CS 2100: Data Structures & Algorithms 1

Introduction to Queues

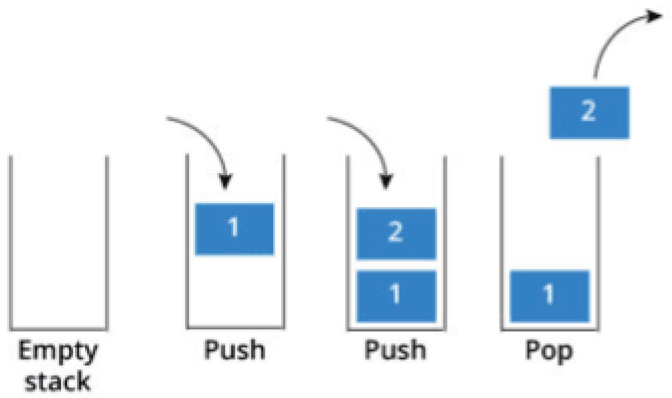
Dr. Nada Basit // basit@virginia.edu

Spring 2022

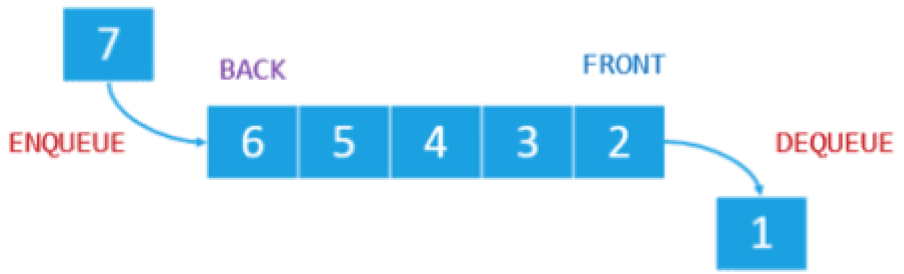
Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Stacks and Queues



Stack



Queue

Abstraction: Stacks and Queues

- We can think of a **stack** or a **queue** as an abstraction
  - We can implement them in different ways
  - They have operations that manipulate the data (in specific ways)

Queue

First In – First out (FIFO)

Remember: work is done at BOTH ends of the queue:

Adding to the **tail**; Removing from the **head**.

Queues

- Also a list, but **inserts** happen at one end (e.g. “back”) and **removals** happen at the other end (e.g. “front”)
- First In-First Out (**FIFO**)
  - Fields:
    - `front` – reference to the **front** of the stack (list)
    - `back` – reference to the **back** of the stack (list)
  - Operations:
    - `add/enqueue` – **insert** at one end (e.g. **back/tail**) of the queue (item is passed in)
    - `remove/dequeue` – **delete** at the other end (e.g. **front/head**) of the queue
  - Work is done at **both ends**, adding to the **back/tail** and removing from the **front/head**
- Java Collections provides the `Queue<T>` interface (implemented by `LinkedList<T>`)

Queue Implementations

- **Linked list** and **array implementations** are **constant time for all operations**
  - Disclaimer about a full vector queue:
    - When the internal array is full, you have to resize which isn't constant time (and contradicts the statement above)

- Array or vector
  - **theArray**
  - **front** position/index
  - **back** position/index
  - current **size**
- LinkedList

Application of Queue

- **Scheduling / Lines in general**
  - Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.

- **Asynchronous data transfer / File serving**
  - When data is transferred asynchronously between two processes. Queue is used for synchronization. Examples : IO Buffers, pipes, file IO, etc.

Application of Queue

- **Print spooling**
  - Documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.

- **Handling of interrupts in real-time systems**
  - The interrupts are handled in the same order as they arrive, First come first served.

Application of Queue

- **Call Center phone queues**
  - In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

- **Breadth First search**

- … and many more!

Queue: Array Implementation

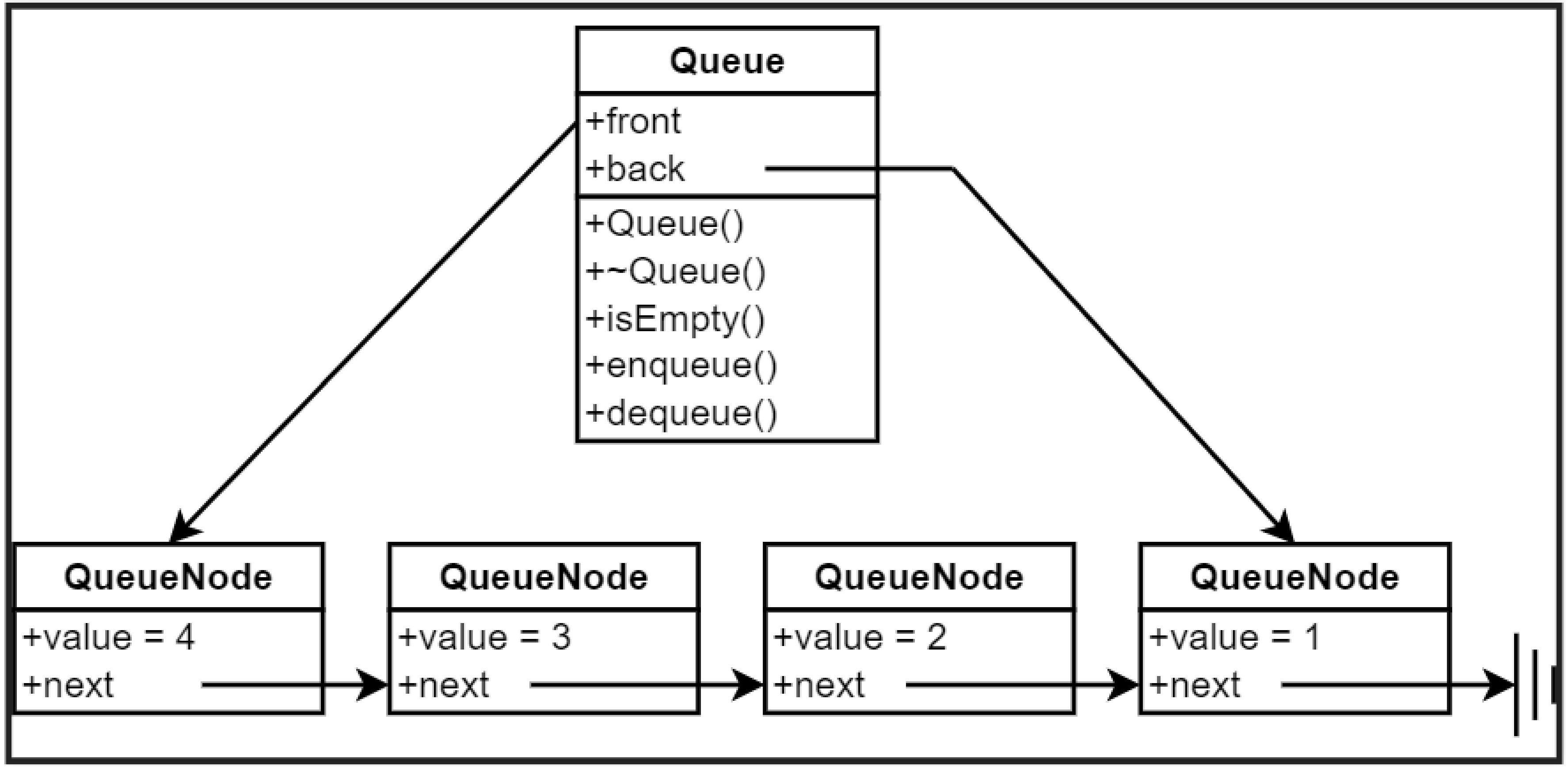
- Operations [tail/back is pointing at actual last element]
  - **enqueue** *[add at tail/back; element to be added is passed into the method]*
    - **check** if there is enough room, if so…
    - **increment** current size,
    - **increment tail/back**
    - set `theArray[tail] = element`
  - **dequeue** *[remove at head]*
    - set return value to `theArray[head]`
    - **decrement** current size,
    - **increment head/front**

Queue: LinkedList Implementation

- Also used "head" and "tail" instead of "front" and "back"

```
public class Queue< T >{

    // pointers to front and back of list
    private QueueNode< T > front, back;

    // place item on back of list
    public void enqueue(T value);

    // remove item from front of list
    T dequeue();

    // other supporting methods...
}
```

Queue: LinkedList Implementation Diagram



Queue: LinkedList Implementation

- In the Queue<T> class, we use a **LinkedList** as the underlying implementation for the Queue.
- **Constructor** (and class attribute):

```
// field: LinkedList called list representing the queue
private LinkedList<T> list;

/**
 * Constructor: Initialize the inner list
 */
public Queue(){
    list = new LinkedList<T>();
}
```

Queue: LinkedList Implementation

- **enqueue(T data)** method:

```
// Simply add the data to the tail of the linked list
public void enqueue(T data) {
    // Body ... Simply call the appropriate method in LinkedList class
}
```

- **dequeue()** method (with T return type):

```
// Simply remove data from the head of the list
public T dequeue(){
    // Body ... Simply call the appropriate method in LinkedList class
}
```

LinkedList Class: insertAtHead() method

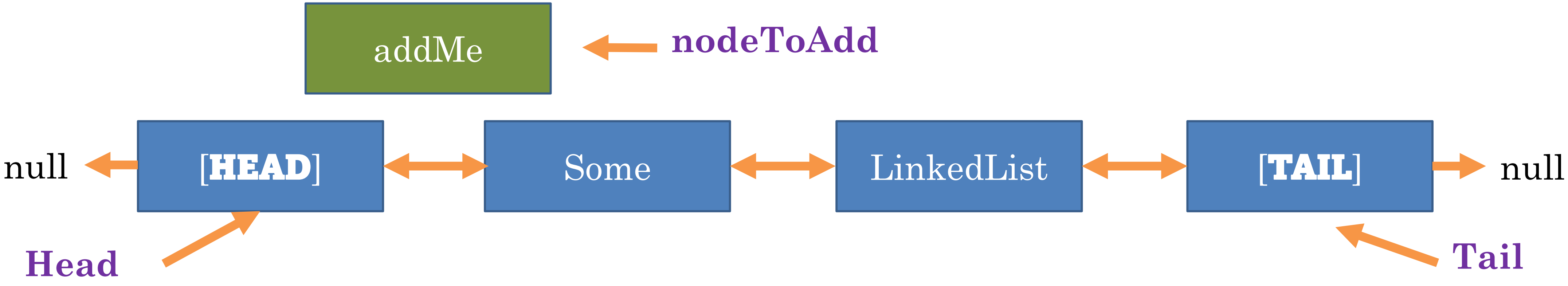
- How might we accomplish this?

```
public void insertAtHead(T data)
```

- // The method takes in **data** to be included in a node in the Queue
  - // Create a brand new node of type ListNode<T> and include the data

```
ListNode<T> nodeToAdd = new ListNode<T>(data);
```

// use ListNode non-default constructor

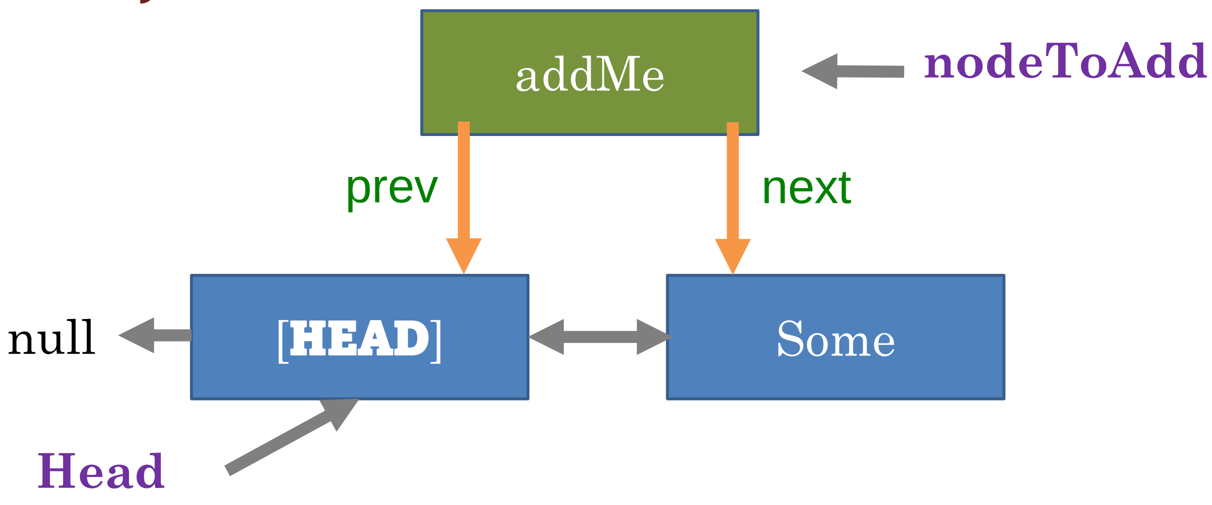


LinkedList Class: insertAtHead() method (cont'd)

```
public void insertAtHead(T data)
```

- // Let's set up the **new node's** next and previous pointers
  - // nodeToAdd's **next** pointer should point to what the **head** node's **next** pointer was pointing to (node with data="Some")
  - // since nodeToAdd is to become the first actual node, it's **prev** pointer should point at the dummy **head** node

```
nodeToAdd.next = head.next;
nodeToAdd.prev = head;
```

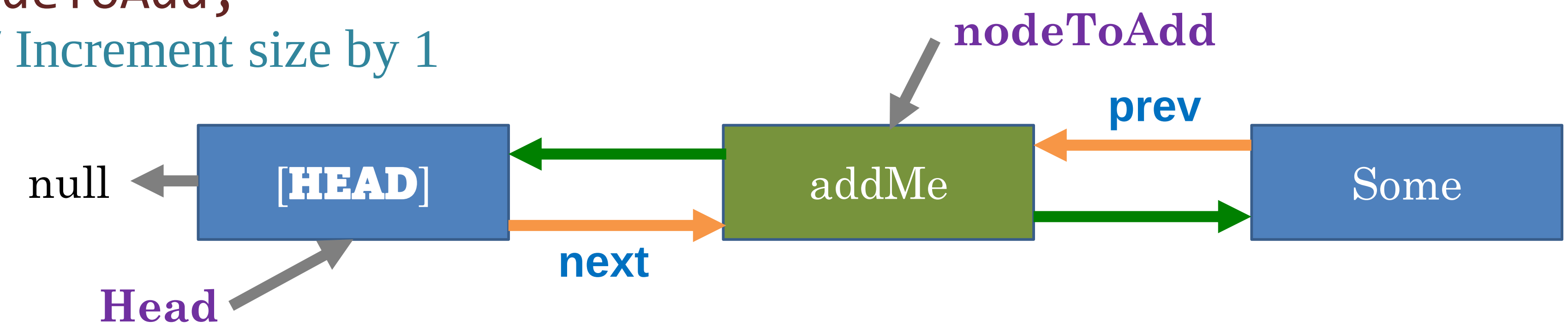


LinkedList Class: insertAtHead() method (cont'd)

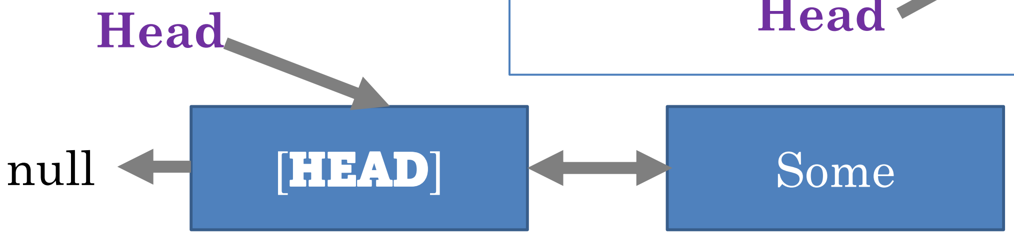
```
public void insertAtHead(T data)
```

- // Let's set up the **dummy head node's** next and previous pointers
  - // The prev pointer of the **node** the dummy head was pointing to (data="Some") should point at **nodeToAdd**
  - // The next pointer of the **dummy head node** should now point to the new **nodeToAdd**

```
head.next.prev = nodeToAdd;  // head.next is the "Some" node
head.next = nodeToAdd;
this.size++;  // Increment size by 1
```



Remember how it was:



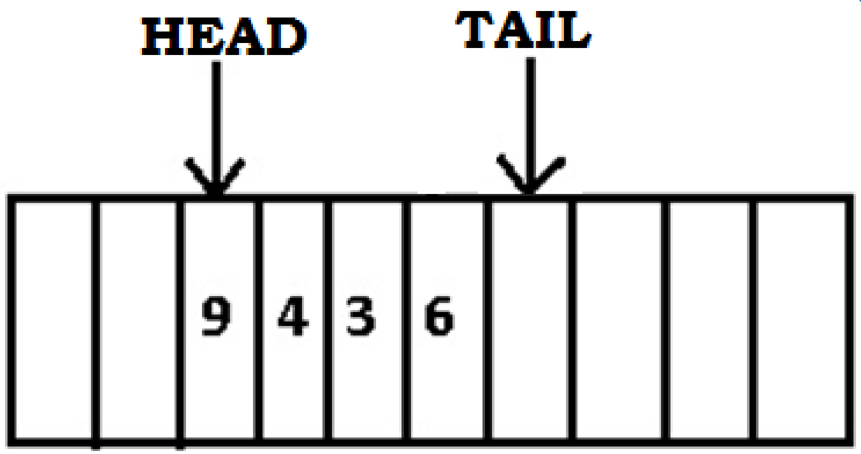
[Queue] What would an array-based implementation look like?

We are adding at one end, and removing from the other!

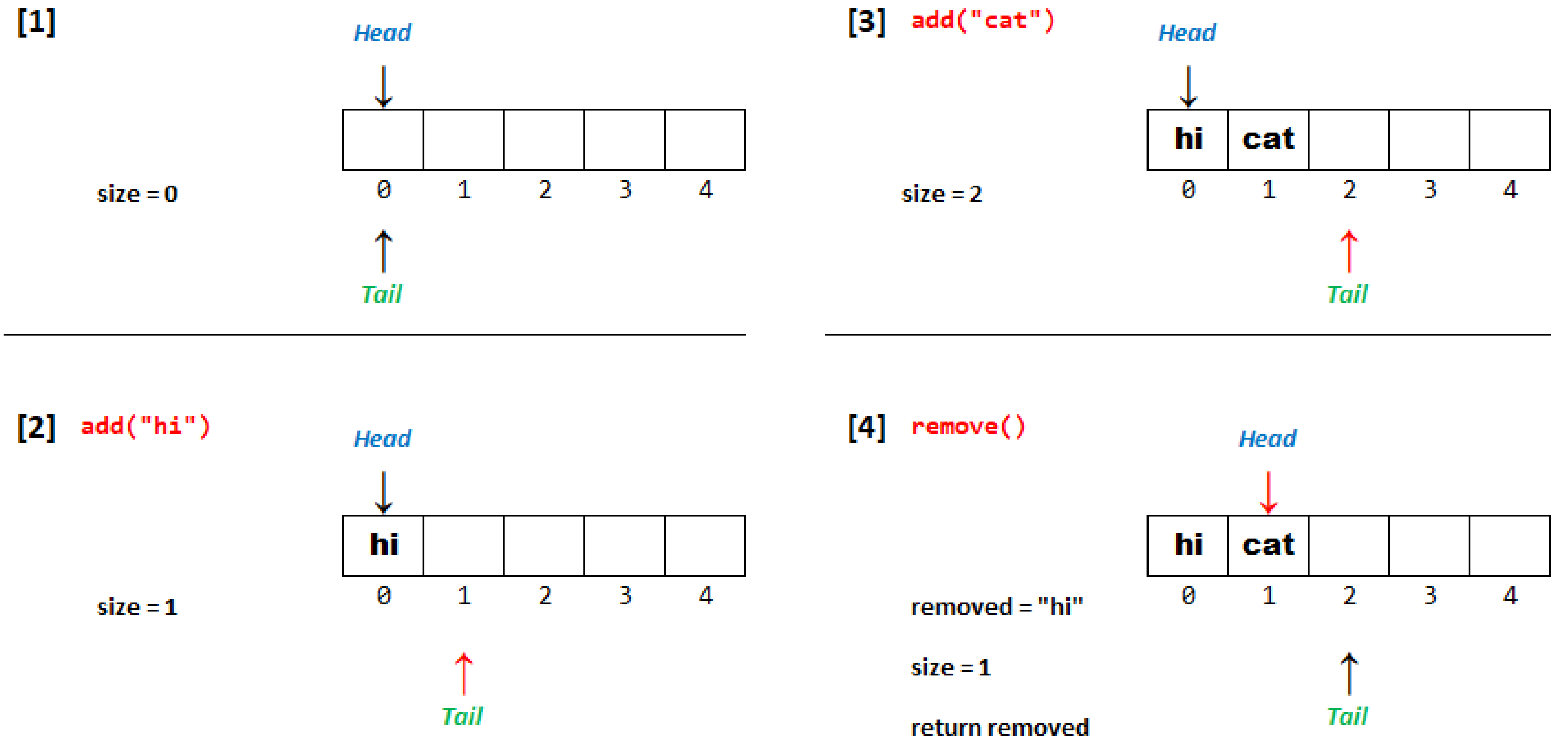
Alternate example where ‘tail’ is **floating** next to last item

Queue – add( ) and remove( ) methods

- Think about how you would keep track of where to **insert** into the array and where you would **remove** from the array [Hint: pointers]
- Think about how you would handle the fact that when you remove from an array, you have an empty slot
  [Hint: either shift all the elements inside the array, or just keep track, via int pointers, of the location of the head and tail]
- *Are there other ways you can think of to do this?*



Example of a Queue (FIFO)

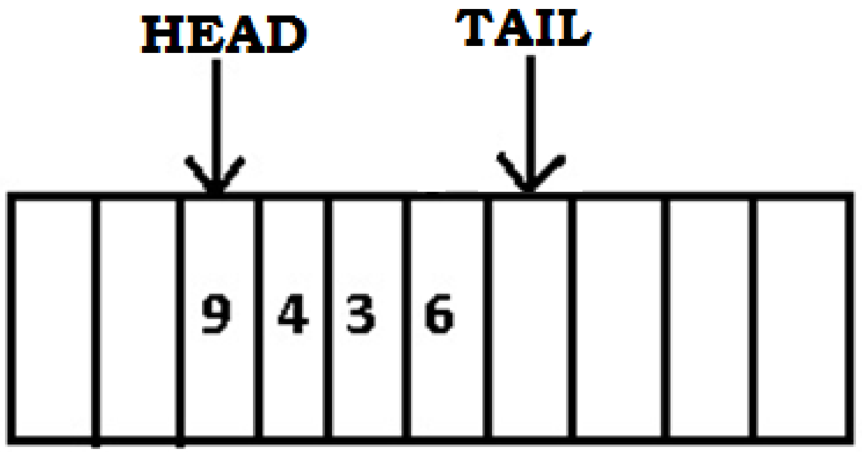


Although "hi" isn't removed, what is valid is between the "Head" and "Tail" pointers

Queue (where 'tail' is **floating** next to last item)

```
final int INITIAL_SIZE = 4; // a constant
String[] elements;
int currentSize, head, tail; // head and tail are position pointers

public Queue() { // constructor
    this.elements = new String[this.INITIAL_SIZE];
    this.currentSize = this.head = this.tail = 0;
                                        // all initialized to 0
}
```

Queue – Implementing add() and remove() methods



- **`Add() with parameter`** *[ADD AT "TAIL" (END) OF QUEUE]*
  - Increment size counter
  - Add at the tail: `myQueueArr[tail] = v;` *// v is the value*
  - Adjust tail to be: `tail = (tail + 1) % myQueueArr.length;` *// can loop*
- **`Remove()`** *[REMOVE FROM "HEAD" (FRONT) OF QUEUE]*
  - Check if queue is empty, if so return null
  - Remove at the head: `int removed = myQueueArr[head];` *// to return*
  - Adjust head to be: `head = (head + 1) % myQueueArr.length;` *// can loop*
  - Decrement size counter
  - return `removed`

By using modulus (%), the **head** chases the **tail** around ends of the array