Queues

Axel Månson Lokrantz

Spring Fall 2022

Introduction

This assignment will focus on the queue data structure. A queue is, as the name suggest, a row of items in a waiting line. It follows the first-in-first-out principle (FIFO), just like a queue would in our everyday life. The principle can be compared ta a stack, which uses a last-in-first-out (LIFO) logic. The following program was written in Java together with Axel Lystam and Björn Formgren at KTH.

Task 1: Linked List and Queues

The simplest way of implementing a queue is through a singly linked list. One way would simply be to set a pointer to the head of the list, add elements to the rear and remove them at the front of the list. Insertion would then have the time complexity of $O(1)$ since it is at the rear of the list all we have to do is add an extra node. To delete a node we need to traverse the list n number of elements until we reach the last node, which would result in a time complexity of $O(n)$.

A more convenient implementation is to add an extra pointer which points at the last element of the list. Insertion and deletion would then be $O(1)$ since we now have a reference to the last node and know exactly which node we want to delete.

```java
public void enqueue(int key){

        Node temp = new Node(key);

        if (front == null && rear == null){
            this.front = this.rear = temp;
            return;
        }

        this.rear.next = temp;
        this.rear = temp;
    }
```

Enqueue function for a linked list.

We check if the list is empty, this only occurs when the front and rear pointers are null. If not, then we set rear.next to the new node (temp) and update our rear pointer.

```
Node dequeue() throws NullPointerException{

        Node node = front.node;

        if (front == null)
            throw new NullPointerException ("List is empty.");
        if(front == rear)
            front = rear = null;
        else
            front = front.next;
        return node;
    }
```

Dequeue function for a linked list.

In the dequeue function we make a copy of the node we want to delete and return. If front and rear are pointing at the same node, we only have one node in our list which means we have to set both pointers to null. Else, we simply set front to point to the next element. Since there is no pointer pointing towards the node we want to remove java's garbage collector will take care of it.

Task 2: BinaryTree and Queues

In this next task we were asked to reuse our binary tree implementation and replace the stack with a queue. We were also asked to make use of a traversal called breadths-first. Breadths-first search (BFS) is a search algorithm for the binary tree which starts at the root and explore all nodes at the present depth prior to moving on to the nodes at the next depth level. This method for traversal is often implemented through a queue to keep track of the child nodes that were encountered but not yet explored.

```
    TreeIterator(BinaryTree.Node root)
    {
        pointer = root;
        que = new ListQueue();
        que.enqueue(root);
    }

    @Override
    public boolean hasNext()
    {
        return que.front != null;
    }

    @Override
    public Integer next()
    {
        pointer = que.dequeue();

        if (pointer.getLeft() != null)
            que.enqueue(pointer.getLeft());
        if (pointer.getRight() != null)
            que.enqueue(pointer.getRight());
        return pointer.getValue();
    }
```

Implementation of a queue in a binary tree.

We set our pointer at the root node and delete it from our queue. If there are nodes either to the left or the right of the root we store them inside our queue structure. Next iteration we set our pointer at the node to the left of our root and delete it from the queue. If there are any nodes connected, either to the left or right, we move them to the queue. The same pattern repeats over and over until the condition in the hasNext function no longer is met. With the insertion below we got the following printout:

```
tree.add(5,105);
tree.add(2,102);
tree.add(7,107);
tree.add(1,101);
tree.add(8,108);
tree.add(6,106);
tree.add(3,103);
```

```
Depth level 1, value: 105
Depth level 2, value: 102, 107
Depth level 3, value: 101, 103, 106, 108
```

Task 4: Array and Queues

In the forth and final task we made two implementations of a queue in an array structure. The implementation is quite tricky as there are a lot more corner cases to consider. The idea is to have two pointers i and k where i points at the first element and k points at the last element at position k-1. When adding / removing items from the array we simply increment k and i. At some point k will be equal to the array length which means we have to wrap around to see if there are more slots to fill. The following cases were listed in the assignment to take into consideration:

- If i is equal to k the queue is empty. An attempt to remove an item from an empty que will return an IndexOutOfBoundsException.
- If k is equal to the length of the array - 1, k should be set to index 0 to implement wrap around.
- If k equals i after being incremented the queue is full.
- If i is equal to the length of the array -1, i should be set to index 0 to implement wrap around.

```
Integer remove(){
    if (i == k)
        throw new IndexOutOfBoundsException("Array is empty.");
    Integer value = array[i];
    array[i] = null;

    // Wrap around.
    if (i == size - 1)
        i = 0;
    else
        i++;
    return value;
    }

    void add (Integer value){
        // Wrap around.
        if(k == array.length)
            k = 0;
        // Array is full.
        if(i == k && array[i] != null){
            throw new ArrayIndexOutOfBoundsException("Array is full.");
        }
        array[k++] = value;
    }
```

Implementation of a queue in an array.

For the second implementation we had to address the fact that our array eventually will run out of space. To solve the problem we created a new function called expand. Much like the dynamic stack from previous assignments we implemented a dynamic queue. Once the array gets full we allocate a new temp array with double the length of the original one. The copying was done as follow:

- Index i is set to 0 and all items from i to the array length -1 are copied starting at 0.
- All items from 0 to k - 1 are then copied and k is set to the array length.

```
Integer remove(){
    if (array[i] == null)
        throw new IndexOutOfBoundsException("Array is empty.");
    Integer value = array[i];
    array[i++] = null;
    if(i == array.length)
        i = 0;
    return value;
}

void add (Integer value){
    if(k == array.length)
        k = 0;
    if(array[k] != null)
        expand();
    array[k++] = value;
}

private void expand(){
    Integer[] expandedArray = new Integer [2*array.length];
    int j = i;
    int n = 1;
    expandedArray[0] = array[j++];
    while(j != i){
        if(j == array.length)
        j = 0;
        else
        expandedArray[n++] = array[j++];
    }
    k = array.length;
```

```
            i = 0;
            array = expandedArray;
        }
```

Implementation of a dynamic queue in an array.