Queue based on Arrays with Wrap Around in Java

Malte Berg - ID1021

HT 2024

Introduction

This report covers the implementation and explanation of a dynamically sized queue using arrays in Java as well as problems and solutions that arises, an assignment for a higher letter grade in the course ID1021.

Background

While a queue can be easily implemented with a linked list using nodes, it is not the most memory efficient as the queue grows. Using an array to store items in a queue is a better idea, but also a more complex idea. There are a few cases that will lead to problems, especially due to the nature of an array being static in size and not able to add new items beyond the bounds of it, instead having to copy over the items to a bigger array every time one reaches the end of the array, which is costly in time, but necessary to hold more items.

Another problem that arises is when dequeuing an item from the queue. The first item of the array is removed, and to take use of the now created hole in the start, an idea is to shift all the items in the array one step so one can continue to add items at the end. This might be fine for smaller arrays, but is a very costly operation as the queue grows longer.

The solution that needs to be found is a way to utilize all the available spots in the array before increasing the size, and to not unnecessarily shift the values in the array.

A Solution and an Implementation

The solution for this problem is to create a queue that does a "wrap around" of the array. For this to be able to work, a few properties have to be added to a regular queue class that will be brought up in the subsection for implementation.

What does wrap around mean?

When dequeuing items from the queue, a gap is created in the beginning of the array. When enqueing items at the end of the queue, the last index of the array will finally be reached, but if there are spaces left at the start it is wasteful to not use these before creating a larger array. The wrap around happens when the index for the first free spot is not increased to beyond the bounds of the array but instead set back to zero, or incremented by 1, and modulo the maximum size of the array, thereby wrapping around to the beginning again to take use of the empty spots.

Code and Explanation

```
1     public class QueueWrap {
2         Integer[] queueArr;
3         int maxSize;
4         int firstItem;
5         int firstFree;
6         int length;
7
8         public QueueWrap(int initialSize) {
9             maxSize = initialSize;
10            queueArr = new Integer[maxSize];
11            firstItem = 0;
12            firstFree = 0;
13            length = 0;
14        }
```

With the constructor of the `QueueWrap` class, the properties needed for the wrap around to function are initialized. The constructor first takes in an argument of the initial size the queue array `queueArr` should have, and initializes the array with that size. The index of first item, `firstItem`, of the queue and the index of the first free spot, `firstFree` are both set to 0 at the start, but will change later when items are enqueued. The number of items in the queue, the `length`, is of course also set to 0 as no items have yet been added.

One might spot the use of `Integer` instead of `int`, the reason for this is to be able to set an empty spot back to `null` when array copies will happen to make sure the garbage collector in Java does not have to make a copy of an already dequeued item.

Before moving on to the methods `enqueue()` and `dequeue()`, take a look at two help functions that were implemented to the class to make the main methods easier to read.

```
32      private int increment(int toIncrement) {
33          return (toIncrement + 1) % maxSize;
34      }
35
36      private void increaseSize() {
37          int newMaxSize = maxSize * 2;
38          Integer[] increasedQueueArr = new Integer[newMaxSize];
39
40          for (int i = 0; i < length; i++)
41              increasedQueueArr[i] = queueArr[(firstItem + i) % maxSize];
42
43          firstItem = 0;
44          firstFree = length;
45          queueArr = increasedQueueArr;
46          maxSize = newMaxSize;
47      }
```

The help method `increment()` returns a correctly incremented value of a given index, `toIncrement`. When passing either the `firstItem` or `firstFree`, it is incremented by 1 and then modulo `maxSize` to make it wrap around to the start of the queue array.

The help method `increaseSize()` is called on in the method `enqueue` when the queue is full and needs to be increased. The array size is doubled, saving the size in a variable `newMaxSize`, and a new array `increasedQueueArr` of that new size is created. The items in the current queue `queueArr` can then be copied over to the new array starting at index 0. To preserve the order of the current queue, the index from where the loop starts copying over the items from is `(firstItem + i) % maxSize` *(take notice that the old* `maxSize` *is used here as this is the size of the array that is being copied from)*, with the modulo helping with the wrap around. The loop is ran as many times as the value of `length`, as that is exactly how many items are in the queue.

An example of the loop: A full queue with `maxSize` of 4, and index `firstItem` at 3, the loop's accesses to the queue looks as follows:

`(firstItem + i) % maxSize` = `(3 + 0) % 4 = 3` → `queueArr[3]`
`(firstItem + i) % maxSize` = `(3 + 1) % 4 = 0` → `queueArr[0]`
`(firstItem + i) % maxSize` = `(3 + 2) % 4 = 1` → `queueArr[1]`
`(firstItem + i) % maxSize` = `(3 + 3) % 4 = 2` → `queueArr[2]`

And the queue preserves its order. After this, the class variable `firstItem` is set to 0 to point at the first item in the new queue, the variable `firstFree` is set to `length` as that is the index of the array cell after the last item of the queue, `queueArr` is updated to the `increasedQueueArr`, getting rid of

any reference to the old queue as well, and the `maxSize` is set to its new doubled value `newMaxSize`.

```
16      public void enqueue(Integer item) {
17          if (length == maxSize) increaseSize();
18          queueArr[firstFree] = item;
19          firstFree = increment(firstFree);
20          length++;
21      }
22
23      public Integer dequeue() {
24          if (length == 0) return null;
25          Integer dequeued = queueArr[firstItem];
26          queueArr[firstItem] = null;
27          firstItem = increment(firstItem);
28          length--;
29          return dequeued;
30      }
```

Taking a look at the method `enqueue()`, it can be called on with an argument `Integer item`, providing an integer to place in the array. It is first checked if the `length` of the queue is equal to the `maxSize` of the queue. If true, the previosuly mentioned help method `increaseSize()` is called on to make room before adding a new item. It then continues by adding the `item` to the index of the next empty spot `firstFree`. That index is then appropriately updated with the help method `increment()` to make sure the queue does a wrap around. Finally, the `length` of the queue is incremented by 1 in a regular fashion as the variable only keeps track of the number of items in the list and is not affected by any wrap around.

Taking a look at the method `dequeue()`, it is called on with no argument but returns an integer. If the queue is empty, i.e. `length` is equal to 0, `null` is returned. One could instead choose to throw an exception, but for this implementation a null-return was chosen instead. If the queue is not empty, the return variable `dequeued` is set to the first item in the queue, and `null` is set at the index where the item was in the queue to avoid keeping the reference. Instead of incrementing `firstFree` like in the previoud method, the index `firstItem` is instead correctly incremented to the next item in the queue. Before the collected item `dequeued` is returned, the `length` is decremented by 1 in a regular fashion as well just like in `enqueue()`.

With this implementation, the program can be made more memory efficient compared to a queue based on a linked list as the need for every node to keep a reference to the next node in the sequence is removed and an array can be used instead. The array only grows when all possible spots have been filled which removes both unnecessary shifting and expanding.