Queues implemented using linked lists in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment I looked at the abstract data type called queues and implemented two queues with linked lists. A queue is a First In First Out datatype, so it enqueues (adds) new elements to the end while dequeues (removes) elements from the front, so it can make good use of linked data structures.

Implementation

My queue is represented with a linked list which contains a reference to the first node which will contain some data (an integer in this case) and some other reference to the rest of the list. It has two rather simple methods as shown in the code below.

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

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

```
    return r;
}
```

Since we write at the end of the list, we need to traverse the entire list first, so it has a time complexity of O(n). Reading, on the other hand happens in constant time, so we are not worried about that, but O(n) writing is generally not considered great. In most use-cases of queues, the ratio of enqueue to dequeue operations are more or less balanced, so that means the total handling of the queue is leaning more towards O(n) so it would be worth a little more thinking to improve writing.
We could of course switch it around and write to the `head` of the list while deleting from the `tail`, which is rather similar in terms of time complexity, except for if the use case suggests that not all elements written to the array will be read, in which case it is better to have writing be O(1) as it happens more often. Still, this is not satisfactory for general purposes, so let's see how else we can improve our queue.

Improved implementation

The idea is that in order to have both reading and writing available in constant time, we need to have direct access to both the first and the last node of the list. This also means we need to store two references in each node, one for the element before and one for the element after. It is a little bit trickier to implement since we need to keep track of two references in our methods instead of one.

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

For writing, we need to insert a node at the end of the list, so we look at the `tail` property. If it is `null`, then there is no item in the list, so we insert one that has both references set to null and both the `head` and the `tail` points to its address. Otherwise, we just need to instantiate a new node with its `prev` being the current `tail`, `next` being `null`, update the `tail` property of our list, and set the `next` field of the previous node to the new node's address.

```
public Integer dequeue(){
    if (this.head==null) return null;
```

```
    Integer r = this.head.item;
    if (this.head.next==null){
        this.head=this.tail=null;
    }
    else{
        this.head.next.prev=null;
        this.head=this.head.next;
    }
    return r;
}
```

For reading we look at the `head` of the list. If it is `null`, then there is nothing to read so we return `null` (we could also use exceptions, but that is beside the point of this assignment). If `head.next` is `null`, then there is only one element in the queue, so we set both `head` and `tail` to `null`. In all other cases, we need to set the `prev` field of the next node to `null` (since that will be the new first element) and the `head` property of the queue to the next elements address.

Comparison

In the second implementation we managed to get both the `enqueue()` and `dequeue()` methods to have a time complexity of O(1), though for small n it may be a little slower due to the added if statement. It is also a little harder to implement, and is easier to make mistakes, but of course that doesn't affect using it afterwards.
The real drawback however is the increase in memory complexity, from having to store an additional reference in each node. In theory, this may also come back and affect execution times too because of the memory overhead. I have tried to find some kind of indication of this in my benchmarks, but was unsuccessful.

Benchmarks

To see how using doubly linked lists improve the execution time, I set up a benchmark for adding an element to a queue already of size n with both implementations. I expect to see a linear trend in the first implementation and a constant execution time for the second. My measurements are in the table below:

array size (n)	regular linked list [ns]	doubly linked list [ns]
100	100	4
200	200	4
400	400	4
800	900	4
1600	1700	4
3200	3700	4

The measurements confirm the time complexities I calculated previously.
I also measured the dequeue operation, which in both implementations was around 1-2 ns, so consistently just a little bit less than enqueue. This makes sense, since the memory allocation when adding a new element takes up a little more time.

Summary

We have seen that an abstract data type like a queue can be implemented in many ways (another interesting one would be using arrays, which I cover in Assignment D) having significant differences in both time and space complexity. Though it makes sense to hide implementation details from the programmer, it is also sometimes necessary to provide these details.