Data Structures

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

- Queue (array based)

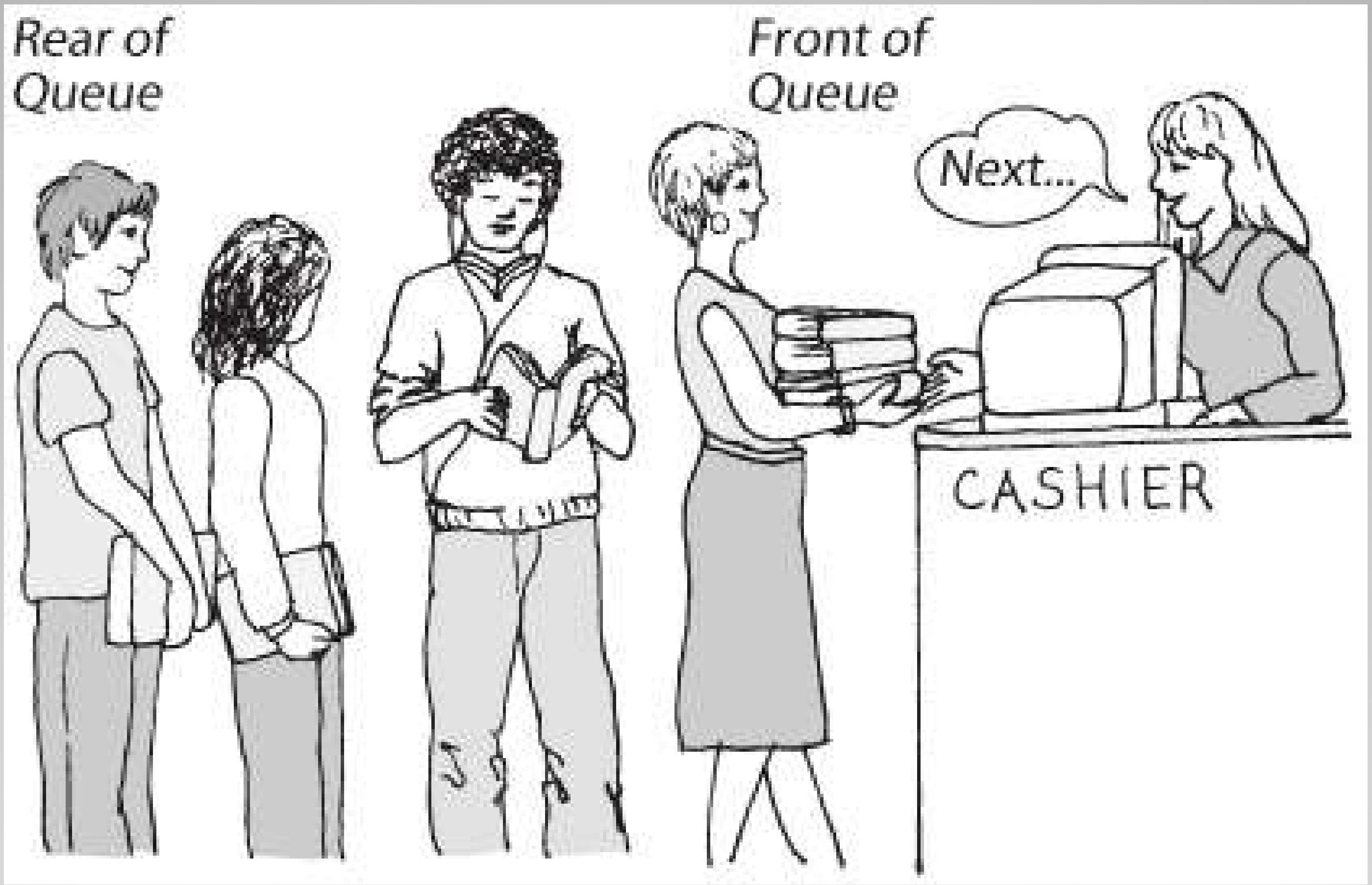
Today’s Lecture

Goals

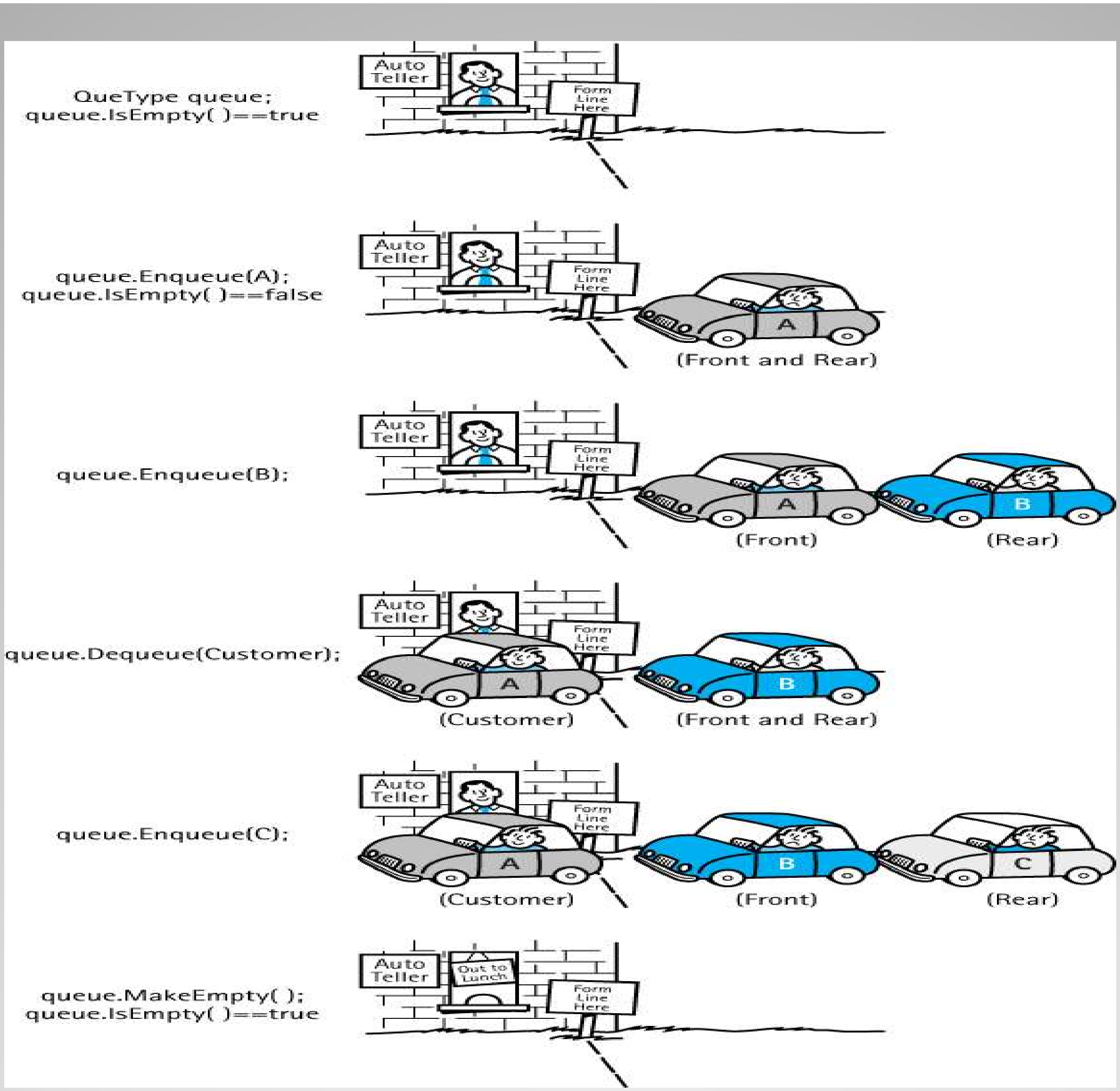
- Describe the structure of a queue and its operations at a logical level
- Demonstrate the effect of queue operations using a particular implementation of a queue
- Implement the Queue ADT, using both a an array-based implementation and a linked implementation
- Discuss Big O runtimes of operations for array-based and linked implementations.

Queue – Logical View

- We will start by looking at the logical view of a queue…



Queues



```
QueType queue;
queue.IsEmpty( )==true

queue.Enqueue(A);
queue.IsEmpty( )==false

queue.Enqueue(B);

queue.Dequeue(Customer);

queue.Enqueue(C);

queue.MakeEmpty( );
queue.IsEmpty( )==true
```

Queues

Queue

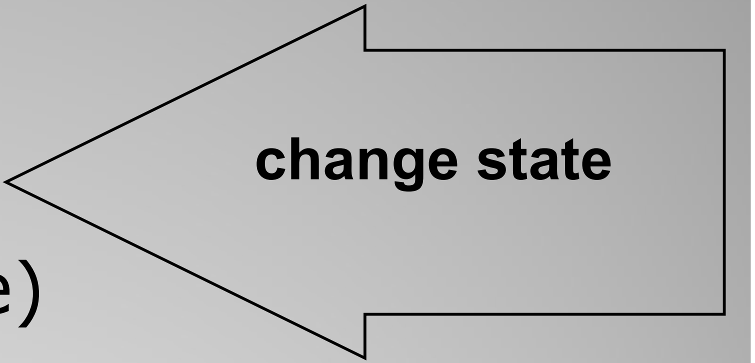
An abstract data type in which elements are added to the rear and removed from the front; a “first in, first out” (FIFO) structure

- What operations would be appropriate for a queue?

Queue – Logical View

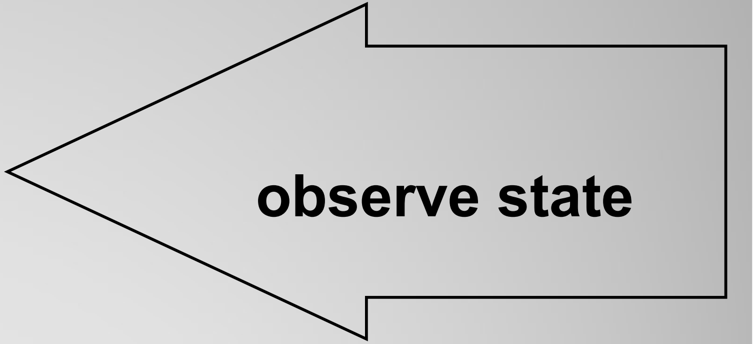
Queue – Logical View

- **Transformers**
  - makeEmpty
  - enqueue (Insert or Add)
  - dequeue (Delete or Remove)



change state

- **Observers**
  - isEmpty
  - isFull



observe state

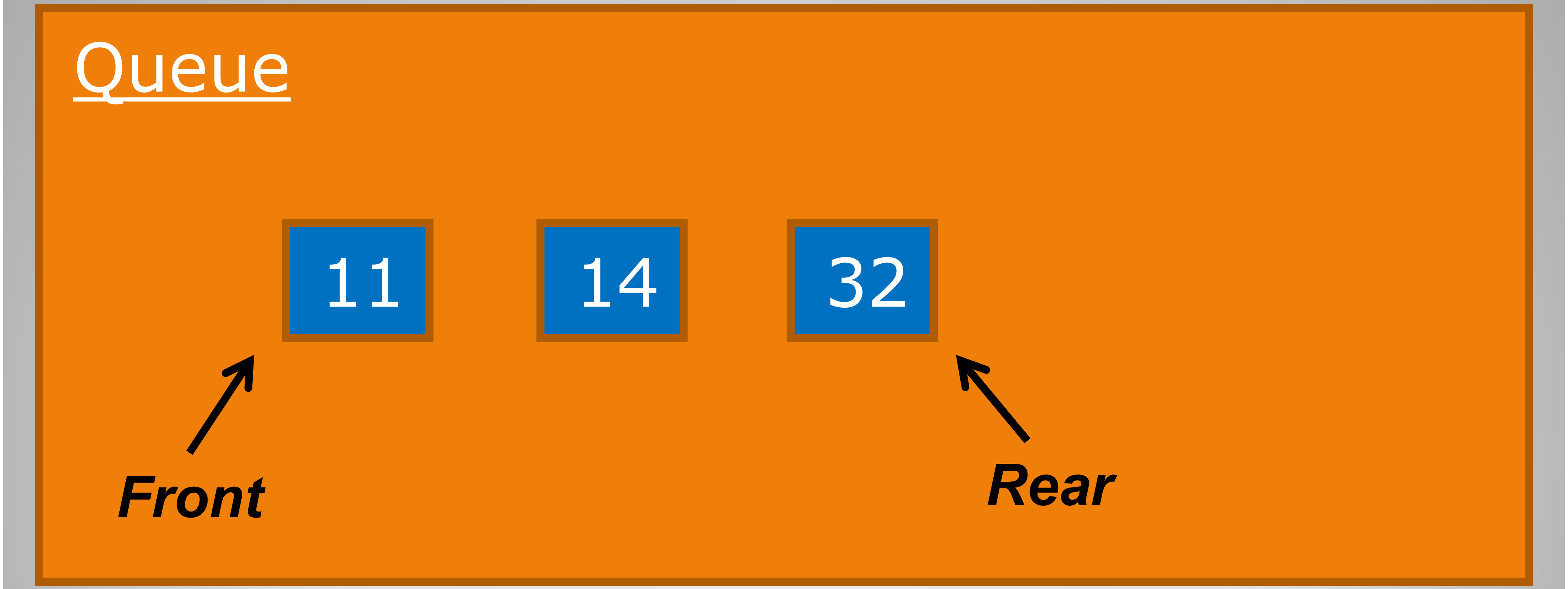
- What does a queue look like if we insert the following elements (in the given order):

  11, 14, 32

Queue – Logical View

Queue – Logical View

- *Insert: 11, 14, 32*
- *Here is the resulting queue…*



Queue

11

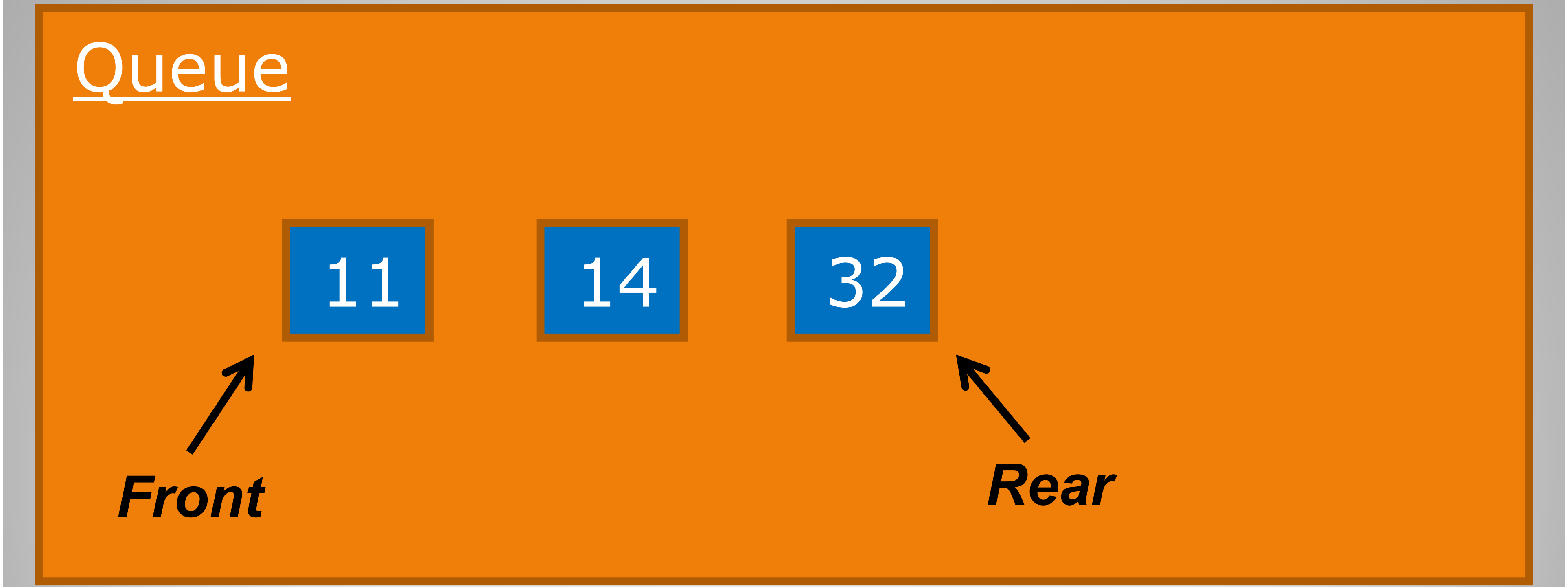
14

32

Front

Rear

- *What if we remove an element?*
- *Where does it get removed from?*
- *Can we remove from in the middle?*



Queue

11

14

32

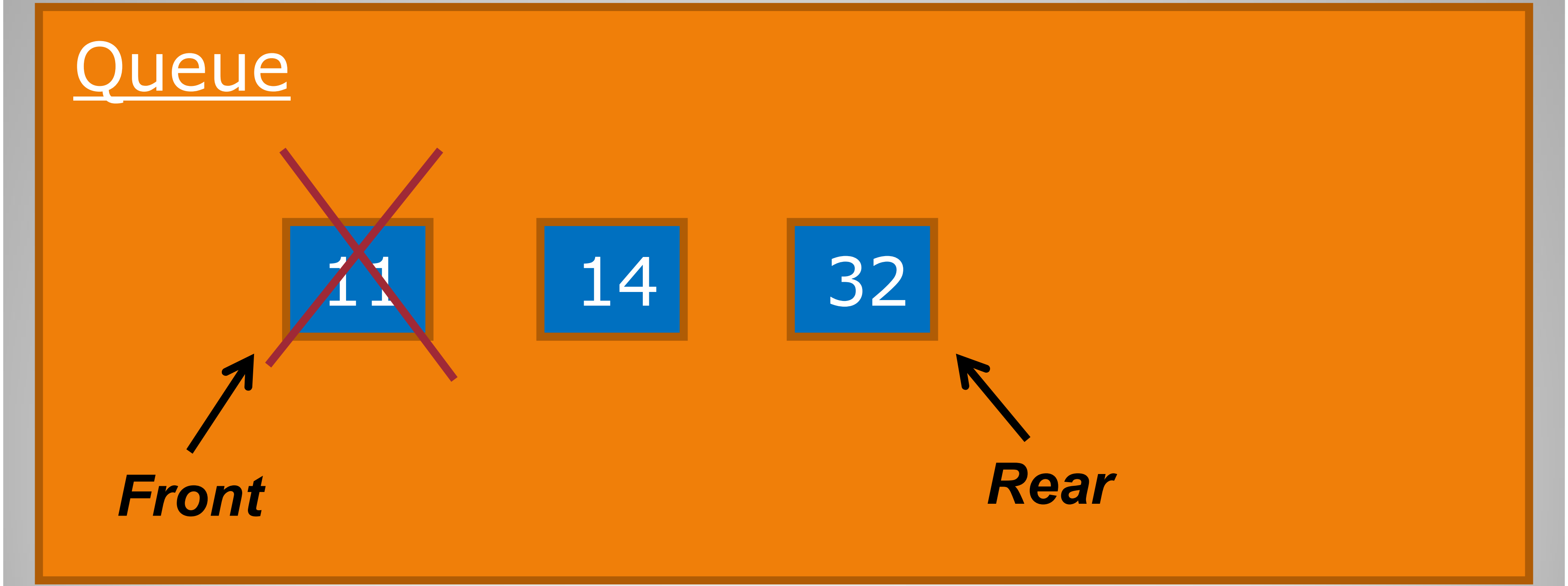
Front

Rear

Queue – Logical View

Queue – Logical View

- *What if we remove an element?*
- *Where does it get removed from? THE FRONT*
- *Can we remove from in the middle? NO*



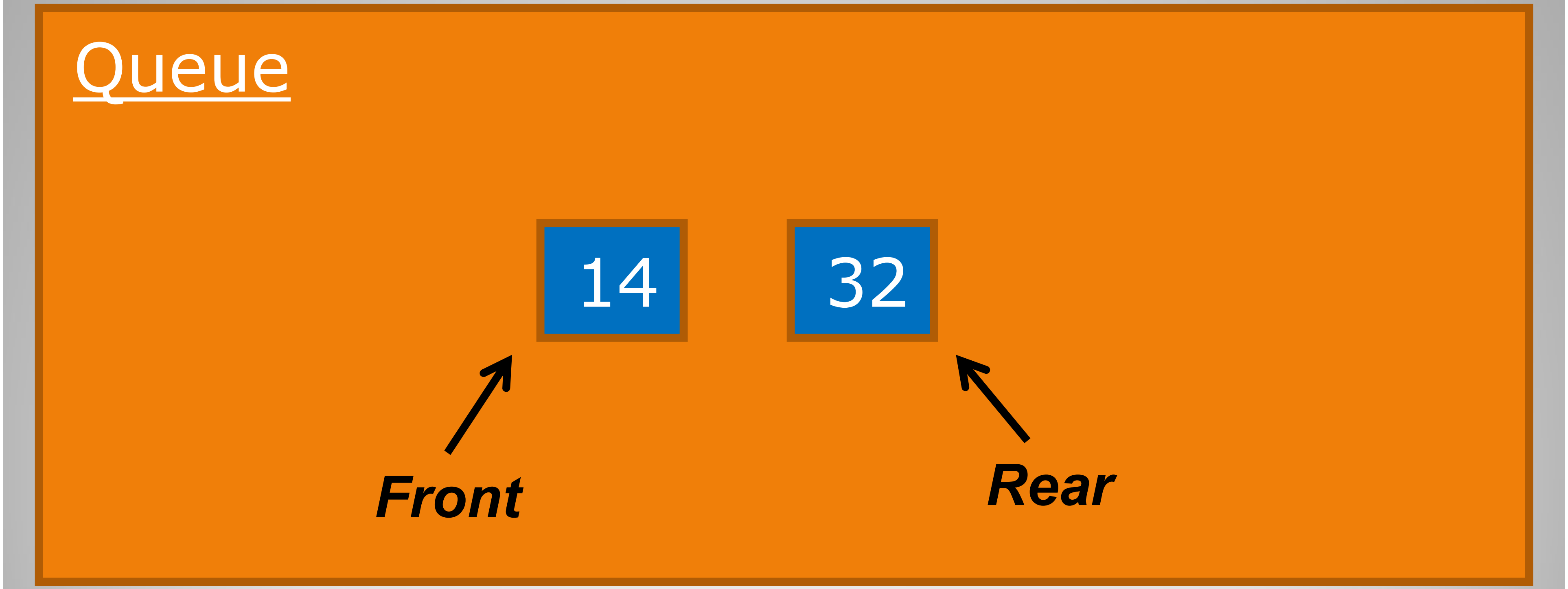
Queue

~~11~~ 14 32

Front

Rear

- *Queue after removing one element.*
- *Can we add an element after we remove. For example, Enqueue(77)?*
- *Where does it get added?*



Queue

14

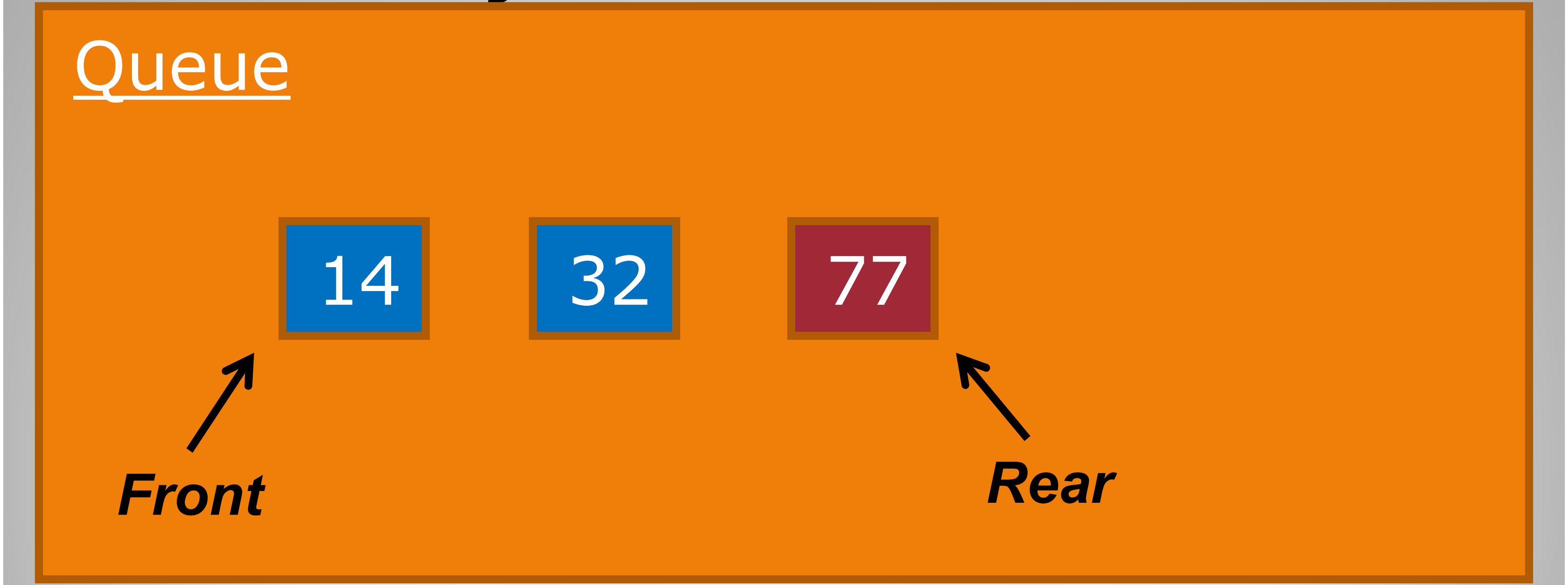
32

Front

Rear

Queue – Logical View

- *Queue after removing one element.*
- *Can we add an element after we remove. For example, Enqueue(77)?*
- *Where does it get added? REAR*



Queue – Logical View

- Now we will look at an array-based implemenation of a queue.

- ***Exam questions will be based on the slide implementation of the array-based queue*** and not one from another source.

Queue (Array)

Queue

Here is the interface for the Queue ADT:

```
public interface Queue {
   boolean isEmpty();
   boolean isFull();
   void enqueue(int item) throws Exception;
   int dequeue() throws Exception;

   void makeEmpty();
}
```

The public interface of a queue should be the same for both the array-based and linked implementations

Queue Array-based Implementation

- Keep track of the front and rear indexes

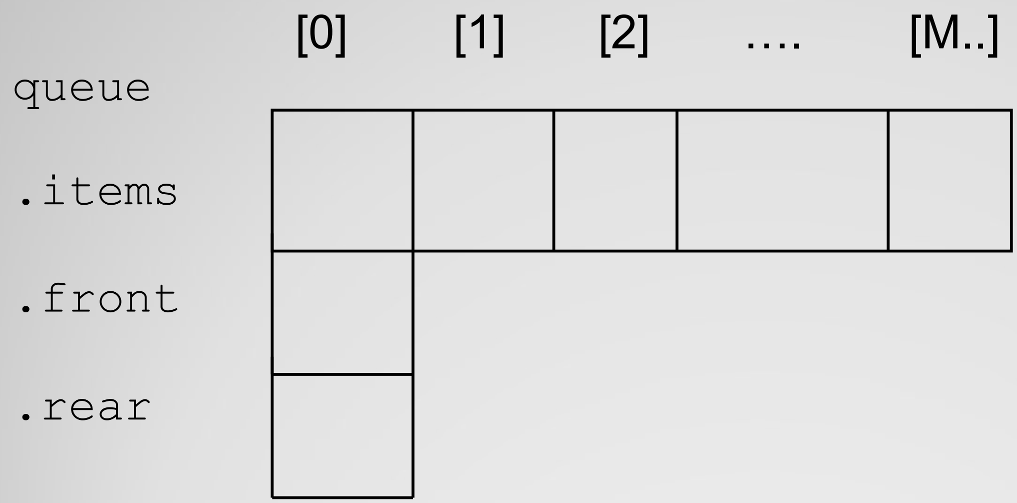
- Rear is the actual index of the last element.
- Front is positioned one before the front element.
- If Front and Rear are equal then the queue is empty.

- Note: Other implementation may give code for an array-based queue where front indicates the actual first element and rear indicates the actual last element. This is not the same as the slide implementation given here.

Queue (Array) - Implementation

Physical Level

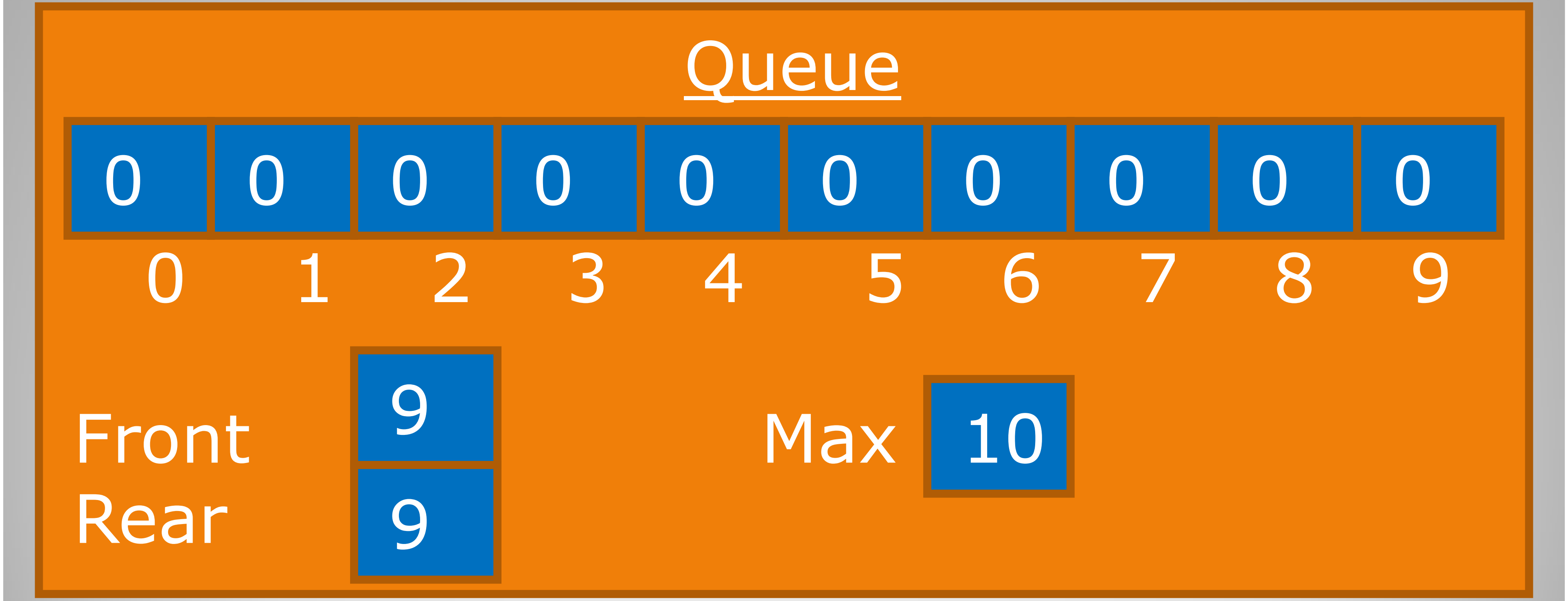
```
public class QueueArrayBased implements Queue {
    Declare int front      // An array index. One BEFORE the first element.
    Declare int rear        // An array index. The actual last element.
    Declare int max        // Need to know the size of the array
    Declare int items[]   // Array stores the queue data
    // Public members go here…
```



Queue (Array) - Implementation

Queue (Array)

- Rear is the actual index of the last element.
- Front is positioned one before the front element.
- If Front and Rear are equal then the queue is empty.



Queue

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Front 9

Rear 9

Max 10

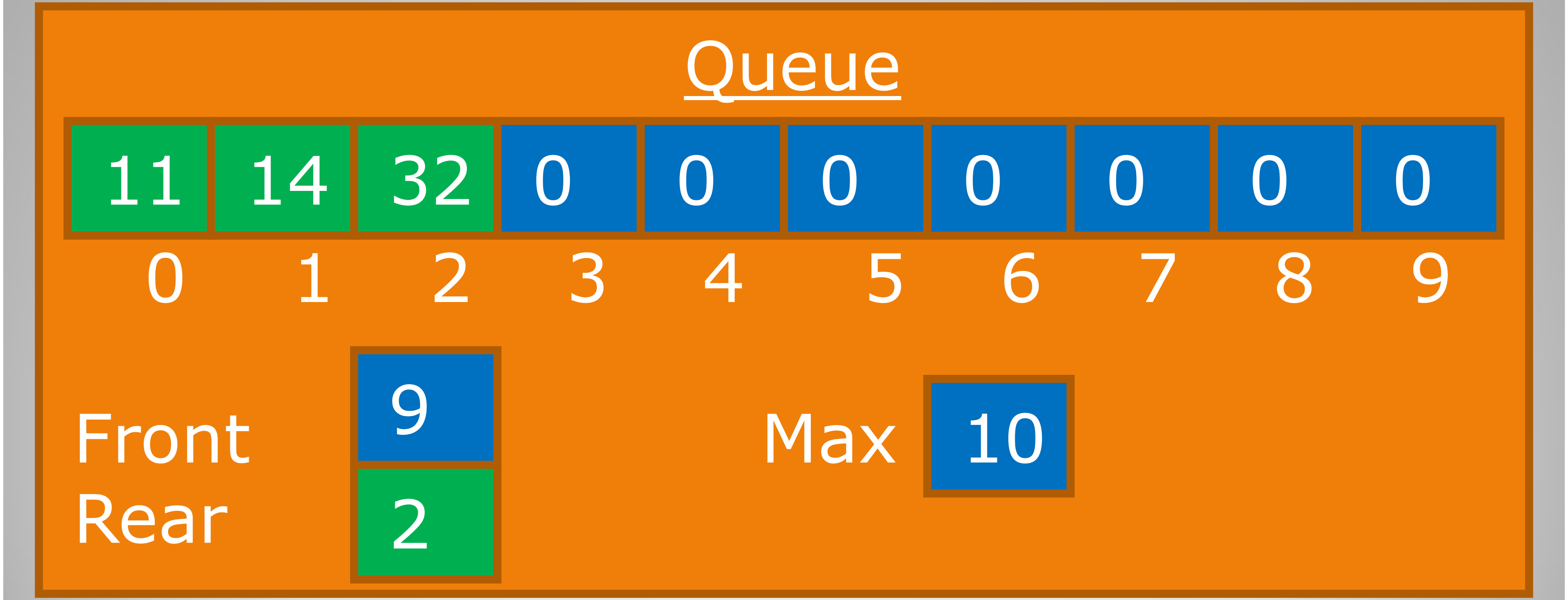
Queue (Array)

- What does a queue look like internally using an array-based implementation assuming we run the following code…

```
Declare Queue q
Set q to new QueueArrayBased() instance

q.enqueue(11)  // Adds to queue
q.enqueue(14)  // Adds to queue
q.enqueue(32)  // Adds to queue
```

Queue (Array)

- Rear is the actual index of the last element.
- Front is positioned one before the front element.
- *Only the rear index changes when you enqueue.*



Queue

11	14	32	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

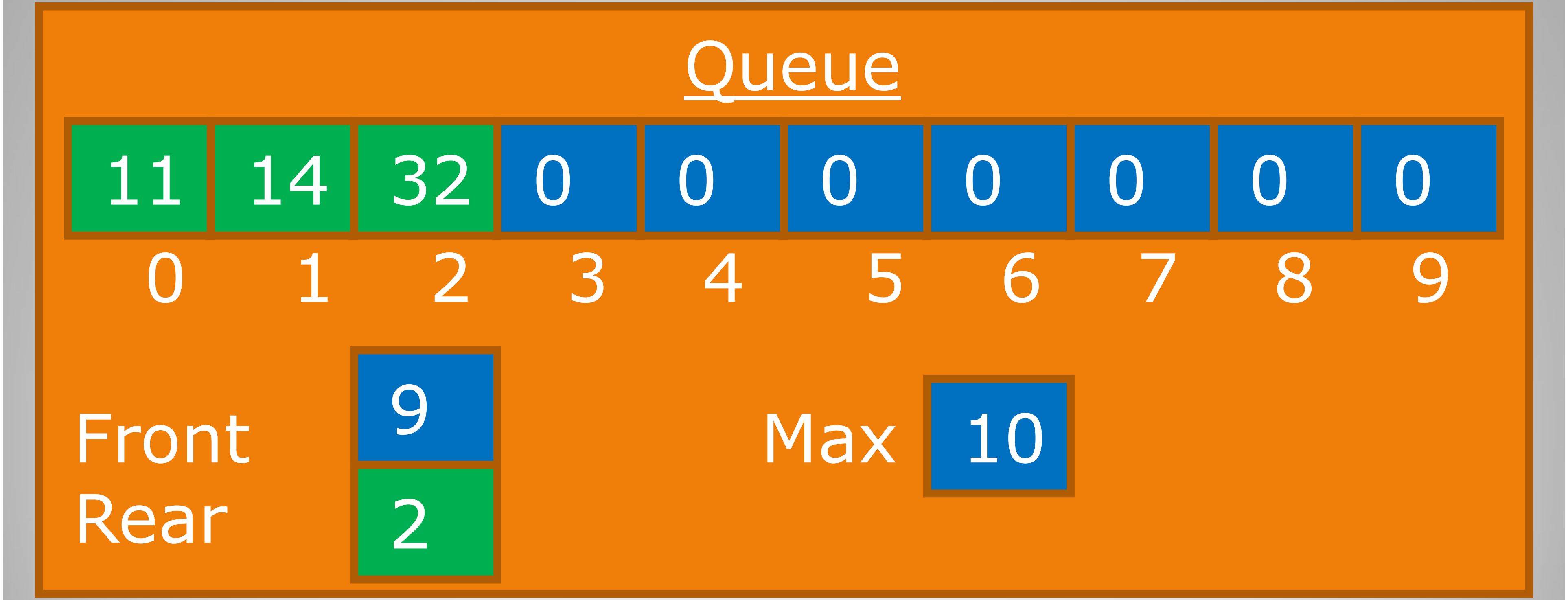
Front	9
Rear	2
Max	10

Queue (Array)

- Now remove an element from the queue…

```
Declare item // Gets returned value
Set item to q.dequeue()  // Removes
```

What happens?



Queue

11	14	32	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Front 9

Rear 2

Max 10

Queue (Array)

Declare int item

Set item to q.dequeue()

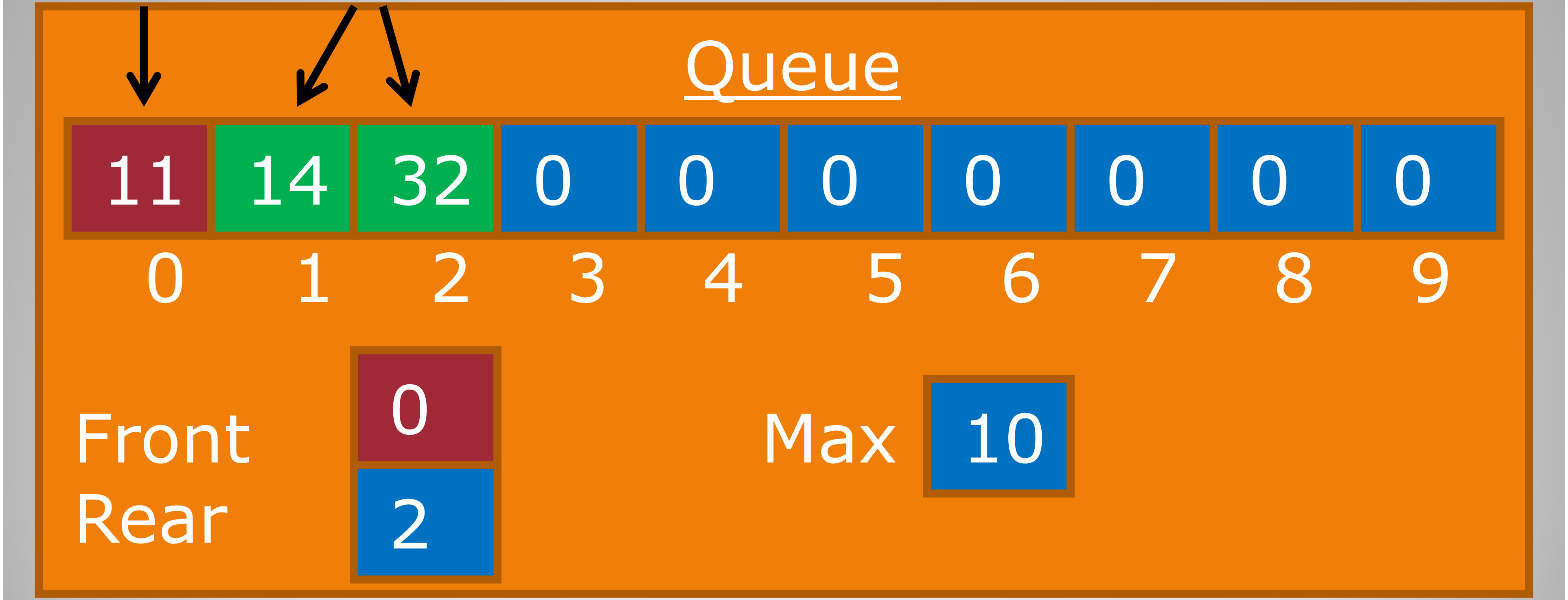
Front is moved one element.

Logically removes first element.

Note: *Front* stands for the index that is *one before* the first element.

Removed

Queue elements



Queue

0	1	2	3	4	5	6	7	8	9
11	14	32	0	0	0	0	0	0	0

Front	0
Rear	2
Max	10

Queue (Array) - Dequeue

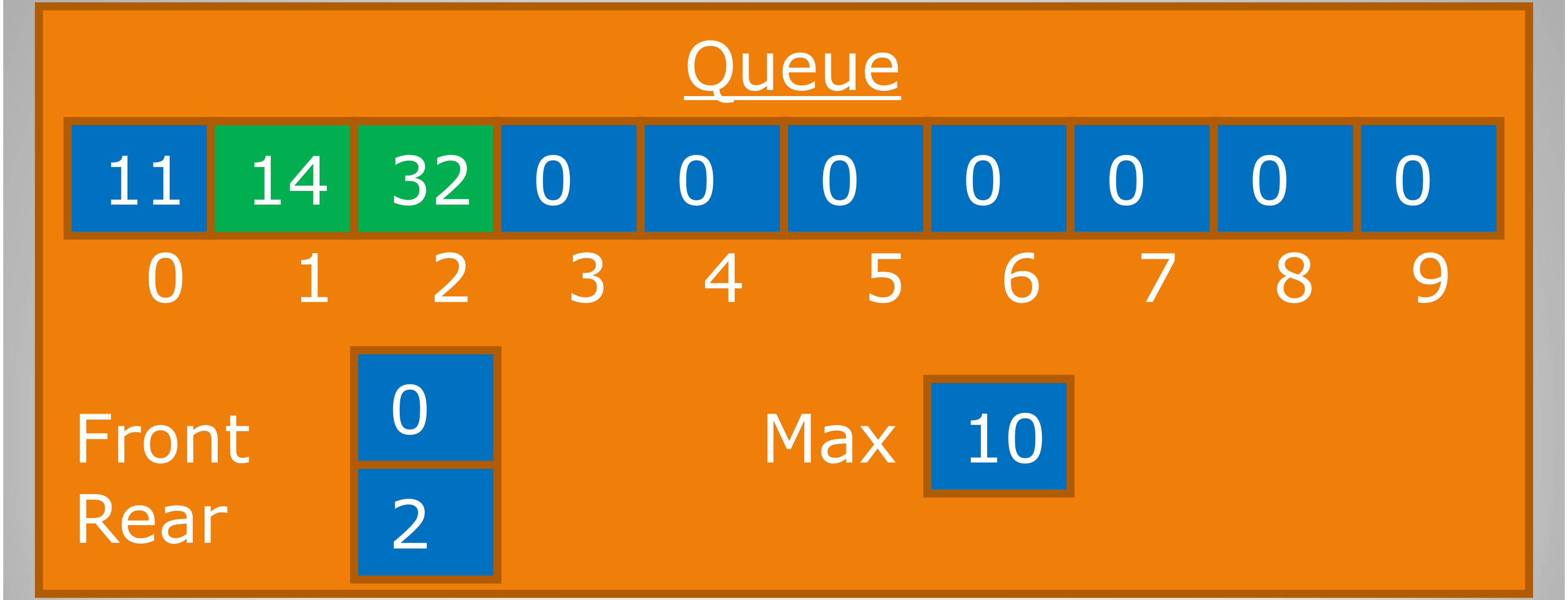
```
dequeue() returns int throws Exception
    Declare int item

    if (queue is empty)
        throw exception "Empty Queue"
    else
        Set front to (front + 1) % max
        Set item to items[front]

    return item
```

1. Increment the front index

2. Get the data to return

- Now run the following…

q.enqueue(77)



Queue

11	14	32	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

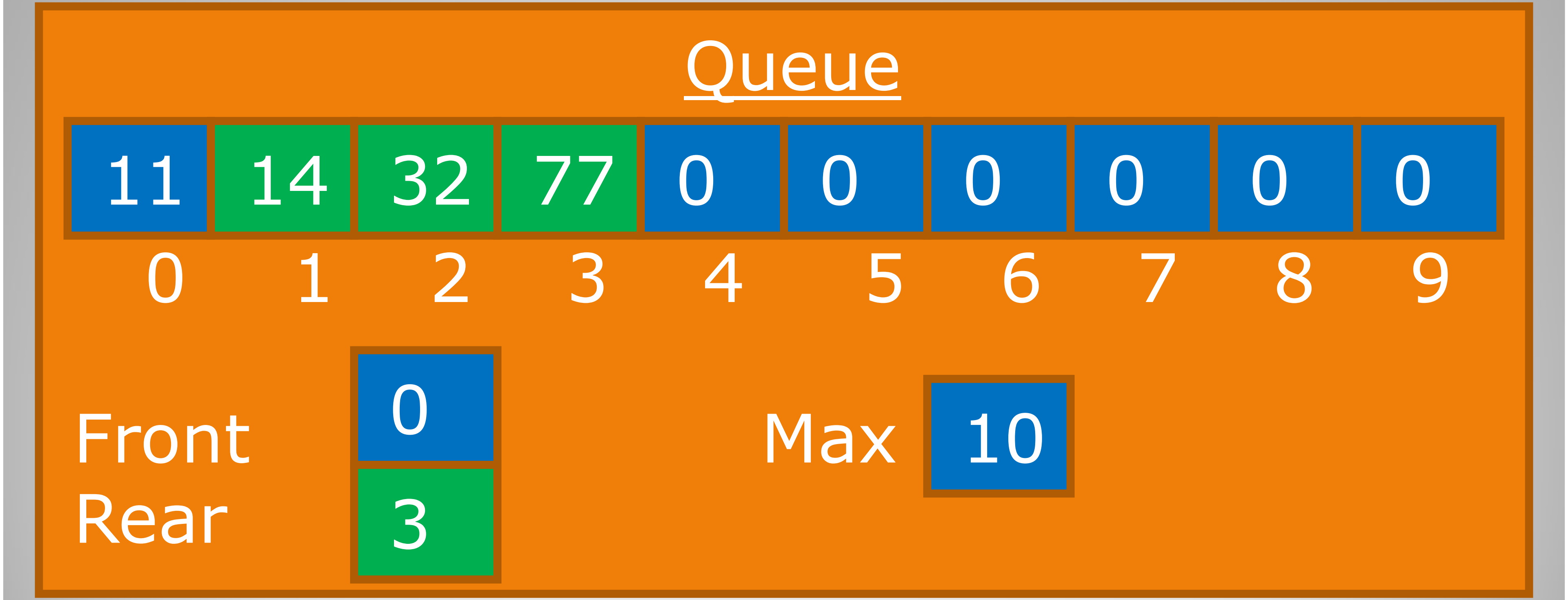
Front 0

Rear 2

Max 10

Queue (Array)

- ***Rear is now index 3. 77 is on the queue.***
- Note: Only 14, 32 and 77 are actually on the queue.



Queue

11	14	32	77	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Front	0
Rear	3
Max	10

Queue (Array)

Queue (Array) - Enqueue

```
enqueue(int item) throws Exception
    if (queue is full)
        throw exception "Full Queue"
    else
        Set rear to (rear + 1) % max
        Set items[rear] to item
```

Update rear.

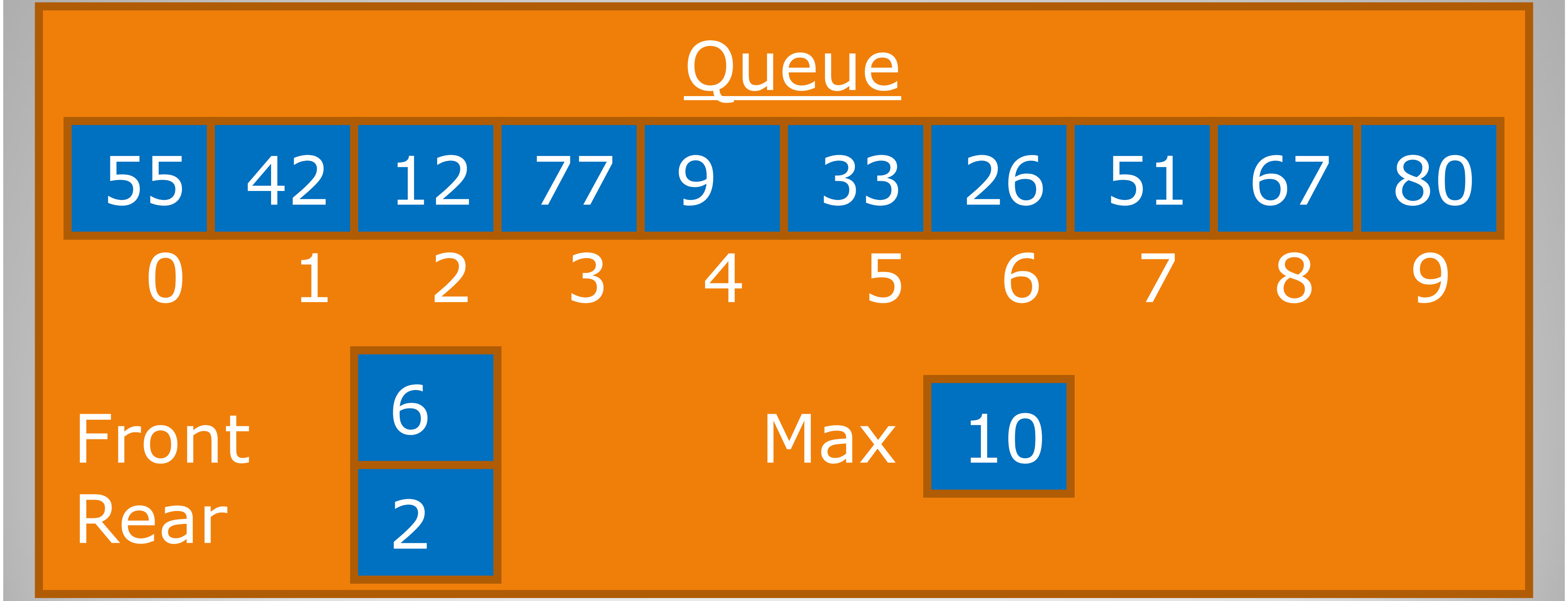
Add 1 to rear to make it go to the next index. Max is the size of the underlying array, so you need to mod by that in case rear went off the end.

- Now look at the next queue and determine which elements are on it…

Queue (Array)

Queue (Array)

- Which elements are actually on the queue?



Queue

0	1	2	3	4	5	6	7	8	9
55	42	12	77	9	33	26	51	67	80

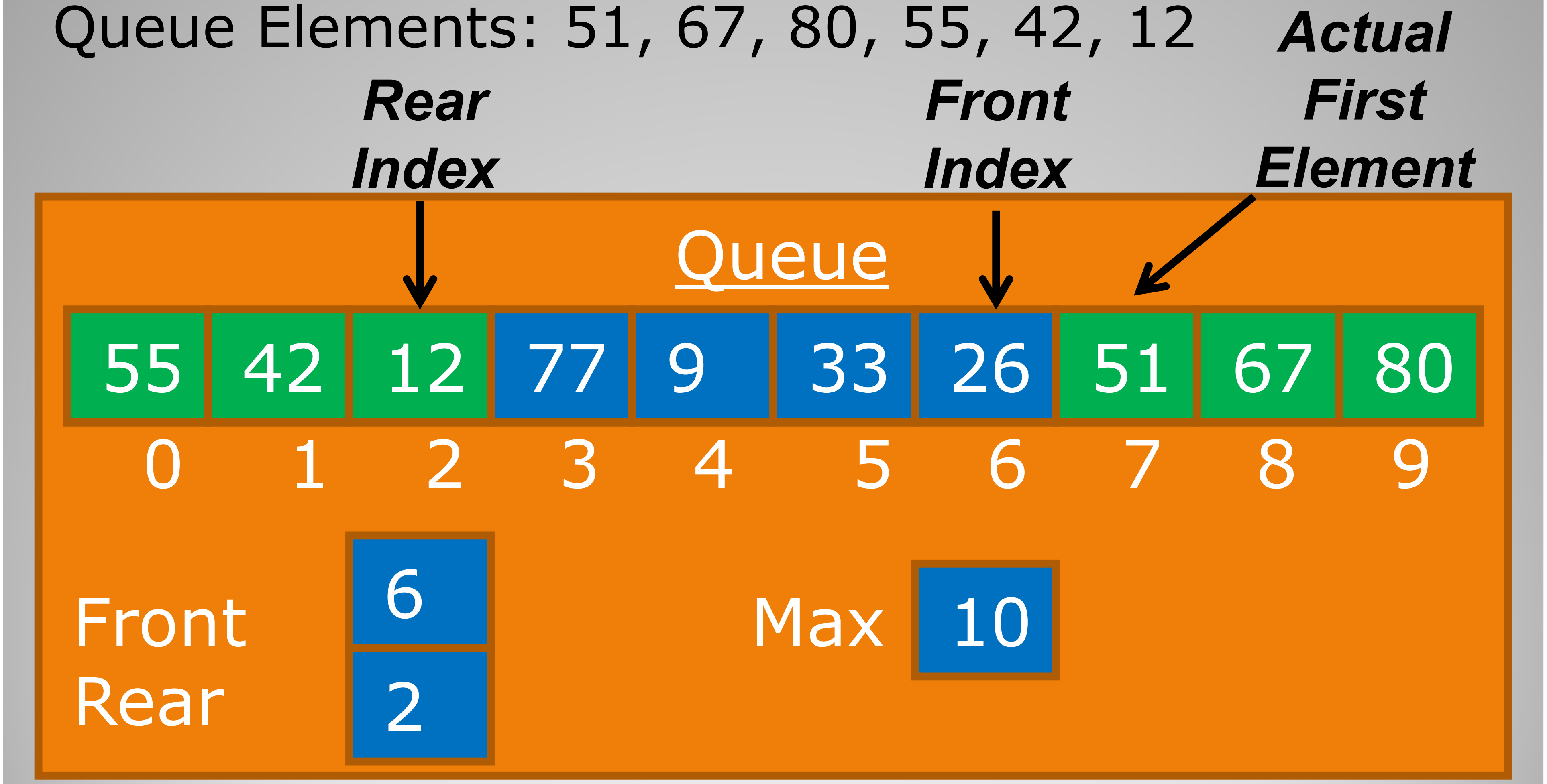
Front 6

Rear 2

Max 10

LOGICALLY THE QUEUE IS:

Queue Elements: 51, 67, 80, 55, 42, 12



Rear Index

Front Index

Actual First Element

Queue

55	42	12	77	9	33	26	51	67	80
0	1	2	3	4	5	6	7	8	9

Front	6
Rear	2

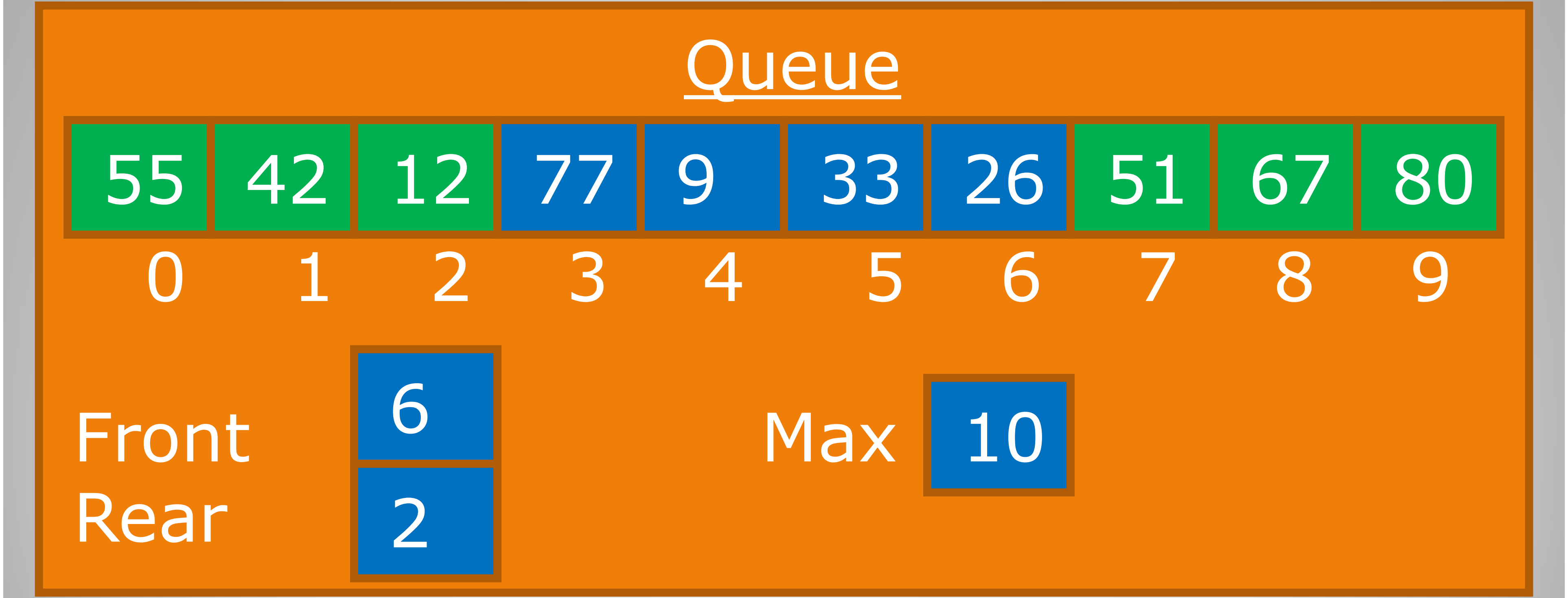
Max 10

Queue (Array)

- Now run the following code…

```
q.makeEmpty()
```

What happens?



Queue

0	1	2	3	4	5	6	7	8	9
55	42	12	77	9	33	26	51	67	80

Front 6

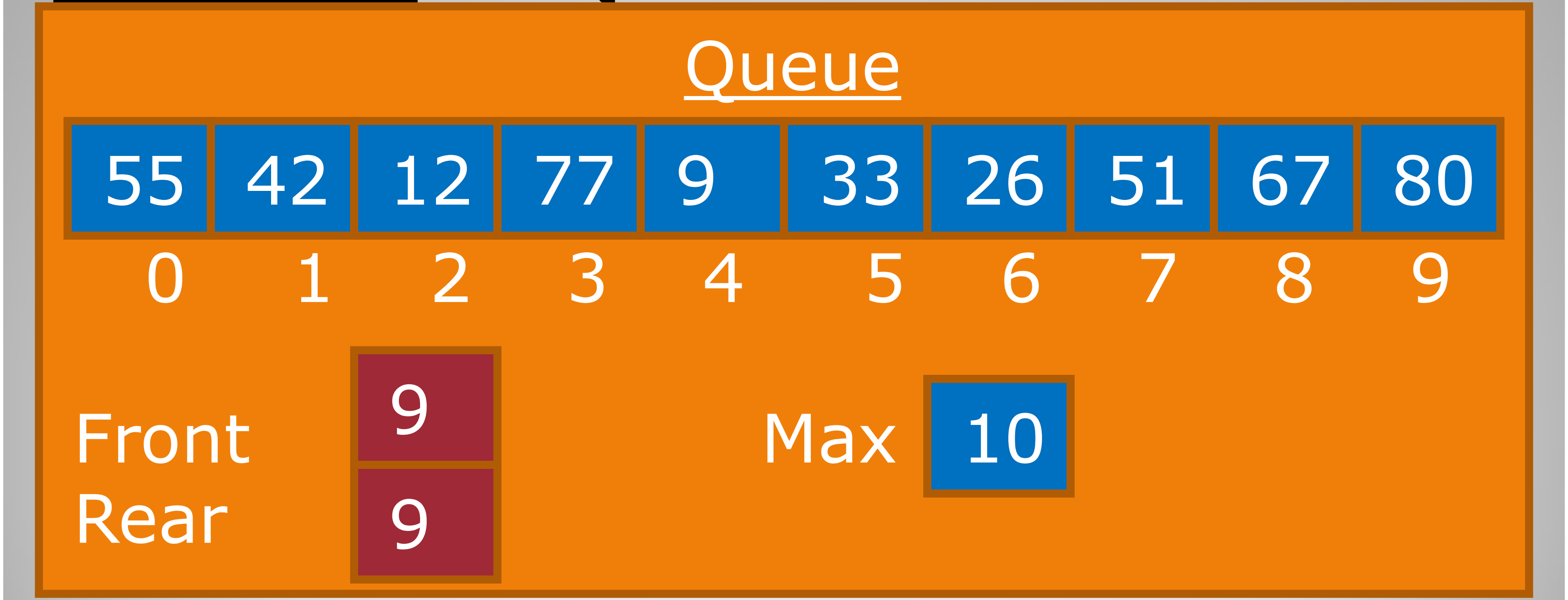
Rear 2

Max 10

Queue (Array)

- **<u>EMPTY QUEUE</u>**
- Front and rear indexes are equal.

<u>LOGICALLY</u> THE QUEUE IS EMPTY!



<u>Queue</u>

55	42	12	77	9	33	26	51	67	80
0	1	2	3	4	5	6	7	8	9

Front 9

Rear 9

Max 10

Queue (Array)

Queue (Array)

- What does a full queue look like internally?

Queue (Array)

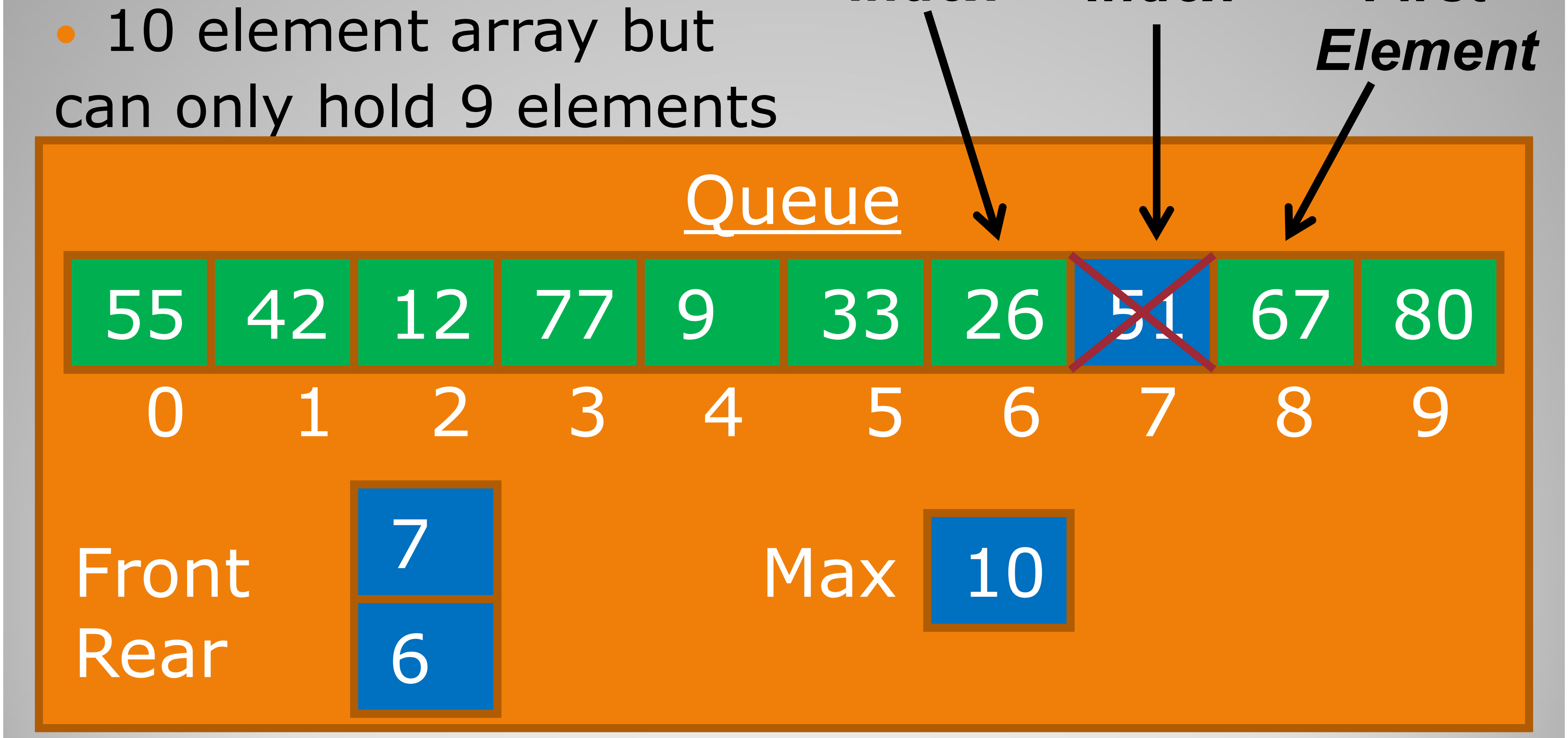
- **FULL QUEUE**
- Front element unusable!
- 10 element array but

can only hold 9 elements

Rear Index

Front Index

Actual First Element



Queue

55	42	12	77	9	33	26	~~51~~	67	80
0	1	2	3	4	5	6	7	8	9

Front	7
Rear	6
Max	10

Queue (Array) – Other functions

```
QueueArrayBased Constructor
    Set max to 10
    Set items to new int[max]
    Set front to max - 1
    Set rear to max - 1


isEmpty() returns boolean
    return ( rear == front)


isFull() returns boolean
    return ( (rear + 1) % max) equals front


makeEmpty()
    Set front to rear
```

Note: This queue only holds 9 elements. If you want the queue to hold 10 elements then you need to set max to 11.

Operation	Cost
makeEmpty	O(1)
isFull	O(1)
isEmpty	O(1)
enqueue	O(1)
dequeue	O(1)
Constructor	O(1)

Big-O Comparison – Queue (Array)

- **End of Slides**

End of Slides