Assignment 7 Report - Queue in an Array - Java

Dean Tsankov

October 5, 2024

Introduction

In this report I will present an alternative way to implement the Queue data structure. Instead of using a linked list as an underlying structure here I will instead rely on an array. The reason we might want to not use linked lists depends on the cost of memory allocation. With optimizing this property we introduce the fact that the implementation itself becomes a bit more complicated.

General Idea

The main way this queue is intended to work is the following: The queue has a property which is the array holding the data and pointer values indicating the indices of the first element in the queue and the one after the last (the next possible position for enqueue-ing). When we add an element to the queue we write it at the current `last` index and increment it. Conversely when removing an element we delete the data in the array at the `first` index and increment it as well.

The complications arise when we reach a `last` value equal to the array length. At that point it is most probably the case that there are still a couple of empty slots before the `first`. We would like to utilize this space and not mindlessly expand the array disregarding this optimisation. This will be done by, in essence, wrapping the `last` index back to the beginning. The way to do this is by incrementing that index not only by 1 but then also taking its modulo with regard to the current size of the array.

That sort of thinking allows us to fully use the current size of the array, but there will come a time when the whole array will in fact be filled. In order for us to reference this it would be useful to introduce a new property into the structure, keeping track of the actual elements in the queue. The work we do with this variable is fairly self explanatory - its increased when enqueue-ing and decreased when dequeue-ing.

With the current wrap-around setup we will know that we are in an edge case if the first and last indices have the same value. This happens either when there are no elements in the queue, i.e we removed the last element or we are trying to add the first element, these cases should be accounted for with simple checks. The more important case is the one where we are trying to override the first value with a new element. This is when the dynamic size increase should happen. That is we transfer the data to a twice as large array. During this transfer we do not need to preserve the wrap-around order and so we can "straighten" the elements.

Essentially that is the idea of the array implementation of a queue.

The Code

This is how the class is defined and initiated by the constructor:

```
(...)
public class ArrayQueue<T> extends JQueue<T> {

private T[] queue;
private int first; //index of first element in queue
private int last; //index of last element in queue
private int size; //number of elements in queue
public Class<T> clazz; //allows this class to use a generic array

public ArrayQueue(Class<T> clazz) {
    this.queue = (T[]) Array.newInstance(clazz, 4);
    first = -1;
    last = -1;
    size = 0;
    this.clazz = clazz;
}
(...)
```

The first and last indices are set to -1 on initiation so as to signal that the queue is empty. The class itself extends an abstract Queue class which ensures the abstraction of this implementation.

Next is the enqueue function, the most involved part of the structure:

```
(...)
public void enqueue(T item) {
    if (size == 0) {
        first = 0;
        last = 0;
```

```
        }
        if(last == first && size!=0){
            int n = queue.length*2;
            int tempfirst = first;
            T[] tempQueue = (T[]) Array.newInstance(clazz, n);
            for (int i = 0; i < size; i++) {
                tempQueue[i] = queue[tempfirst];
                tempfirst = (tempfirst + 1) % queue.length;
            }
            queue = tempQueue;
            first = 0;
            last = size;
        }
        queue[last] = item;
        last = (last + 1) % queue.length;
        size++;

    }
    (...)
```

It generally follows the explanations given above. The first element case is handled by setting both indices to zero and later incrementing the `last`. There is the array expansion routine and finally the actual writing to the array.

Lastly is the dequeue method:

```
    (...)
    public T dequeue() {
        if (first == -1) {
            return null;
        }
        size--;
        T temp = queue[first];
        queue[first] = null;
        first = (first + 1) % queue.length;
        if (size == 0) {
            first = -1;
            last = -1;
        }
        return temp;
    }
    (...)
```

At first there is a check if the queue is already empty otherwise it pro-

ceeds to delete the correct data from the queue and increment the pointer. There also happens to be a peace keeping check if the removed element was the last in the queue and if so, then the indices get reset.

Benchmark

I decided to also present a comparison between this array queue and the linked list queue I had already done for the previous assignment. This is a graphical plot of the time with respect to the amount of elements being enqueued:

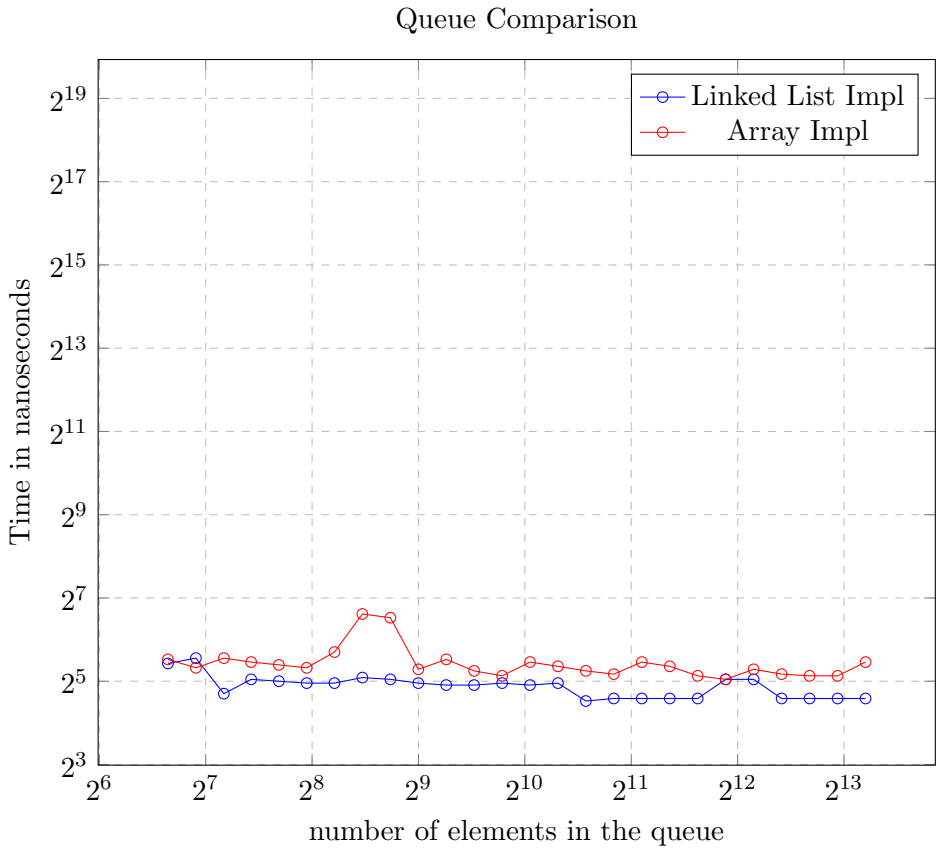

Figure 1: Queue Comparison

Conclusion

It is generally apparent that the array implementation requires more time, since it is expanding its size quite a few times while the linked list has a constant cost for adding elements. Even still the array implementation also

has a constant time complexity since its expand operations get amortised. The difference between the two alternatives will become apparent when we consider the memory accesses and the amount of cash misses the linked list yields each time it operates on the queue.