Queue and Iterator

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2018

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

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 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

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 the **last** too.
- **front** is the first one in line. **back** is the last.
- I'll draw a diagram, but it should be pretty obvious.

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

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 front=2 and back=3.

People in Chairs, continued

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

ArrayQueue Implementation

ArrayQueue

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

Why **size**?

- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=4, back=3)
- Lance leaves, but we don't set to null.
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=0, back=3)
- Ana leaves.
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=1, back=3)
- Philip leaves.
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=2, back=3)
- Alex leaves.
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=3, back=3)
- Sam leaves.
- 0:Ana 1:Philip 2:Alex 3:Sam 4:Lance (front=4, back=3)

Same front and back! But now it's empty!!

Storing null

Does

- 0:null 1:null 2:null 3:null 4:null (front=4, back=3)

mean completely empty?

Or completely full of nulls?

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.

- **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

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

LinkedQueue

- just need to keep track of the next element you haven't looked at yet.
- Initialize to **front**.
- What does **next** return?
- What change does it make front?
- What does **hasNext** return?

ArrayQueue

- Store index of next element,
- plus a count of the number of elements you have returned.
- Why?
- Otherwise if the seats are all full,
- you won't know when to stop!

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 **front** and last at **back**.
- Array implementation uses **front** index and **back** index.
- If **back** $<$ **front**, that means elements go from **front** to length-1 and 0 to **back**.
- Reallocate and start from zero if adding (offer) when size$=$length.

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 next node.
- ArrayQueue implementation keeps track of next index and count.