Queue Using an Array

Emanuel Paraschiv

Fall 2024

Introduction

In the past assignment we implemented a queue using linked lists. Even though that way seems very versatile and one may not understand the importance of the array implementation, there are some things to consider.

For example, the linked list approach can pay an expensive price depending on memory allocation. This is a good reason to explore other options.

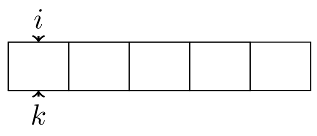
Queue using Arrays

We will try to implement a queue using an array, and therefore need a new constructor for this. Here `n` represents the capacity of the array, `i` is the pointer to the first element and `k` points to the next free sport in the queue:

```
public QueueArray(int capacity)
    queue = new int[capacity];
    i = 0;
    k = 0;
    n = capacity;
```

This gives as a good start. Now we however have to understand the behavior of such a data structure;

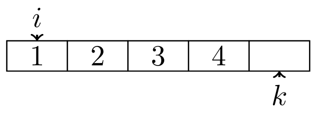
Let us take a practical example, where the user creates a queue using an array of capacity 5. When we first create such an object, both pointers `i` and `k` will point to the first cell, 0:



A starting point for adding elements can be a method such as:

```
public void enqueue(int val){
    queue[k]= val;
    k++;
```

If we add an element at that position (enqueue), then k should be incremented by one. Let us enqueue the next numbers: 1, 2, 3 and 4. Now the queue should look something like this:



There are some things to consider in this case. If we are to add one more element, then the array will be full and k will point out of bounds. We don't want that to happen, but what can we really do to prevent it?

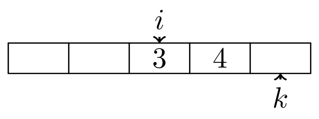
One possible solution can be to keep track of this situation and create a bigger array, maybe one to be double the size, as we did in a previous experiment involving a dynamic stack. This option seems the best and the most intuitive. However, it can be very inefficient if we think about the wasted memory.

Let us take the following scenario: Suppose that the elements of these queue can also be `dequeued`. By 'dequeuing' we mean 'remove elements from the beginning'. For creating a method to perform this, firstly we would like to now if the queue is already empty, case in which we will notify the user. Do we have to technically remove the values inside each cell? The answer is not really, but there are some advantages from doing that as we don't want to keep references to invalid spots:

```
public int dequeue()
    if(i==k)
        throw new NoSuchElementException("No elements to dequeue")
    int val=queue[i];
    queue[i]=0;
    i++;
    return val;
```

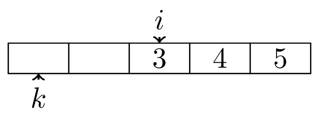
This implementation is however not the greatest. Notice the condition in the if statement: `i` can be equal to `k` both when the array is completely empty but also when it is completely filled (as we will see later), so how can we tell the difference? One idea I came up with was to create a variable `size` and initialize it in the constructor. If `size` is equal to `n`, then we know it is full. If it is 0, then the array must be empty. In the beginning, `size` is initialized to 0. We increment it when we `enqueue` and decrement it when we `dequeue`.

If we dequeue the first 2 elements, our queue will look like this:

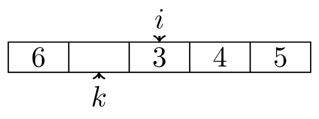


Now, we can take advantage of this situation. Before we were talking about crating a larger array every time we need to enqueue one element in a full queue, but now we can build up on this specific case: We have an array able to fit 5 elements and we only use 2 cells. We want to add the third one, so what if we use the free slots after?

We will do this starting from the beginning of the array, as the queue is a FIFO data structure and all elements are removed in order. Thus, in the `enqueue()` method that we are going to modify and enhance, instead of incrementing `k` by 1 and risking an overflow, we will increment it with respect to % n. In our case, `k` will be able to be maximum 4. If it is on position 4 (as currently) and we try to enqueue one more element, let's say the value 5, then `k` would be incremented to position 0 (because k= 5 % 5 =0):



So far everything is likely to work great, but we have to consider one more case. If we enqueue more elements than we dequeue, at some point `k` will catch up on `i` from behind.

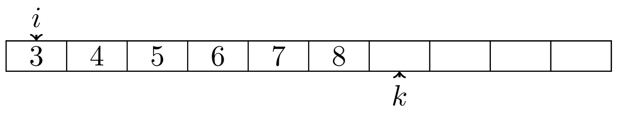


We can enqueue one more element here, but what are we going to do after? With a full queue, the only thing we can do is create a bigger one. One way to be aware of this change will be to check is both i and k point to the same cell, nut also if the size is equal to n this time (to not mistake it with an empty queue). Therefore, if `k` is exactly before `i` it means that at the next enqueue `k` will pass over `i`. To avoid we can call a method to resize the queue.

Let us call it `change()` for simplicity. What we do is create an array double in size to store the queue elements:

```
public void change(){
    // create new array
    int[] newArr=new int[n*2];
    //copy
    for(int j=0;j<size;j++){
        newArr[j]=queue[i];
        i=(i+1)%n;}
    // update
    queue=newArr;
    n=n*2;
    i=0;
    k=size;}
```

Let us come back to our practical example. Now if we enqueue 2 more elements, one to fill the array and one more to activate the exception case (ex. 7 and 8), we will activate this procedure and our new queue will be:



After exploring all these cases, we have everything we need to complete our `enqueue()` and `dequeue()` methods.

First the `enqueue()` method will check if it is the case for a change of size. If not, then the element passed will we enqueued to position `k` and `k` will be incremented one position with respect to `mod n`:

```
public void enqueue(int val){
    if (i ==k && size==n){
        change();}
    queue[k] = val;
    size++;
    k = (++k) % n;}
```

Similarly, our `dequeue()` method will take care of the special cases. First it will check if the queue is empty, case in which it will throw an exception to the user. If it is not the case, then we will return the dequeued value, nullify the removed element as discussed previously, and ultimately increment `i` with respect to `mod n`. This way, the next element in the queue will become the first:

```
public int dequeue() {
    if (i==k && size==0){
        throw new NoSuchElementException("No elements to dequeue");}
    int val = queue[i];
    queue[i] = 0;
    i = (++i) % n;
    size--;
    return val;}
```

The time complexity for both of these operations in constant, $\mathcal{O}(1)$. However, when we need to perform a resize operation, the time complexity becomes $\mathcal{O}(n)$ for `enqueue()` as we need to copy `n` elements to a new array.

Conclusion

Even though this implementation can be tricky compared to the linked lists one, it comes with advantages.

The memory usage is more efficient, elements are accessed much faster and there is no overhead caused by references to other nodes. On the other hand, some disadvantages include the fixed size of the array and the need for resizing when we are to go out of bounds.

One should therefore pay close attention when choosing the best implementation for their queue depending on the specific application.