The Queue ADT

Computer Science 112
Boston University

Christine Papadakis

Queue ADT

- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (**FIFO**) (vs. a stack, which is last in, first out)
  - you can only access the item that is currently at the front
- Operations:
  - insert: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full

Queue ADT

- A queue is a sequence in which:
  - items are removed from the front
    - first in, first out (or FILO, which is last in, first out)
  - you can only access what is currently at the front
- Operations:
  - **insert**: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full

enqueue

Queue ADT

- A queue is a sequence in which:
  - items are removed from the front
    - first in, first out (or FIFO, which is last in, first out)
  - you can only access the item that is currently at the front
- Operations:
  - insert: add an item at the rear of the queue
  - **remove**: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full

dequeue

Queue ADT

- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - you can only access the item that is currently at the front
- Operations:
  - insert: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full
- Example: a queue of integers

start: 12 8

insert 5: 12 8 5

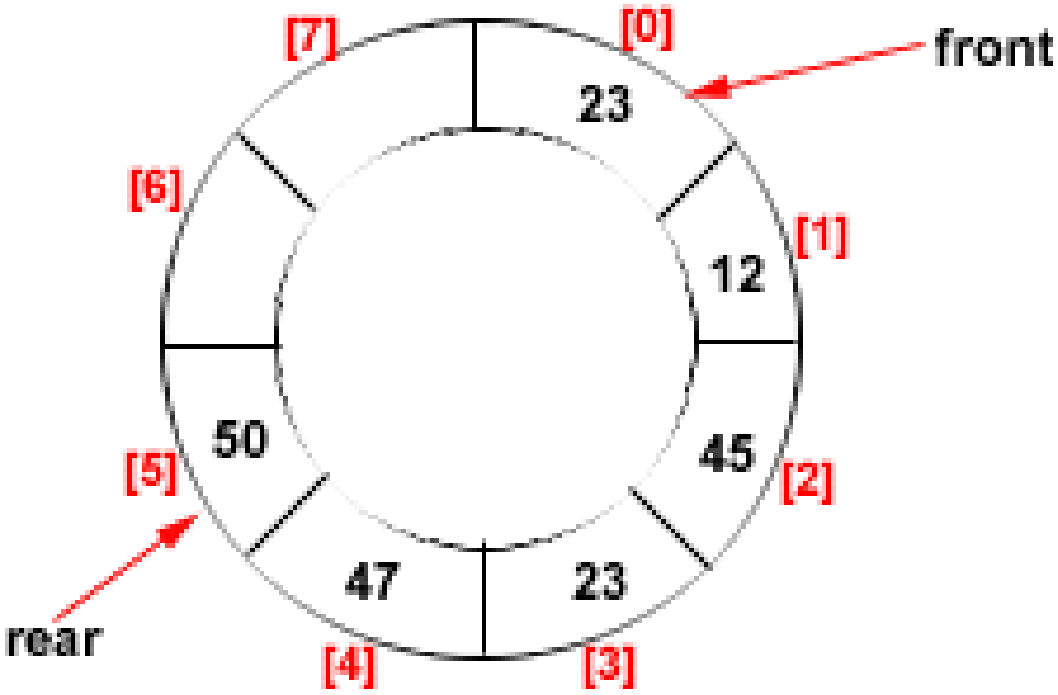
remove: 8 5

Our Generic Queue Interface

```
public interface Queue<T> {
    boolean insert(T item);
    T remove();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `insert()` returns `false` if the queue is full, and `true` otherwise.
- `remove()` and `peek()` take no arguments, because we know that we always access the item at the front of the queue.
  - return `null` if the queue is empty.
- Here again, we will use encapsulation to ensure that the data structure is manipulated only in valid ways.

Using an array to implement a Queue



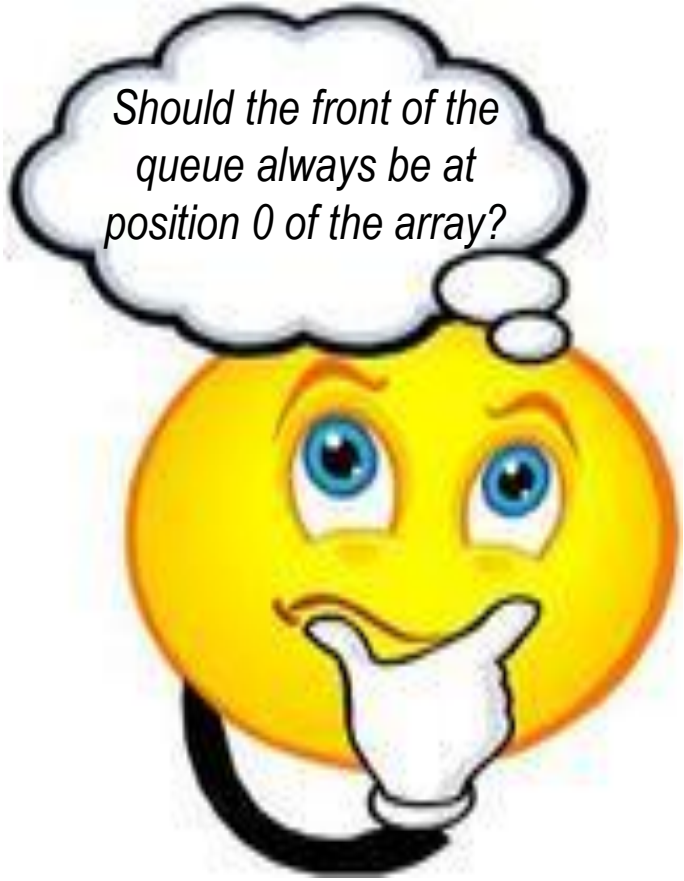
Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```



- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

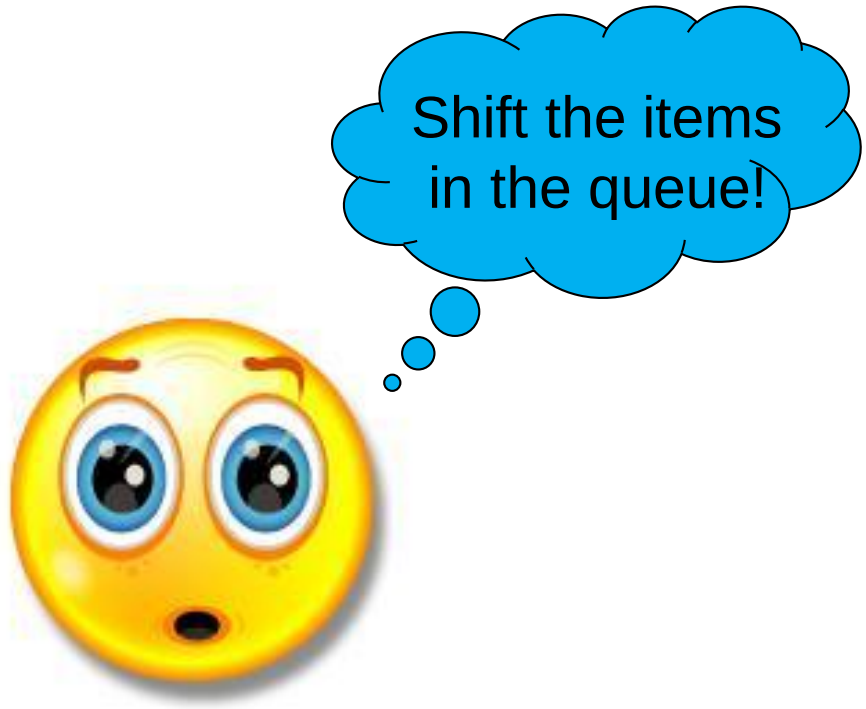
Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

Implementing a Queue Using an Array

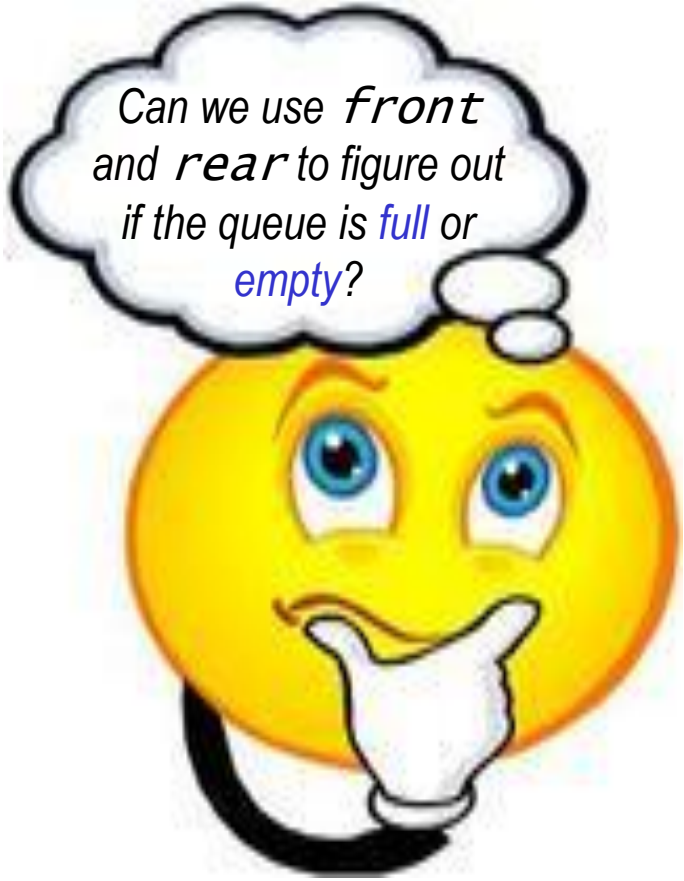
```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```



- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```



- We also need to maintain the number of items in the queue:
  - Why? Could we just rely on `front` and `rear` for all operations we need to perform on the queue?

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- We also need to maintain the number of items in the queue:
  - No! and I will explain why shortly…

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

But first, let's create an instance of our ArrayQueue:

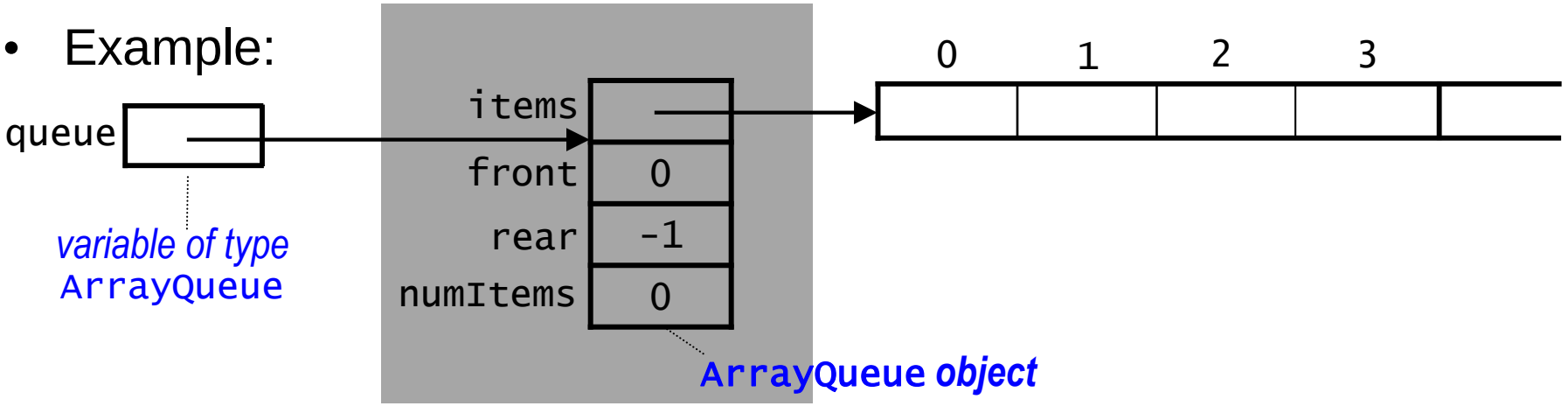
Queue queue = new ArrayQueue<Integer>(*size*);

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

After creating an instance of our ArrayQueue:

- Example:

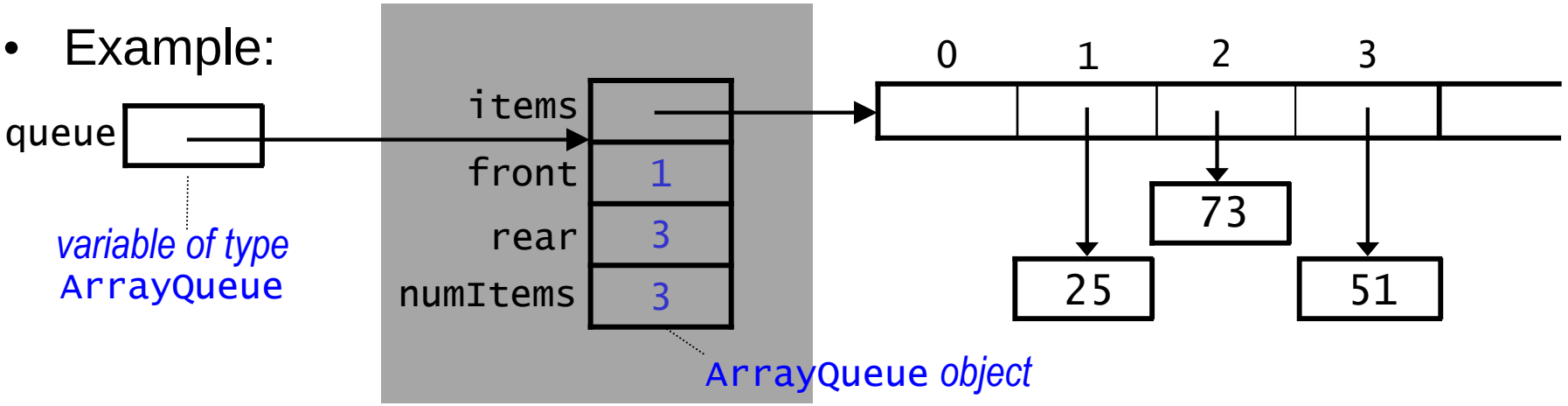


Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

…. And with some items in the queue…

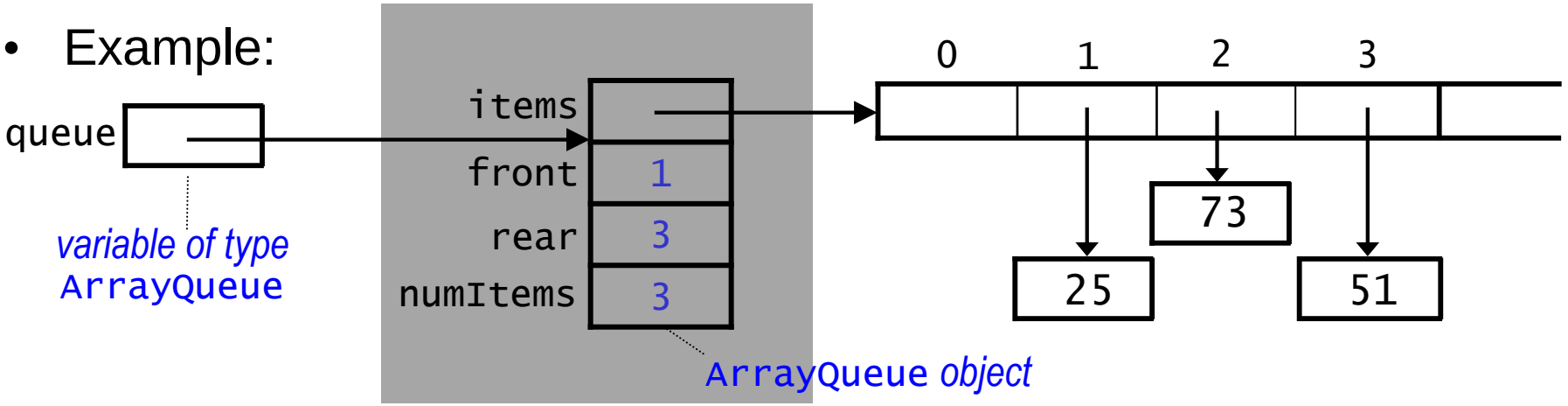
- Example:



Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- Example:

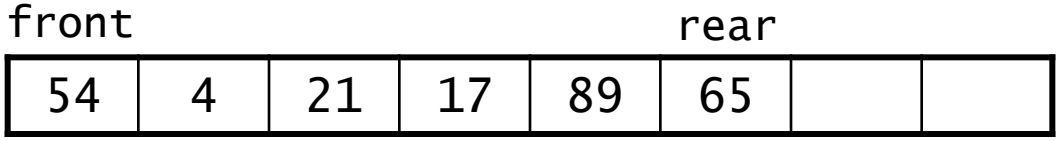


In an array implementation, to optimize the efficiency of the operations on our queue we have to avoid ….

Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

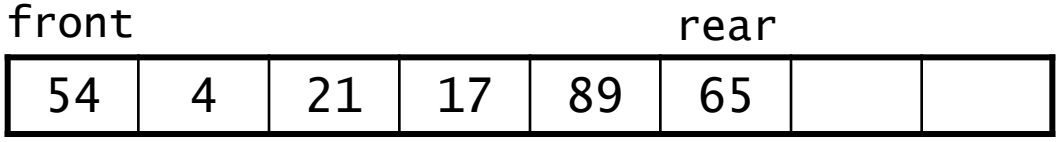
  example: a queue of integers:



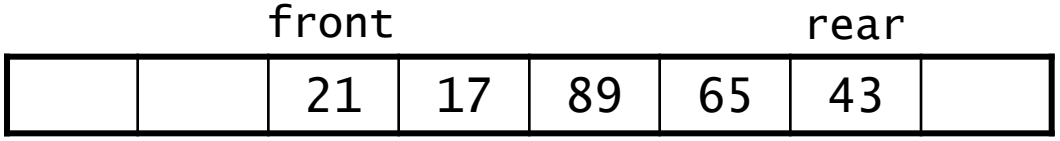
Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

  example: a queue of integers:



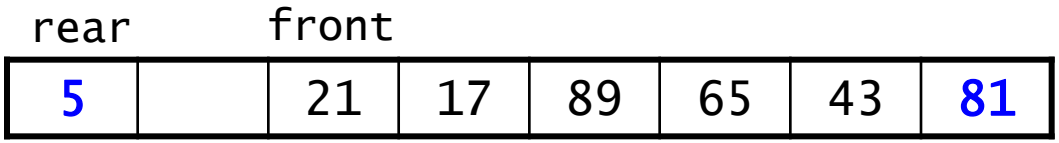
  the same queue after removing two items and inserting one:



  To insert two or more additional items, would need to shift items left.

- Solution: maintain a ***circular queue***. When we reach the end of the array, we wrap around to the beginning.

  the same queue after inserting two additional items:



Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

example: a queue of integers:

front rear

54	4	21	17	89	65		

the same queue after removing two ite… one:

front

		21	1…

To insert two or more additi… tems left.

> *How should we compute the index to know that we need to wrap around back to the beginning of the array?*

- Solution: maintain a *circula*… ch the end of the array, we wrap arou… to th… g…

insert 5: wrap arou…!

rear front

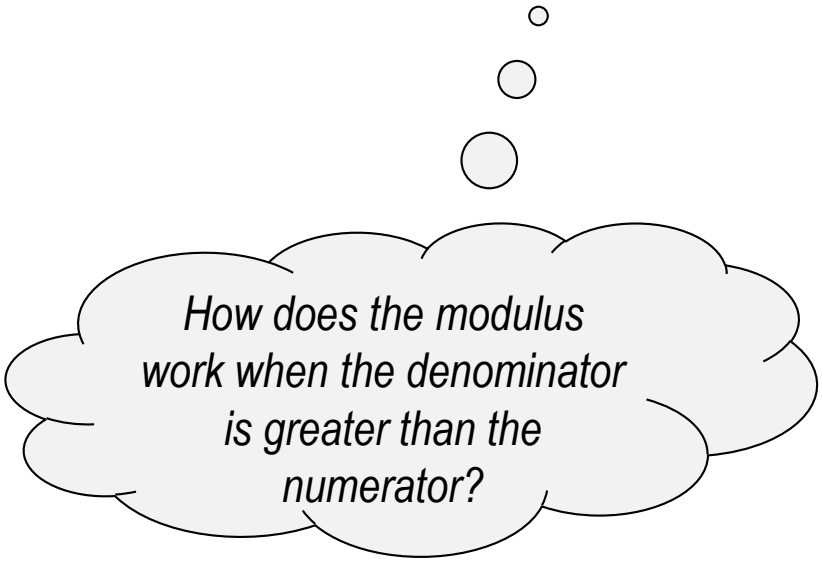
5		21	17	89	65	43	81

A Circular Queue

- To get the front and rear indices to wrap around, we use the modulus operator (%).

- x % y = the remainder produced when you divide x by y
  - examples:
    - 10 % 7 = 3
    - 36 % 5 = 1

```
3 % 7 = 3
5 % 7 = 5
```

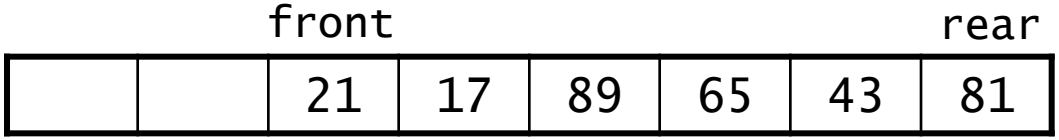


A Circular Queue

- To get the front and rear indices to wrap around, we use the modulus operator (%).
- `x % y` = the remainder produced when you divide `x` by `y`
  - examples:
    - `10 % 7 = 3`
    - `36 % 5 = 1`
    - `3 % 7 = 3`
    - `5 % 7 = 5`
- Whenever we increment `front` or `rear`, we do so modulo the length of the array.

```
front = (front + 1) % items.length;
rear = (rear + 1) % items.length;
```

- Example:



`items.length = 8, rear = 7`

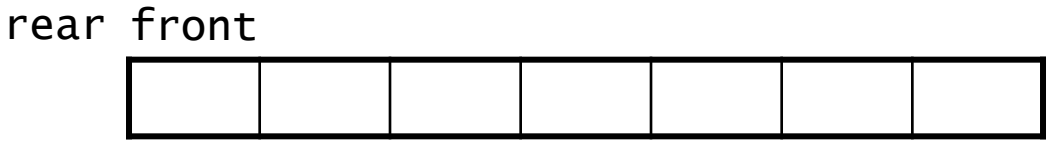
before inserting the next item: `rear = (7 + 1) % 8 = 0`

which wraps `rear` around to the start of the array

Testing if an `ArrayQueue` is Empty:
the problem with using **`rear`** *and* **`front`** *indices*

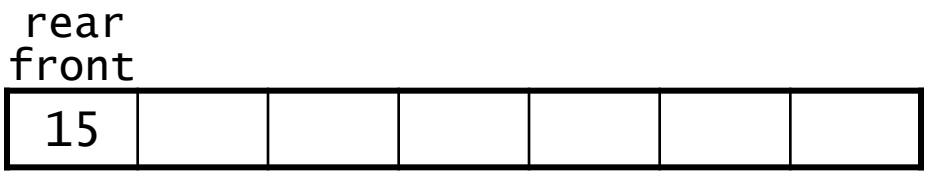
- Initial configuration:

```
rear = -1
front = 0
```

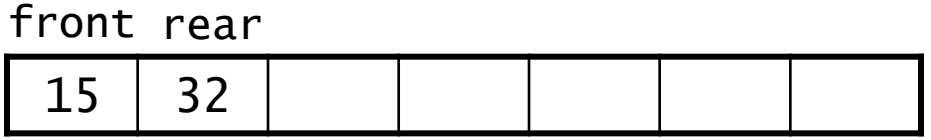


- We increment `rear` on every insertion, and we increment `front` on every removal.

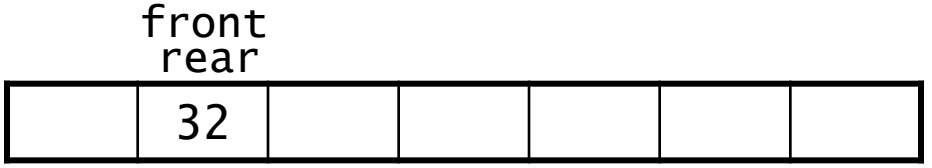
after one insertion:



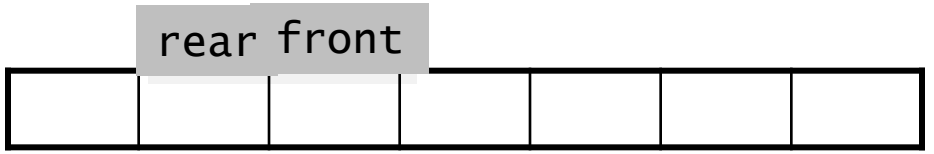
after two insertions:



after one removal:



after two removals:



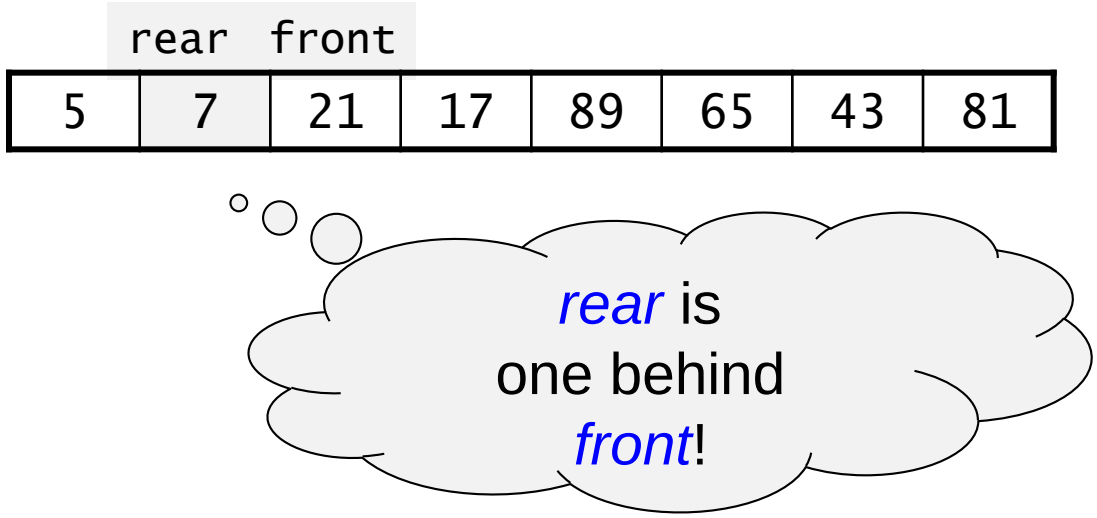
- The queue is empty when `rear` is one position "behind" `front`:

```
((rear + 1) % items.length) == front
```

Testing if an `ArrayQueue` is Full:
the problem with using **`rear`** *and* **`front`** *indices*

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

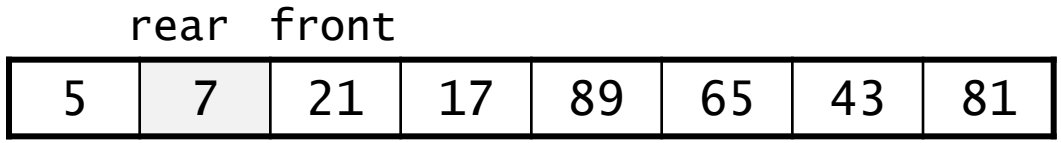
example: what if we added one more item to this queue?



Testing if an `ArrayQueue` is Full:
the problem with using **`rear`** *and* **`front`** *indices*

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

example: what if we added one more item to this queue?

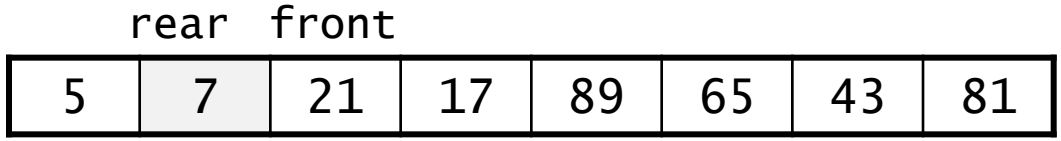


- This is why we maintain `numItems`!

Testing if an `ArrayQueue` is Full:
the problem with using **`rear`** *and* **`front`** *indices*

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

example: what if we added one more item to this queue?

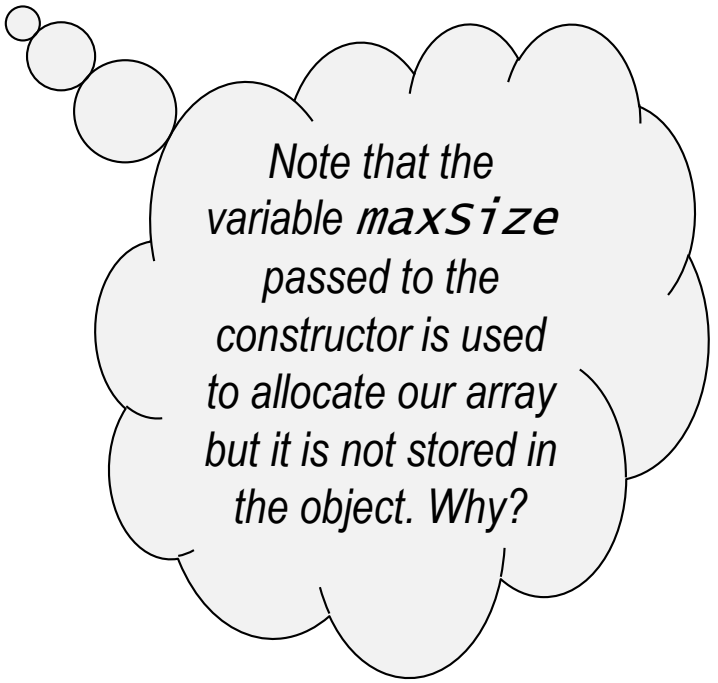


- This is why we maintain `numItems`!

```
public boolean isEmpty() {
    return (numItems == 0);
}

public boolean isFull() {
    return (numItems == items.length);
}
```

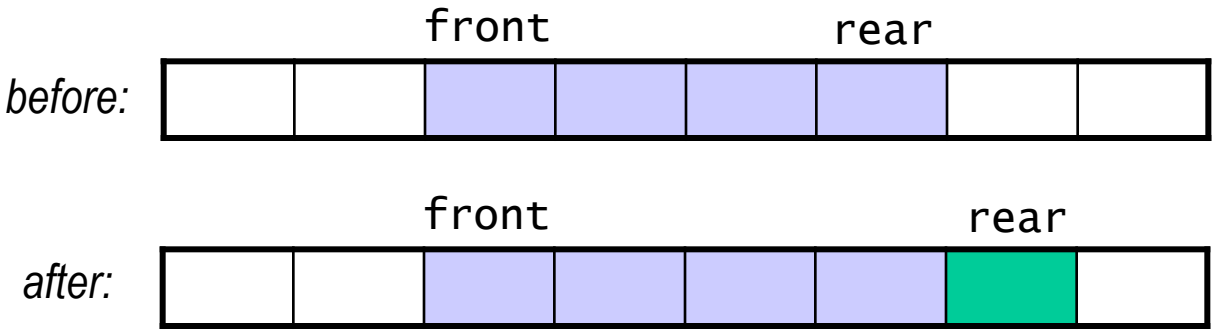
Constructor

```
public ArrayQueue(int maxSize) {
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```



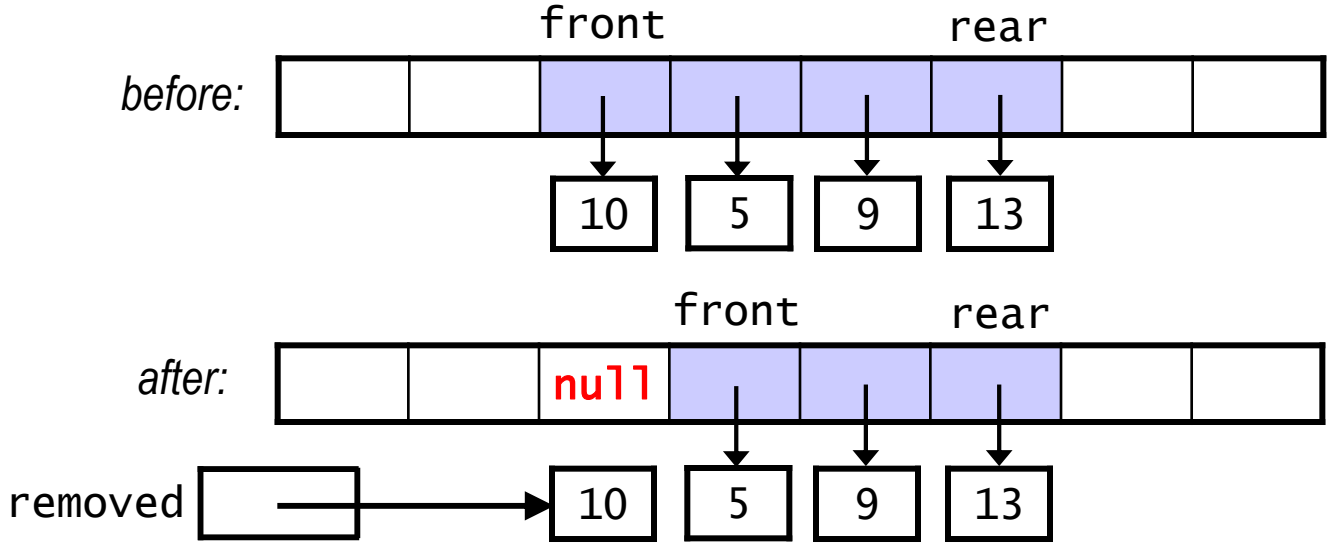
Inserting an Item in an `ArrayQueue`

- We increment `rear` before adding the item:



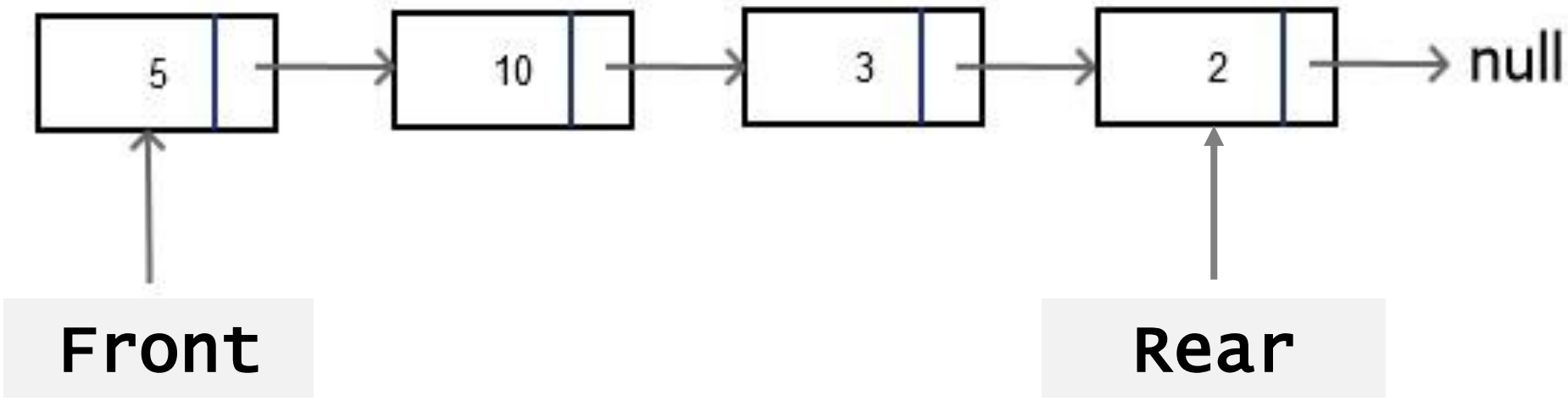

```
public boolean insert(T item) {
    if (isFull()) {
        return false;
    }
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}
```

ArrayQueue remove()




```
public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[front];
    items[front] = null;
    front = (front + 1) % items.length;
    numItems--;
    return removed;
}
```

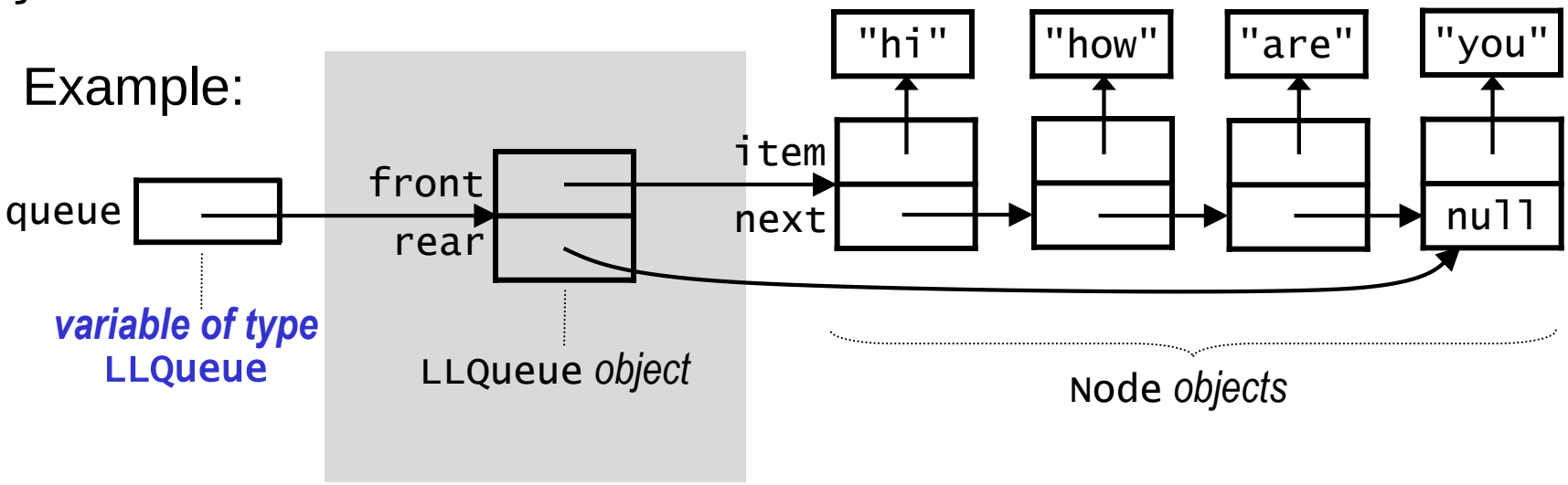
Linked List Implementation



Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;     // front of the queue
    private Node rear;      // rear of the queue
    …
}
```

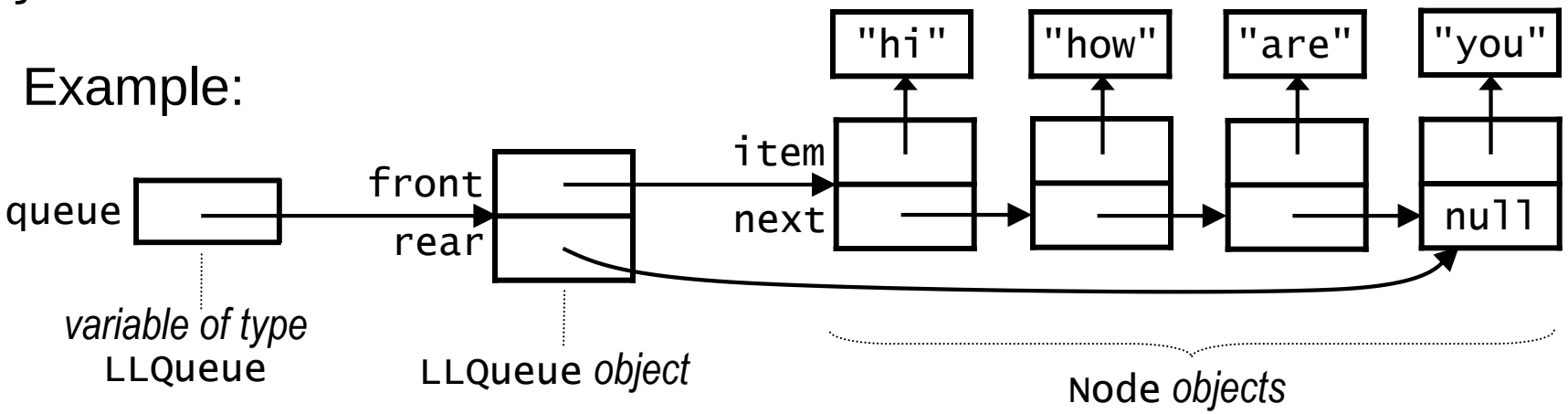
- Example:



Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;     // front of the queue
    private Node rear;      // rear of the queue
    …
}
```

- Example:



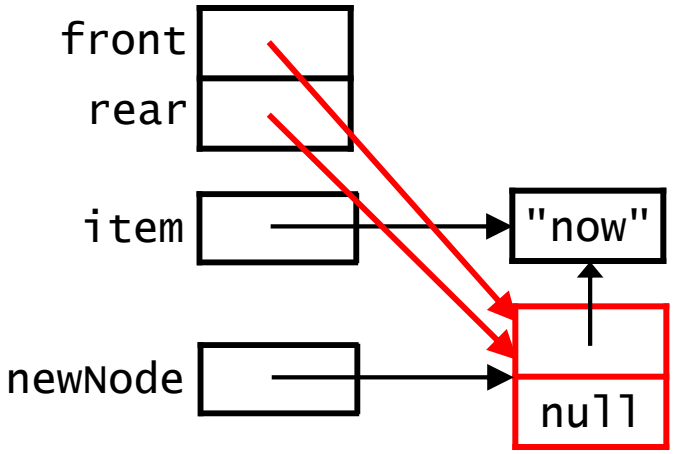
- Because a linked list can be easily modified on both ends, we don't need to take special measures to avoid shifting items, as we did in our array-based implementation.

Other Details of Our LLQueue Class

```
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node front;
    private Node rear;

    public LLQueue() {
        front = null;
        rear = null;
    }
    public boolean isEmpty() {
        return (front == null);
    }
    public boolean isFull() {
        return false;
    }
    …
}
```

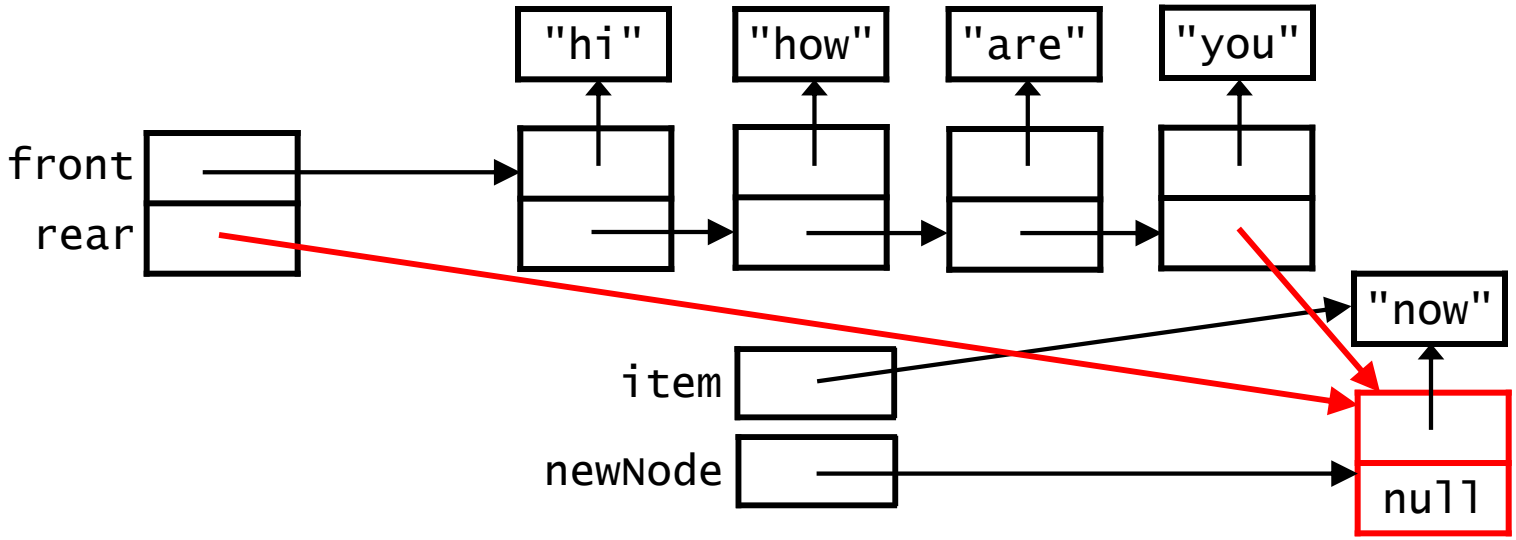
- Much simpler than the array-based queue!

Inserting an Item in an Empty LLQueue



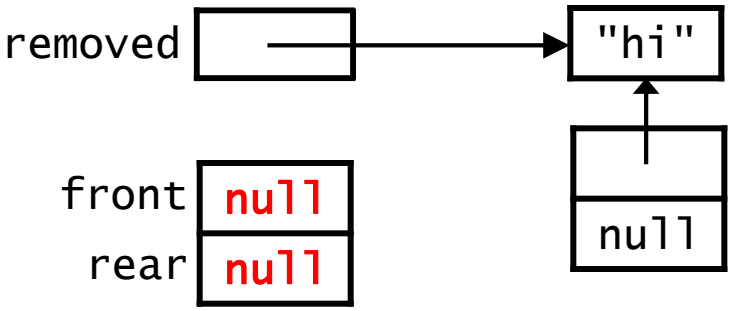

```
public boolean insert(T item) {
    Node newNode = new Node(item, null);

    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {



    }
    return true;
}
```

Inserting an Item in a Non-Empty LLQueue



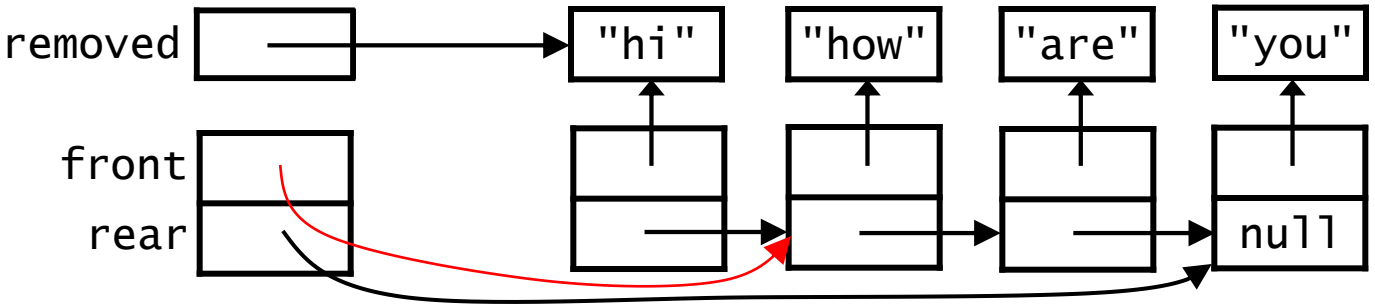

```
public boolean insert(T item) {
    Node newNode = new Node(item, null);

    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        rear.next = newNode;
        rear = newNode;
    }
    return true;
}
```

Removing from an LLQueue with One Item




```
public T remove() {
    if (isEmpty()) {
        return null;
    }

    T removed = front.item;
    if (front == rear) {      // removing the only item
        front = null;
        rear = null;
    } else {

    }

    return removed;
}
```

Removing from an LLQueue with Two or More Items




```
public T remove() {
    if (isEmpty()) {
        return null;
    }

    T removed = front.item;
    if (front == rear) {      // removing the only item
        front = null;
        rear = null;
    } else {
        front = front.next;
    }

    return removed;
}
```

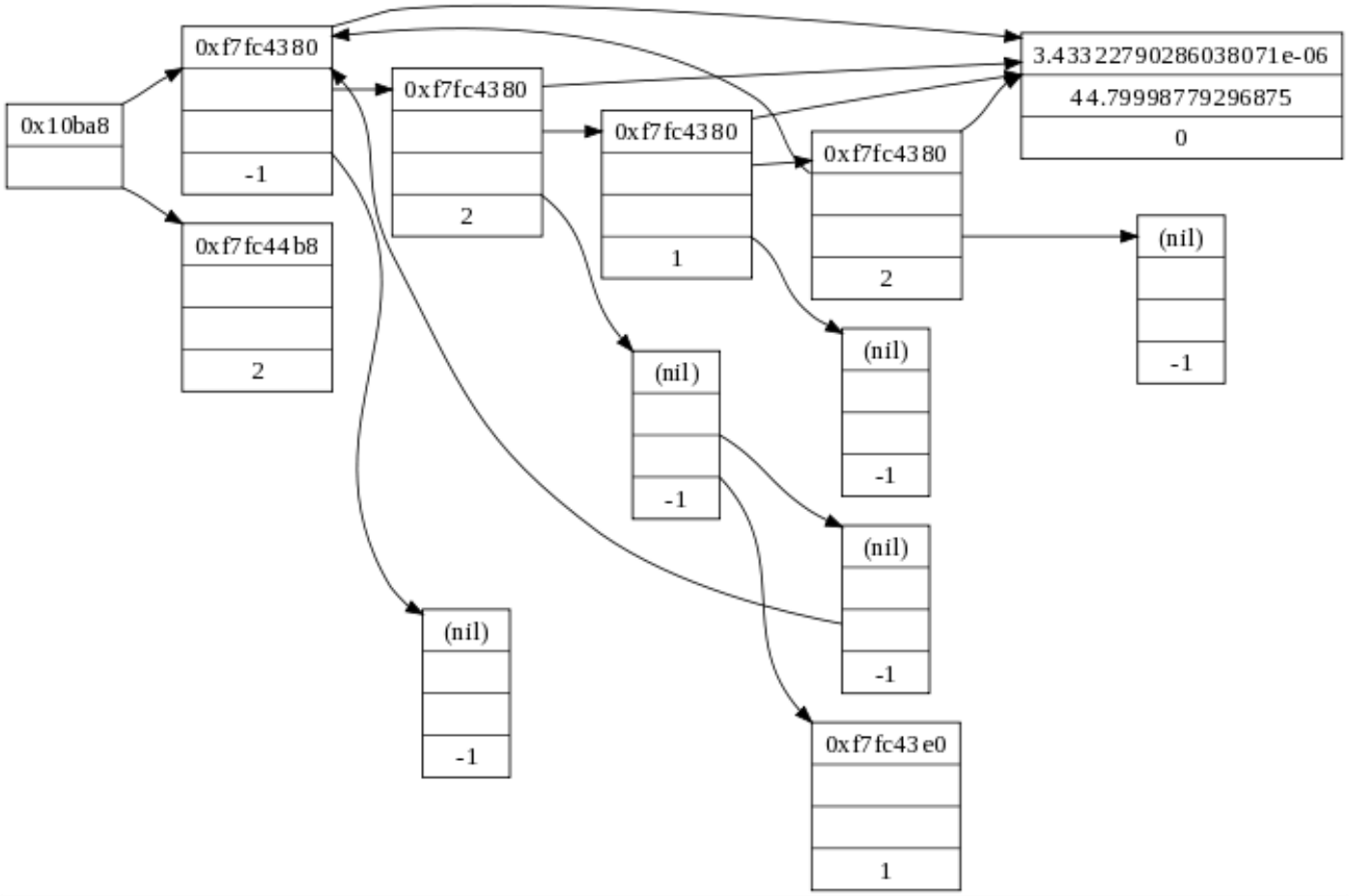
Efficiency of the Queue Implementations

	`ArrayQueue`	`LLQueue`
`insert()`	*O*(1)	*O*(1)
`remove()`	*O*(1)	*O*(1)
`peek()`	*O*(1)	*O*(1)
space efficiency	*O*(m) where m is the *anticipated* maximum number of items	*O*(n) where n is the number of items currently in the queue

Applications of Queues

- first-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- simulations of banks, supermarkets, airports, etc.
- level-order traversal of a binary tree (more on this later)

There's an app for that!



There's a data structure for that!

Lists, Stacks, and Queues in Java's Class Library

- Lists:
  - interface: `java.util.List<T>`
    - slightly different methods, some extra ones
  - array-based implementations: `java.util.ArrayList<T>`
    `java.util.Vector<T>`
    - the array is expanded as needed
    - `Vector` has extra non-`List` methods
  - linked-list implementation: `java.util.LinkedList<T>`
    - `addLast()` provides *O*(1) insertion at the end of the list
- Stacks: `java.util.Stack<T>`
  - extends `Vector` with methods that treat a vector like a stack
  - problem: other `Vector` methods can access items below the top
- Queues:
  - interface: `java.util.Queue<T>`
  - implementation: `java.util.LinkedList<T>`.