

# CS350: Data Structures

## Queues

---

James Moscola

Department of Engineering & Computer Science

York College of Pennsylvania



# Queues

---

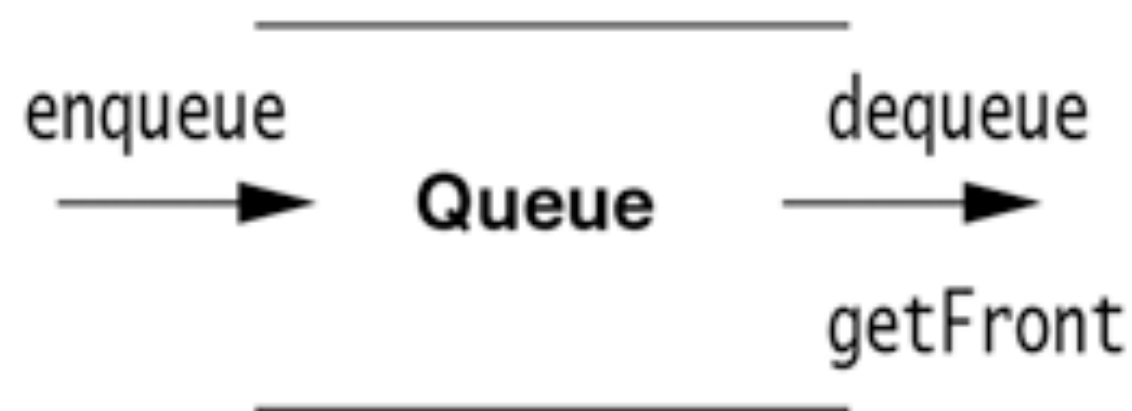
- **Queues are another very common data structure that can be used for a variety of data storage purposes**
- **Similar to a line of people waiting for a ride at an amusement park**
  - People enter the line/queue at the rear
  - People wait behind others that entered the line/queue before them
  - People exit from the front of the line/queue to get on the ride
  - People in the middle of the line/queue cannot get out without first advancing to the front of the line
  - There is no cutting
- **May also be referred to as a **FIFO** (First-In-First-Out)**

# Queue Operations

---

- **Stacks have two main operations**

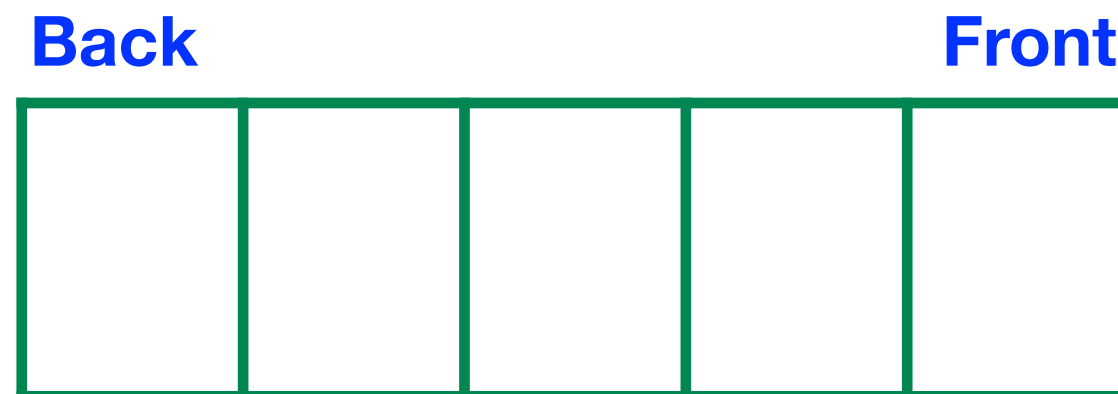
- **Enqueue** - inserts an element into the back of the queue
- **Dequeue** - removes a single element from the front (i.e. the head) of the queue



# Queue Operations

---

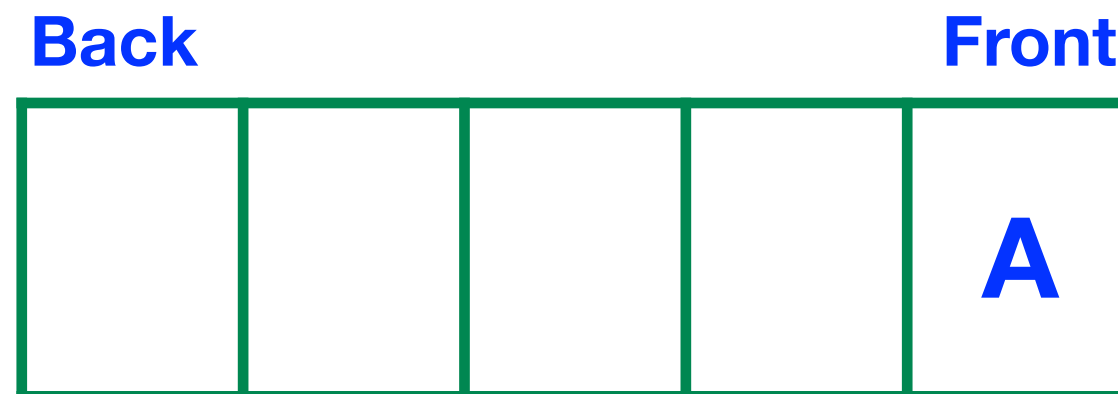
## Start with Empty Queue



# Queue Operations

---

**Enqueue Value: A**



# Queue Operations

---

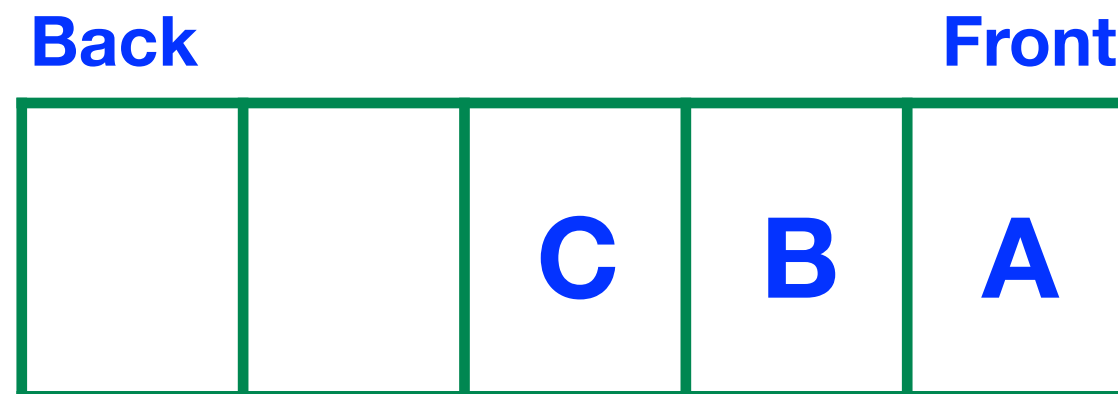
**Enqueue Value: B**



# Queue Operations

---

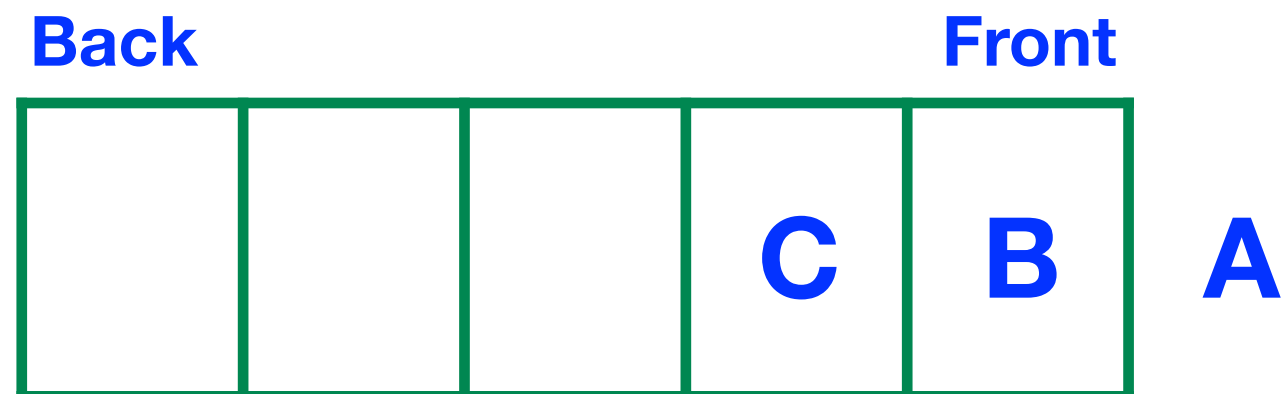
**Enqueue Value: C**



# Queue Operations

---

**Dequeue the Front of the Queue:**





# Queue Interface (Java)

---

```
public interface Queue<AnyType> {  
    public void      enqueue( AnyType x );  
  
    public AnyType dequeue( );  
  
    public AnyType getHead( );  
  
    public boolean isEmpty( );  
  
    public void      makeEmpty( );  
}
```

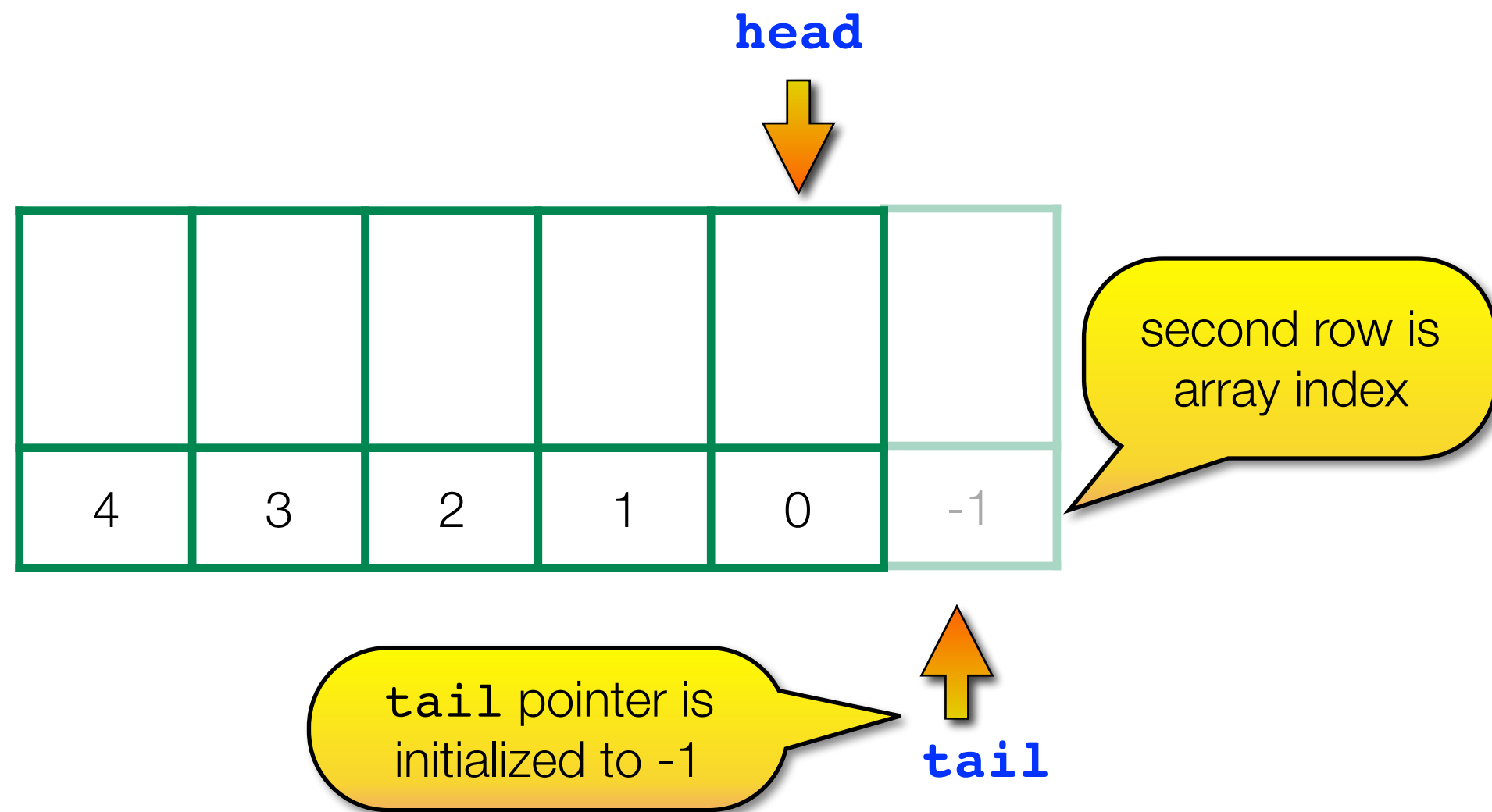
# Queue Implementations

---

- **Queues can be implemented in multiple ways**
- **Two popular methods for implementing queues include**
  - (1) Arrays
  - (2) Linked Lists
- **Both of these implementation approaches allow for constant time operations -  $O(1)$**

# Queue Implementation Using an Array

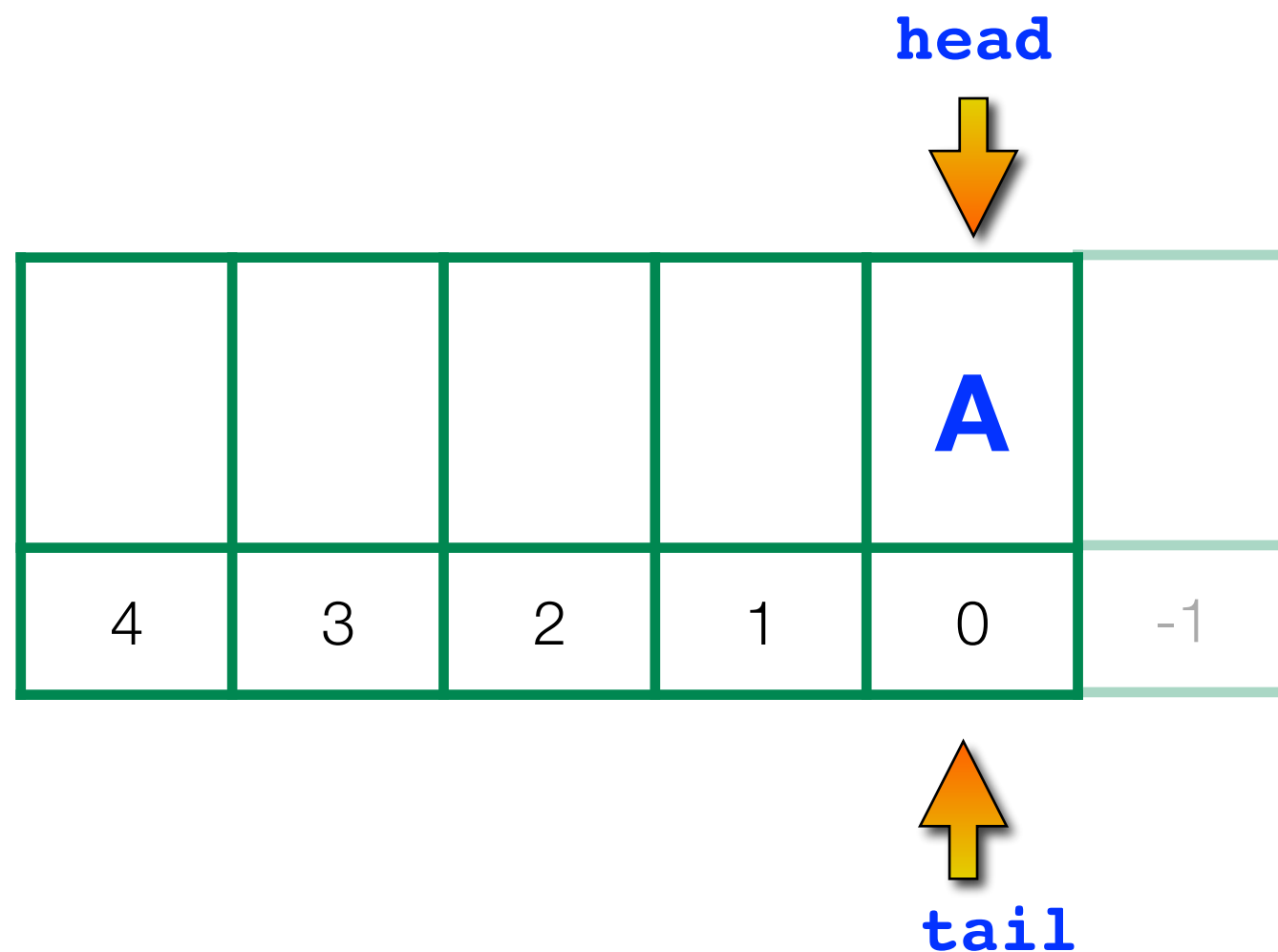
**Start with Empty Queue (i.e. an array)**



# Queue Implementation Using an Array

---

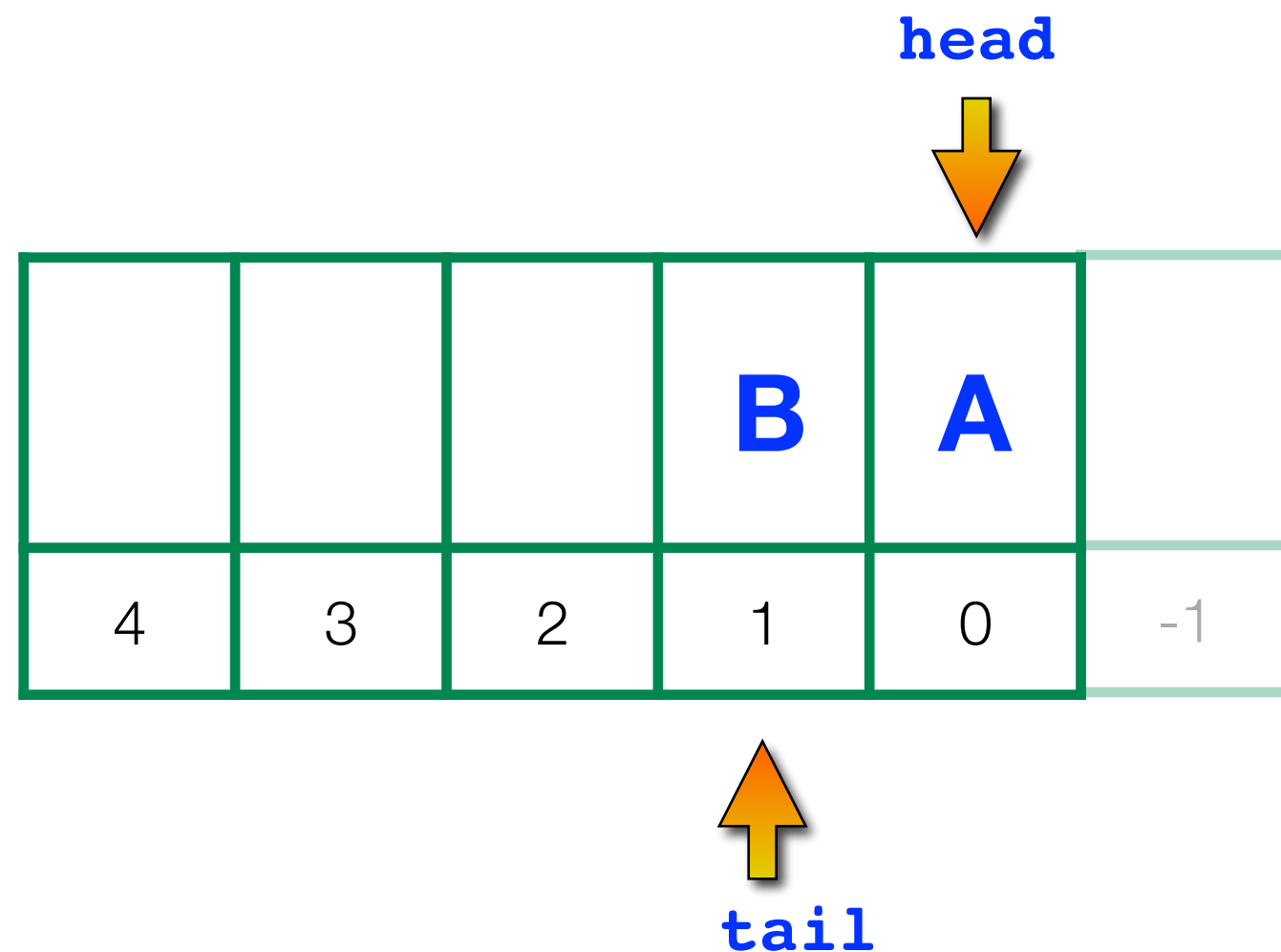
**Enqueue Value: A**



# Queue Implementation Using an Array

---

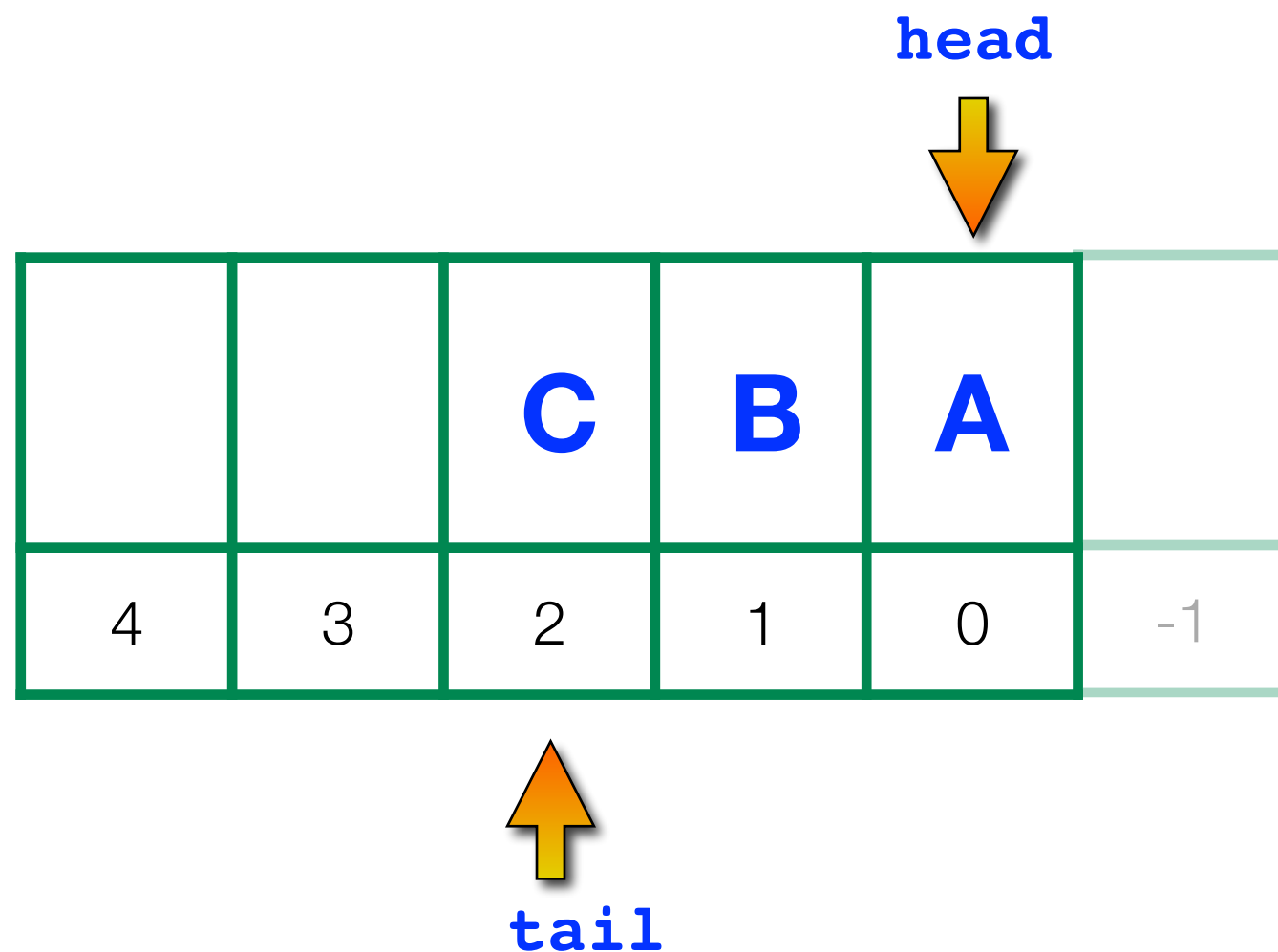
**Enqueue Value: B**



# Queue Implementation Using an Array

---

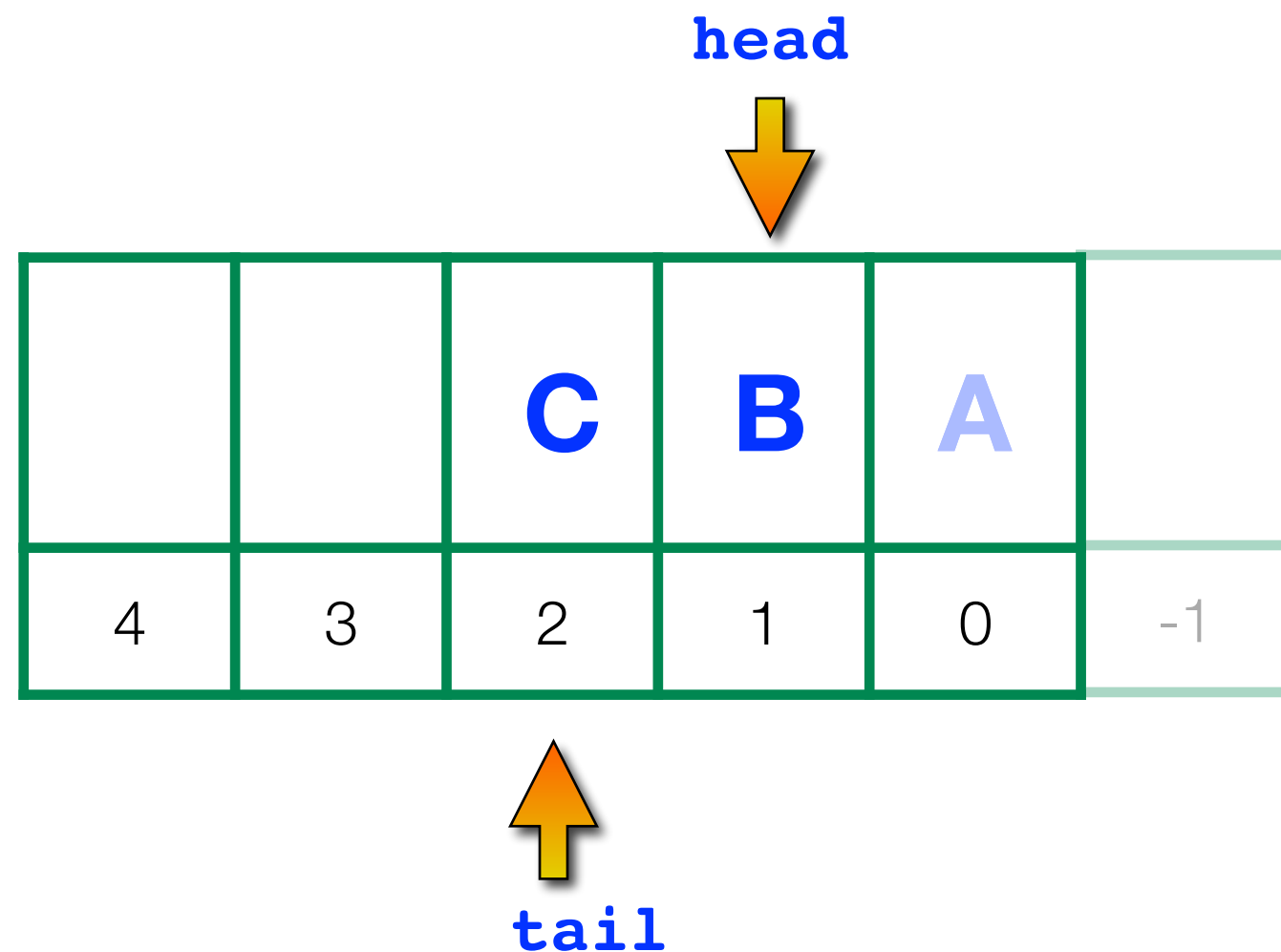
**Enqueue Value: C**



# Queue Implementation Using an Array

---

## Dequeue the Head of the Queue:



# Queue Implementation Using an Array

---

- **Considerations when using an array implementation**

- Enqueue, and Dequeue operations run in constant time ... in most cases
- What happens when your array is full and you want to Enqueue another element?
  - Array must be increased in size which takes time and more memory
  - Time to copy and create new array is  $O(N)$
  - Time to copy array is amortized over the lifetime of the array
  - May not be suitable for all types of systems (e.g. RTOS)



# Queue Implementation Using an Array (Java)

---

```
/**
 * Queue constructor
 */
public ArrayQueue( )
{
    theArray = (AnyType [ ]) new Object[ DEFAULT_CAPACITY ];
    makeEmpty( );
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Test if the queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return currentSize == 0;
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Make the queue logically empty.
 */
public void makeEmpty( )
{
    currentSize = 0;
    head = 0;
    tail = -1;
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Insert a new item into the queue.
 * @param x the item to insert.
 */
public void enqueue( AnyType x )
{
    if( currentSize == theArray.length )
        doubleQueue( );
    tail = increment( tail );
    theArray[ tail ] = x;
    currentSize++;
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Return and remove the least recently inserted item
 * from the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType dequeue( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ArrayQueue dequeue" );
    currentSize--;

    AnyType returnValue = theArray[ head ];
    head = increment( head );
    return returnValue;
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Internal method to increment with wraparound.
 * @param x any index in theArray's range.
 * @return x+1, or 0 if x is at the end of theArray.
 */
private int increment( int x )
{
    if( ++x == theArray.length )
        x = 0;
    return x;
}
```

# Queue Implementation Using an Array (Java)

---

```
/**
 * Internal method to expand theArray.
 */
private void doubleQueue( )
{
    AnyType [ ] newArray;

    newArray = (AnyType [ ]) new Object[ theArray.length * 2 ];

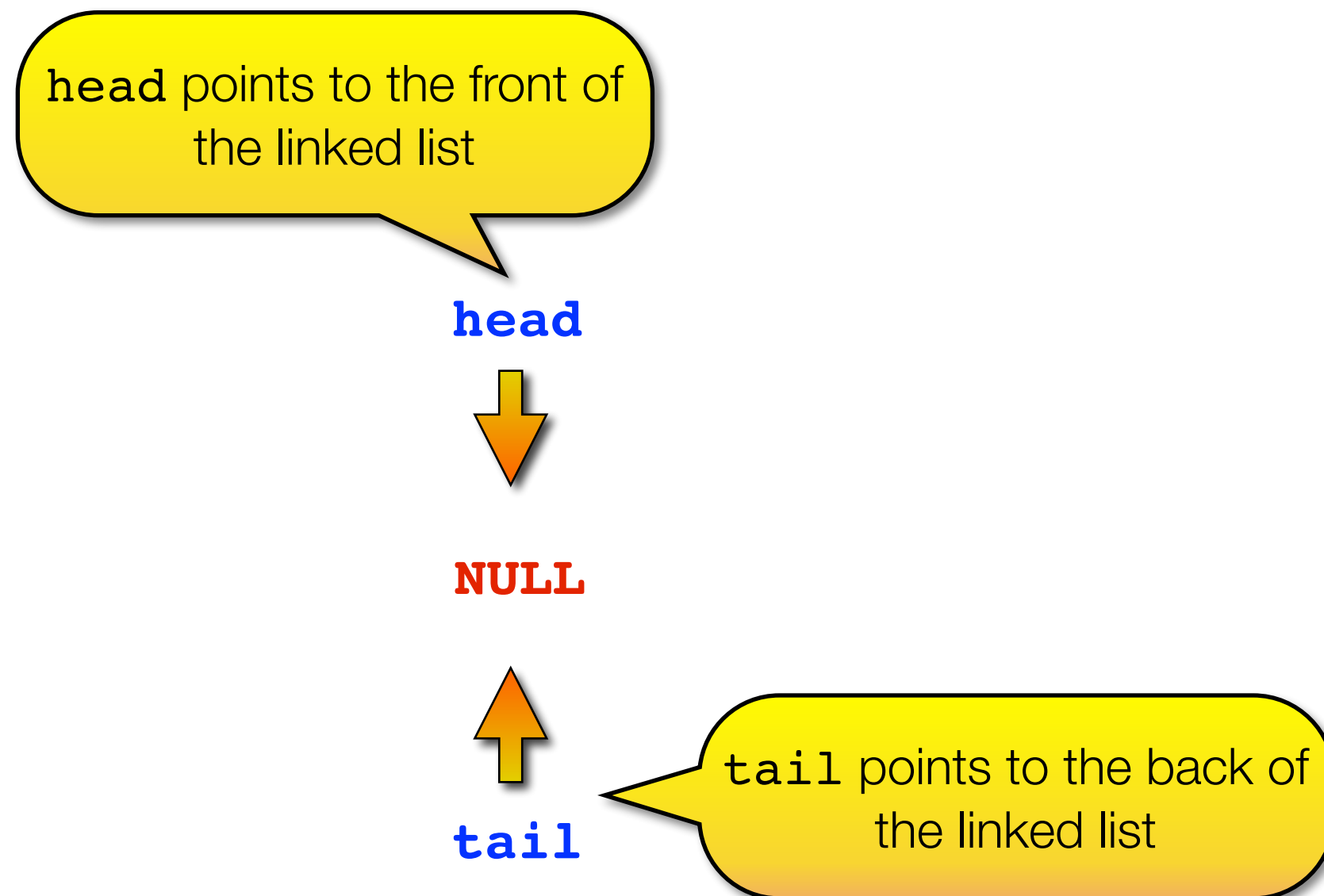
    // Copy elements that are logically in the queue
    for( int i = 0; i < currentSize; i++,
        head = increment( head ) )
        newArray[ i ] = theArray[ head ];

    theArray = newArray;
    head = 0;
    tail = currentSize - 1;
}
```

# Queue Implementation Using a LinkedList

---

## Start with Empty Queue (i.e. a null LinkedList)

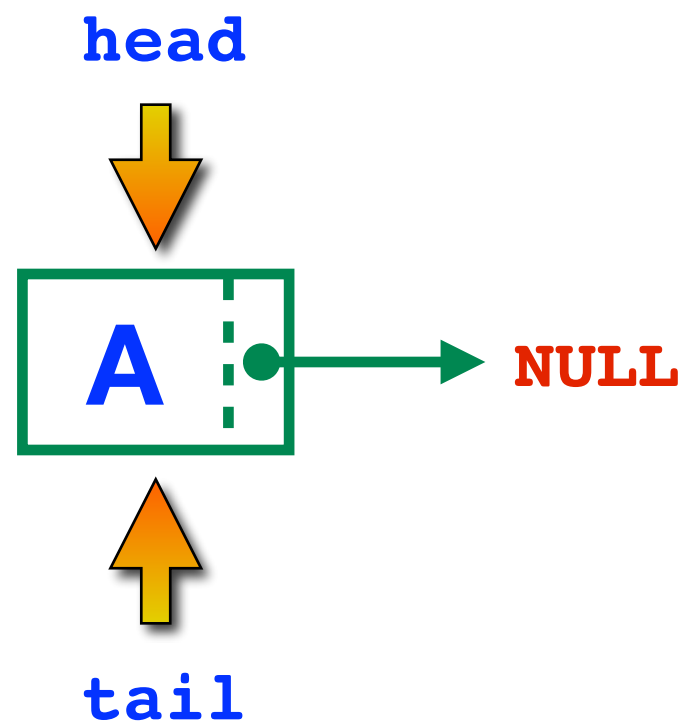




# Queue Implementation Using a LinkedList

---

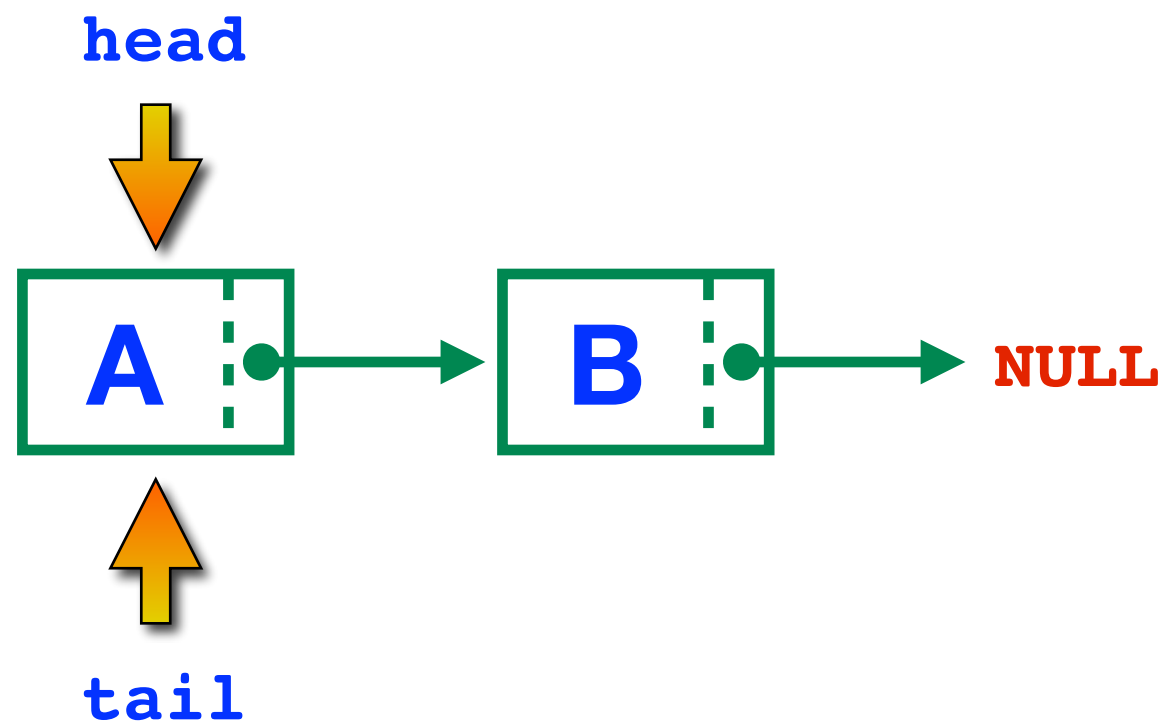
**Enqueue Value: A**



# Queue Implementation Using a LinkedList

---

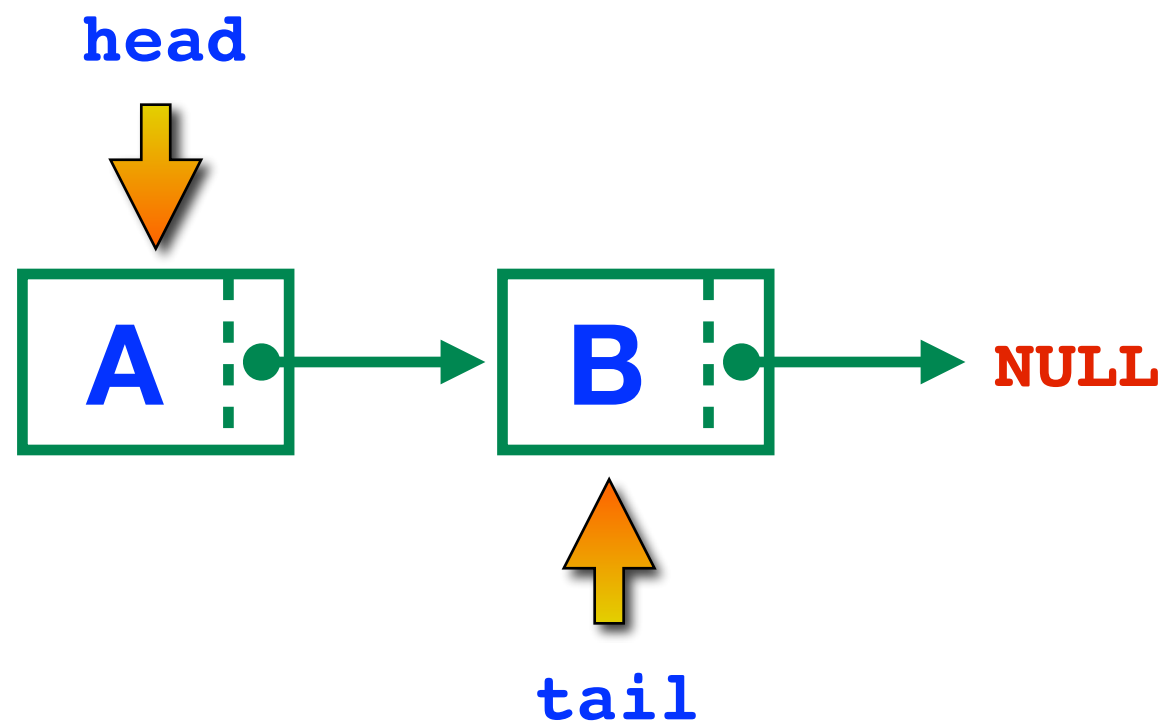
**Enqueue Value: B**



# Queue Implementation Using a LinkedList

---

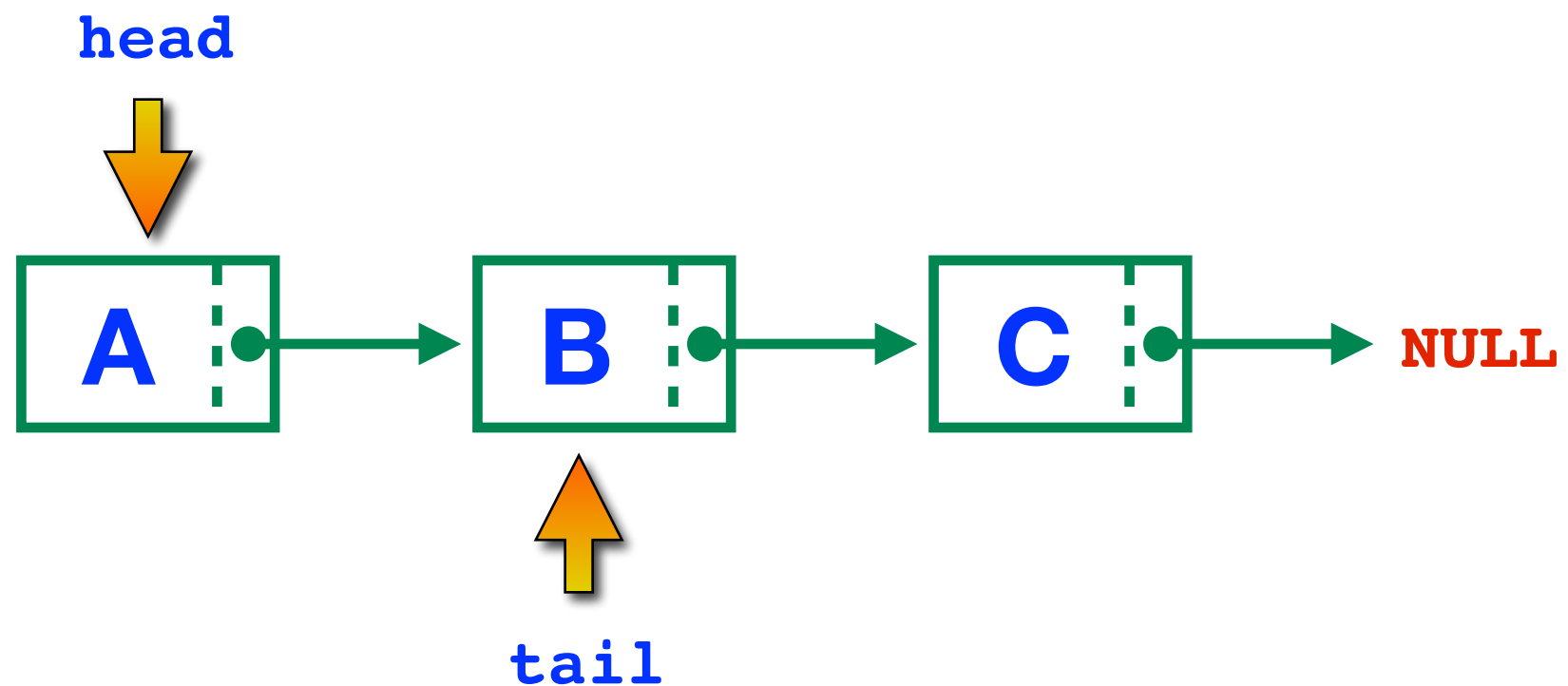
**Enqueue Value: B**



# Queue Implementation Using a LinkedList

---

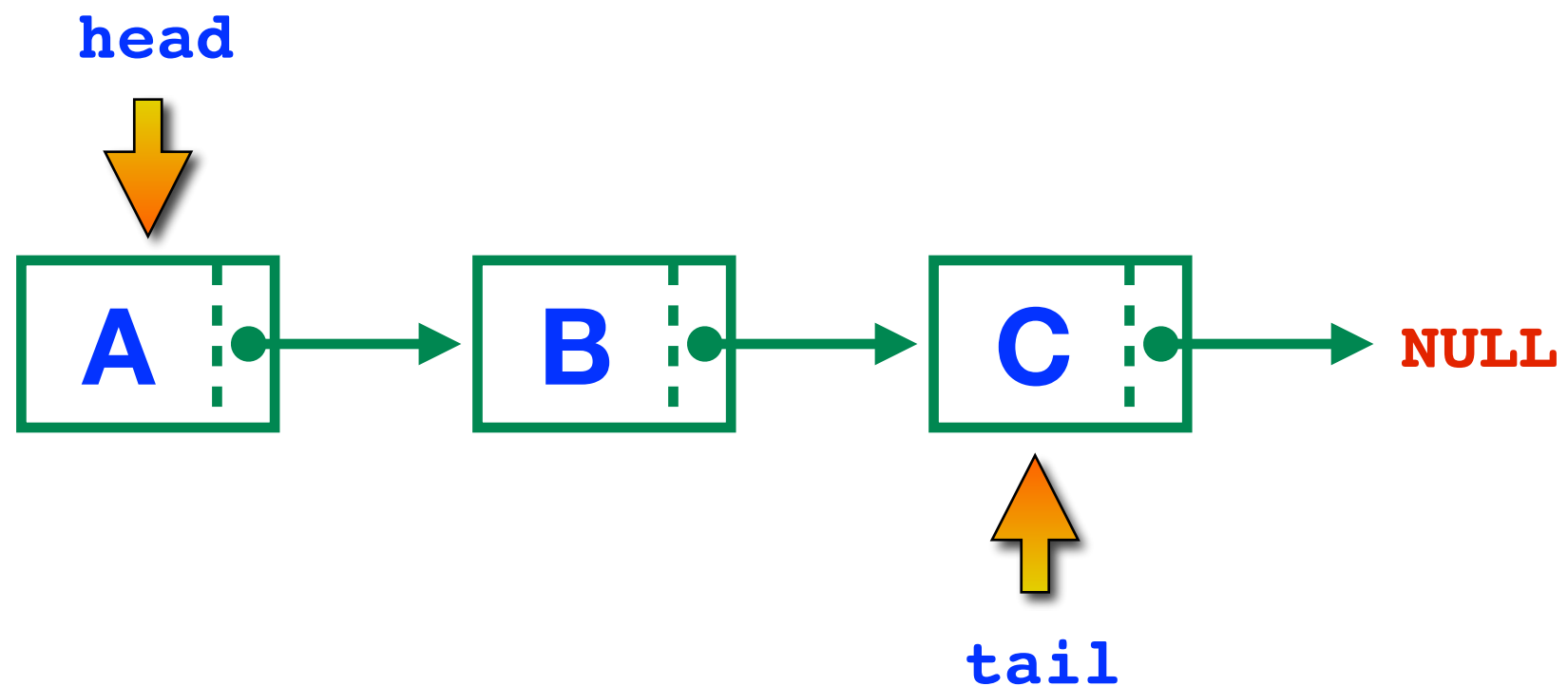
**Enqueue Value: C**



# Queue Implementation Using a LinkedList

---

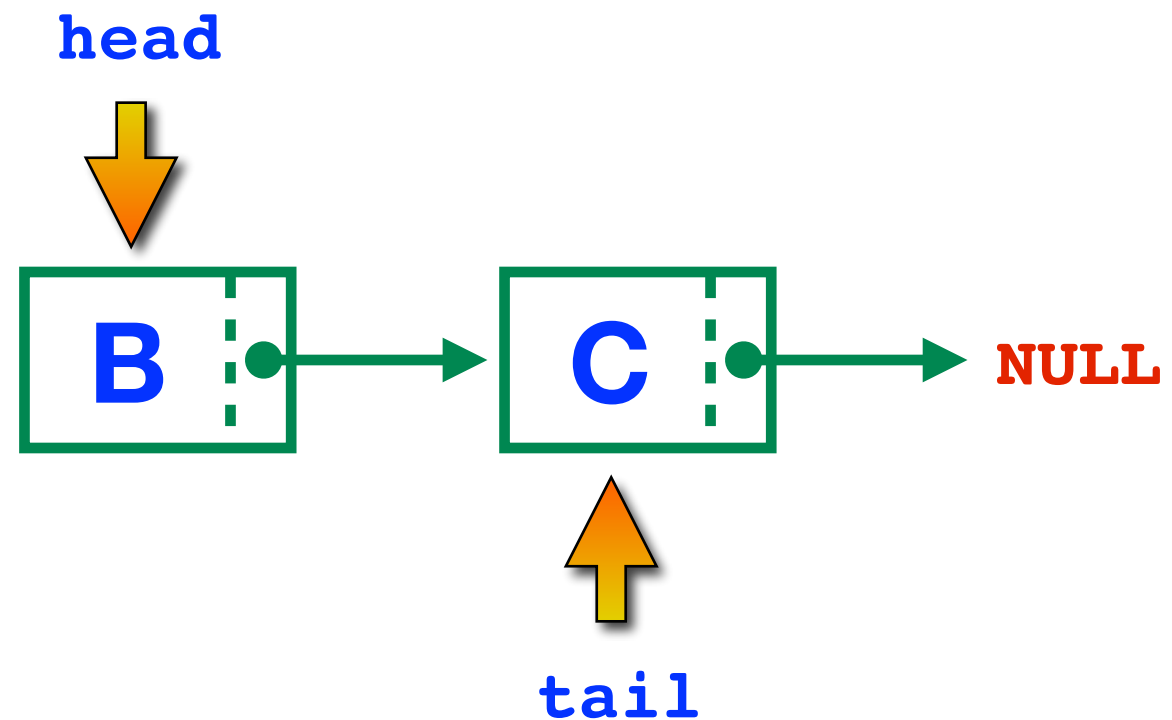
**Enqueue Value: C**



# Queue Implementation Using a LinkedList

---

## Dequeue the Front of the Queue:



# Queue Implementation Using a LinkedList

---

- **Considerations when using an queue implementation**

- Enqueue and Dequeue operations run in constant time ... still
- Each element inserted into the queue requires a pointer to the data and a second pointer to the next node in the LinkedList

# Queue Implementation Using a LinkedList (Java)

---

```
/**
 * Queue constructor
 */
public ListQueue( )
{
    head = tail = null;
}

private ListNode<AnyType> head;
private ListNode<AnyType> tail;
```



# Queue Implementation Using a LinkedList (Java)

---

```
/**
 * Test if the queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return head == null;
}
```

# Queue Implementation Using a LinkedList (Java)

---

```
/**
 * Make the queue logically empty.
 */
public void makeEmpty( )
{
    head = null;
    tail = null;
}
```

# Queue Implementation Using a LinkedList (Java)

---

```
/**
 * Insert a new item into the queue.
 * @param x the item to insert.
 */
public void enqueue( AnyType x )
{
    if( isEmpty( ) )    // Make queue of one element
        tail = head = new ListNode<AnyType>( x );
    else                // Regular case
        tail = tail.next = new ListNode<AnyType>( x );
}
```

# Queue Implementation Using a LinkedList (Java)

---

```
/**
 * Return and remove the least recently inserted item
 * from the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType dequeue( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue dequeue" );

    AnyType returnValue = head.element;
    head = head.next;
    return returnValue;
}
```