Queue and Iterator

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2024

Queue

Queue

A Queue is like a Stack

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.
Like Stack, the Queue interface has some strange names.

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.
Like Stack, the Queue interface has some strange names.

- **offer** put something in the queue.

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.
Like Stack, the Queue interface has some strange names.

- **offer** put something in the queue.
- **poll** take something out.

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does. Like Stack, the Queue interface has some strange names.

- **offer** put something in the queue.
- **poll** take something out.
- **peek** who is next to be served?

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.
Like Stack, the Queue interface has some strange names.

- **offer** put something in the queue.
- **poll** take something out.
- **peek** who is next to be served?
- **size** how many are in the queue?

Queue

A Queue is like a Stack

- EXCEPT THAT things that go in first come out FIRST (not last).
- So it's like waiting in line.

I will run a MaintainQueue program for you to illustrate what a Queue does.
Like Stack, the Queue interface has some strange names.

- **offer** put something in the queue.
- **poll** take something out.
- **peek** who is next to be served?
- **size** how many are in the queue?

Have a look at the Queue interface.

AbstractQueue

AbstractQueue

The Queue interface has other methods,

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them
- thanks to the AbstractQueue class.

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them
- thanks to the AbstractQueue class.
- AbstractQueue is an abstract class,

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them
- thanks to the AbstractQueue class.
- AbstractQueue is an abstract class,
- meaning it is partially implemented.

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them
- thanks to the AbstractQueue class.
- AbstractQueue is an abstract class,
- meaning it is partially implemented.
- To create an implementation of Queue, you just have to finish it.

AbstractQueue

The Queue interface has other methods,

- but we do not have to implement them
- thanks to the AbstractQueue class.
- AbstractQueue is an abstract class,
- meaning it is partially implemented.
- To create an implementation of Queue, you just have to finish it.

“A Queue implementation that extends this class must minimally define a method Queue.offer(E) which does not permit insertion of null elements, along with methods Queue.peek(), Queue.poll(), Collection.size(), and a Collection.iterator() supporting Iterator.remove(). Typically, additional methods will be overridden as well. If these requirements cannot be met, consider instead subclassing AbstractCollection.”

AbstractQueue

AbstractQueue

AbstractQueue implements other Queue methods:

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**$>$ 0**?** True!

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**$>$ 0**?** True!
- So both of us call **remove()**.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**$>$ 0**?** True!
- So both of us call **remove()**.
- One of us is going to crash!

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**$>$ 0**?** True!
- So both of us call **remove()**.
- One of us is going to crash!
- **poll()** would just return **null** to the unlucky one, not crash.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**>0**?** True!
- So both of us call **remove()**.
- One of us is going to crash!
- **poll()** would just return **null** to the unlucky one, not crash.

AbstractQueue implements **add** by calling *your* **offer** method.

AbstractQueue

AbstractQueue implements other Queue methods:

- **add**, like **offer** but throws an exception if out of memory;
- **remove**, like **poll** but throws an exception on empty queue;
- **element**, like **peek** but throws an exception on empty queue.

poll returns null on an empty queue.

- Why do we need that? Think about parallel processing.
- Suppose **size()** equals 1, and two of us have access to the queue.
- Both of us ask if **size()**$>$ 0**?** True!
- So both of us call **remove()**.
- One of us is going to crash!
- **poll()** would just return **null** to the unlucky one, not crash.

AbstractQueue implements **add** by calling *your* **offer** method.
Can you write it?

Implementations of Queue

Implementations of Queue

We will learn two ways to implement a Queue:

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.
- Peek and poll are just like peek and pop for a stack.

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.
- Peek and poll are just like peek and pop for a stack.
- Except you should set **bottom** to null when you set **top** to null when you pop the last element.

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.
- Peek and poll are just like peek and pop for a stack.
- Except you should set **bottom** to null when you set **top** to null when you pop the last element.
- Offer on an empty queue sets both **top** and **bottom** to the (sole) new Node.

Implementations of Queue

We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.
- Peek and poll are just like peek and pop for a stack.
- Except you should set **bottom** to null when you set **top** to null when you pop the last element.
- Offer on an empty queue sets both **top** and **bottom** to the (sole) new Node.
- Offer on a non-empty queue adds a new Node below **bottom**.

Implementations of Queue

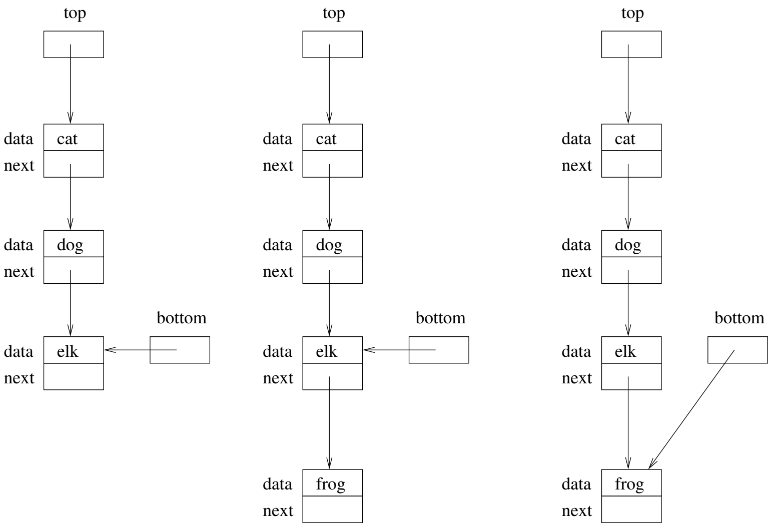
We will learn two ways to implement a Queue:

- **LinkedQueue** which uses a linked list
- **ArrayQueue** which uses an array

LinkedQueue

- Like **LinkedStack** except that it keeps track of **bottom** too.
- **top** is the first one in line. **bottom** is the last.
- Peek and poll are just like peek and pop for a stack.
- Except you should set **bottom** to null when you set **top** to null when you pop the last element.
- Offer on an empty queue sets both **top** and **bottom** to the (sole) new Node.
- Offer on a non-empty queue adds a new Node below **bottom**.
- and sets **bottom** to the new Node.

LinkedQueue offer



Implementations of Queue

ArrayQueue

Implementations of Queue

ArrayQueue

- Like **ArrayStack**, adds at the “end”.

Implementations of Queue

ArrayQueue

- Like **ArrayStack**, adds at the “end”.
- But how can it remove at the beginning (index 0),

Implementations of Queue

ArrayQueue

- Like **ArrayStack**, adds at the “end”.
- But how can it remove at the beginning (index 0),
- without moving everyone last one,

Implementations of Queue

ArrayQueue

- Like **ArrayStack**, adds at the “end”.
- But how can it remove at the beginning (index 0),
- without moving everyone last one,
- which takes $\mathrm{O}(n)$ time?

ArrayQueue Idea

ArrayQueue Idea

Suppose people want to wait “in line”

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,

ArrayQueue Idea

Suppose people want to wait "in line"

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

- 0:Victor 1:Irina 2:Parul 3:Joe 4:null

ArrayQueue Idea

Suppose people want to wait "in line"

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

- 0:Victor 1:Irina 2:Parul 3:Joe 4:null

Serve Victor and then Irina.

ArrayQueue Idea

Suppose people want to wait "in line"

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

- 0:Victor 1:Irina 2:Parul 3:Joe 4:null

Serve Victor and then Irina.

- For the sake of clarity, I will set those locations to null,

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

- 0:Victor 1:Irina 2:Parul 3:Joe 4:null

Serve Victor and then Irina.

- For the sake of clarity, I will set those locations to null,
- but I don’t really have to.

ArrayQueue Idea

Suppose people want to wait “in line”

- but there is a row of five seats to sit in.
- It makes sense to fill in the seats from left to right,
- but when the first person gets served,
- does everyone really need to move over?

Four people are waiting:

- 0:Victor 1:Irina 2:Parul 3:Joe 4:null

Serve Victor and then Irina.

- For the sake of clarity, I will set those locations to null,
- but I don’t really have to.
- Actually, what I do is set first=2 and size=2.

People in Chairs, continued

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)
- Sam arrives

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)
- Sam arrives
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (first=4, size=5)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)
- Sam arrives
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (first=4, size=5)
- Song arrives (Uh oh!). NOW we have to buy more chairs!

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)
- Sam arrives
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (first=4, size=5)
- Song arrives (Uh oh!). NOW we have to buy more chairs!
- 0:Lance 1:Ana 2:Philip 3:Alex 4:Sam 5:Song 6:null 7:null 8:null 9:null (first=0, size=6)

People in Chairs, continued

- 0:null 1:null 2:Parul 3:Joe 4:null (first=2, size=2)
- Lance arrives.
- 0:null 1:null 2:Parul 3:Joe 4:Lance (first=2, size=3)
- Parul gets served.
- 0:null 1:null 2:null 3:Joe 4:Lance (first=3, size=2)
- Ana arrives. Where should she sit? Do we need to buy more chairs??
- 0:Ana 1:null 2:null 3:Joe 4:Lance (first=3, size=3)
- Philip arrives. Alex arrives.
- 0:Ana 1:Philip 2:Alex 3:Joe 4:Lance (first=3, size=5)
- Joe is served (whew!)
- 0:Ana 1:Philip 2:Alex 3:null 4:Lance (first=4,size=5)
- Sam arrives
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (first=4, size=5)
- Song arrives (Uh oh!). NOW we have to buy more chairs!
- 0:Lance 1:Ana 2:Philip 3:Alex 4:Sam 5:Song 6:null 7:null 8:null 9:null (first=0, size=6)
- Notice that we take the opportunity to put the first person in chair 0.

ArrayQueue Implementation

ArrayQueue Implementation

ArrayQueue

ArrayQueue Implementation

ArrayQueue

- **first** index

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?
- (first $+$ size) % length

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?
- (first $+$ size) % length
- (3 $+$ 3) % 5 = 1

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?
- (first $+$ size) % length
- (3 $+$ 3) % 5 = 1
- The mod (%) operator effectively subtracts length until the result is less than length.

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?
- (first $+$ size) % length
- (3 $+$ 3) % 5 = 1
- The mod (%) operator effectively subtracts length until the result is less than length.
- We will always mod by length whenever we access the array. That makes the array "circular".

ArrayQueue Implementation

ArrayQueue

- **first** index
- **size** (number of elements in the queue).

Assuming size $<$ length, where does the next person go?

- first $+$ size? Not always.
- 0:Ana 1:null 2:null 3:Sam 4:Lance (first=3, size=3)
- first $+$ size $=$ 6, but that's out of range
- What formula "wraps around" when it is "out of range"?
- (first $+$ size) % length
- (3 $+$ 3) % 5 = 1
- The mod (%) operator effectively subtracts length until the result is less than length.
- We will always mod by length whenever we access the array. That makes the array "circular".
- Also known as a RING BUFFER.

Printing a Queue

Printing a Queue

In order to implement MaintainQueue,

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

We know a Queue is like a List,

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

We know a Queue is like a List,

- so maybe we can just use get(index). But this is no good.

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

We know a Queue is like a List,

- so maybe we can just use get(index). But this is no good.
- A get on a linked list implementation of a Queue would take O(n) each time.

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

We know a Queue is like a List,

- so maybe we can just use get(index). But this is no good.
- A get on a linked list implementation of a Queue would take O(n) each time.
- We need some way to keep track of where we are in the Queue

Printing a Queue

In order to implement MaintainQueue,

- we need to be able to visit all the elements of the Queue
- but leave it the same afterwards.
- We could use a second Queue, like we did for a Stack,
- but there is a better way that is a standard technique in Java.

We know a Queue is like a List,

- so maybe we can just use get(index). But this is no good.
- A get on a linked list implementation of a Queue would take O(n) each time.
- We need some way to keep track of where we are in the Queue
- that does not depend on the implementation.

Iterator

That way is the Iterator.

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

```
void print (Queue<String> queue) {
  Iterator iterator = queue.iterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
}
```

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

```
void print (Queue<String> queue) {
  Iterator iterator = queue.iterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
}
```

This is a little cumbersome still, so Java makes it easy.

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

```
void print (Queue<String> queue) {
  Iterator iterator = queue.iterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
}
```

This is a little cumbersome still, so Java makes it easy.
The following code is equivalent

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

```
void print (Queue<String> queue) {
   Iterator iterator = queue.iterator();
   while (iterator.hasNext())
     System.out.println(iterator.next());
}
```

This is a little cumbersome still, so Java makes it easy.
The following code is equivalent

```
void print (Queue<String> queue) {
   for (String string : queue)
     System.out.println(string);
}
```

Iterator

That way is the Iterator.

- **hasNext** tells you if there are any more elements.
- **next** gets the next element and moves you one forward.

```
void print (Queue<String> queue) {
  Iterator iterator = queue.iterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
}
```

This is a little cumbersome still, so Java makes it easy.
The following code is equivalent

```
void print (Queue<String> queue) {
  for (String string : queue)
    System.out.println(string);
}
```

It's a new kind of for-loop!

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.
This won't work because it doesn't wrap around.

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.
This won't work because it doesn't wrap around. How can you fix it?

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.
This won't work because it doesn't wrap around. How can you fix it?

LinkedQueue:

```
for (Node node = ?; node != ?; node = ?)
  System.out.println(node.element);
```

n is the current node.

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.
This won't work because it doesn't wrap around. How can you fix it?

LinkedQueue:

```
for (Node node = ?; node != ?; node = ?)
  System.out.println(node.element);
```

n is the current node.

Outside the queue, you can't do these because these variables are private or protected.

Implementing Iterator

First think how you would iterate over each type of queue if you didn't have an iterator.
ArrayQueue:

```
for (int count = ?; count < ?; count++)
  System.out.println(theElements[first + count]);
```

count is the number of elements we have printed so far.
This won't work because it doesn't wrap around. How can you fix it?

LinkedQueue:

```
for (Node node = ?; node != ?; node = ?)
  System.out.println(node.element);
```

n is the current node.

Outside the queue, you can't do these because these variables are private or protected.
The Iterator "deconstructs" these for-loops.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

- just need to keep track of the current Node node you are about to look at.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

- just need to keep track of the current Node node you are about to look at.
- Initialize node to **top**.

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

- just need to keep track of the current Node node you are about to look at.
- Initialize node to **top**.
- What does **next()** return?

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

- just need to keep track of the current Node node you are about to look at.
- Initialize node to **top**.
- What does **next()** return?
- What change does it make to node?

Implementing Iterator

To implement iterator(), you need an inner class which implements Iterator.

ArrayQueue

- Store a count of the number of elements returned from **next()**.
- Initialize it to zero.
- What does **next()** return?
- What change does it make to count?
- When does **hasNext()** return true? false?

LinkedQueue

- just need to keep track of the current Node node you are about to look at.
- Initialize node to **top**.
- What does **next()** return?
- What change does it make to node?
- When does **hasNext()** return true? false?

Summary

Summary

A Queue is a Java interface.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first $+$ size) % length.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to "wrap around": (first $+$ size) % length.
- When size$=$length, reallocate and start from zero if adding (offer).

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first + size) % length.
- When size=length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first + size) % length.
- When size=length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

- Iterator has **next** and **hasNext**.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first + size) % length.
- When size=length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

- Iterator has **next** and **hasNext**.
- Use a new for-loop.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first $+$ size) % length.
- When size$=$length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

- Iterator has **next** and **hasNext**.
- Use a new for-loop.
- Implement iterator() by returning an implementation of Iterator.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to "wrap around": (first $+$ size) % length.
- When size$=$length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

- Iterator has **next** and **hasNext**.
- Use a new for-loop.
- Implement iterator() by returning an implementation of Iterator.
- LinkedQueue implementation keeps track of current node.

Summary

A Queue is a Java interface.

- Usually stores items FIFO (first in first out). (Stack is LIFO.)
- Queue operations are **offer**, **poll**, **peek**, and **size**.
- Singly linked list implementation stores first at **top** and last in **bottom**.
- Array implementation uses **first** index and **size**.
- Use mod (%) to “wrap around”: (first $+$ size) % length.
- When size$=$length, reallocate and start from zero if adding (offer).

Visit all the elements using a Iterator.

- Iterator has **next** and **hasNext**.
- Use a new for-loop.
- Implement iterator() by returning an implementation of Iterator.
- LinkedQueue implementation keeps track of current node.
- ArrayQueue implementation keeps track of offset from first.