Queues

Emanuel Paraschiv

Fall 2024

Introduction

The scope of this assignment is to implement a data structure that resembles a situation in everyday life: a queue. We will do this by taking advantage of the linked list structure.

The queue

A queue is a FIFO data structures. All elements are organized linearly and the first element that is `enqueued` (added to the queue) is the one to be `dequeued` (removed from the queue). To create such a structure with linked lists first we need to create individual nodes:

```
private class Node
    Integer item;
    Node next;
    private Node(Integer item, Node next)
        this.item = item;
        this.next = next;
```

With nodes we will store values and create references. This was the first step, now to create a real Queue we need to merge these nodes.

In the beginning, when we create a Queue object, the data structure will be empty:

```
public Queue()
    this.head = null;
```

As we can see, the implementation is straightforward. Elements of the queue are organized in individual nodes. The first element in the queue has the reference `head` and thus we can keep track of it when doing operations on this data structure.

In order to add elements to this queue, I created an enqueue method. This method starts by checking if the queue is empty, case in which it will simply make the head reference point to the new Node. Otherwise, it will iterate through all the nodes in the queue and simply add it at the last position, at the end of the queue.

```
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);
```

Now, if we need to delete an element we need to start from the beginning of the queue. We delete the elements only in the order that they were added. To do so, we have a `dequeue()` method that when called will remove the first node from the queue. It achieves that by simply moving the head pointer one position, eliminating thus the first node in the queue:

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

With this implementation we can draw a conclusion. The `dequeue()` method should be constant because all it does is changing the position of the head pointer, thus its time complexity should be $\mathcal{O}(1)$.

However, the `enqueue()` method has a limitation which we will address in the next section. We can only add elements to the end of the queue. In order to reach that end we have therefore to traverse the whole queue. As the size increases, so does the execution time. The expected time complexity is thus linear, $\mathcal{O}(n)$.

In order to prove that, I created a `benchmark()` method that checks the execution time for this operation. It creates a queue of different input sizes. We then call the `enqueue()` method to add a random set number (1 for the sake of simplicity) to these queues. We get the execution time by using the `System.nanoTime()` function:

```
private static long benchmarkEnqueue(int size)
    //*create queues of different sizes*
    long t0 = System.nanoTime();
    queue.enqueue(1);
    long t1 = System.nanoTime();
    return t1 - t0;
```

The results are plotted in the graph below:

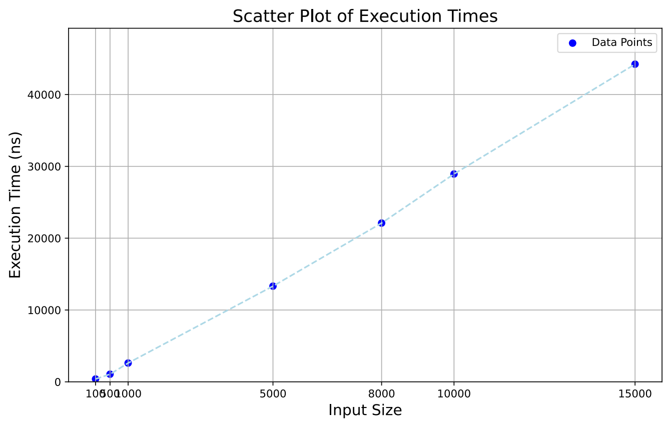

Figure 1: Enqueue execution time

The Improved Queue

With one simple trick we can make the enqueue method to have a constant execution time, and also make the information more organized and easy to follow.

Currently our queue consists of individual nodes and one `head` reference that keeps track of the first element. A great idea is to create a new reference, let's call it `tail`, that will keep track of the last element in the queue.

Even though this doesn't seem like a big change, the improvement will be visible. For example, we can change the way that our `enqueue()` method operates: instead of traversing all the queue from the beginning, now we can simply add it at the end without the need of iterating.

```
public void enqueue(Integer item)
    Node newNode = new Node(item, null);
    if (tail == null)
        head = newNode;
        tail = newNode;
     else
        tail.next = newNode;
        tail = newNode;
```

This new enqueue method works as follows: first we check if the queue is empty, case in which we make both references point to this new node. Otherwise we simply add it at the end using the tail reference and update it accordingly.

As we can see from the following graph, the execution time is constant as expected:

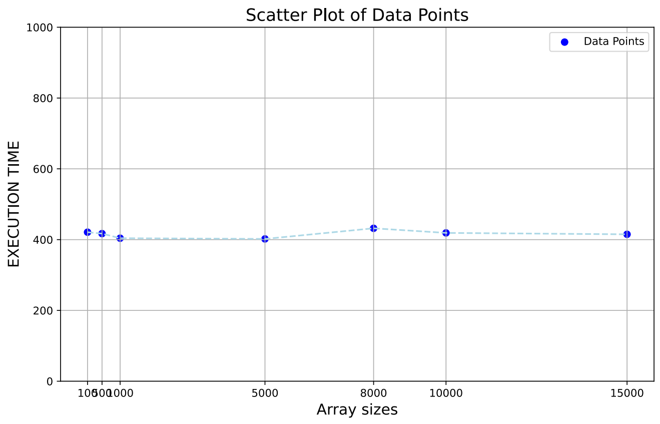

Figure 2: Enqueue execution time

Also, the `dequeue()` method has the same constant time complexity $\mathcal{O}(1)$ as before.

Conclusion

Therefore we can conclude that the queue is a great data structure for keeping track of the order of adding elements. The later implementation offers an enhanced and more organized structure to it, helping us deal with it easier.