Queues implemented using arrays in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment, I look at the abstract data type queue. Previously in an assignment I have implemented one in a (doubly) linked list and saw that it worked well with a linked implementation having O(1) time complexity for both the enqueue and dequeue operations. Now I will implement one in an array and see how it compares.

Implementation

The idea here is that we represent the start and end of the queue with two indices called `first` and `last`. `First` points to the head of the queue which is the element that a `dequeue` returns, and `last` points to the first empty position where an `enqueue` can place a new item.

complications

The problem, of course, is that arrays have fix sizes, so we need to decide what happens if we run out of space. In this case, we have already placed n items in the queue, so `last`=n points outside the array. Writing there would throw an ArrayIndexOutOfBoundsException in Java, so we either need to

1. let the program crash
2. catch it and do something about
3. or prevent it by checking that we are inside the array before attempting a write.

Needless to say, the first option is not really an option, at least not a good one. So if we were to go for the second one, we could catch the exception, double the size of the array, copy everything over and attempt the enqueue

again. Afterwards the program can continue running regularly. This is similar to how we implemented the dynamic stack in Assignment 1. However, now that we read from the other end, it is possible, that even though we reached the end of the array, the queue is not actually full. The problem with this, is that with this implementation, after a while we would have an excessively large array while only using the last few positions.
To counter this, we naturally might come up with the idea, that before lengthening an array, we should make use of the place we already have. One way to do this, would be to take a loop that goes from `first` to `last` and start copying them one-by-one to the beginning of the array, update `first` and `last` and keep working in the same array. The nice thing about this, is that we can do it in place, it is pretty simple, and we don't end up with too much unused space and we only need to reallocate an array if it is actually full. It doesn't sound too bad, though there are still a lot of copying involved.

circular queue

An even better way to do this, is to just wrap around. This way, instead of moving everything to the beginning of the array when we reach the end, we will just continue placing items to the front as long as there is space. The implementation of this takes a little thinking (and a lot of visualisation), but we can come up with the idea to use `last` mod `n` to find the position we want to write to (and `first` mod n̂ for reading). This way we need to make sure that the queue is not full before writing. When `first` and `last` are equal, we know something needs to happen, but we don't know if the queue is full or empty. There are once again several ways to solve this, e.g.

1. declaring one position always empty, so that (last+1)=first only when the array is full
2. having a `count` variable to keep track of the number of elements in the queue.

I decided to go with the latter one as it is easier to implement. It starts as 0 and is incremented with each successful enqueue and decremented with each dequeue. This way, all I need to check before writing is if `count` is equal to the length of the array.
When I find that the array is full, I still need to double the size, and copy my elements over from `first` to `last`. I take this opportunity to "unwrap" my queue (have the first element be at position 0) and decrease the value of `first` and `last` to 0 and `last-first`, respectively.

my code

For lengthening the array, I wrote the following code:

```
public void lengthen(){
    int[] new_q = new int[2*this.q.length];
    for (int i = first; i<last; i++){
        new_q[i-first]=q[i%this.q.length];
    }
    last=last-first;
    first=0;
    q=new_q;
}
```

After this, the first element in the queue is at position 0 and the last is (`last-first`) steps towards the end. In the last step I update the reference to the array stored in my queue object. This method is called from enqueue if the array is full:

```
public void enqueue(int item){
    if(this.count==this.q.length){
        lengthen();
    }
    this.q[last%this.q.length]=item;
    last++;
    count++;
}
```

As you can see, after lengthening the array, the rest of the code still executes and places the item in the correct position using the new array and the new value for `last`.
Similarly in the dequeue I need to take care of shrinking the array. I decided to decrease its size by half only when the queues length is 1/3 the arrays size.

```
public Integer dequeue(){
    if(count<this.q.length/3&&this.q.length>=6){
        shrink();
    }
    if (first!=last){
        int r = this.q[first%this.q.length];
        first++;
        count--;
        return r;
    }
    else return null;
}
```

where `shrink()` is the same as `lengthen()` except for the new size. It would be more elegant to have these two as one `resize()` method, but I added shrinking later and forgot to change it.

testing

For testing purposes, I added two different print functions, one of which prints the entire array from index 0 to length, and another that prints the queue in order (so "unwrapped", without the empty positions). This way I was able to find and correct my mistakes and confirm that the methods work as intended.

```
public void print(){
    for (int i=0; i<this.q.length; i++){
        System.out.print(this.q[i]+",");
    }
    System.out.println("first: "+this.first+" last: "+last+"
    count: "+count);
}

public void print_queue(){for (int i=this.first%this.q.length;
    i<this.first%this.q.length+count; i++){
    System.out.print(this.q[i%this.q.length]+",");
    }
    System.out.println("first: "+this.first+" last: "+last+
    " count: "+count);
}
```

Comparison

The implementation itself is definitely more complicated compared to the linked list version. The enqueue and dequeue operations usually have a time complexity of O(1), however there is the occasional penalty with the resizing that have O(n) time complexity, but all in all, an amortized cost of O(1). Space complexity is where the array implementation stands out, as it stores only the items themselves and the two integers for `first` and `last` (+ our `count` variable, but that is negligible).

Benchmarks

I decided to run similar benchmarks on the array and linked implementation, to see how they compare.

enqueue

First, I started with an empty queue and added n elements without any dequeues. For the array implementation, I set the length=100. This way, for n=100 there were no resize penalties, for n=200, there was 1, and so on. My measurements are in the table below:

queue size (n)	array version [µs]	linked version [µs]
100	0.2	0.5
200	0.7	1.2
400	1.6	3
800	3	6
1600	5	13
3200	10	17
6400	20	40
12800	40	80

As you can see, the array version dominates. But what would happen, if I started with only length=10? Or length=2?

n	length=100	length=10	length=2
100	0.2	0.7	0.6
200	0.7	1.4	1.3
400	1.6	2.1	2
800	3	4.5	4
1600	5	6	5.5
3200	10	12	11
6400	20	26.5	24
12800	40	55	50

`Enqueue()` execution times with the different starting lengths

Interestingly, I found that starting with a length of 2 is actually better than starting with 10 elements, even though there are more resizing. But still, even an array of size 10 is more efficient than the linked implementation was.

dequeue

Next I performed n dequeue operations (on a queue of size n).

queue size (n)	array version [µs]	linked version [µs]
100	0.3	0.3
200	0.8	0.5
400	1	1.2
800	2	1.5
1600	4	3
3200	8	8
6400	20	11
12800	40	20

Here, in the array implementation there are also penalties associated with resizing, since I decrease the length of the array when the utilised length is less than third of the arrays length, so for example for n=100, this will occur 5 times, which is obviously more resizing than what we had for measuring `enqueue()`. Also remember, that in the linked list implementation, I found that the `dequeue()` operation was about $1/4^{\text{th}}$ of the `enqueue()`. These together mean that for n `dequeue()` operations, the linked implementation is more efficient.
Now the natural question to ask, is how the two would perform in a more general setting, where both `enqueue()`-s and `dequeue()`-s occur frequently.

more real-life test

To get an idea of the general performance of both queues, I decided to set up an array of length n with a random bit sequence that decides whether to perform an `enqueue()` or a `dequeue` operation (0=enqueue, 1=dequeue).

queue size (n)	array version [µs]	linked version [µs]
100	0.5	0.5
200	1.5	1.7
400	3	3.5
800	4.5	4
1600	8.5	9
3200	16	16
6400	32	32.5
12800	64	65

Here, you can see that the two are pretty close, however, the array implementation seems to take the lead. I also confirmed this by testing it for n=1,000,000 for which I got 5.5 ms with the array implementation and 4.8 ms with the linked.

Summary

In this assignment, I carefully implemented a circular queue in an array and compared it with my previous implementation of the same queue in a doubly linked list. I found that the array implementation usually performs better.