Queues

Fatima Mohammad Ali

Fall 07/10/2023

Introduction

The purpose of this assignment was to explore the properties of *queues* represented in both arrays and linked lists. Queues are vital data structures that can be utilized for task scheduling or handling concurrent programs. Since queues are FIFO (*first-in-first-out*), their implementation becomes interesting when comparing it to their counterpart *stacks* which are *last-in-first-out* structures. Queues can also be interesting in regards to traversal and dynamic allocation aspects, which will be analysed in this report.

A linked list

To implement a queue for a linked list, a `class Queue<T>` that includes a node structure was used. The queue was generic for flexible use (in this task, the queue was solely used for `Integer` type). The queue kept track of two pointers, `first` and `last` which each pointed to the first respective last nodes in the queue. Each node held a value (`T item`) and a next pointer (`Node next`).

A method `void enqueue(T itm)` was used to add an item at the end of the queue. A new node was initialized with the given item value and a null pointer, then it was added to the queue by considering the `last` pointer. If `last` was not null, `last.next` would point to the new node and then `last` would be set to the new node. If the queue was empty (`first == null`), both the `first` and `last` pointers would point to the new node. Hence, a node was added at the end of the queue while updating `last` accordingly.

Removing a node from the beginning of the queue was done as follows:

```
public T dequeue(){
    if(first == null)   //if the queue is empty
        return null;    //return null

    T itm = first.item; //save the item of the first node
    first = first.next; //set first as the next node in the queue
```

```
        if(first == null)    //if the next node was null
            last = null;     //set last to null
        return itm;
    }
```

In this task, the queue *added* a node to the *end* of the queue and *removed* nodes at the *beginning*. With this same approach, assume that there were no `last` pointer. This would have required traversing through the entire list until the null-pointer was found to add a node at the end, which is costly ($O(n)$ where n are the number of traversed nodes). Removing an element at the beginning would not induce a bigger cost since the unlinking would be done in constant time. In the implemented code above however, the `last` pointer kept track of the lastly added node and so the addition of a new node was done in constant time.

Breadth first traversal

Contrary to the previous assignment where *depth first traversal* (DFS) of a binary tree was done, *breadth first traversal* (BFS) for a queue was implemented in this task. Depth first traversal works by traversing deepest down the left branch until finding a leaf and then exploring the left branches of the right branch (specifically for post-order traversal). Breadth first traversal however, starts at the root node and explores all nodes of the current level before diving into the next depth level.

BFS in this assignment was implemented in the `next()` method of the `TreeIterator` class that was used for a binary tree. The `TreeIterator` class declared a queue and started by adding the root to the queue. Then, the `next()` method was used to traverse the queue, as follows:

```
    @Override
    public Integer next() {
        if (queue.isEmpty()) //return null if the queue is empty
            return null;

        Node cur = queue.dequeue();

        if (cur.left != null)
            queue.enqueue(cur.left); //add the left branch to the queue

        if (cur.right != null)
            queue.enqueue(cur.right);   //add the right branch to the queue
```

```
        return cur.value;
    }
```

DFS could be beneficial when needing to dive deep down a tree structure to find a specific element (e.g. the smallest element). For some cases, DFS would also require less memory than BFS since it does not need to save all nodes from one level before diving into the next one. BFS however, is much more efficient when wanting to find the shortest path from the root to a target node.

Additionally, the main purpose of this task was to traverse a queue. When doing so, BFS was ideal since iterating the tree produced the nodes in FIFO manner. DFS mainly uses a stack and would have produced the values in-depth order, which is not ideal when analysing the sequential order of a queue.

Array queue

In this task, a queue represented in an array was implemented. The first `enqueue` method was done non-dynamically, which was later changed since having a queue of fixed size could result in it becoming full. The queue was also done in "wrap-around" manner. If the array was sequential where the first and last indices were constantly incremented, the queue would need to be extended whenever an element had to be added (while the first indices could be vacant after removing elements). Having the array wrap around would effectively fill the vacant spaces instead of constantly extending the array, which is more efficient.

The `ArrayQueue` class was given an `Integer[] queue`, index i (considered as the *first* index) and index k (considered as the *last* index), and a *size* (n) of the queue. Adding an item was done in the method `void enqueue(Integer itm)`. Several cases were considered when adding elements to a non-dynamic queue to achieve a wrap around behaviour. If i and k were equal, the queue was empty (both i and k were initialized to 0). After adding an element, index k was incremented in modulo *size*. In other words, if k was equal to *size*-1, it was set to 0. This implementation produced a simple queue that could become full if k wrapped around and became equal to i, which was solved by implementing the following dynamic queue:

```
public void enqueue(Integer itm){
    queue[k] = itm; //save the item at index k
    k = (k + 1) % size; //increment k in modulo size
```

```
    if(k == i){ //if the queue is full
        Integer[] newarr = new Integer[2*size]; //allocate bigger array
        int counter = 0;    //counter to iterate the new array

        for(int j = i; j < size; j++){  //copy elements from i to size - 1
            newarr[counter] = queue[j];
            counter++;
        }
        for(int j = 0; j < k; j++){ //copy elements from 0 to k
            newarr[counter] = queue[j];
            counter++;
        }
        size = 2*size;  //update the new size
        i = 0;  //i is set to 0
        k = counter;    //k is set to size
        queue = newarr; //set new queue
    }
}
```

This `enqueue` method started by setting the given item at the end of the queue (at index k). Then, the queue was checked if it was full or not. If i was equal to k, a bigger array with double the size was allocated. What was left now was to transfer all elements to the new array. This was done by copying all items from i to *size*-1 to the new array, and all items from 0 to k to the new array. Lastly, i was updated to being 0 again, k was updated to the last added element in the new array (referenced by *counter*), and the *size* was updated. Hence, a dynamic queue was achieved.

Furthermore, an `Integer dequeue()` method was implemented to remove an item at the beginning of the queue (at index i). If i was equal to k, the queue was empty and null was returned. Otherwise, the value at `queue[i]` was saved in a temporary variable, then `queue[i]` was set to *null*, and the temporary variable was returned. Similarly to how k was incremented in the enqueue method, index i was incremented in modulo *size* after removing the item from the queue. This removing method would not shrink the queue. A shrinking queue would involve moving all elements to a new smaller array (e.g. one quarter the size) when the total amount of elements reached a certain number, but this was not implemented in this assignment.