Queues

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open once again, a window on a different way of storing data sets, namely under the form of a queue. While in this paper, the focus will be oriented more towards the different implementations of this data structure, it will also explore a practical example of a way of using it, namely through the *breadth first traversal* of a binary tree.

Implementing the different queues

1. First implementation: The queue under the form of a linked list

The implementation of the queue under the form of a linked list is pretty similar to the stack in terms of its structure. It consists of a pointer to its *head* and two main methods that serve its functionality, namely *add* and *remove*. Similar to previous assignments, it should be taken into consideration that apart from these methods, others have also been implemented, which helped in terms of debugging (i.e.: length() and asArray()). However, unlike the stack, this time improvements will be implemented in order for the queue to be even more efficient, namely by turning the add operation from an *O(n)* time complexity operation to *O(1)*.

2. Second implementation: The ***updated*** generic queue under the form of a linked list

Improving the time complexity of the *add* operation is therefore, done, by adding an extra pointer that keeps track of the queue's last element. This way, we no longer have to traverse the whole data structure in order to add a new item.

Below the updated version of the *add* method has been included:

```
public void add(K item) { //generic implementation - item of type K
    if(head == null) {
        head = new Node(item, null);
        last = head; //the two pointers are the same if the queue is empty
        return;
    }

    Node n = new Node(item, null);
    last.next = n; //adding the new element after the current last one
    last = n; //updating the new last element
}
```

After these updates have been done, the queue under the form of a linked list has been turned to a generic one in order to help us store in it nodes and implement the breadth first traversal in a binary tree.

Implementing the breadth first traversal in a binary tree

The key in implementing the breadth first traversal in a binary tree is understanding that this time what does matter is to walk through the nodes and display the keys or the values levelwise starting from the root. Similar to last week's assignment, the TreeIterator has been used, however the constructor now checks if the left and right nodes of the root exist and if they do, it adds them in the queue.

Then, as long as the queue is not empty, each node is extracted one by one and goes under the same procedure: adds to the queue their own left and right nodes if they do exist. If the queue is empty, the *next* node is set to *null* and therefore the *hasNext()* method which calls the *next()* method stops its own execution. In other words, we have reached the end of the breadth first traversal comes to an end.

Below, the new TreeIterator constructor has been included since it displays the way the nodes are added in the queue.

```
public TreeIterator() { //the TreeIterator constructor
    next = root; //assigning the root to the 'next' node
    if(root.left != null) //checking that the left node exists
        queue.add(root.left); //adding the left node to the queue
    if(root.right != null) //checking that the left node exists
        queue.add(root.right); //adding the right node to the queue
}
```

3. Third implementation: The queue under the form of a dynamic array

The starting point of implementing the dynamic queue array was its own static version. Having said that, the main difference between the static and dynamic one is that once the queue is full, the elements are carefully copied (more will soon be explained) and a new queue of *double size* is allocated. Now, let us walk through the detailed implementation steps:

Step 1: The object QueueArray (the name I have given to the class) is made up of the following attributes: **an array** of a **given size**, alongside **two integer variables**, which are responsible of keeping track of the indexes where in the array, the *first* and *last* elements find themselves in the queue.

Step 2: The **constructor** is setting the size of the queue and initialising the *first* and *last* variables to zero. In other words, they are both referring to the first position in the queue.

Step 3: Creating the **enqueue method**. In order to fully understand the conditions that have to be imposed on this method, different scenarios have to be taken into consideration. As far as I am concerned, I think the main ones take place when:

- overwriting the queue items with new ones once the old ones have been dequeued; memory efficient usage of the queue

As far as this problem is concerned, in order to solve it, we have to make use of the modulo operation with respect to the queue's size. It should be taken into consideration that when we work with **modulo n**, we basically make use of all the remainders greater than zero and lower than n. This way, once the last element is reached in the array and there is available empty space at the beginning of it, we set the new integer referring to the last queue element, to be to be the value of the previous one, plus one, everything modulo the array's size. In other words when we do so in this particular case, we get zero (size modulo size = 0). And so on and so forth.

```
last = (last + 1) % size;
```

- the queue is full and we have to allocate a new, larger array;

Since the the value of the integer pointing to the last element in the queue points to the first available spot, when it coincides with the one already pointing to the first element, that is precisely when it's time to allocate a new array of double size and copy everything.

- the order of copying the elements to the extended array;

In order to make sure that we can keep on using the remaining new empty space, we start by copying all elements from the first one until the end of the array and then all elements from index zero until the last one. After having done so, we set the value of the index pointing to the last element to be equal to the previous array length, after which we set the new size to double, reset the index of the first element to zero and assign to the queue the new, larger array.

```
int index = 0; //the index keeping track of the copyarray

for(int i = first; i < size; i++) { //copying the first elements in queue
    copyarray[index++] = queue[i]; // moving forward with the index as well
}

for(int i = 0 ; i < last; i++) { //copying the last elements in queue
    copyarray[index++] = queue[i];
}
```

Step 4: Creating the **dequeue method**. Every time this method is called, we start by checking if the *first* and *last* indexes coincide. If they do, we return "null" since the array is, therefore, empty. Otherwise, we once again make use of the modulo operation to move forward with the index of the first element. However, before that we copy in a variable the value that has to be dequeued and in the end, return it.

```
first = (first + 1) % size;
```

Conclusion

All in all, it was very interesting to see how the queue data structure can be implemented in different ways. However, I would say that the most captivating part was by far the idea of making use of Discrete Mathematics, namely of the **modulo n** operation in order to reuse the space left empty by the removed from the queue.