Queue in Java

Malte Berg - ID1021

HT 2024

Introduction

This report covers the sixth assignment assigned in the course ID1021. The assignment covers the implementation of a working queue, as well as an analysis of the time complexity for doing enqueue and dequeue operations with both a simple and a more complex implementation of the queue.

Background

A queue is a data structure built with a FIFO, first in first out, principle. Compared to a stack where the item added latest is the first one to be picked out, a queue preserves the order the items were added, just like a queue in real life. The implementation of a queue is quite similar to how the linked list was implemented in the previous assignment, yet differs in the methods of adding and removing an item.

Implementing a Simple Queue in Java

Just like the previous assigment, the structure builds on using "Nodes" to create a list which contains a reference to the integer value it's holding, the `item` as well as a reference to the next node in the sequence, `next`. The `Queue` class contains a variable `head` which is the first node in the sequence. Creating a `Queue` with no argument gives `head` a value `null`, but providing a size n creates that many nodes in a queue with the values 1 to n.

```
public Queue() {
    head = null;
}

public Queue(int n) {
    Node last = new Node(n, null);
    for (int i = n-1; i > 0; i--) {
        last = new Node(i, last);
    }
    head = last;
}
```

Next is the implementation of the methods `enqueue` and `dequeue`, making the queue functional.

```
    public void enqueue(Integer item) {
        if (head == null) head = new Node(item, null);
        else {
            Node current = head;
            while (current.next != null) current = current.next;
            current.next = new Node(item, null);
        }
    }

    public Integer dequeue() {
        if (head == null) return null;
        Integer val = head.item;
        head = head.next;
        return val;
    }
```

As one might notice when examining the method `enqueue`, the whole queue is traversed before being able to add a new node at the end. This might work for short queues, but as it grows longer, the time taken to enqueue will quickly grow. The method `dequeue` only performs a change at the first node of the queue, and can be suspected to not be affected by the length of the queue.

With this in mind, one can suspect that the time complexity for running the enqueue method should result in a linear time complexity. An improvement can be made.

Implementing a Modified Queue

To avoid the extra time taken to traverse the queue every time an item should be enqueued, a variable `end` should be a reference to the last node in the queue so when adding a new item to the queue, the reference `next` of `end` can immediately accessed. The updated constructors for the class now instead looks like the following.

```
    public QueueMod() {
        head = null;
        end = null;
```

```
}

public QueueMod(int n) {
    Node last = new Node(n, null);
    end = last;
    for (int i = n-1; i > 0; i--) {
        last = new Node(i, last);
    }
    head = last;
}
```

The method `enqueue` of course also needs to be updated accordingly to benefit from the added `end` reference.

```
public void enqueue(Integer item) {
    if (head == null) {
        head = new Node(item, null);
        end = head;
    }
    else {
        end.next = new Node(item, null);
        end = end.next;
    }
}
```

The first if-statement is very similar to the more simple implementation of the queue, with it adding a new node if the `head` is `null`, with an added line of code pointing the `end` to the `head` as well, being the only node in the queue. Whenever a new item is to be added and the list is not empty, i.e `head` is not equal to `null`, then `end.next` is set to a new node and `end` updated to the new last node. The benefit of this is of course the fact that the list does not have to be traversed when trying to add a new item, which in theory should result in a constant time complexity.

The dequeue method has been kept the same as the `end` only needs to be updated when enqueuing and as the method in theory already should be performing at a constant time complexity.

Results

As expected and showed in Figure 1, the optimized variant of the enqueue method performs with constant time complexity, $O(1)$, while the simple variant of the enqueue method has linear time complexity, $O(n)$, with n being the length of the queue. This shows a clear benefit of having an `end` reference in the class.

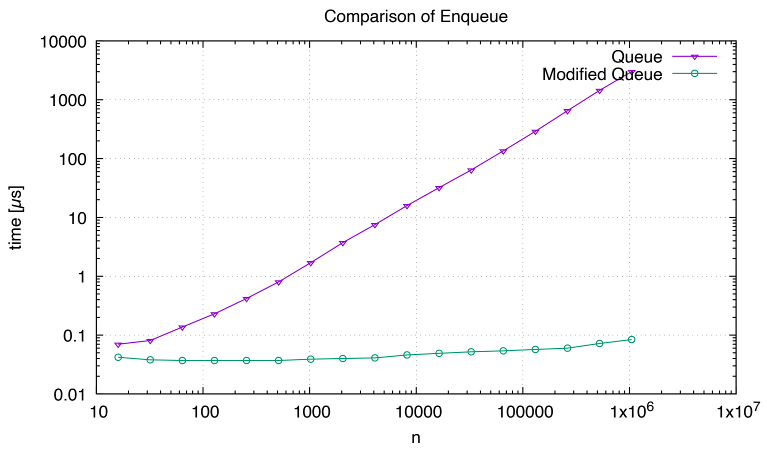

Figure 1: Runtime comparison between enqueue methods on queues of size n, in microseconds

Looking at Figure 2, another hypothesis is confirmed, that being the hypothesis of the dequeue method running at $O(1)$ time for both variants of the queue. The only small variance that can be seen in the graph can be traced to a non-ideal running of the program, and as the variance is so small it can be disregarded in this case.

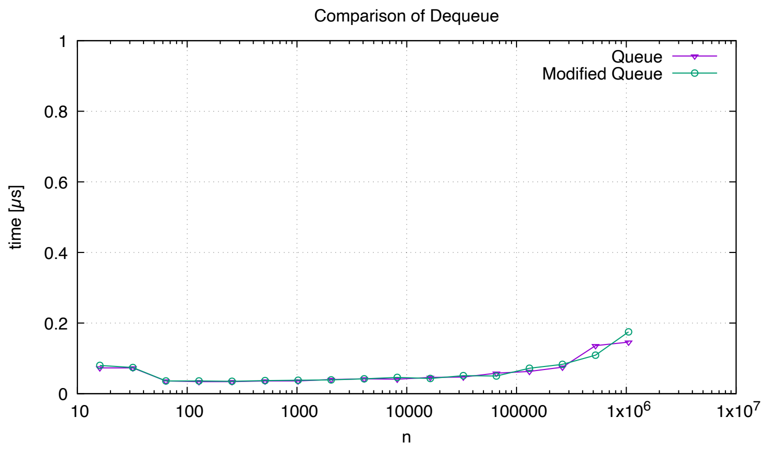

Figure 2: Runtime comparison between dequeue methods on queues of size n, in microseconds