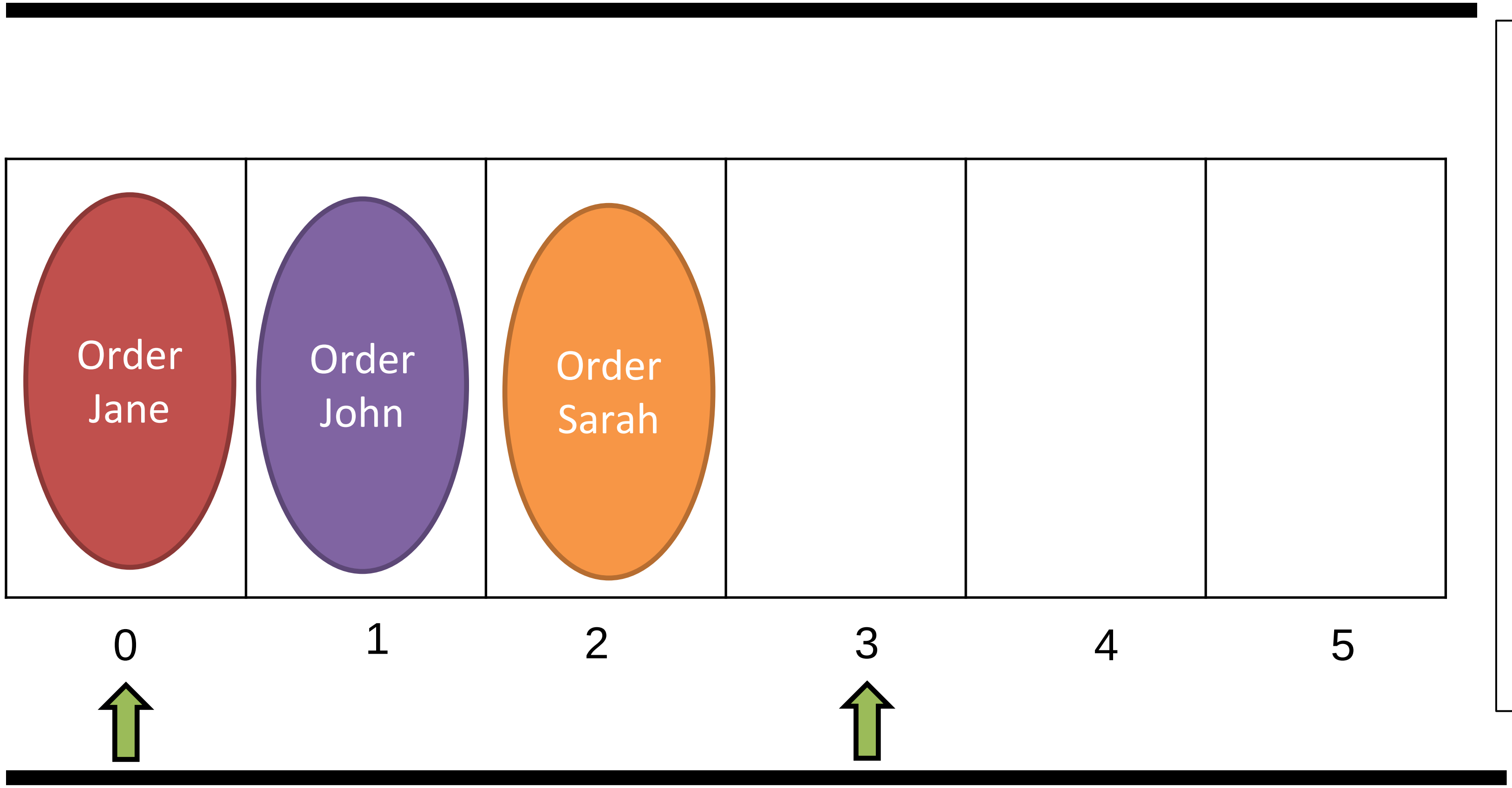



```
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
  }
}
```

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

<u>Shift</u> everything over one spot!

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

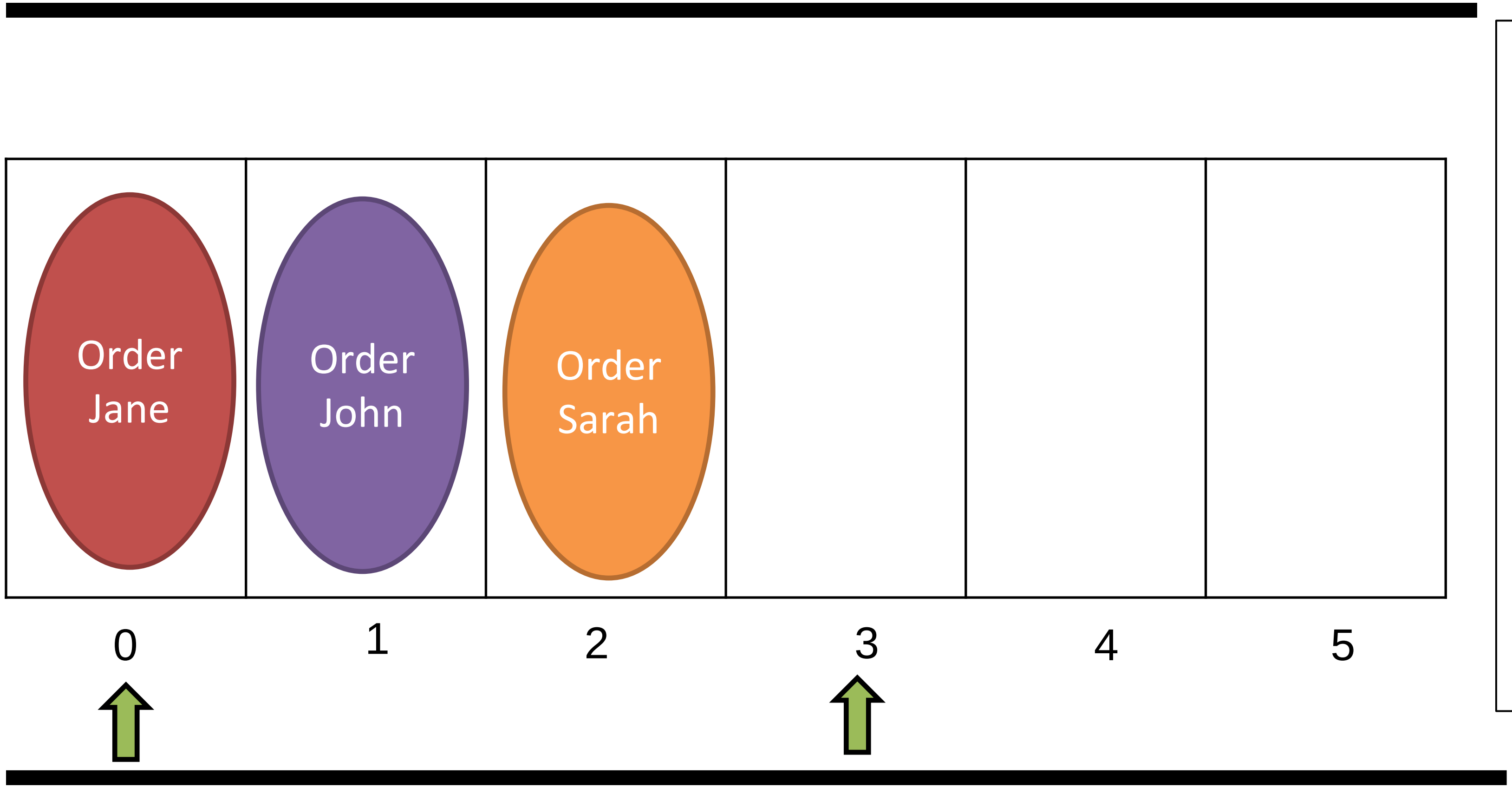



```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
  }
}
```

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

<u>Shift</u> everything over one spot!

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

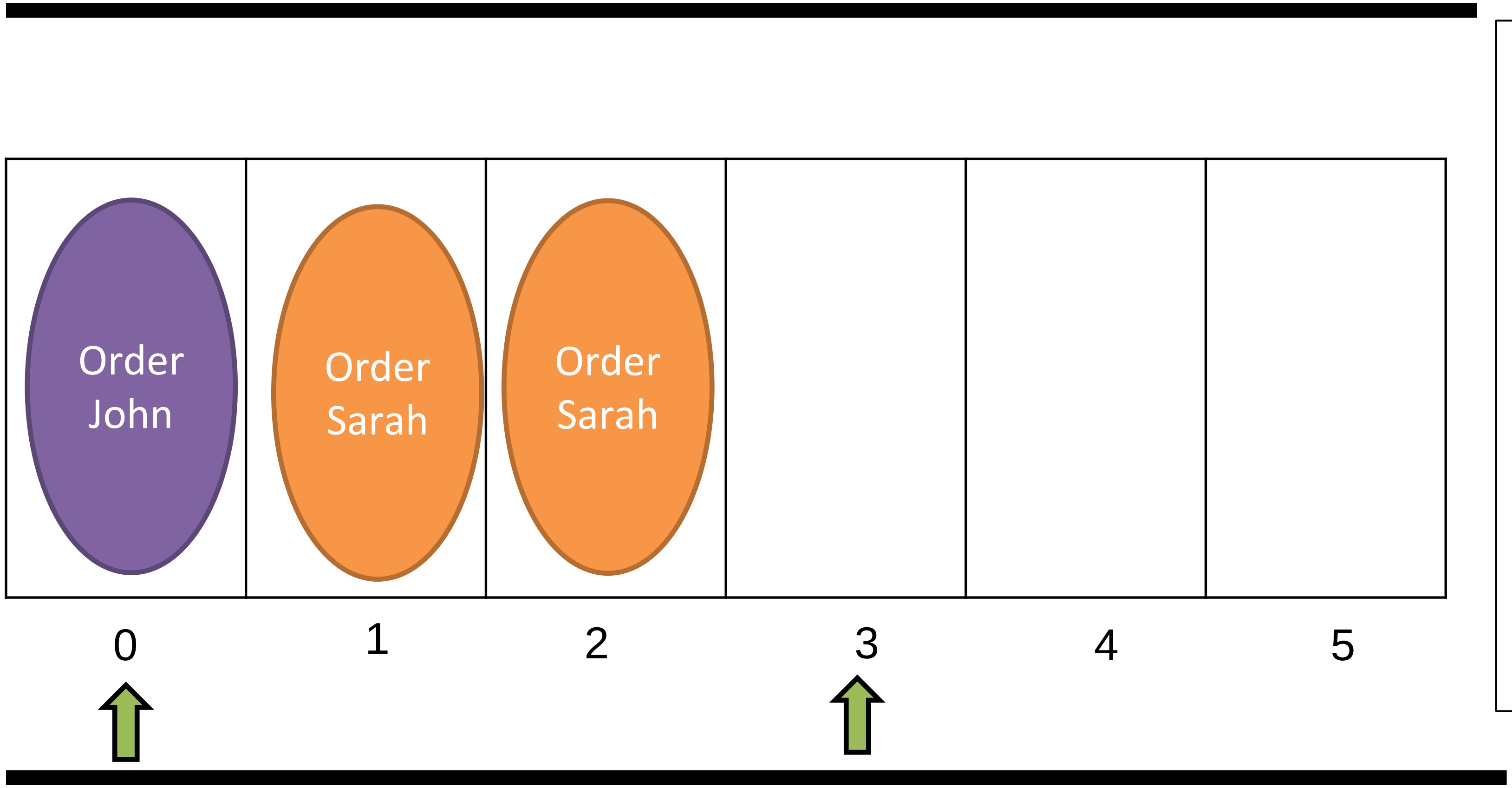



```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
  }
}
```

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

Shift everything over one spot!

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

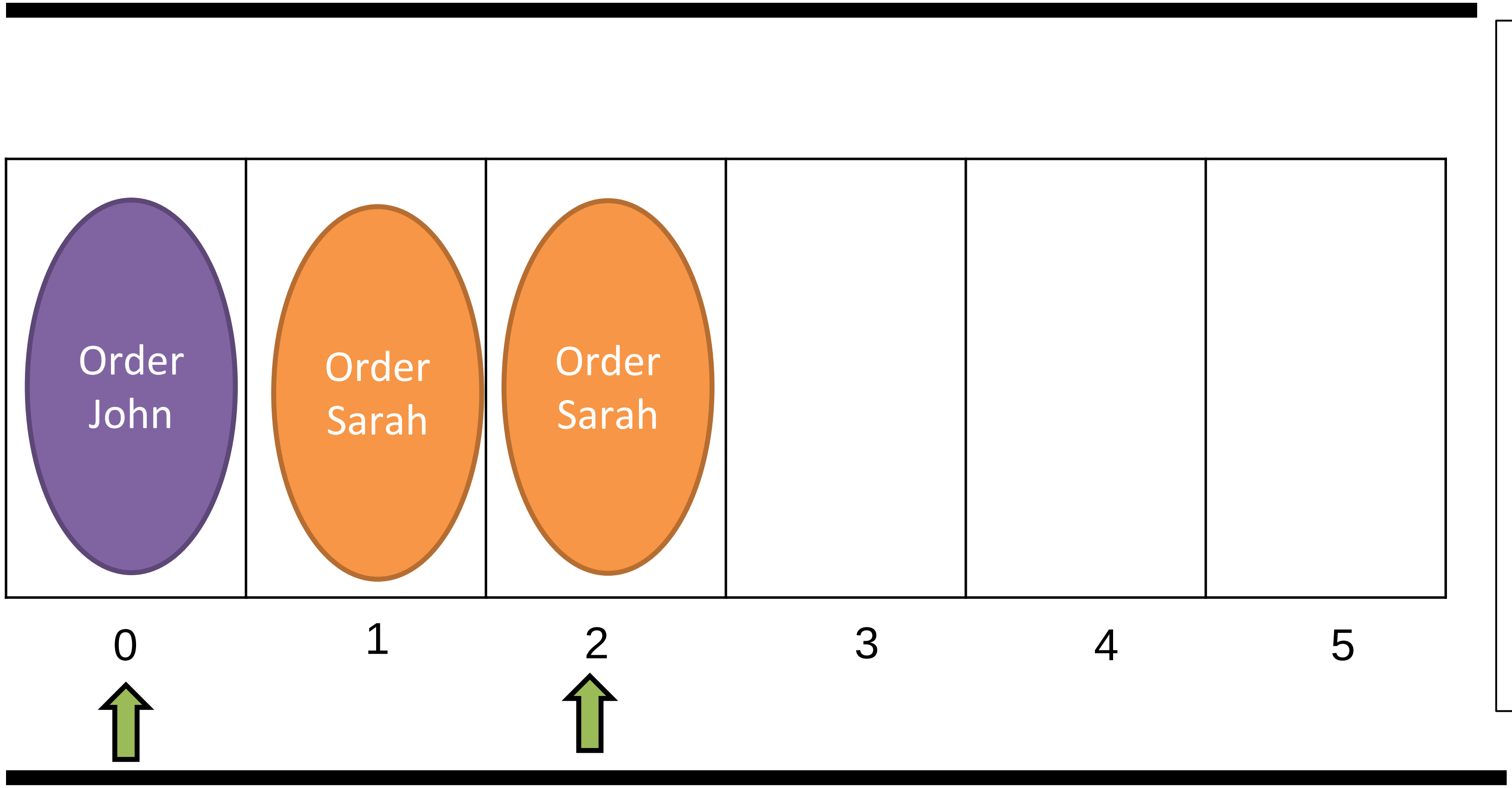



```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

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

<u>Shift</u> everything over one spot!

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

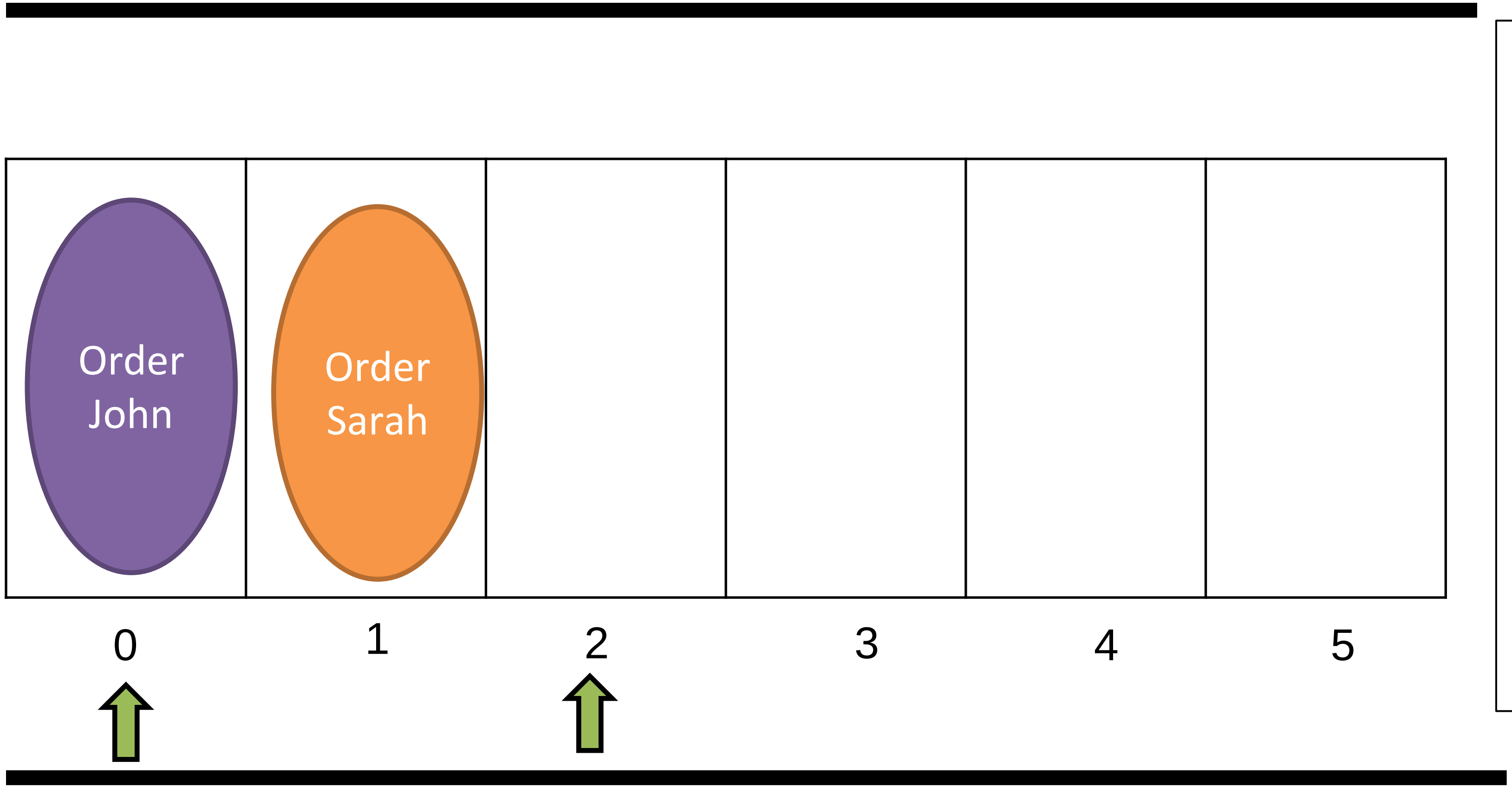



```
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
  }
}
```

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

<u>Shift</u> everything over one spot!

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

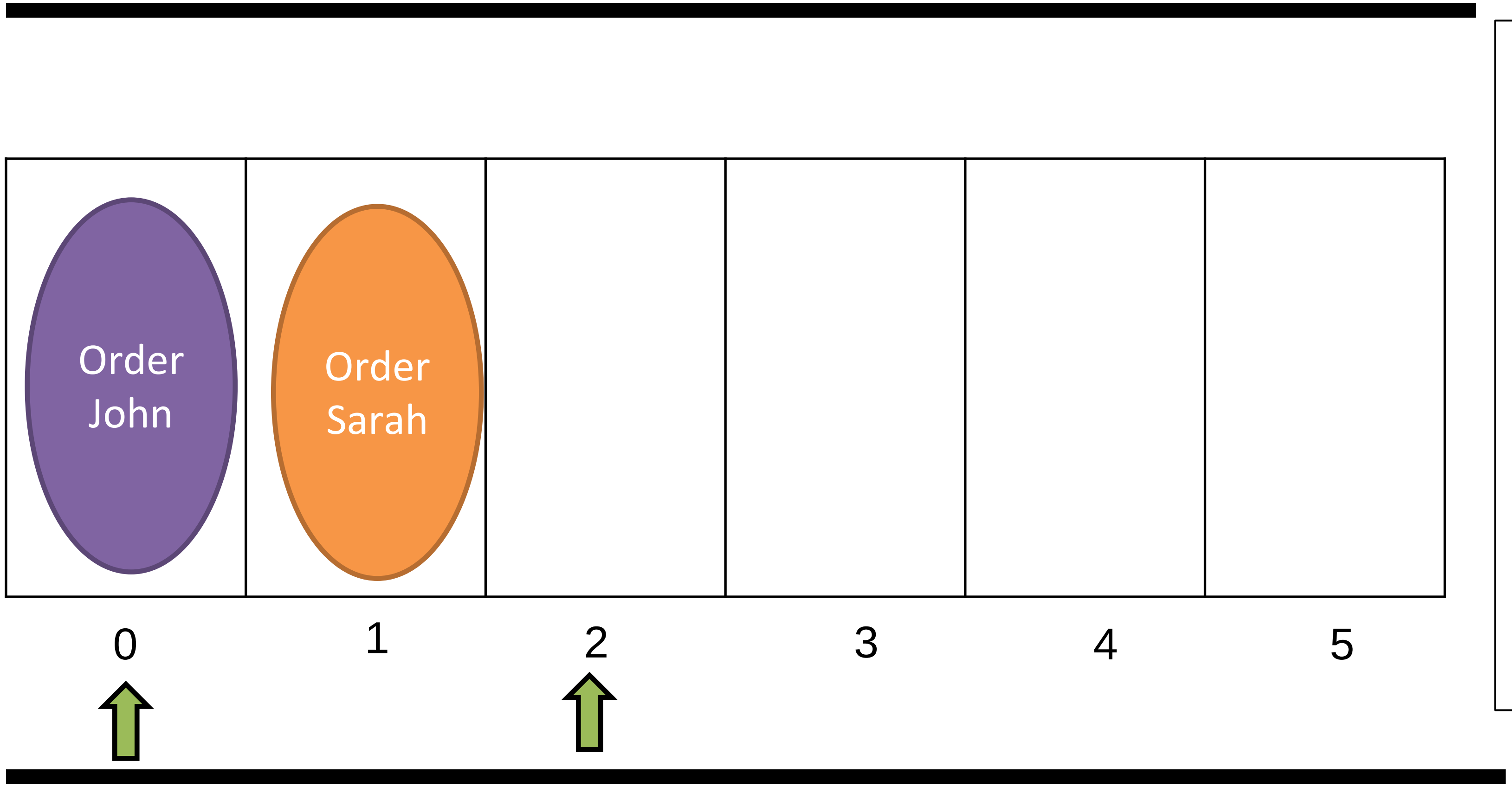



```
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

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

<u>Shift</u> everything over one spot!

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




```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

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

<u>Shift</u> everything over one spot!

```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {
       this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

O(n-1)

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

(think about if this queue has 1000000 things in it→ we must shift 999999 elements!)

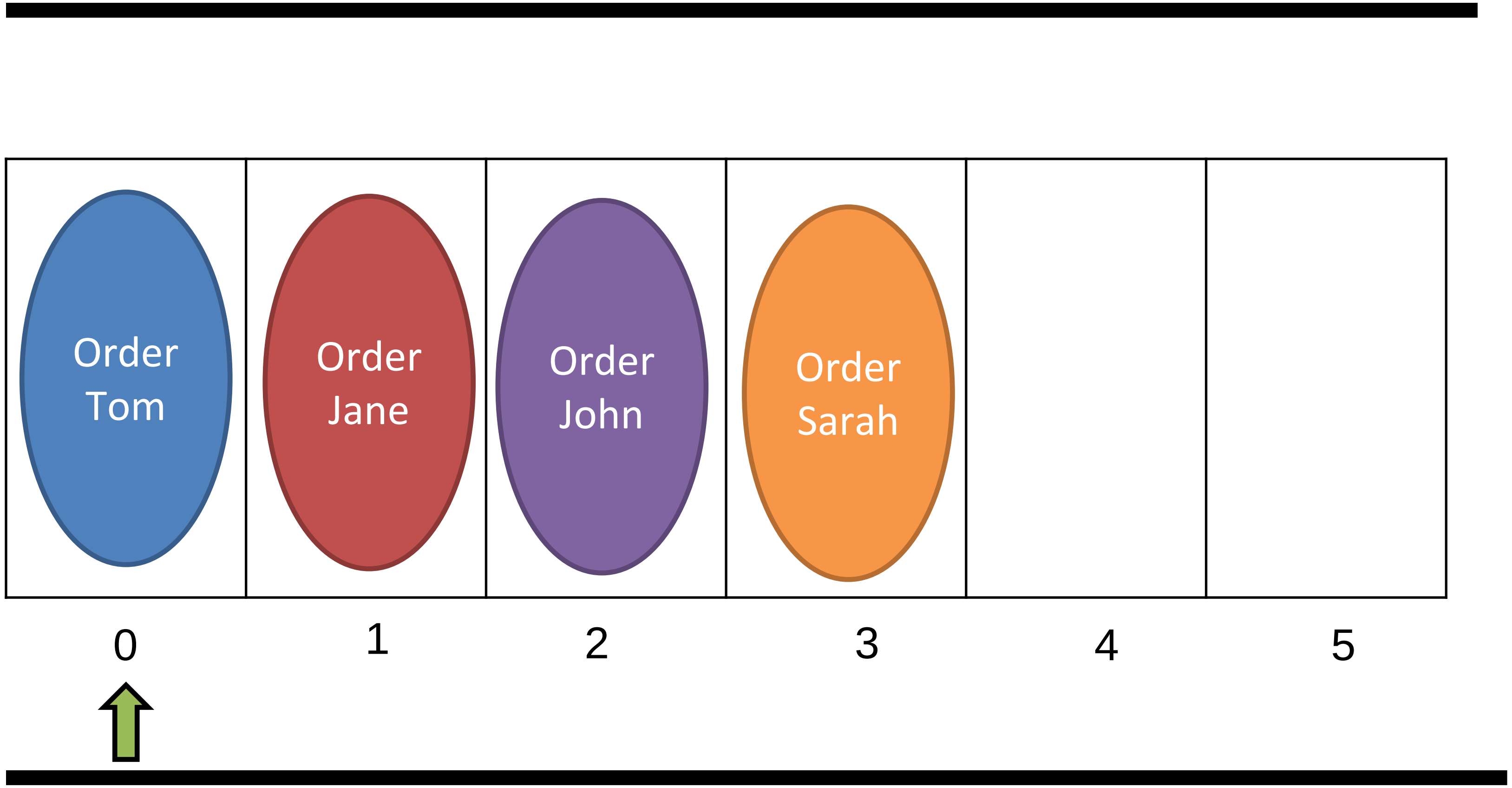
```java
public void dequeue() {
  if(this.size == 0) {
    System.out.println("empty...");
    return;
  }
  else {
    for (int i = 0; i < rear - 1; i++) {   O(n-1)
        this.data[i] = this.data[i + 1];
    }
    rear--;
    this.size--;
    if(rear < capacity) {
       this.data[rear] = null;
    }
}
}
```

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

(think about if this queue has 1000000 things in it→ we must shift 999999 elements!)

We need a better algorithm that runs in **constant time** for enqueuing and dequeuing

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 !

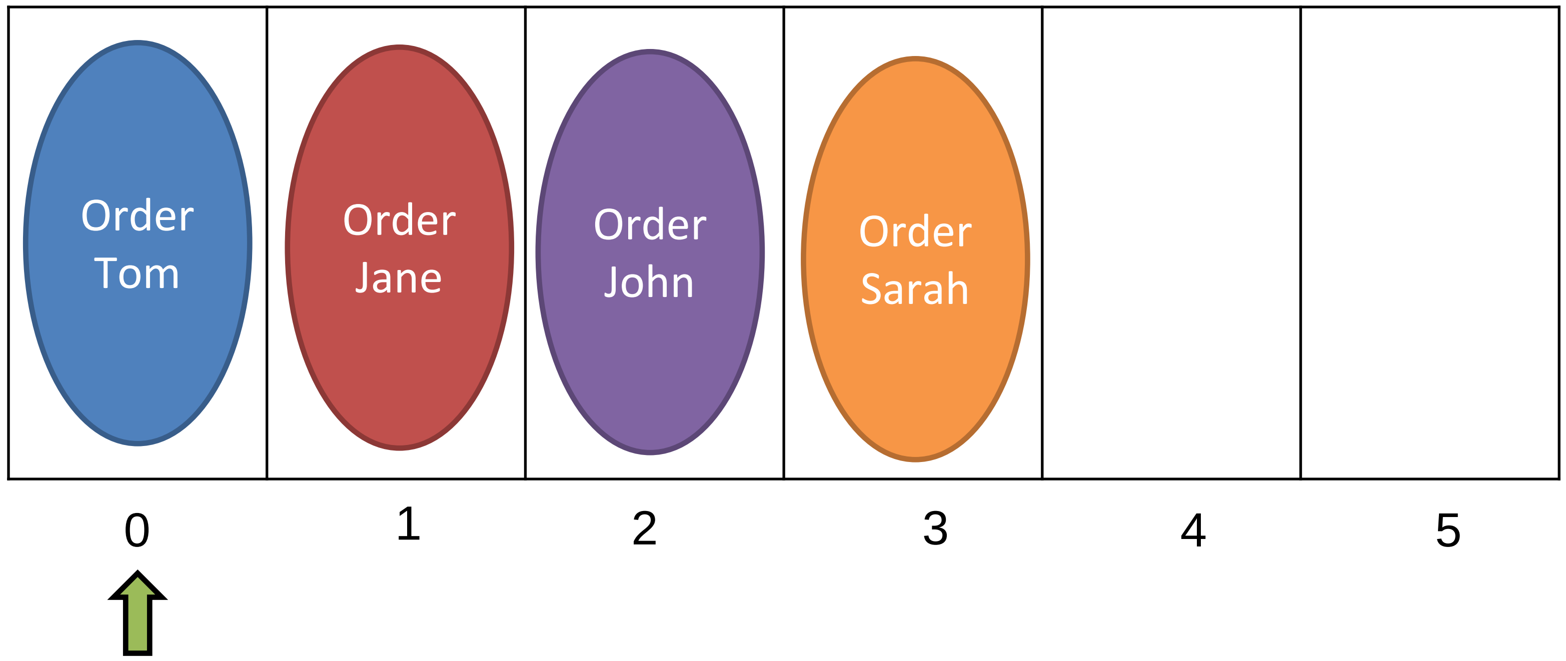
10 % 6 = 4

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



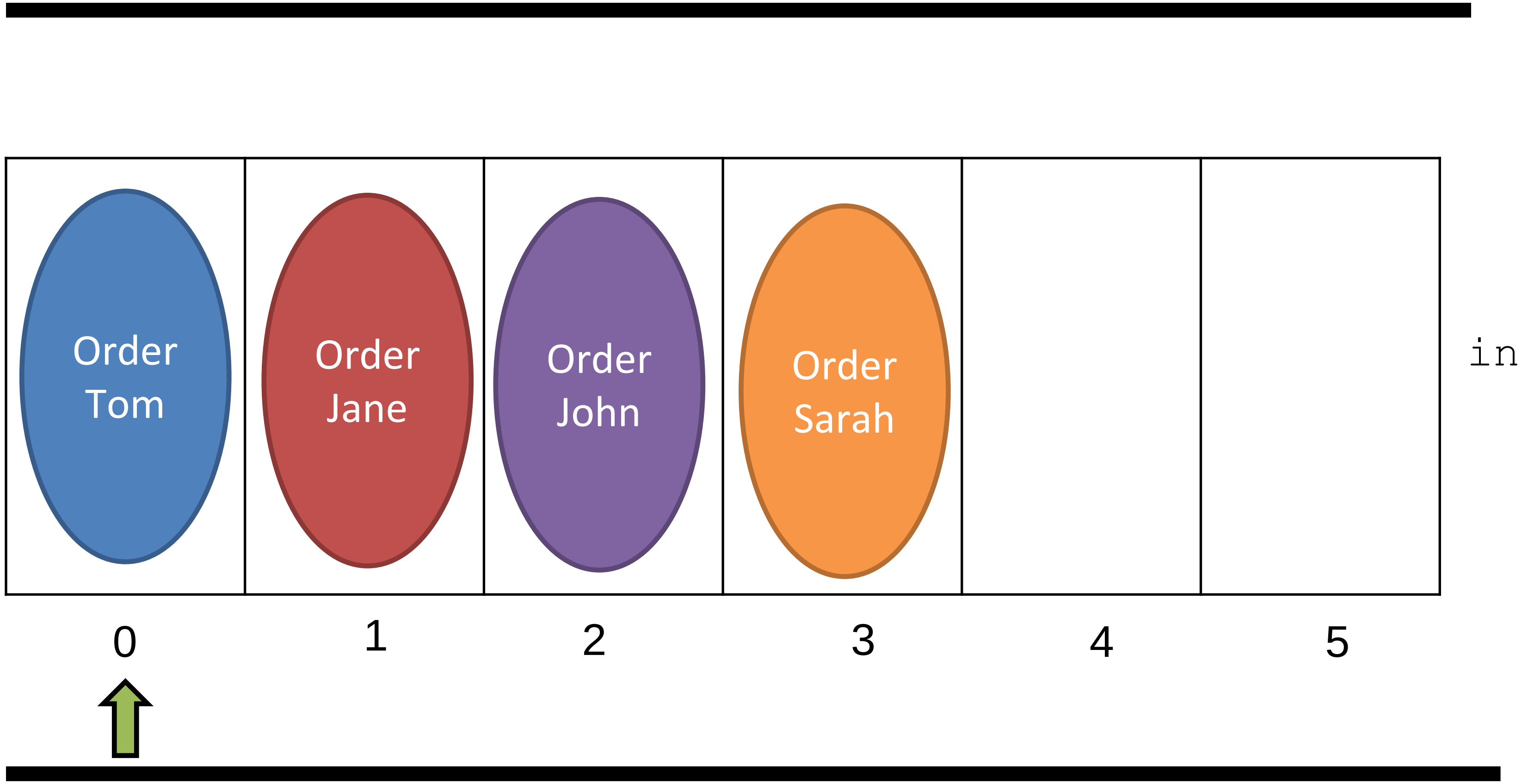
Let's **enqueue**

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

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

```
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



Let’s **enqueue**

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

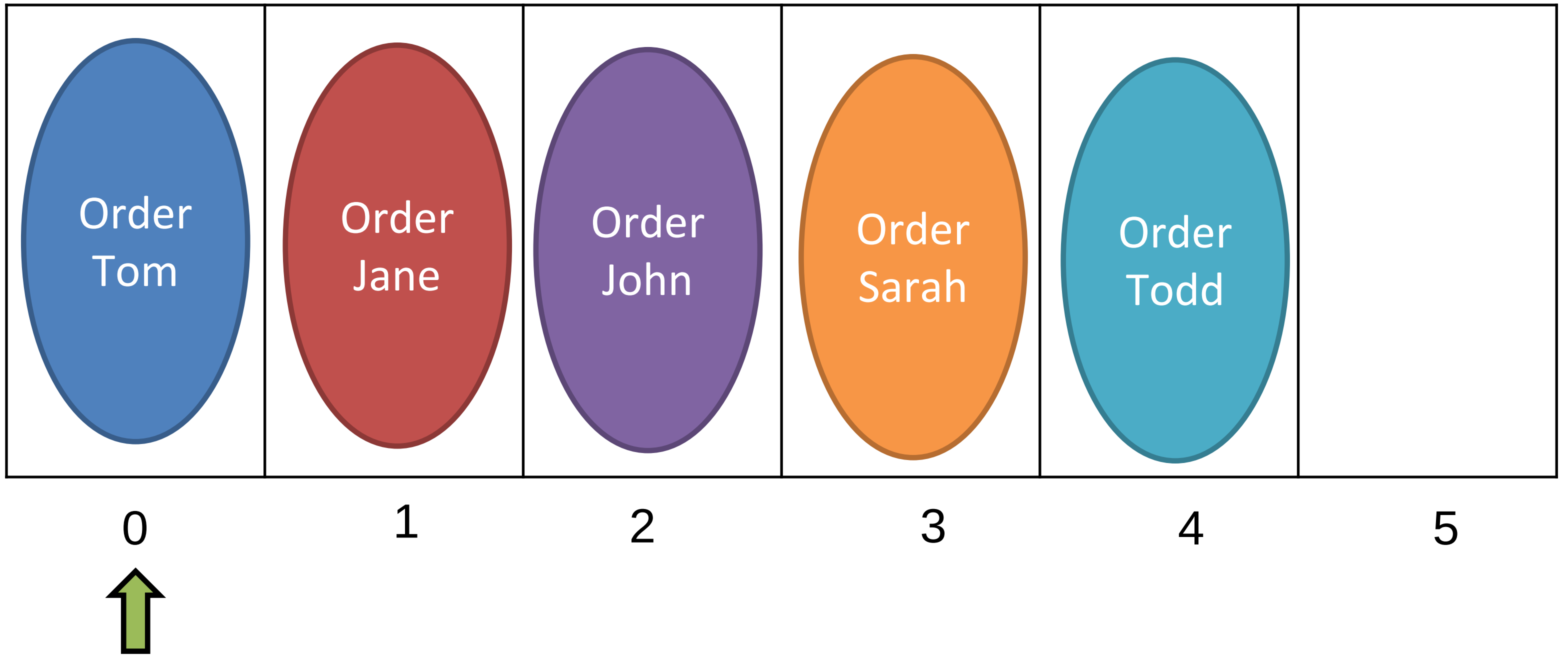
```
insert_spot = (front + size) % 6
```

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

```
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 **enqueue**



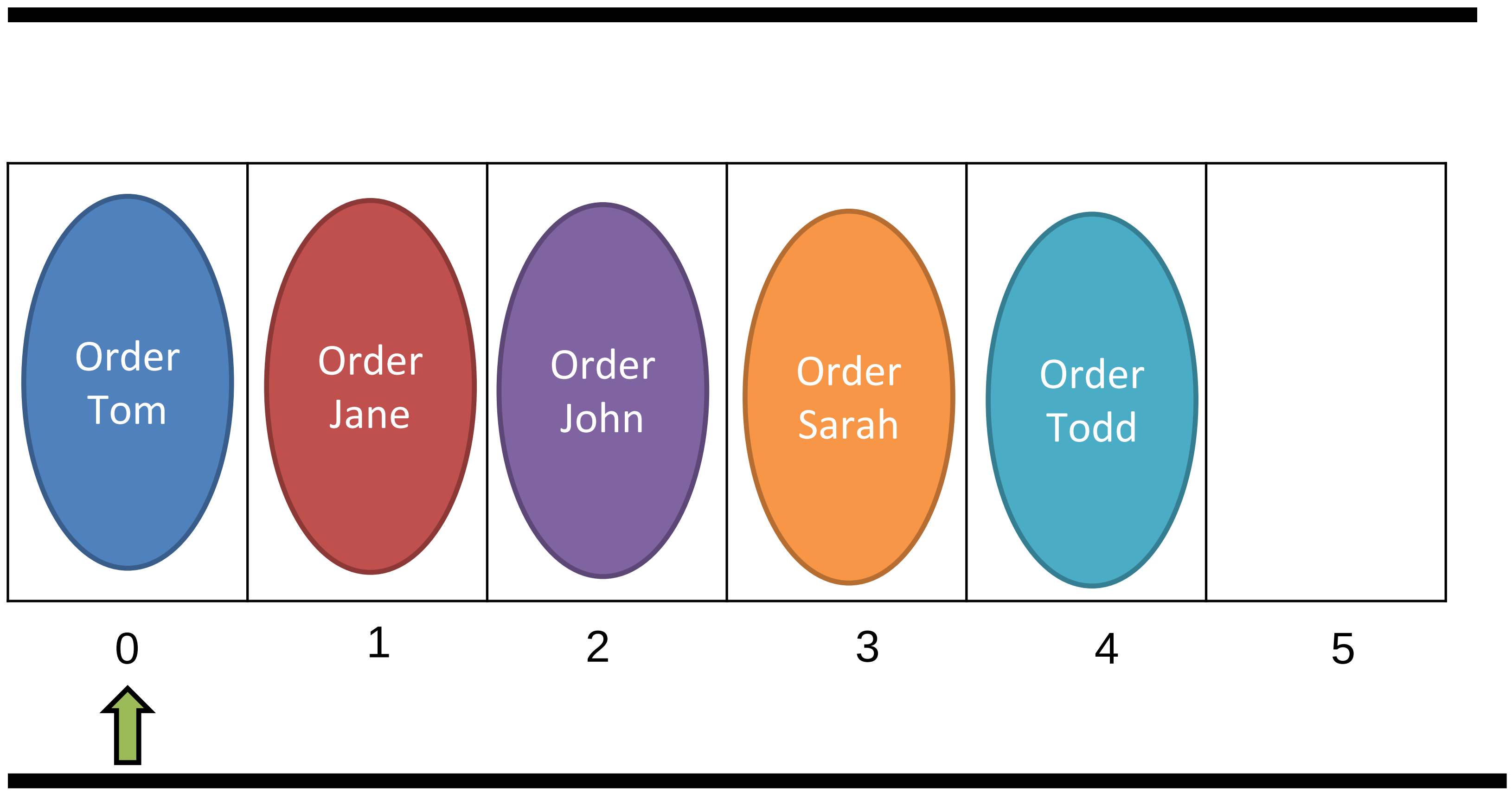
Here is the formula for determining where to insert the new element

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

```
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**

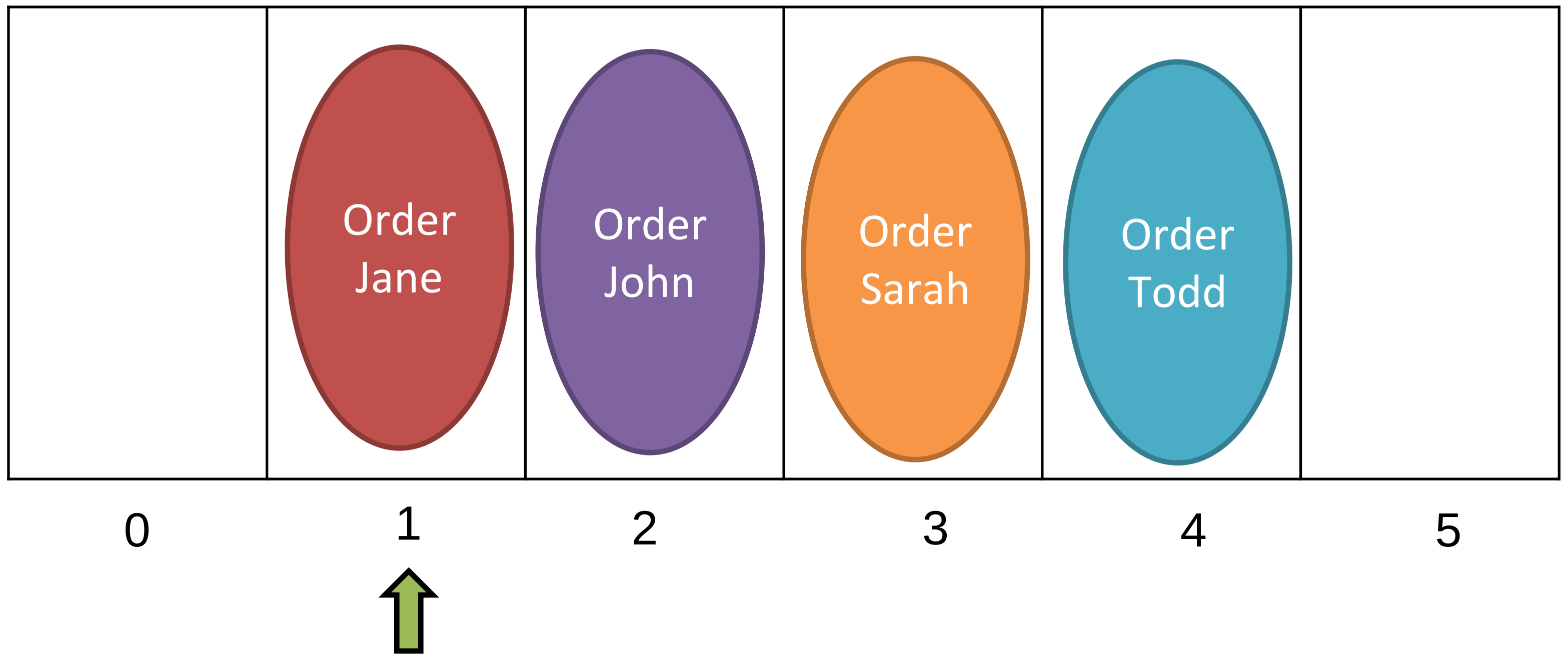



```
data[front] = null
```

```
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
```

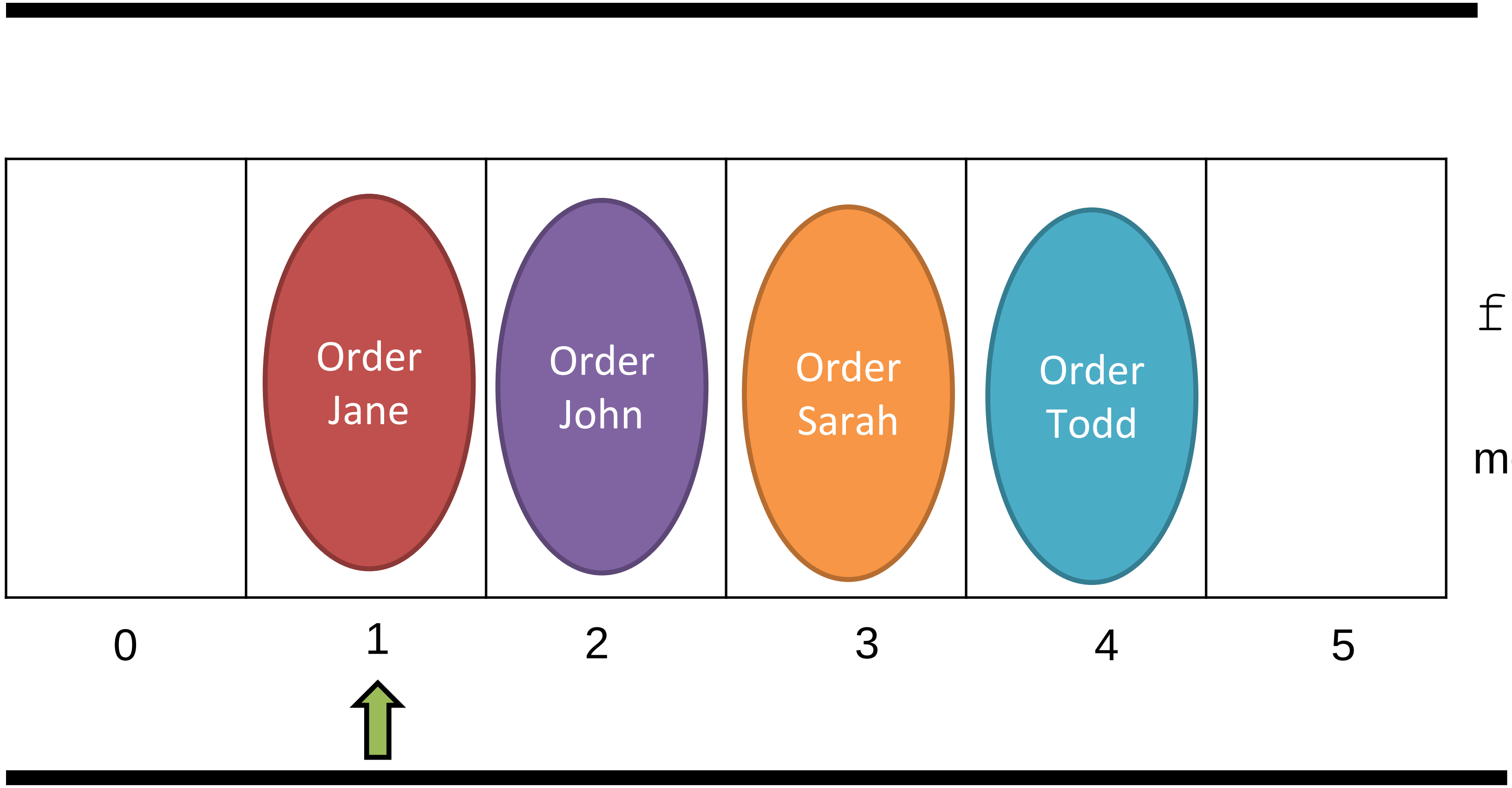
```
front = (front + 1) % 6
```

move the front pointer to the next element

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

```
capacity = 6     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
```

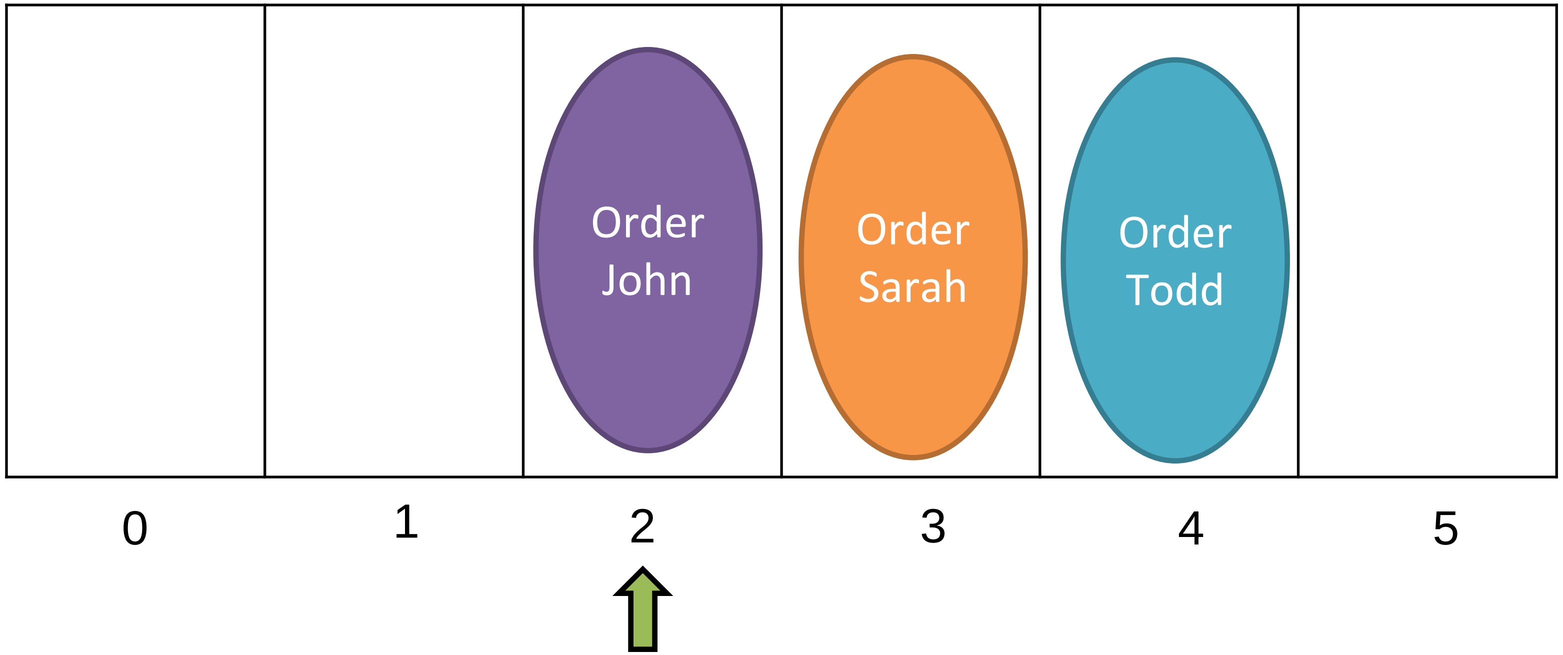
move the front pointer to the next element

= (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`

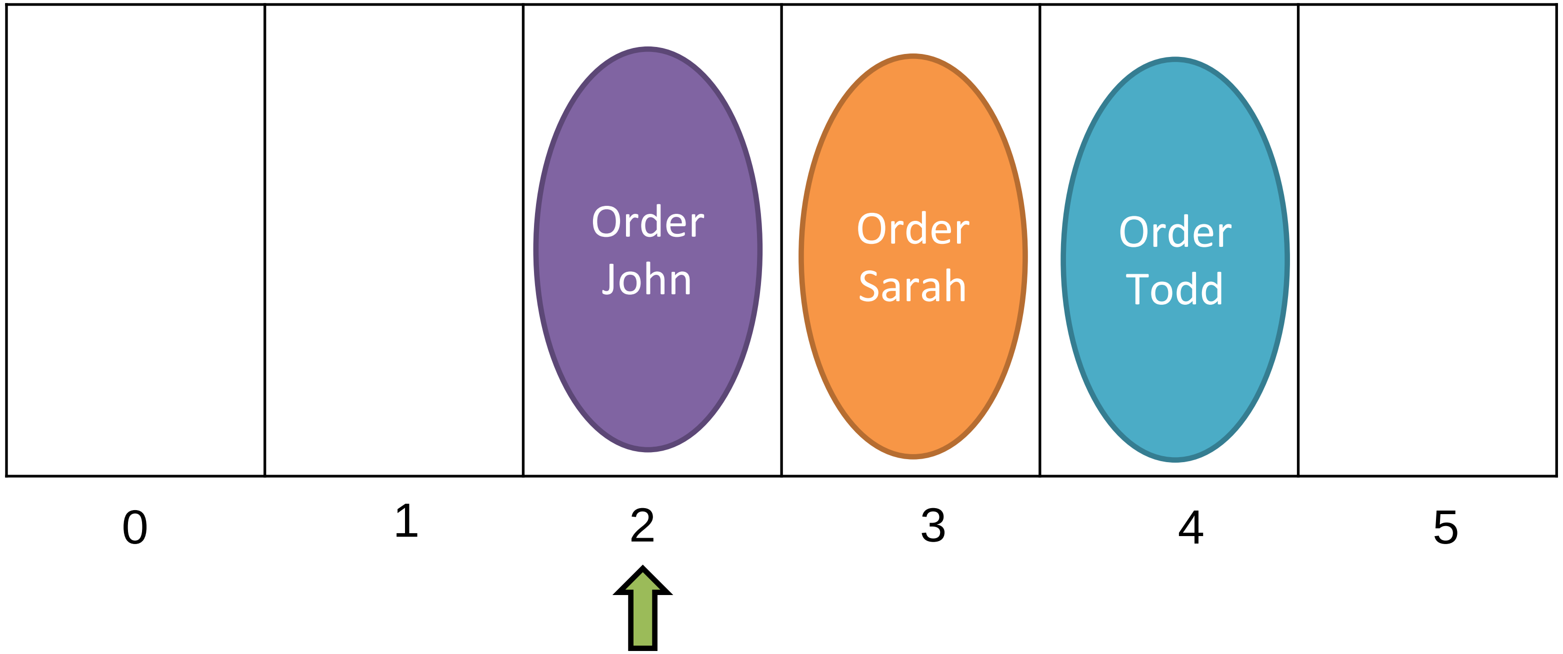
move the front pointer to the next element

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

```
capacity = 6      front = 2
size = 3          insert_spot = 4
```

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

Let’s **enqueue (again)**



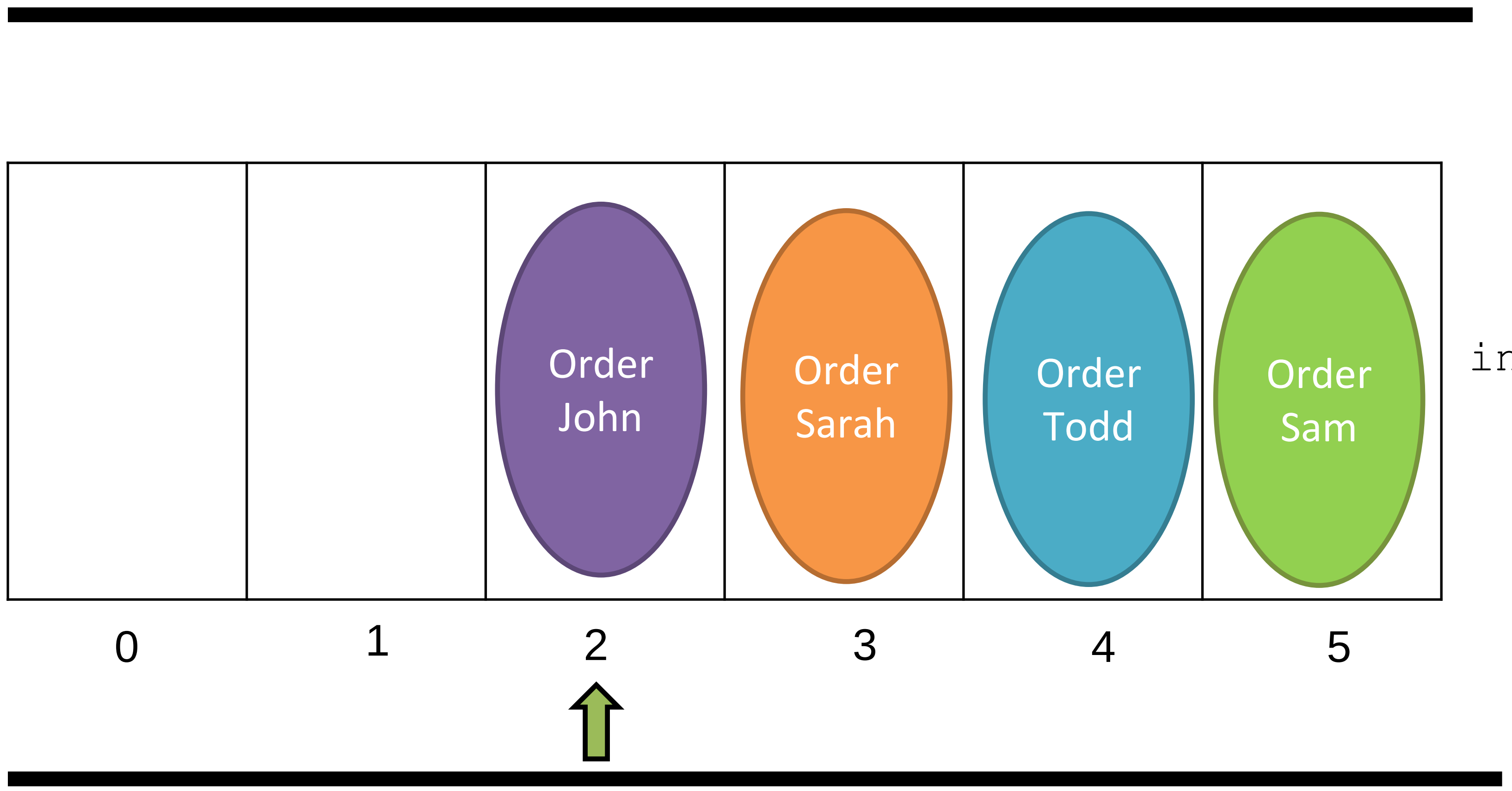

```
insert_spot = (front + size) % 6

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

5%6 = **5**


```
capacity = 6      front = 2
size = 3          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 (again)**



`insert_spot = (front + size) % 6`

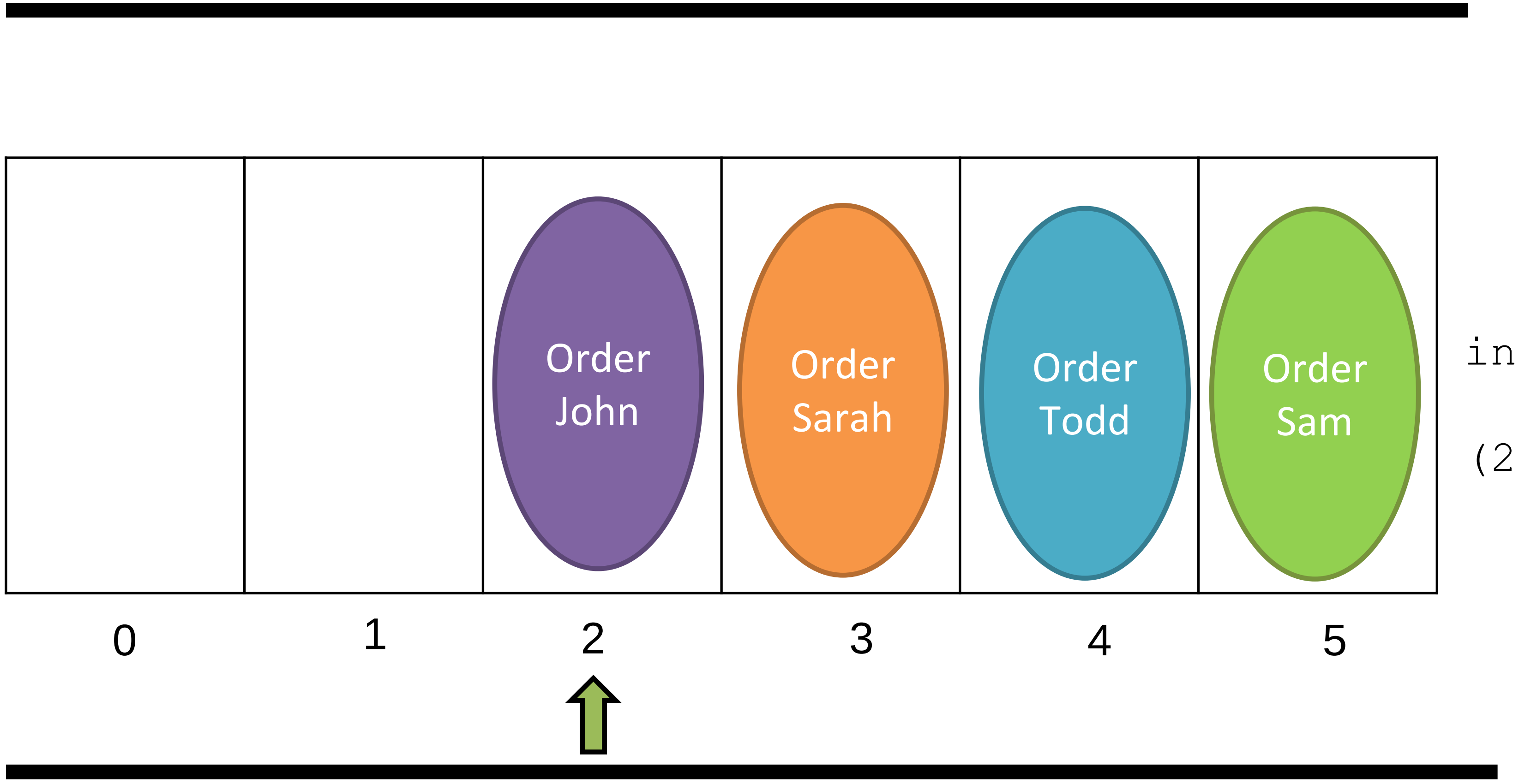
`insert_spot = (2 + 3) % 6`

5%6 = **5**

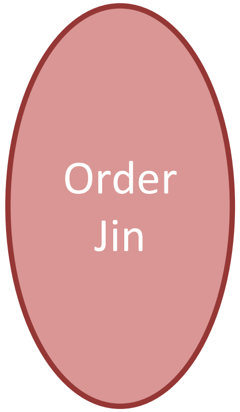
```
capacity = 6      front = 2
size = 4          insert_spot = 5
```

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>)**



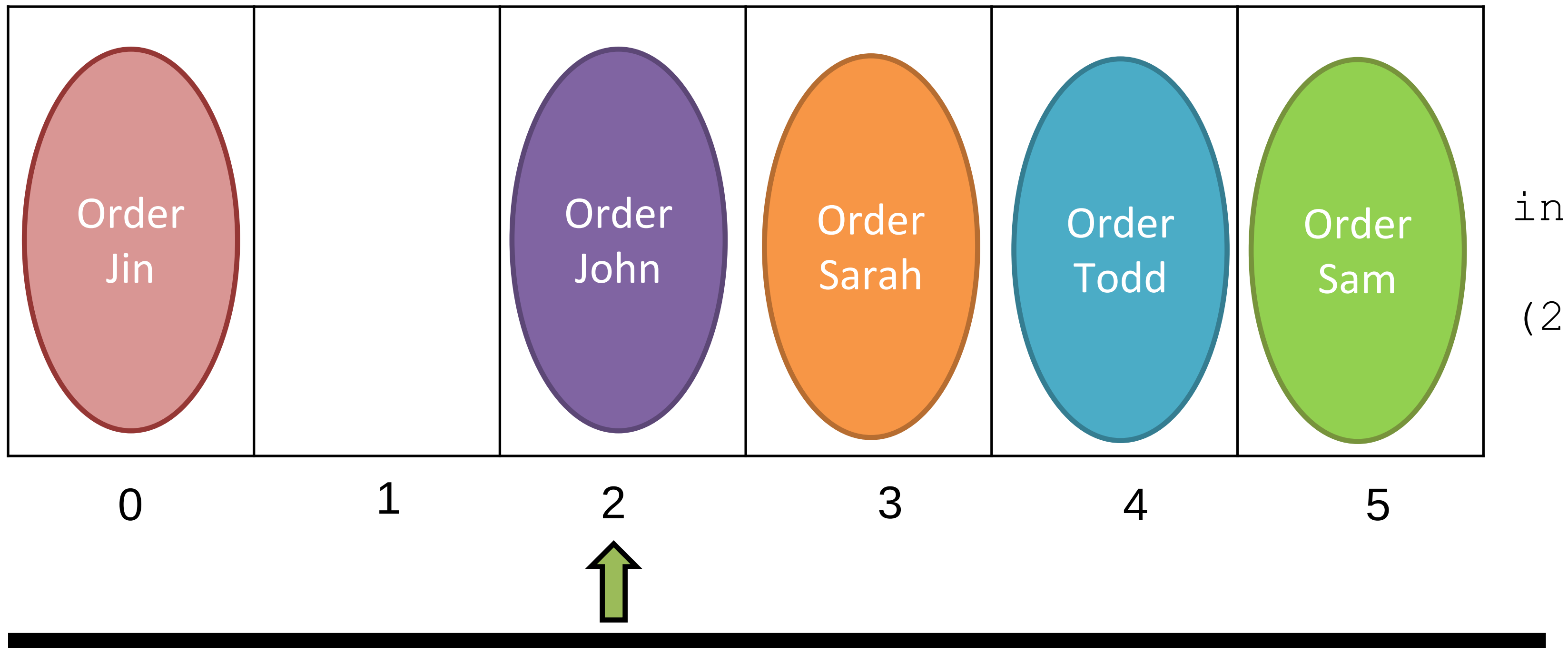

```
insert_spot = (front + size) % 6

(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

Let’s **enqueue (<u>again</u>)**



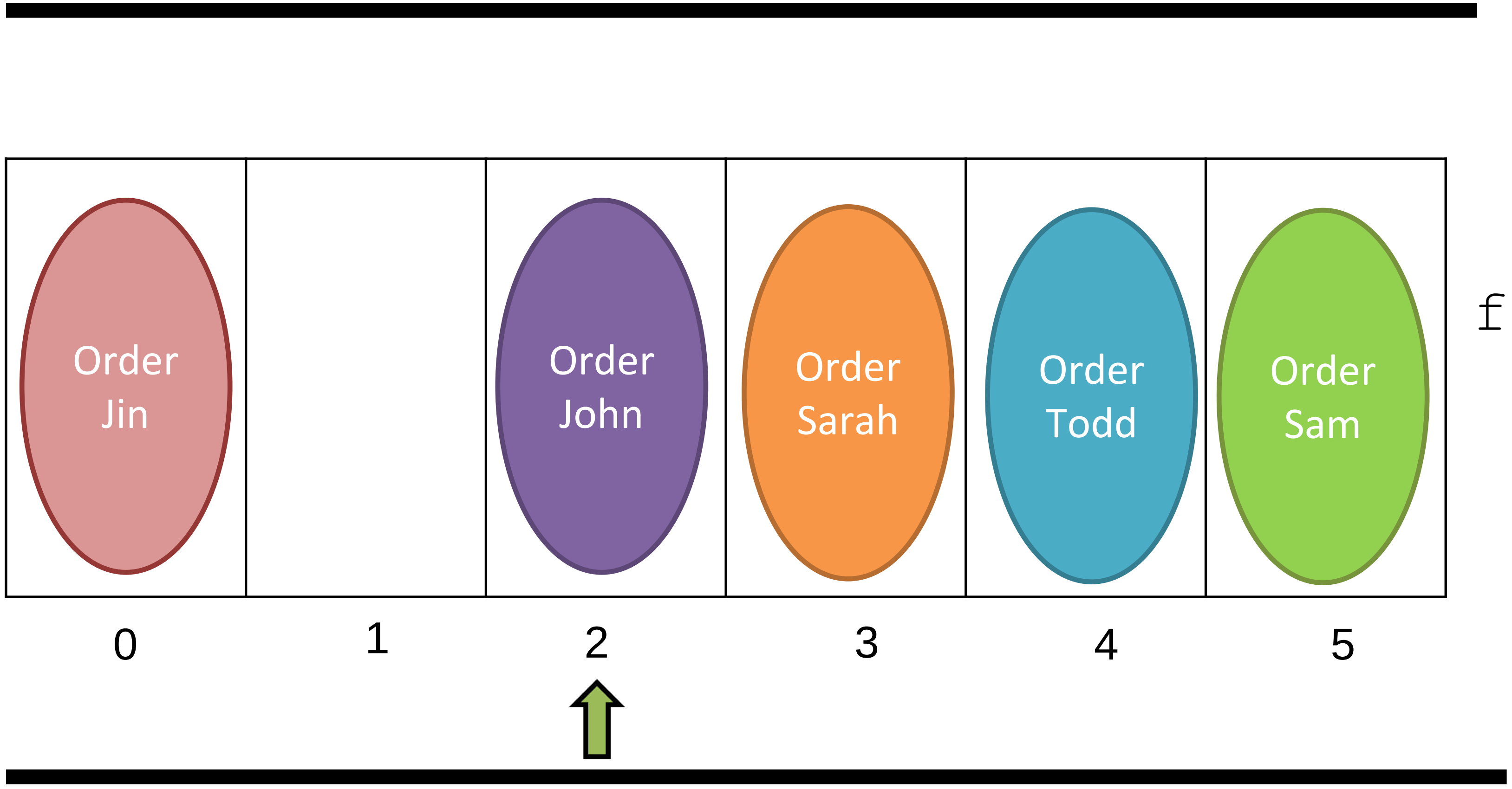
`insert_spot = (front + size) % 6`

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

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

```
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



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

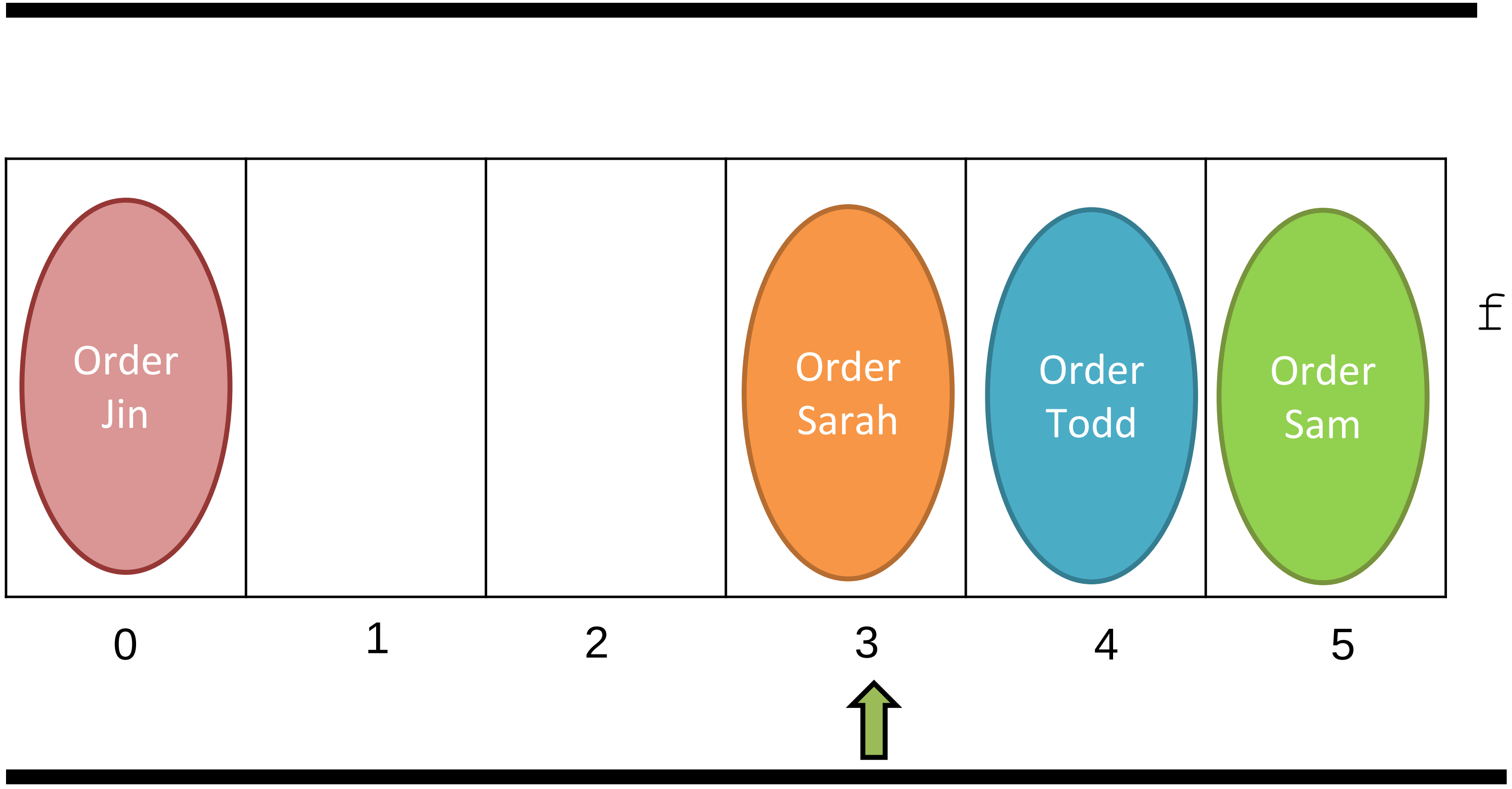
```
data[front] = null
```

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

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

```
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



Let’s **dequqe (<u>again</u>)**

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

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

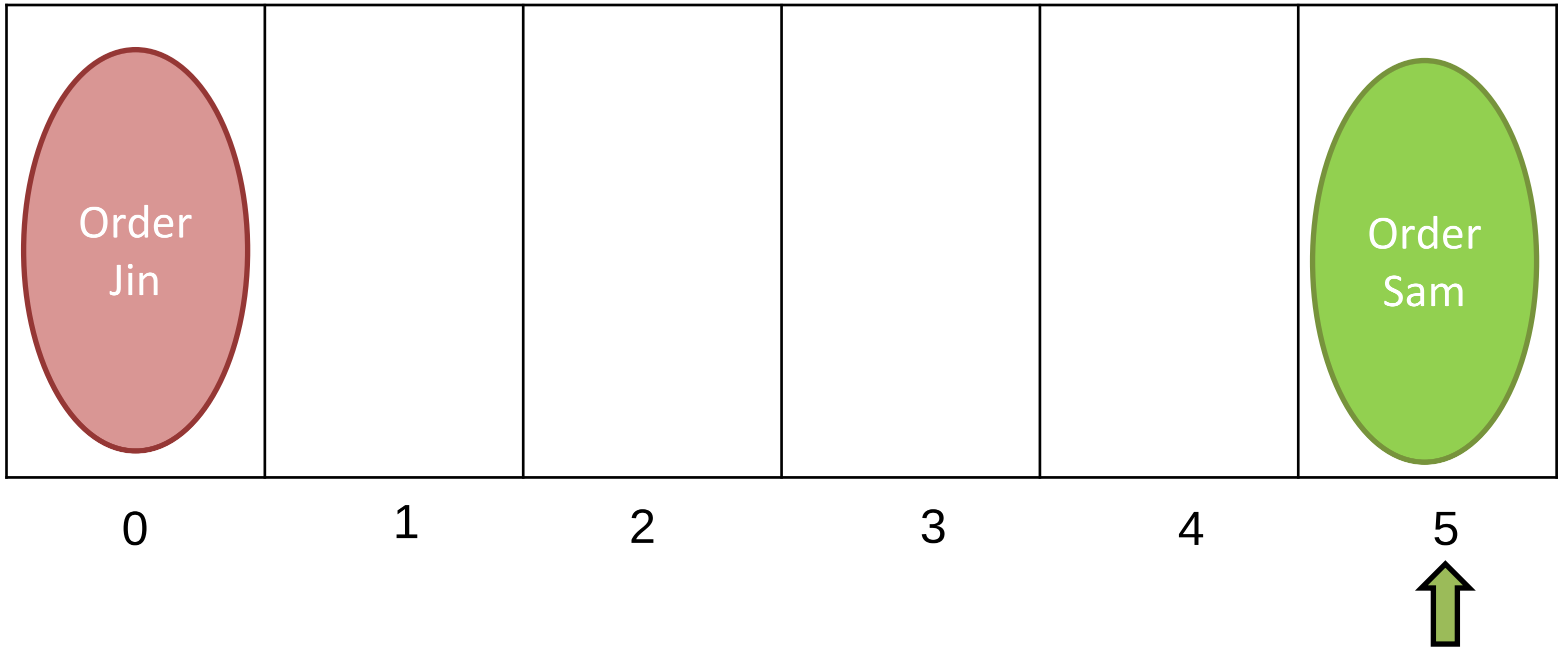
(2+1) % 6 **= 3**

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

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

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

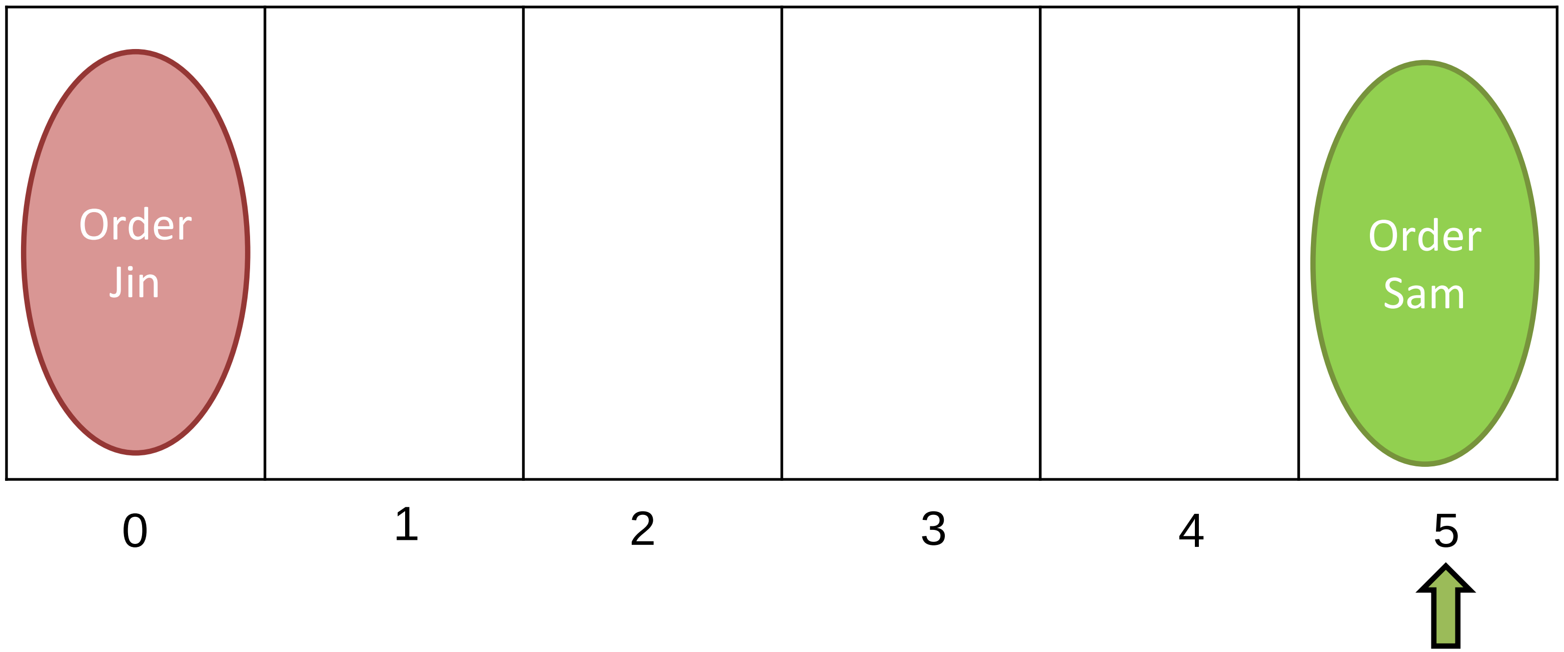
Let’s **dequqe (<u>again</u>)**




```
data[front] = null

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

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

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



Let’s **deque (<u>again</u>)**

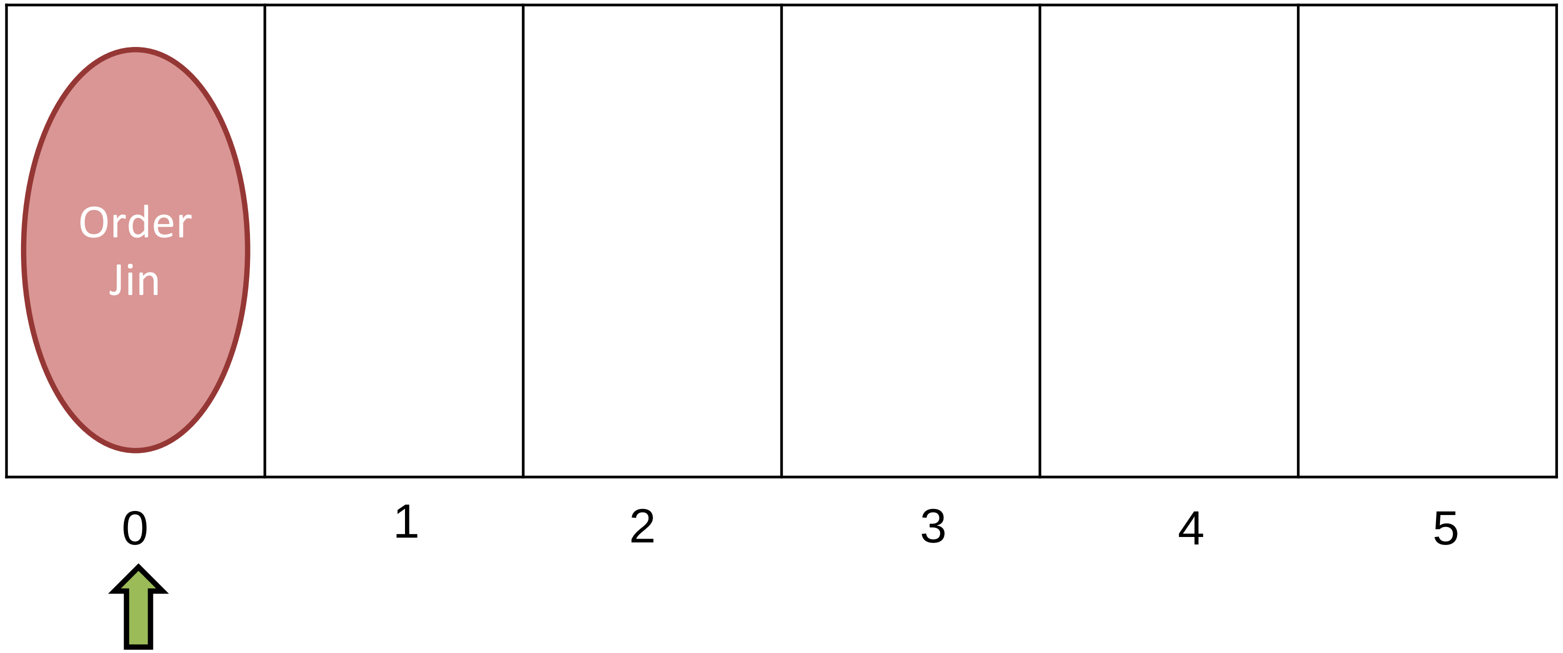
```
data[front] = null

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

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

```
capacity = 6     front = 5
size = 2         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 (<u>again</u>)**




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

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

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

```
capacity = 6     front = 0
size = 1         insert_spot = 0
```

```java
public void enqueue(Order newOrder) {
    if(this.size == this.data.length) {
        System.out.println("Queue is full");
    }

    int insert_spot = (front + size) % (this.data.length);
    data[insert_spot] = newOrder;
    this.size++;

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

```java
public void dequeue() {


    if(this.size == 0) {
        System.out.println("Queue is empty...");
        return;
    }
    else {

        Order o = this.data[front];
        this.data[front] = null;
        front = (front + 1) % this.data.length;
        this.size--;
        System.out.println(o.getName() + " order was removed ");
    }

}
```

```java
public void printQueue() {
	int start = front;
	int counter = 1;
	int n = 0;
	while( n != this.size ) {
		System.out.println(counter + ". " + this.data[start].getName());
		start = (start+1) % this.data.length;
		counter++;
		n++;
	}
}
```

This method will print out the queue in the correct order (there is probably a better way to write this)

The while loop stops once we've printed all N elements in the queue

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		
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) {
                                  O(1)
        this.orders.addLast(newOrder); O(1)
        this.size++; O(1)

}
```

```
public void enqueue(Order newOrder) {

        if(this.size == this.capacity) {
                System.out.println("Error... queue is full");   O(1)
                return;
        }

        int insert_spot = (front + size) % capacity;   O(1)
        this.orders[insert_spot] = newOrder;   O(1)

        this.size++; O(1)
        System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot);
}                                                                           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; O(1)
        for(Order each_order: this.orders) {O(n)
            O(1)each_order.printOrder(counter);
            O(1)counter++;
        }
}
            n = # of elements in queue
```

```
public void printQueue() {

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

}           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)

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)

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

(LIFO)

Stack Runtime Analysis

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