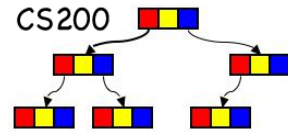


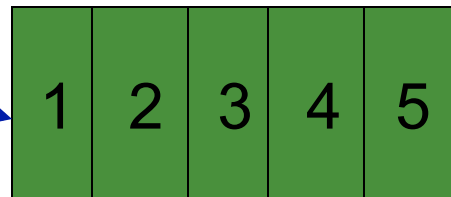
# CS200: Queues

- Prichard Ch. 8

# Queues

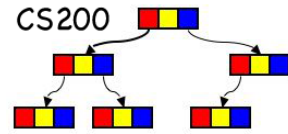


- First In First Out (FIFO) structure
- Imagine a checkout line
- So **removing** and **adding** are done from opposite ends of structure.



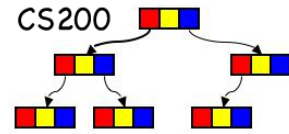
- add to tail (back), remove from head (front)
- Used in operating systems (e.g. print queue).

# Possible Queue Operations



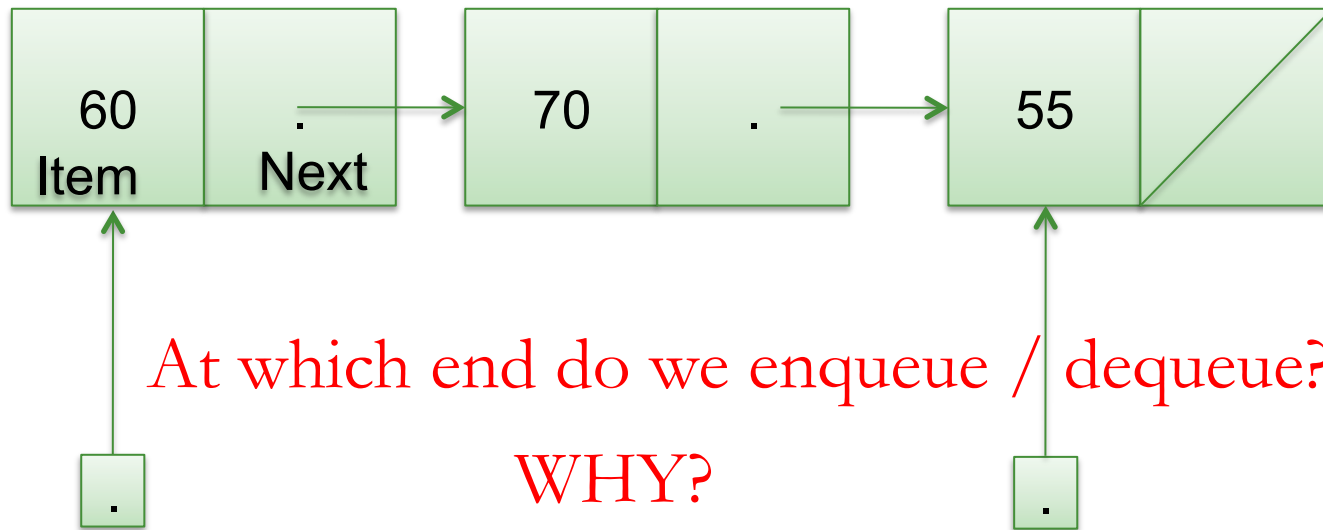
- **enqueue**(in newItem: QueueItemType)
  - Add new item at the back of a queue
- **dequeue**( ) ; QueueItemType
  - Retrieves and removes the item at the *front* of a queue
- **peek**( ) : queueItemType {query}
  - Retrieve item from the *front* of the queue. Retrieve the item that was added earliest.
- **isEmpty**( ) : boolean {query}
- **createQueue**( )

# Reference-Based Implementation 1



A linked list with two external references

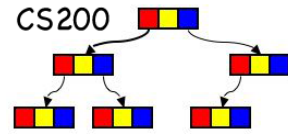
- A reference to the front
- A reference to the back



Reference of the First Node

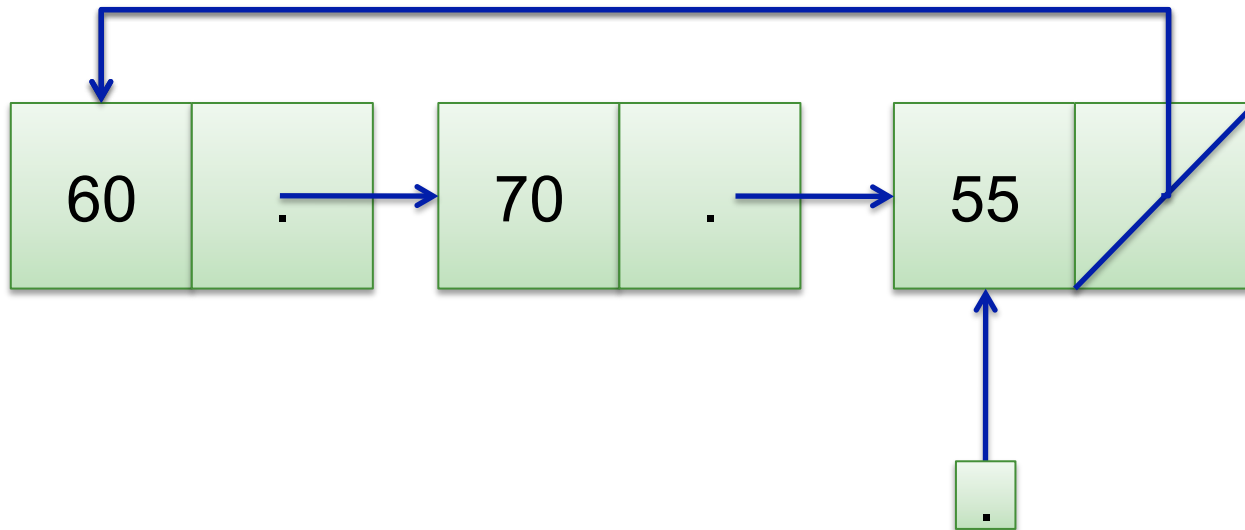
Reference of the Last Node

# Reference-Based Implementation 2



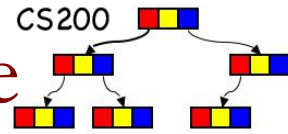
A circular linked list with one external reference

- ❑ `lastNode` references the back of the queue
- ❑ `lastNode.getNext()` references the front

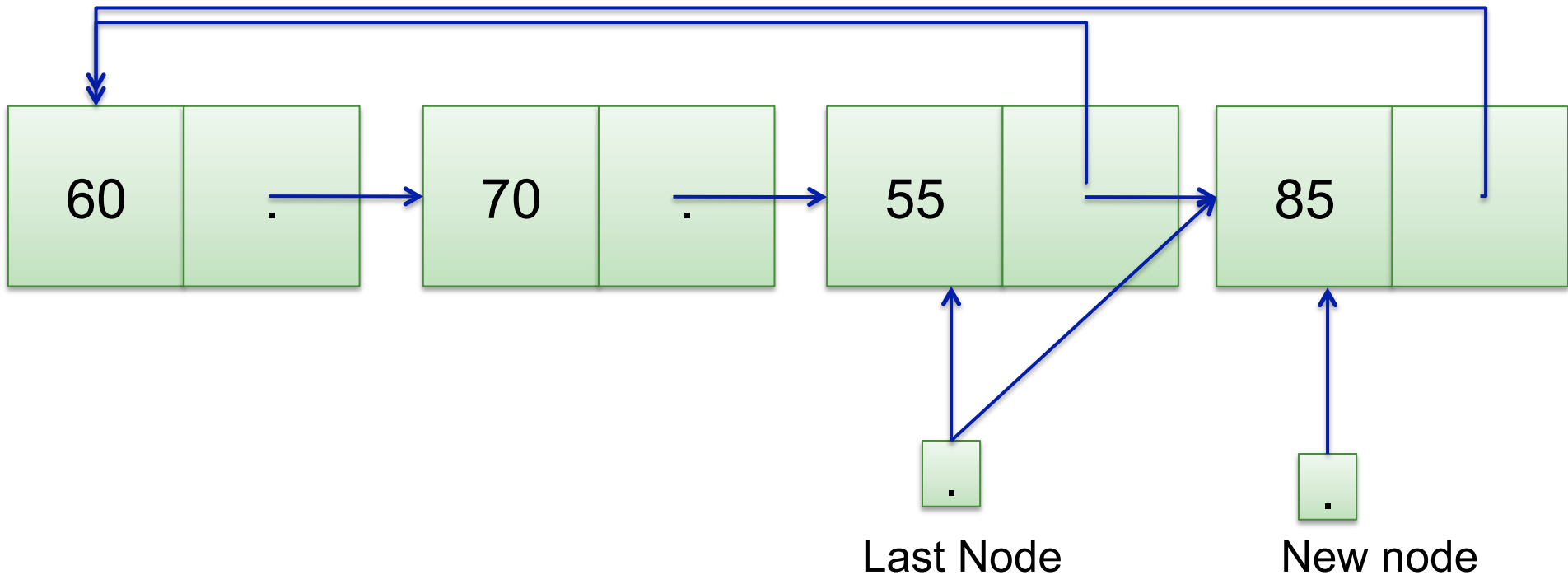


Last Node: node reference

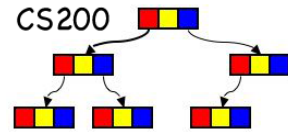
# Inserting an item into a nonempty queue



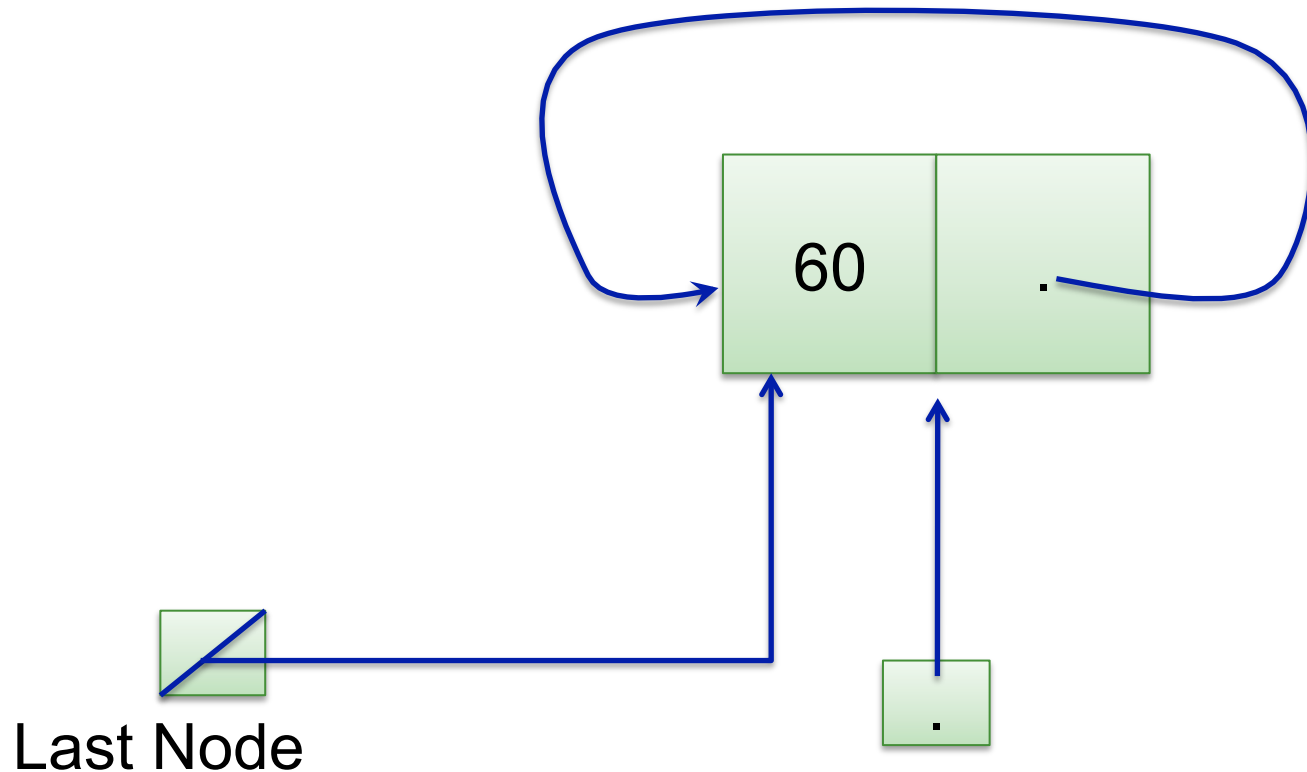
1. `newNode.next = lastNode.next;`
2. `lastNode.next = newNode;`
3. `lastNode = newNode;`



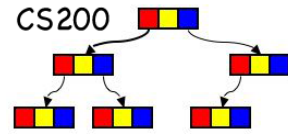
# Inserting a New Item



- Insert a *new item* into the **empty queue**



# Insert new item into the queue



```
public void enqueue (Object newItem) {
    Node newNode = new Node(newItem);

    if (isEmpty()) {
        newNode.next = newNode;
    } else {
        newNode.next = lastNode.next;
        lastNode.next = newNode;
    }

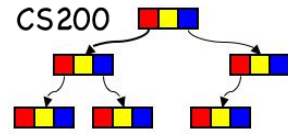
    lastNode = newNode;
}
```

**A. Empty queue**

**B. More than 1 item**



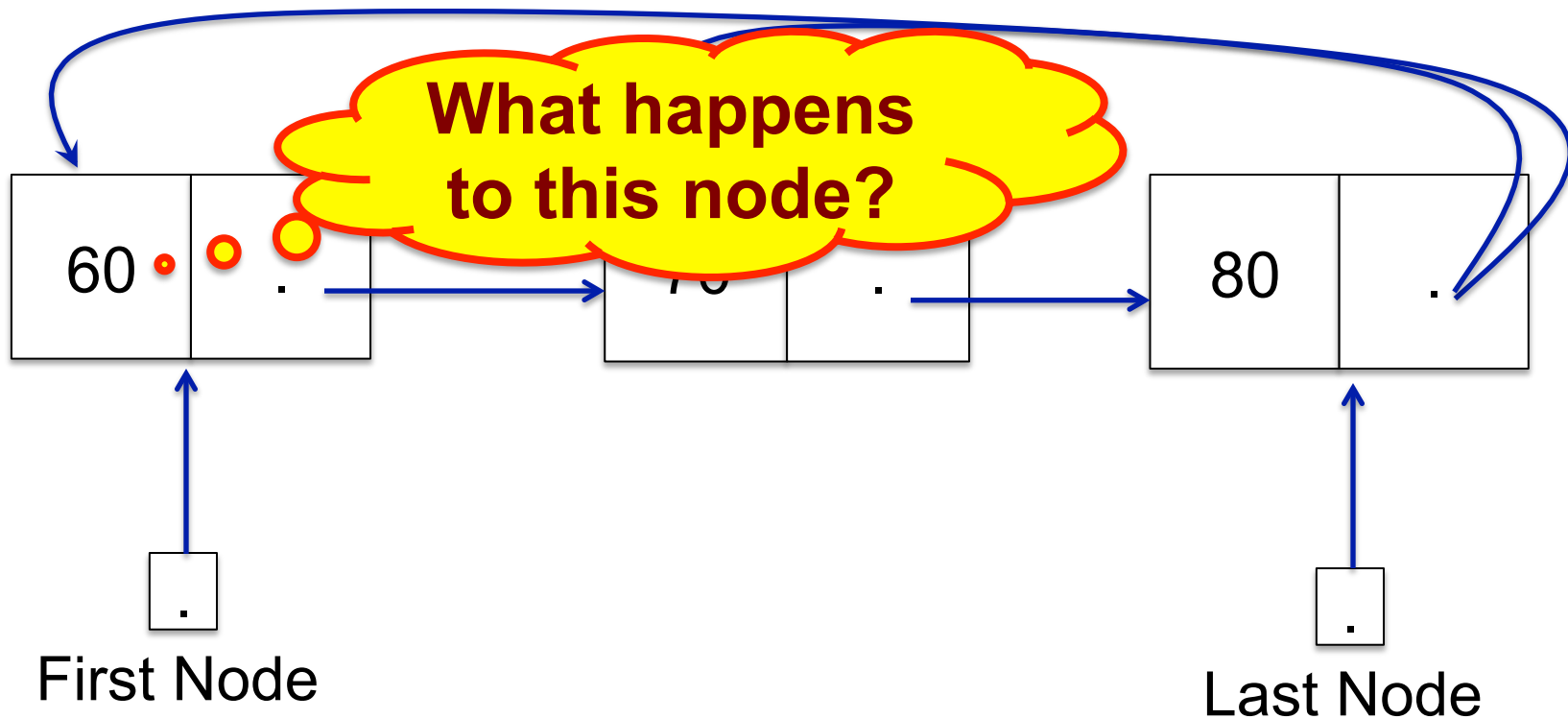
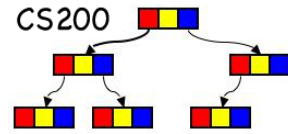
# Removing an item from queue



```
public Object dequeue() throws QueueException{
    if (!isEmpty()){
        Node firstNode = lastNode.next;
        if (firstNode == lastNode) {
            lastNode = null;
        }
        else{
            lastNode.next = firstNode.next;
        }
        return firstNode.item;
    }
    else { exception handling..
    }
}
```

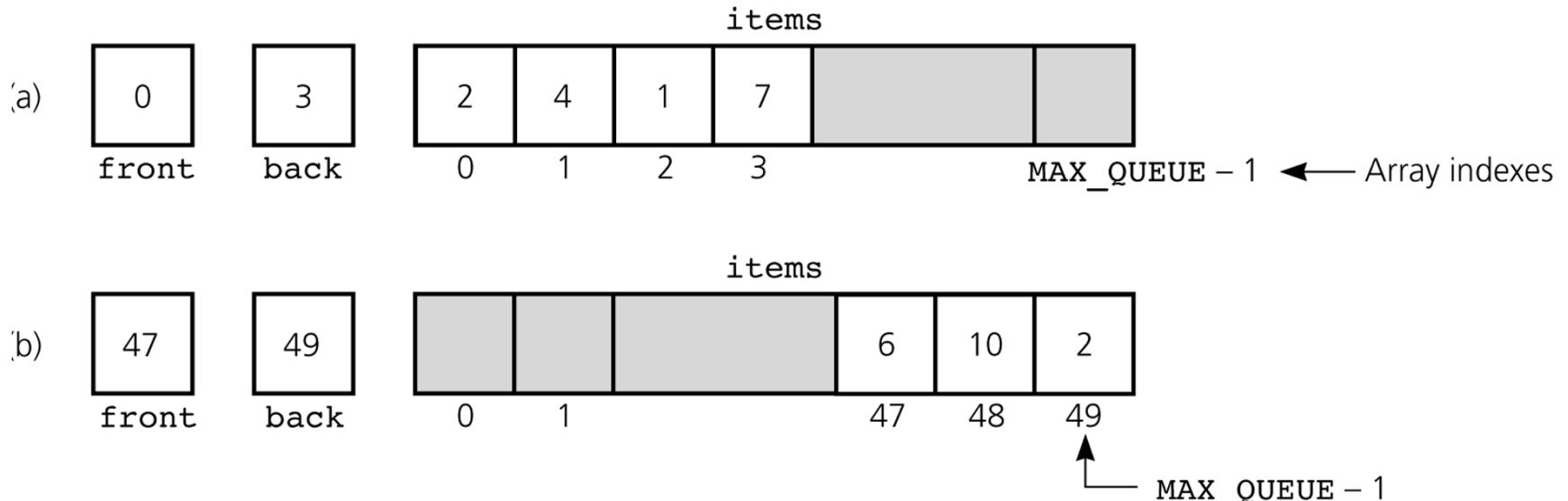
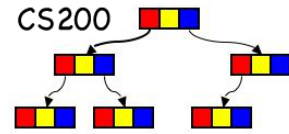
Why?

# Removing an Item



```
Node firstNode = lastNode.next;  
if (firstNode == lastNode) {  
    lastNode = null;  
} else {lastNode.next = firstNode.next;}  
return firstNode.item;
```

# Naïve Array-Based Implementation



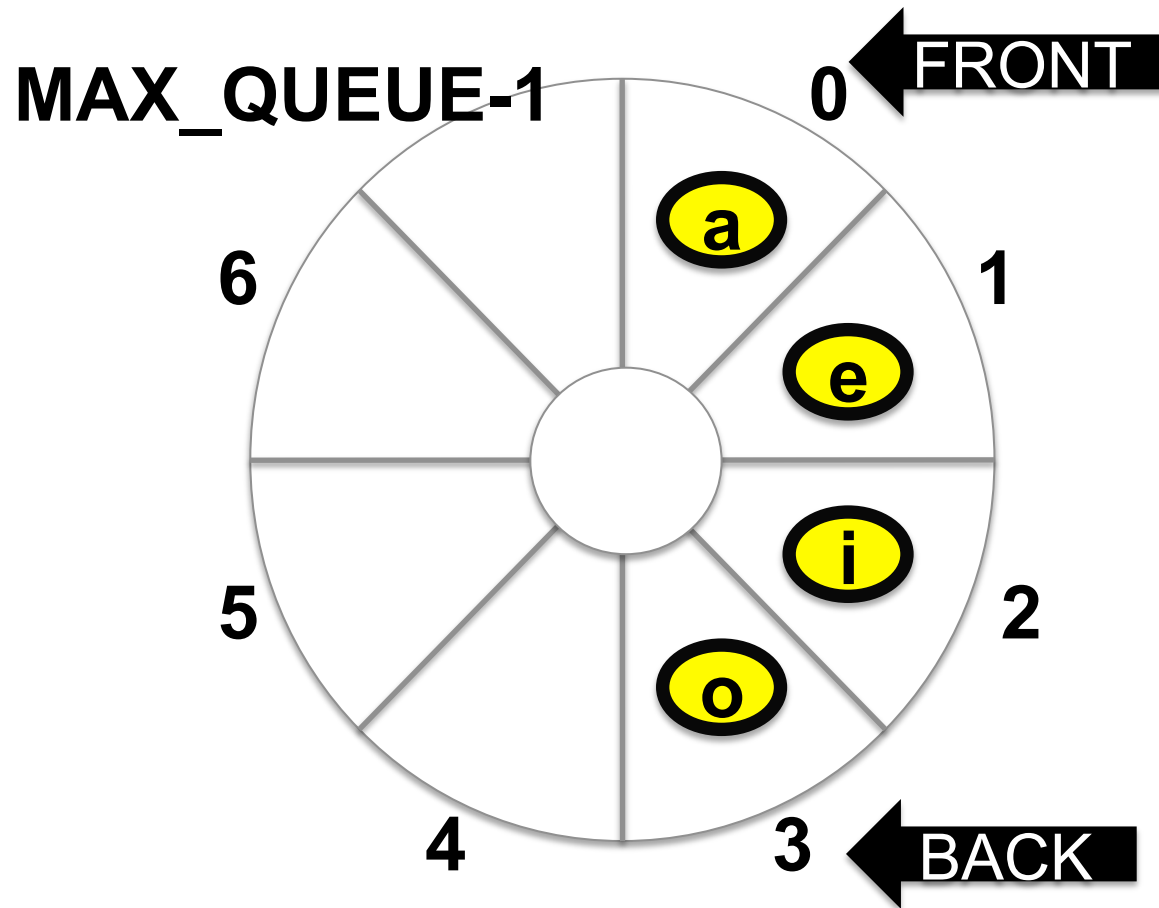
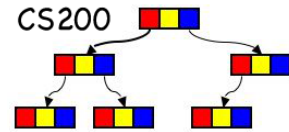
Drift wastes space

How do we initialize front and back?

(Hint: what does a queue with a single element look like?  
what does an empty queue look like?  
)

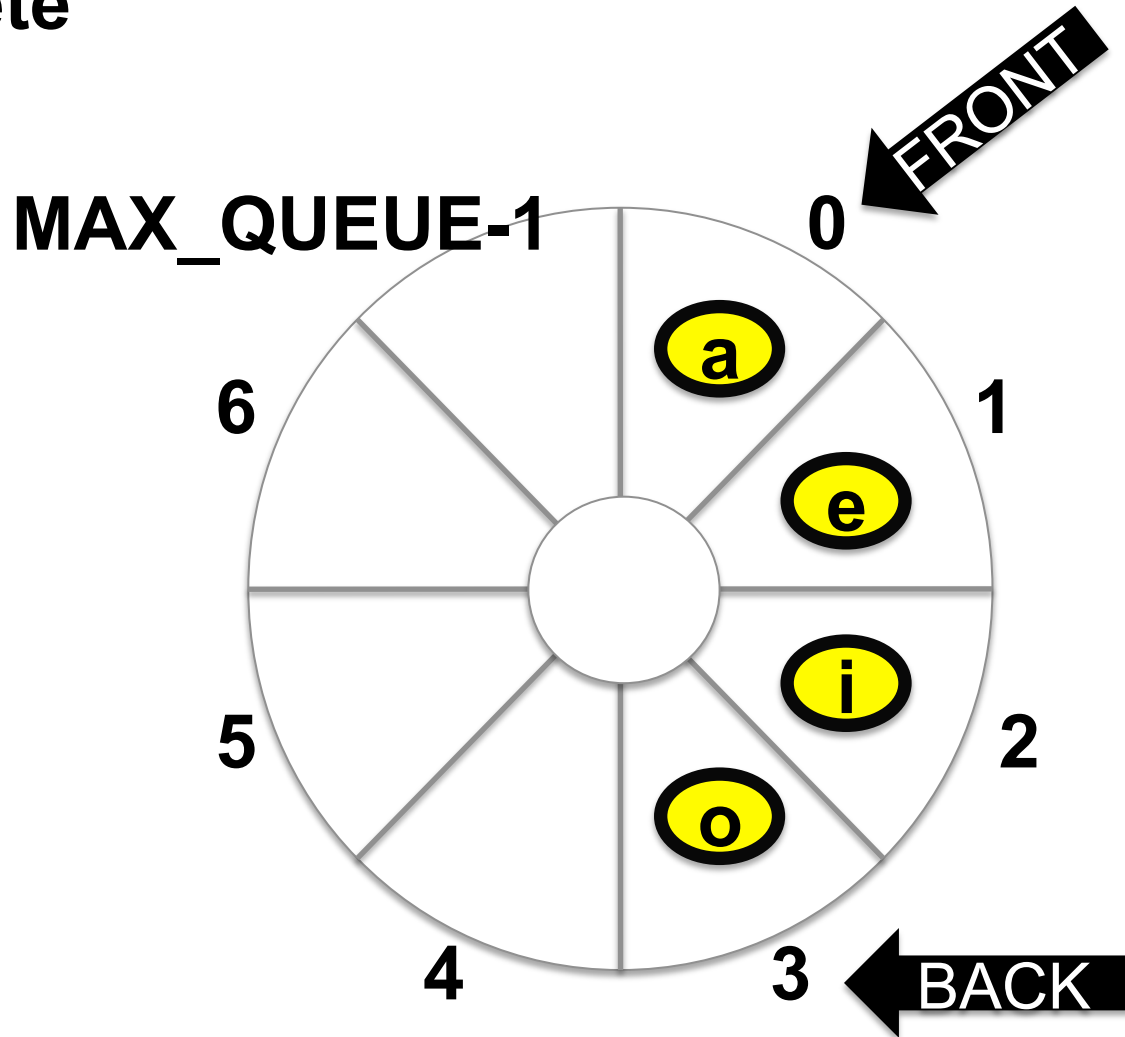
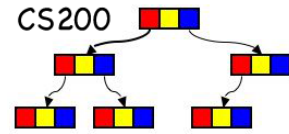
# Solving Drift:

## Circular implementation of a queue

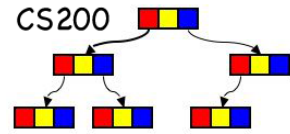


# Solving Drift:

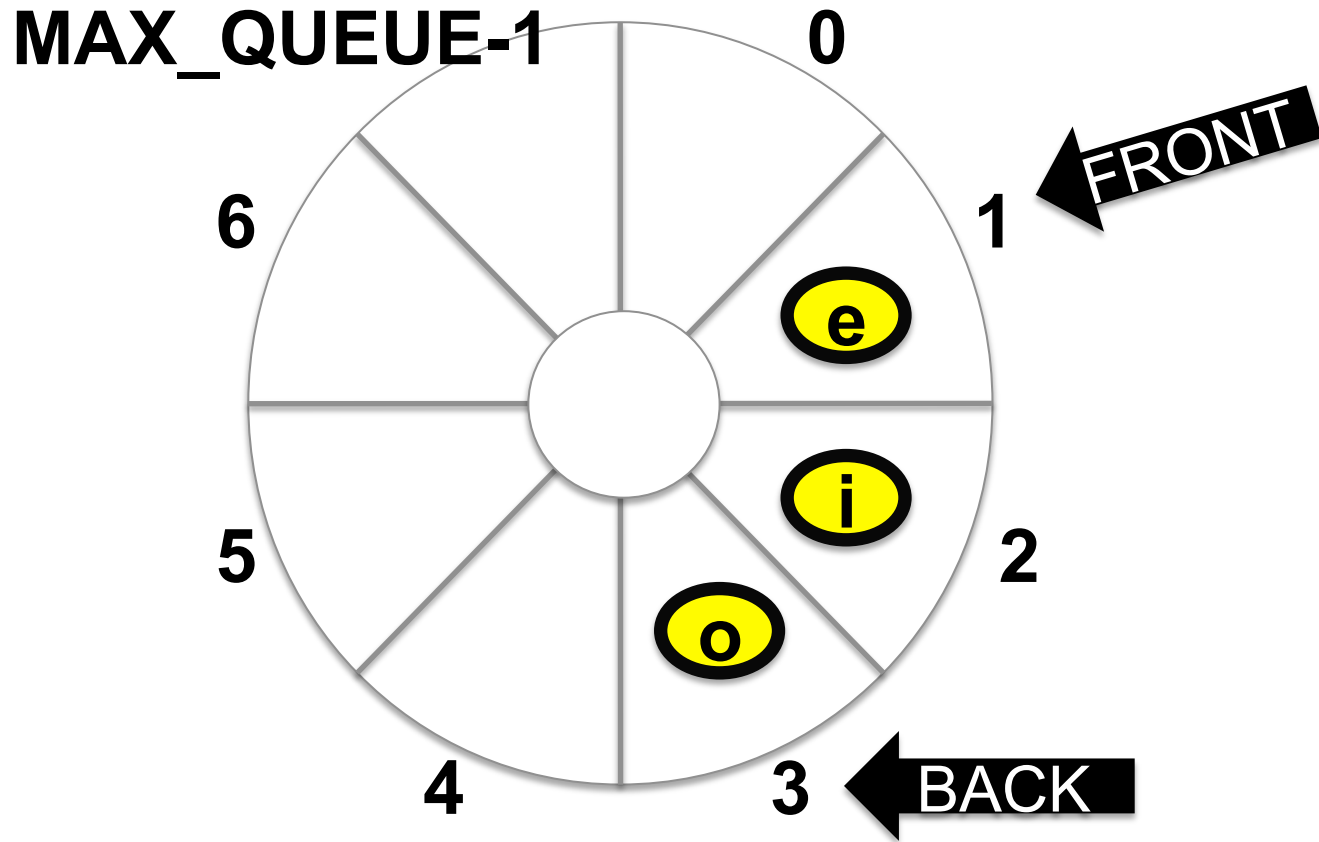
- Delete



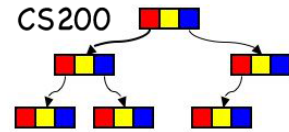
# Solving Drift:



## ■ Delete



# Solving Drift



## ■ Insert u

MAX\_QUEUE-1

0

6

1

5

2

4

3

BACK

FRONT

i

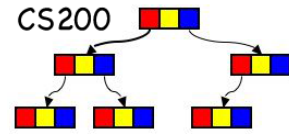
u

o

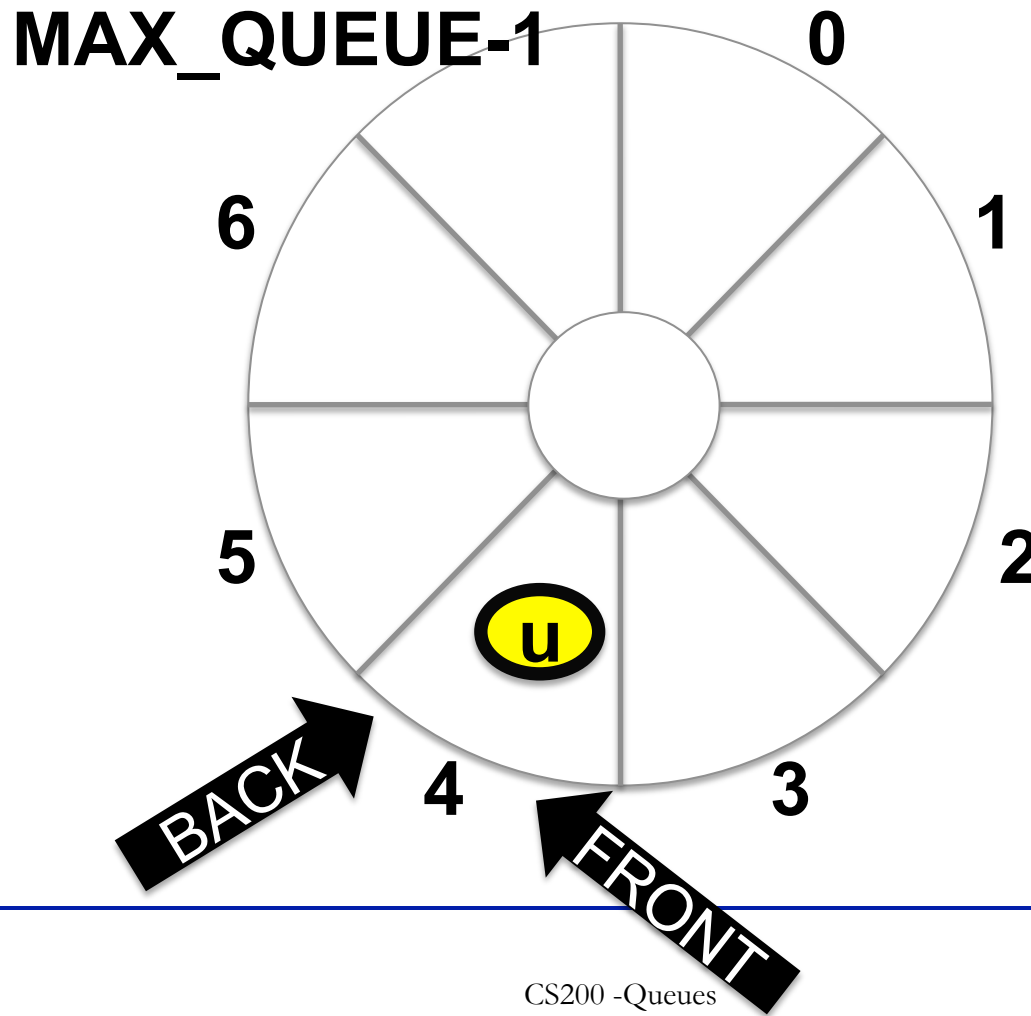
BACK

When either front or back advances past MAX\_QUEUE-1, it wraps around (to 0: using % MAX\_QUEUE)

# Queue with Single Item

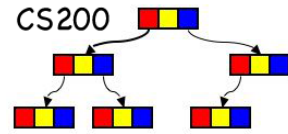


- *back* and *front* are pointing at the same slot.



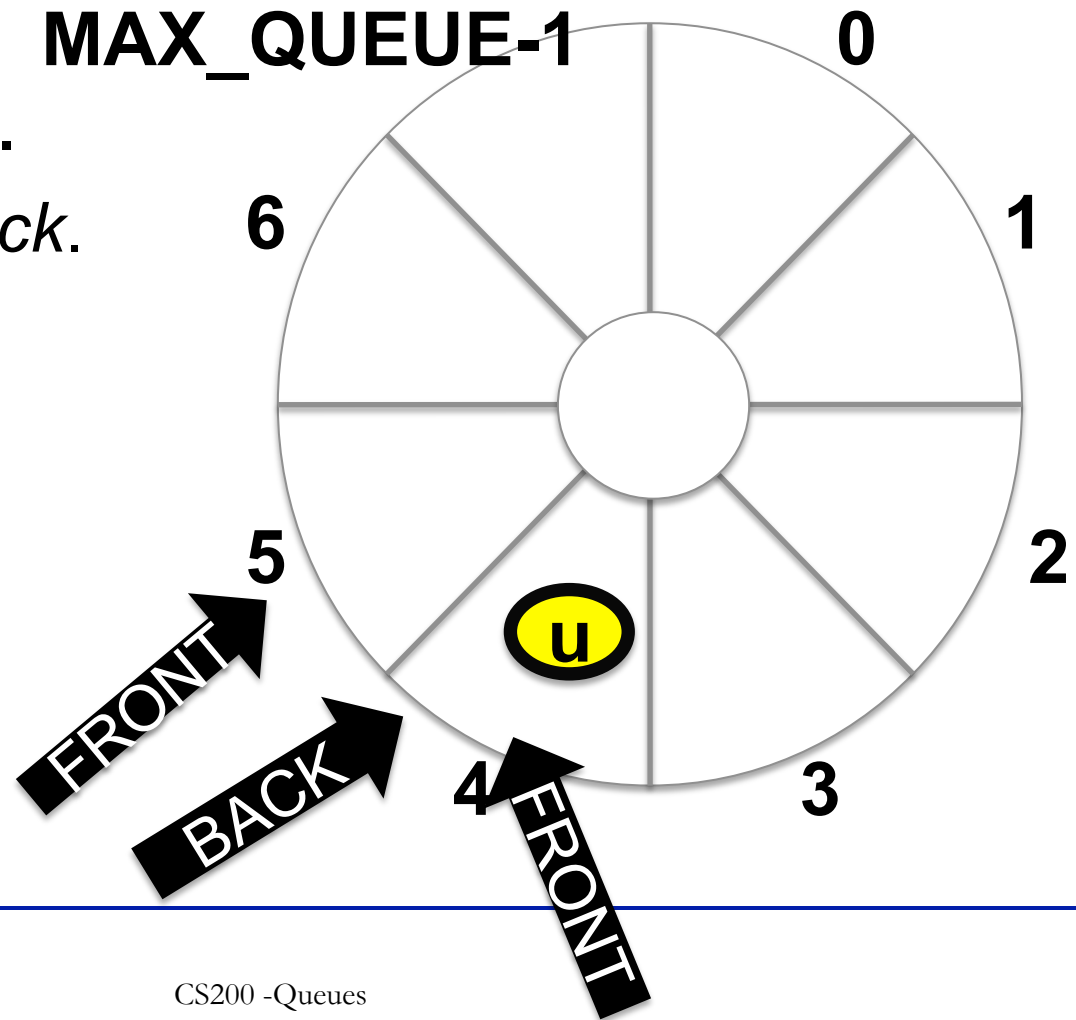


# Empty Queue: remove Single Item

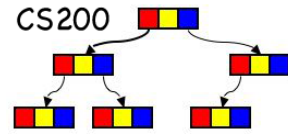


Remove last item.

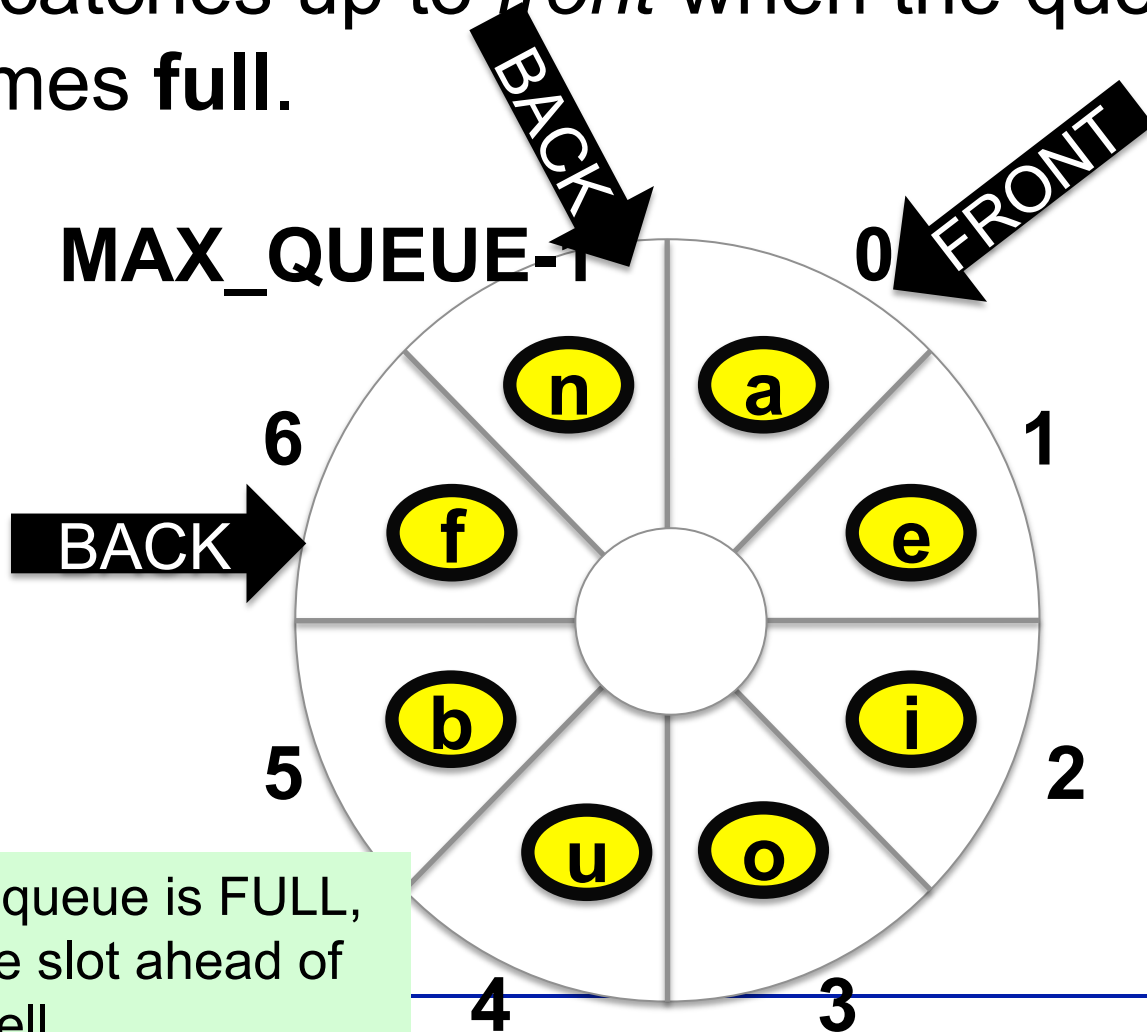
- *front passed back.*



# Insert the last item



*back* catches up to *front* when the queue becomes **full**.



Problem?

Solution?

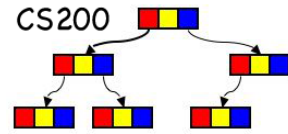
Maintain size:

0:empty

max\_queue: full

When the queue is FULL, *front* is one slot ahead of *back* as well.

# Wrapping the values for front and back



- Initializing

```
front = 0
```

```
back = MAX_QUEUE-1
```

```
count = 0
```

- Adding

```
back = (back+1) % MAX_QUEUE;
```

```
items[back] = newItem;
```

```
++count;
```

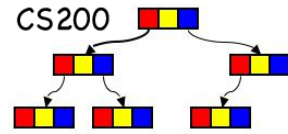
- Deleting

```
deleteItem = items[front];
```

```
front = (front +1) % MAX_QUEUE;
```

```
--count;
```

# enqueue with Array

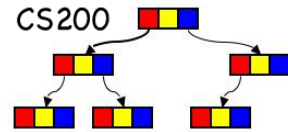


```
public void enqueue(Object newItem) throws
    QueueException{

    if (!isFull()){
        back = (back+1) % (MAX_QUEUE);
        items[back] = newItem;
        ++count;

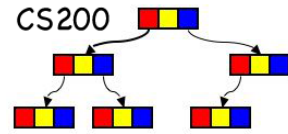
    }else {
        throw QueueException(your_message);
    }
}
```

# dequeue()



```
public Object dequeue() throws QueueException{  
    if (!isEmpty()){  
        Object queueFront = items[front];  
        front = (front+1) % (MAX_QUEUE);  
        --count;  
        return queueFront;  
    }else{  
        throw new QueueException (your_message);  
    }  
}
```

# Implementation with (Array)List



- You can implement operation **dequeue()** as the list operation **remove(0)**.
- **peek()** as **get(0)**
- **enqueue()** as **add(newItem) // at tail**

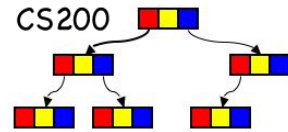
Front of queue

Back of queue



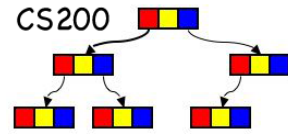
0 1 2 3 ← Position in the list

# Questions



- *What is an advantage of the circular array implementation over linked list?*
  - A. Faster to enqueue
  - B. Uses less memory
  - C. Can more easily fix and enforce a maximum size

# Expressions: infix to postfix conversion



Prichard: 7.4

Let's do some

$$2 + 3 * 4$$

$$2 * 3 + 4$$

$$2 + 3 - 4$$

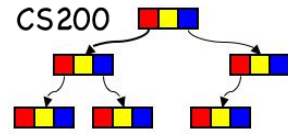
$$2 + (3 - 4)$$

$$1 - (2 + 3 * 4) / 5$$

observations



# Expressions: infix to postfix conversion



$$2 + 3 * 4 \quad \rightarrow \quad 2 \ 3 \ 4 \ * \ +$$

$$2 * 3 + 4 \quad \rightarrow \quad 2 \ 3 \ * \ 4 \ +$$

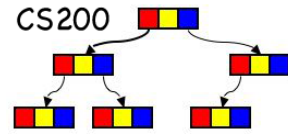
$$2 + 3 - 4 \quad \rightarrow \quad 2 \ 3 \ + \ 4 \ -$$

$$2 + (3 - 4) \quad \rightarrow \quad 2 \ 3 \ 4 \ - \ +$$

$$1 - (2 + 3 * 4) / 5 \quad \rightarrow \quad 1 \ 2 \ 3 \ 4 \ * \ + \ 5 \ / \ -$$

1. operand order does not change
2. operators come after second operand and obey associativity and precedence rules
3. ( ) converts the inner expression to independent postfix

# infix to postfix implementation



- Use a queue to create the resulting postfix expression
  - the operands get immediately enqueued
- Use a stack to store the operators
  - operators get pushed on the stack
- when to pop and enqueue?
  - let's play

$$2 + 3 * 4$$

$$\underline{2} + 3 * 4$$

$$+ \underline{3} * 4$$

$$+ 3 * \underline{4}$$

$$+ 3 * \underline{4}$$

$$\underline{4}$$

**stack**

**queue**

**action**

enqueue

push

enqueue

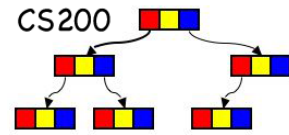
push

enqueue

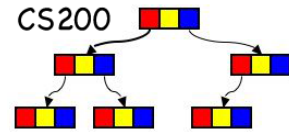
pop; enqueue

pop; enqueue

2 3 4 \* +



2 \* 3 + 4



stack	queue	action
		enqueue
	2	push
*	2	enqueue
*	2 3	push?

2 \* 3 + 4

   \* 3 + 4

   3 + 4

   + 4

   + 4

   + 4

4

  + 4

2 3 \* 4 +

NO!! Because \* has higher precedence than + and so binds to 2 3

\* 2 3 pop; enqueue

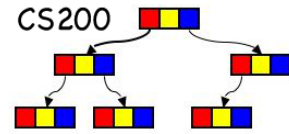
2 3 \* push

2 3 \* enqueue

2 3 \* 4 pop; enqueue

2 3 \* 4 +

2 - 3 + 4



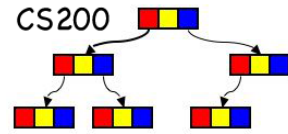
**stack**      **queue**      **action**

<u>2</u> - 3 + 4			enqueue
<u>-</u> 3 + 4		2	push
<u>3</u> + 4	-	2	enqueue
<u>±</u> 4	-	2 3	push?

NO!! Because of left associativity – binds to 2 3

<u>±</u> 4	-	2 3	pop; enqueue
<u>±</u> 4		2 3 -	push
4	+	2 3 -	enqueue
	+	2 3 - 4	pop; enqueue
		2 3 - 4 +	

$$2 - (3 + 4)$$



	stack	queue	action
--	-------	-------	--------

$$\underline{2} - (3 + 4)$$

enqueue

$$\underline{-} (3 + 4)$$

2

push

$$\underline{(} 3 + 4)$$

-

2

enqueue or push?

the expression inside the ( ) makes its own independent postfix, so we push the ( then use the stack as before until we see a ) then we pop all the operators off the stack until we see a ( and delete the (

$$\underline{3} + 4 )$$

(

-

2

enqueue

$$\underline{+} 4 )$$

(

-

2 3

push

+

$$\underline{4} )$$

(

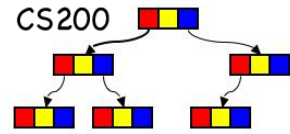
-

2 3

enqueue

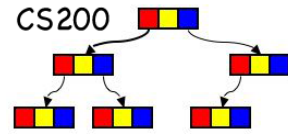
continued next page

$$2 - (3 + 4)$$



	stack	queue	action
	+		
	(		
4 )	-	2 3	enqueue
	+		
	(		
)	-	2 3 4	pop, enqueue until (, delete (
	-	2 3 4 +	pop, enqueue until stack empty
		2 3 4 + -	

# in2post algorithm



when encountering

**operand:** enqueue

**open:** push

**close:**

pop and enqueue operators, until open on stack

pop open

**operator:**

if stack empty: push

pop and enqueue operators with greater or equal precedence, until operator with lower precedence on stack, or open on stack, or stack empty

**end of input:**

pop and enqueue all operators until stack empty

Do it for:  $1-(2+3*4)/5$

**do it, do it**