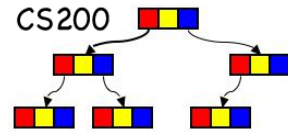


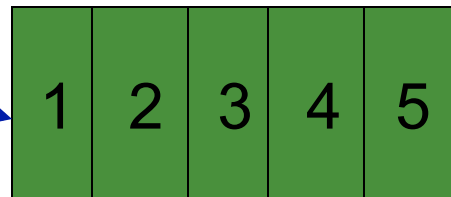
CS200: Queues

- Prichard Ch. 8

Queues

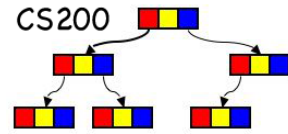


- First In First Out (FIFO) structure
- Imagine a checkout line
- So **removing** and **adding** are done from opposite ends of structure.



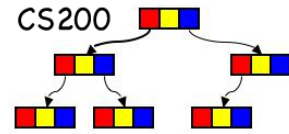
- add to tail (back), remove from head (front)
- Used in operating systems (e.g. print queue).

Possible Queue Operations



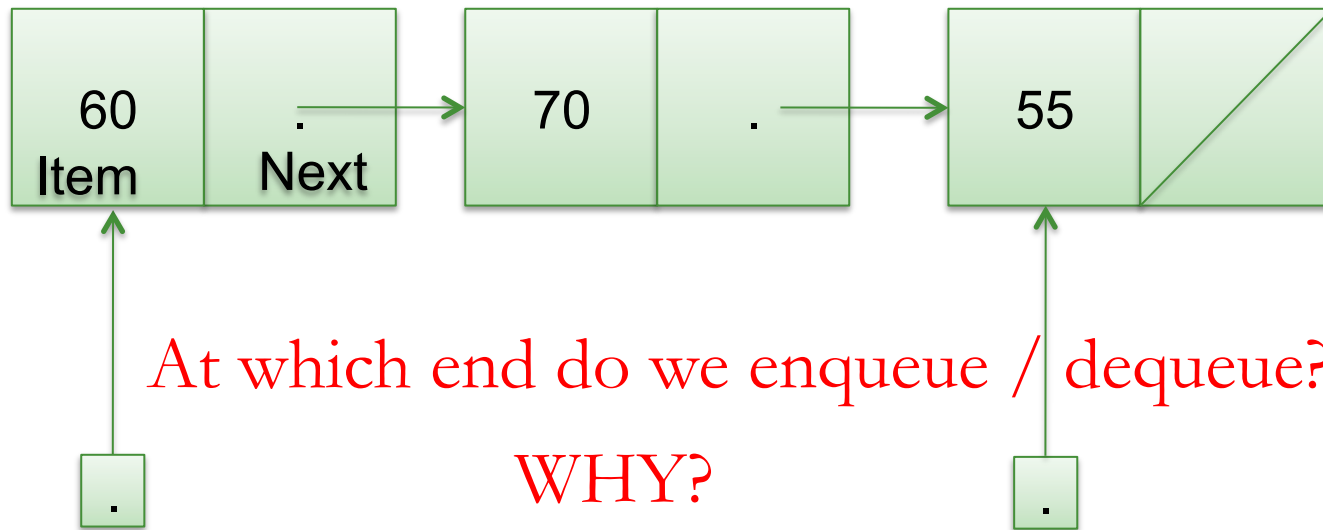
- **enqueue**(in newItem: QueueItemType)
 - Add new item at the back of a queue
- **dequeue**(): QueueItemType
 - Retrieve and remove the item at the *front* of a queue
- **peek**(): QueueItemType
 - Retrieve item from the *front* of the queue. Retrieve the item that was added earliest.
- **isEmpty**():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

Reference-Based Implementation 1



A linked list with two external references

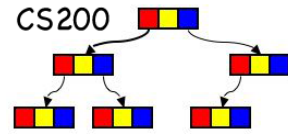
- A reference to the front
- A reference to the back



Reference of the First Node

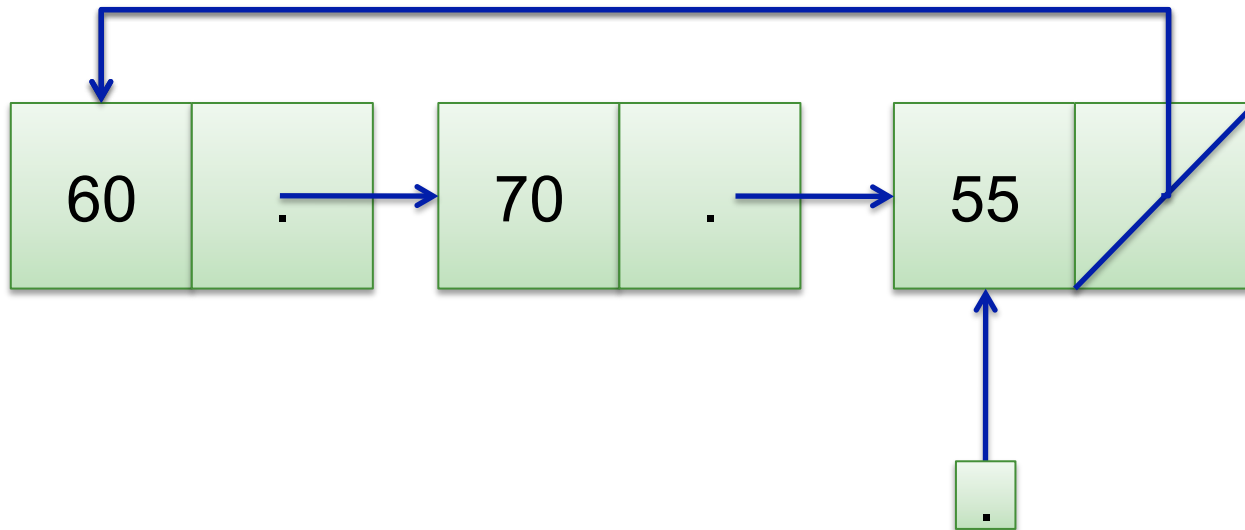
Reference of the Last Node

Reference-Based Implementation 2



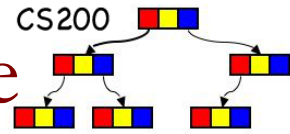
A circular linked list with one external reference

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

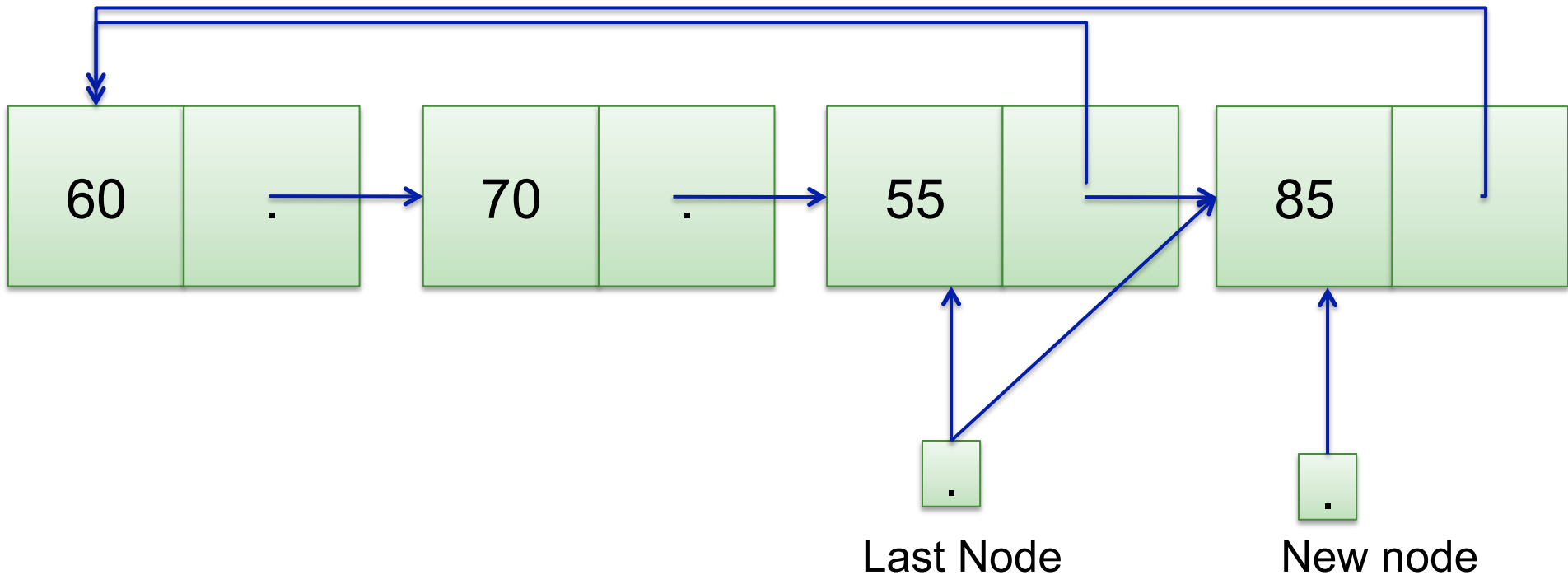


Last Node: node reference

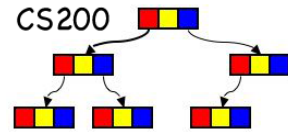
Inserting an item into a nonempty queue



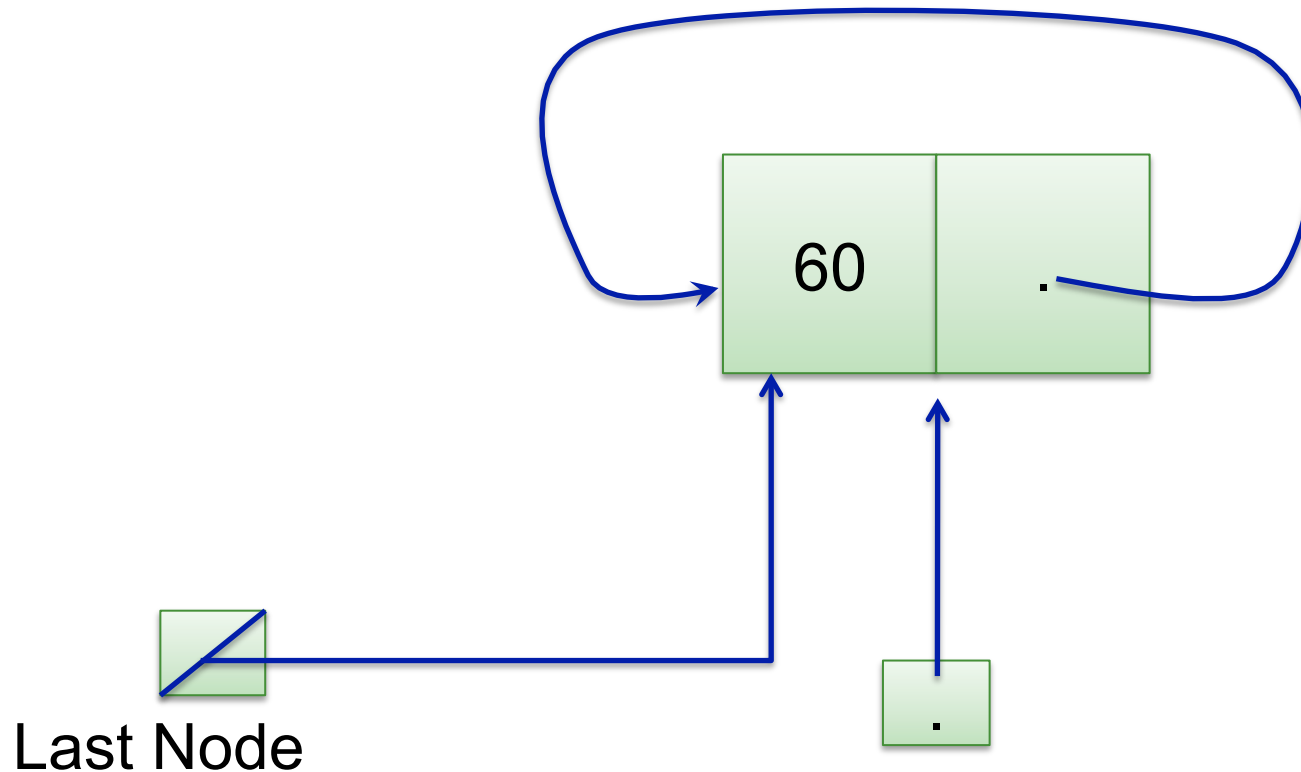
1. `newNode.next = lastNode.next;`
2. `lastNode.next = newNode;`
3. `lastNode = newNode;`



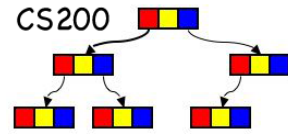
Inserting a New Item



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



Insert new item into the queue



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

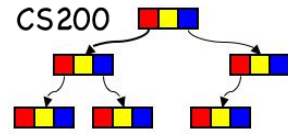
    if (isEmpty()) {
        newNode.next = newNode;
    } else {
        newNode.next = lastNode.next;
        lastNode.next = newNode;
    }

    lastNode = newNode;
}
```

A. Empty queue

B. items in queue

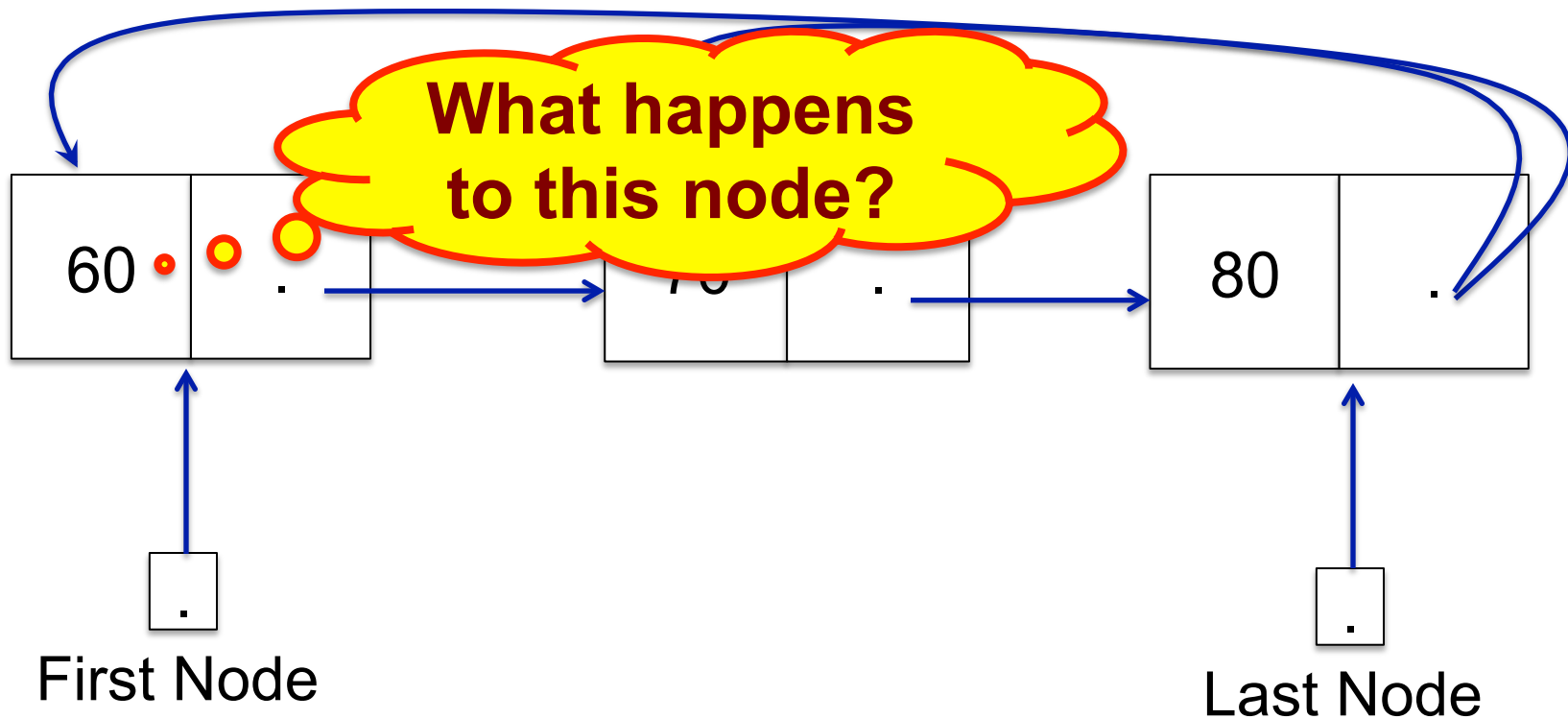
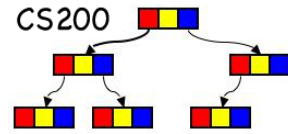
Removing an item from queue



```
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..
    }
}
```

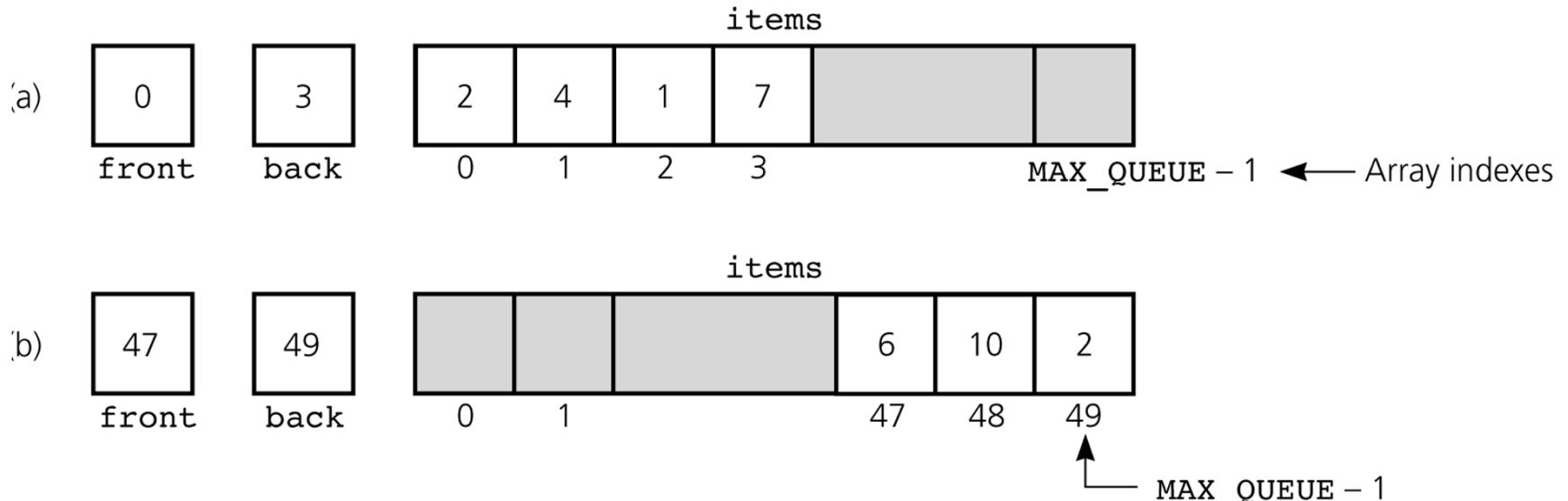
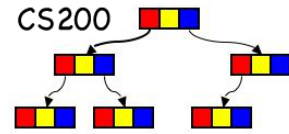
Why?

Removing an Item



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

Naïve Array-Based Implementation



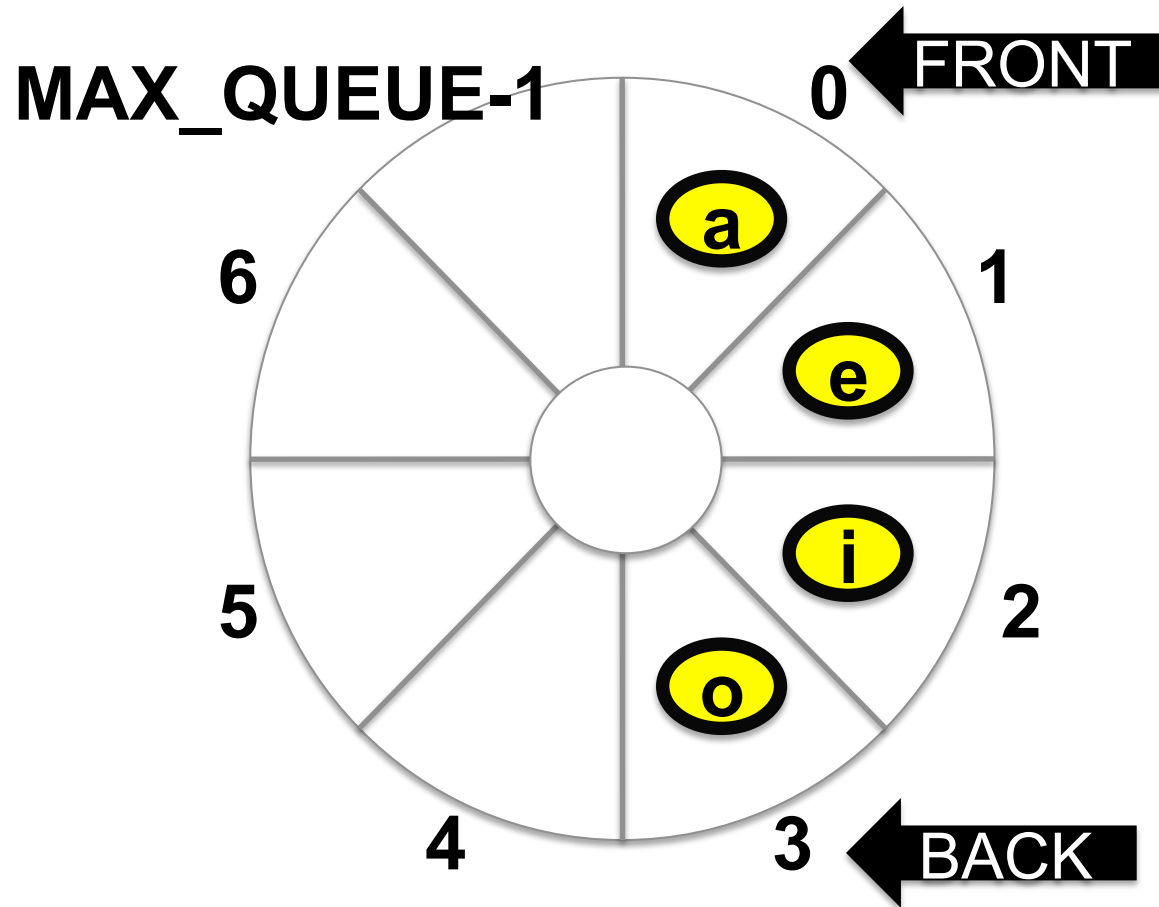
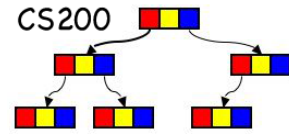
Drift wastes space

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?)

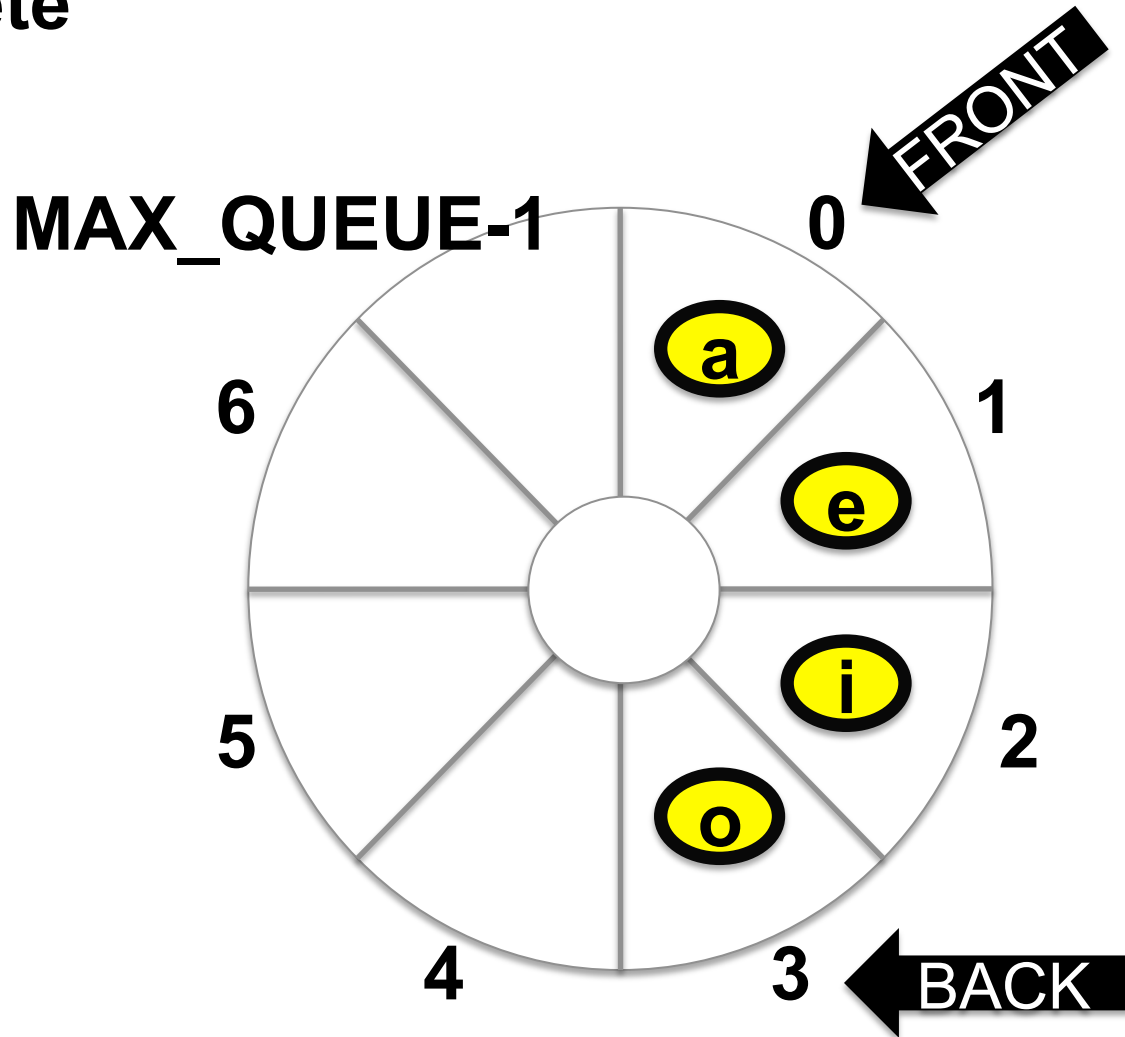
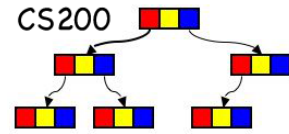
Solving Drift:

Circular implementation of a queue

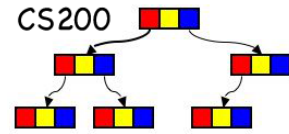


Solving Drift:

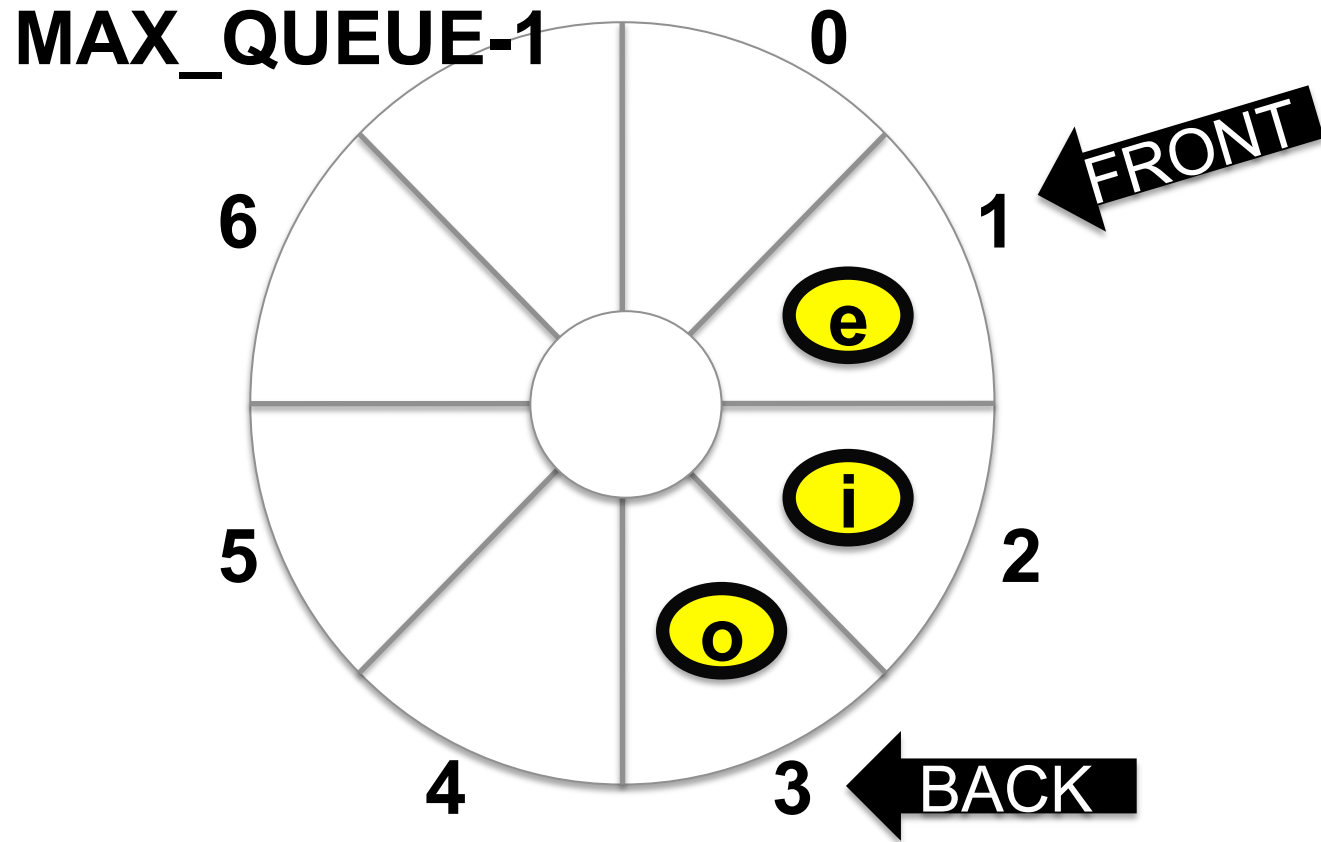
- Delete



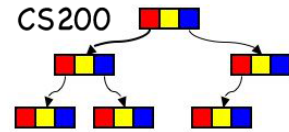
Solving Drift:



■ Delete



Solving Drift



■ Insert u

MAX_QUEUE-1

0

6

1

5

2

BACK

4

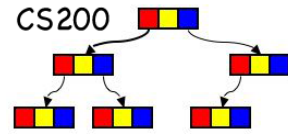
3

BACK

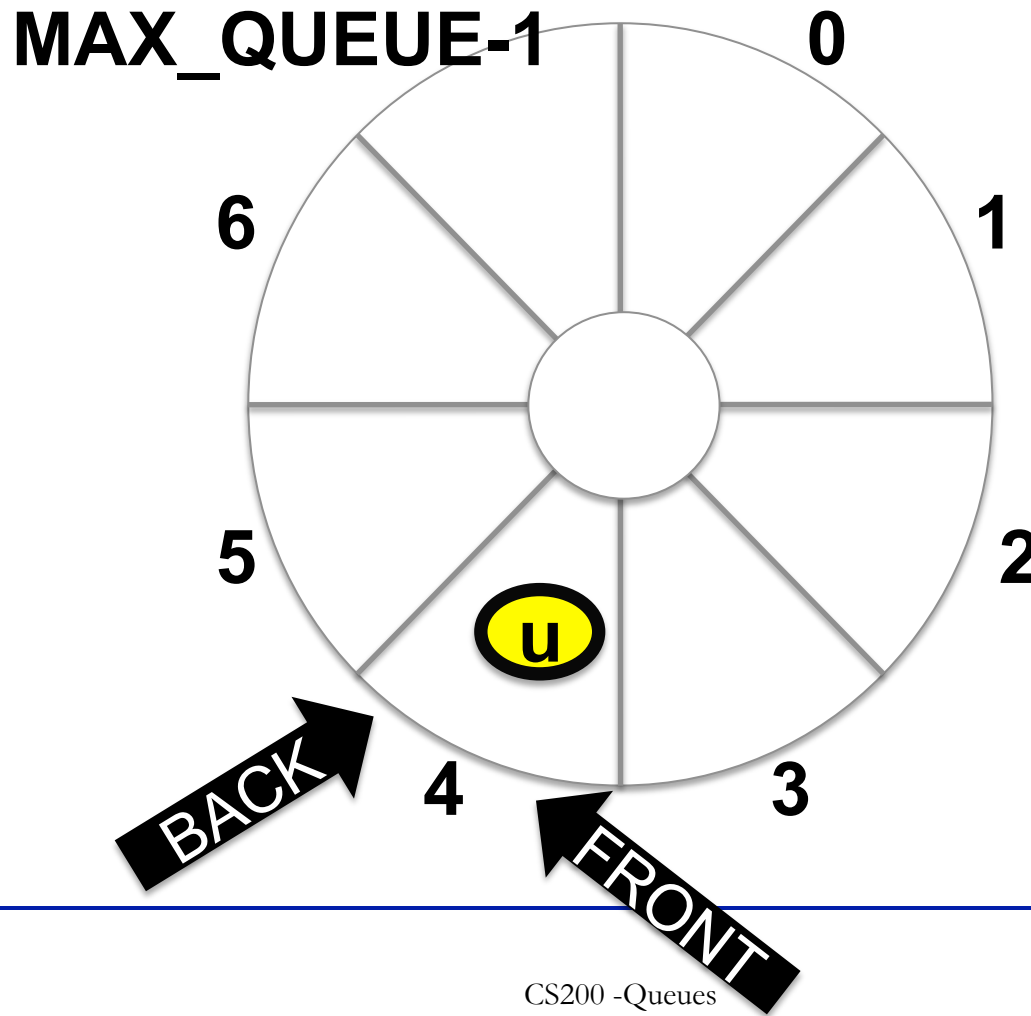
FRONT

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

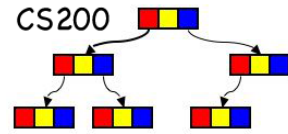
Queue with Single Item



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

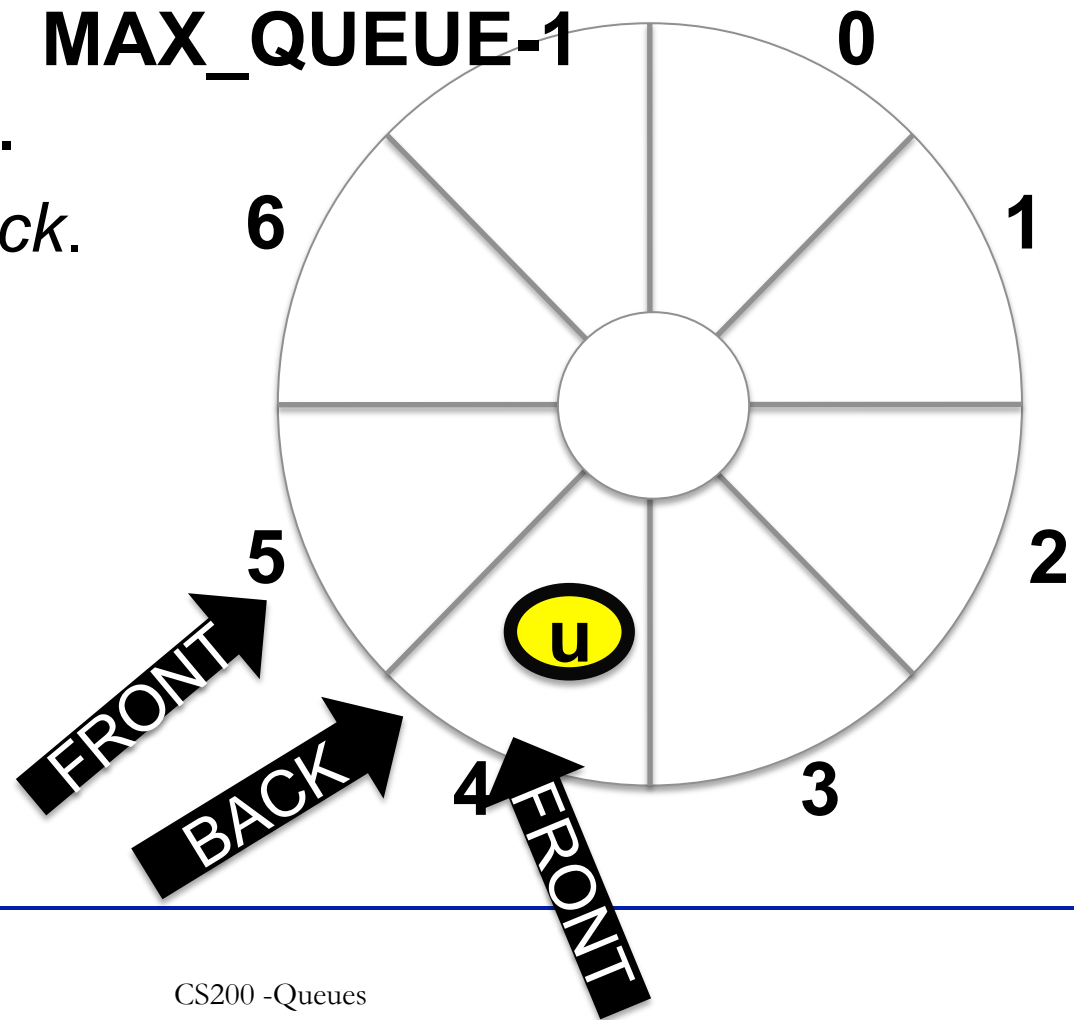


Empty Queue: remove Single Item

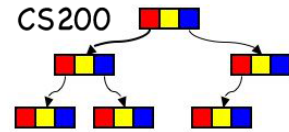


Remove last item.

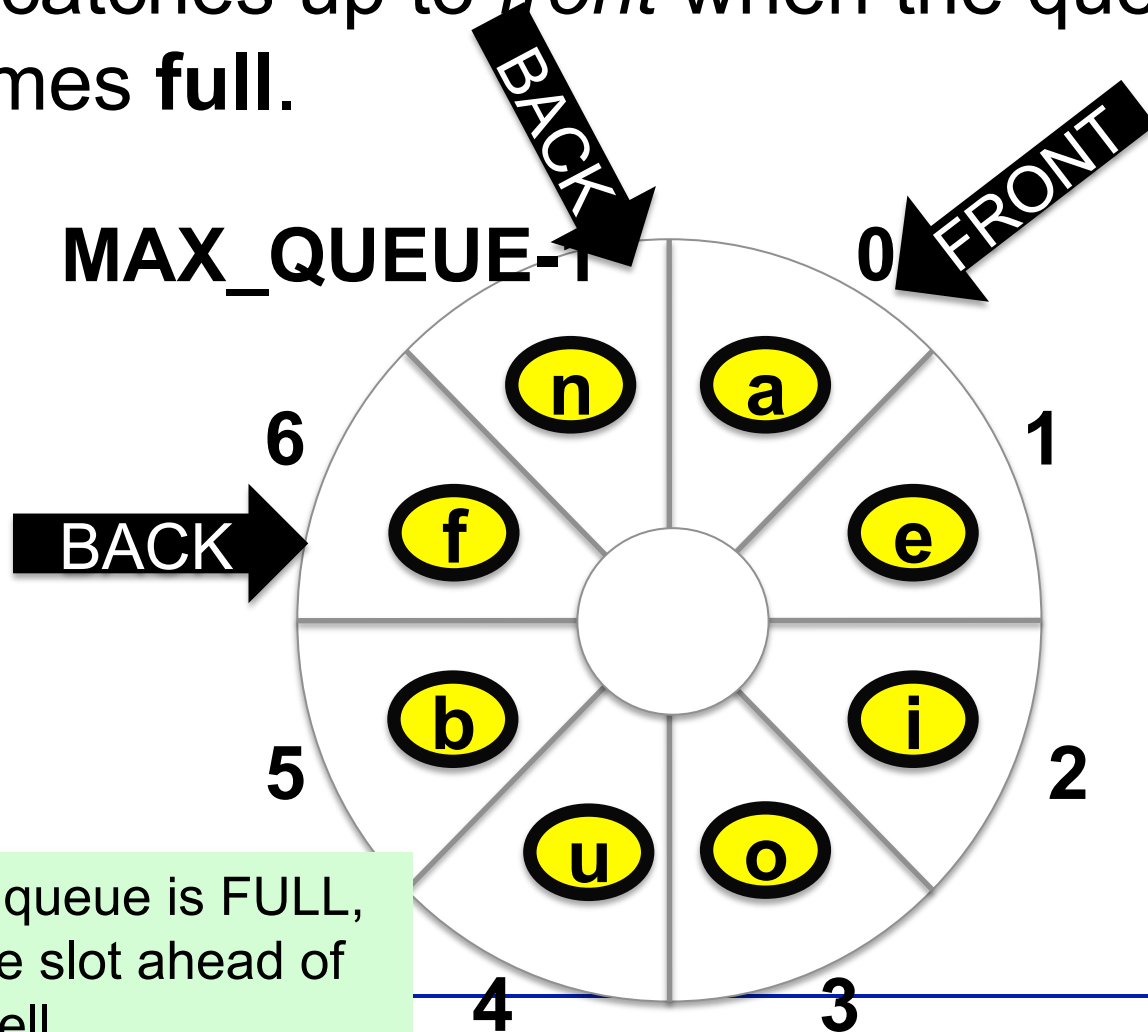
- *front passed back.*



Insert the last item



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



Problem?

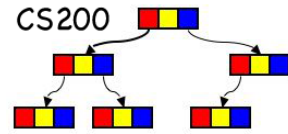
Solution?

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;
```

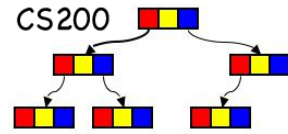
- Deleting

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

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

```
--count;
```

enqueue with Array

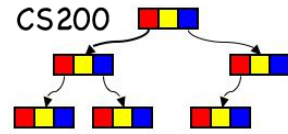


```
public void enqueue(Object newItem) throws
    QueueException{

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

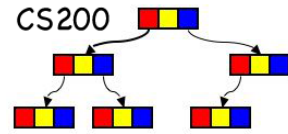
    }else {
        throw new QueueException(your_message);
    }
}
```

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**

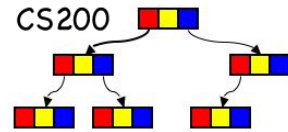
Front of queue

Back of queue



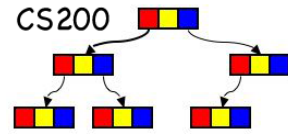
0 1 2 3 ← Position in the list

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
 - D. Fewer allocations

Expressions: infix to postfix conversion



Prichard: 7.4

Let's do some

$$2 + 3 * 4$$

$$2 * 3 + 4$$

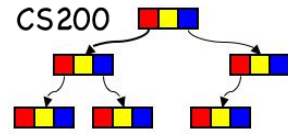
$$2 + 3 - 4$$

$$2 + (3 - 4)$$

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

observations?

Expressions: infix to postfix conversion



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

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

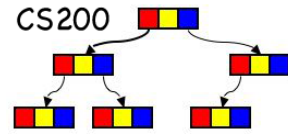
$2 + 3 - 4 \rightarrow 2 3 + 4 -$

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

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

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

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

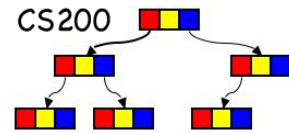
push

enqueue

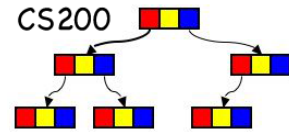
pop; enqueue

pop; enqueue

2 3 4 * +



2 * 3 + 4

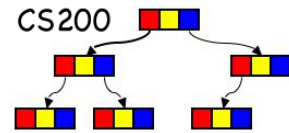


	stack	queue	action
<u>2</u> * 3 + 4			enqueue
<u>*</u> 3 + 4		2	push
<u>3</u> + 4	*	2	enqueue
<u>+</u> 4	*	2 3	push?

NO!! Because * has higher precedence than + and so 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 +	

$$2 - 3 + 4$$

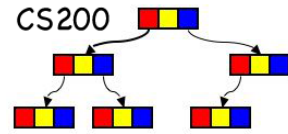


	stack	queue	action
<u>2</u> - 3 + 4			enqueue
<u>-</u> 3 + 4		2	push
<u>3</u> + 4	-	2	enqueue
<u>±</u> 4	-	2 3	push?

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 +	

$$2 - (3 + 4)$$



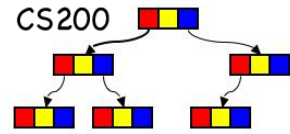
	stack	queue	action
<u>2</u> - (3 + 4)			enqueue
= (3 + 4)		2	push
(3 + 4)	-	2	enqueue or push?

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 and enqueue them, until we see a (and delete the (

(
<u>3</u> + 4)	-	2	enqueue
(
+ 4)	-	2 3	push
	+		
(
4)	-	2 3	enqueue

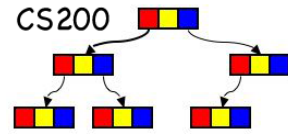
continued next page

$$2 - (3 + 4)$$



	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 or top is open push, else

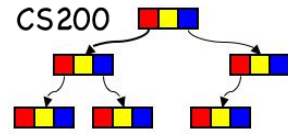
pop and enqueue operators with greater or equal precedence, until operator with lower precedence on stack, or open on stack, or stack empty

end of input:

pop and enqueue all operators until stack empty

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

What about unary operators?



- e.g. **not** in logic expressions such as:

not true and false

not (true or false)

not not true

not has higher priority than **and**,

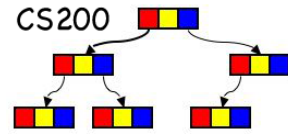
true and not false is true and (not false)

and has higher priority than **or**

not is right associative

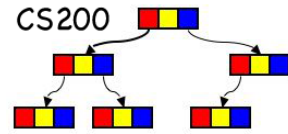
not not true is not (not true)

not true and false



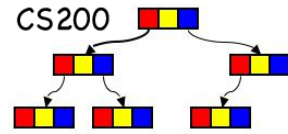
	stack	queue	action
not true and false			push
true and false	not		enqueue
and false	not	true	not higher priority pop, enqueue not
and false		true not	push
false	and	true not	enqueue
	and	true not false true not false and	pop, enqueue

not(true or false)



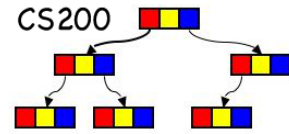
	stack	queue	action
not (true or false)			push
(true or false)	not		push
true or false)	(not		enqueue
or false)	(not	true	push
false)	or (not	true	enqueue

not(true or false) continued



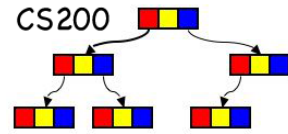
	stack	queue	action
or			
(
false)	not	true	enqueue
	or		
(
)	not	true false	pop, enqueue until (
	not	true false or	pop, enqueue
		true false or not	

not not true



	stack	queue	action
not not true			push
not true	not		push or enqueue?
push! not is right associative, its operand is ahead of it			
true	not		enqueue
	not		
	not	true	pop and enqueue
		true not not	

in2post algorithm



when encountering

operand: enqueue

open: push

close:

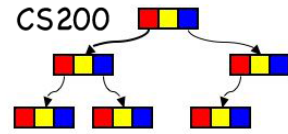
pop and enqueue operators, until open on stack

pop open

end of input:

pop and enqueue all operators until stack empty

in2post continued



when encountering

and, or:

if stack empty or top is open push, else pop and enqueue operators with greater or equal precedence, until operator with lower precedence on stack, or open on stack, or stack empty

not:

push

do it for not (not true or false)