Design of Information Structures - Lab 3

January 25, 2019

1 Introduction

During this lab session we will develop classes for Queues and Stacks, based on linked lists. Be sure to copy the files for this laboratory from the cs126 web site before you start.

You should create *separate* pieces of code for each of the exercises, so that it is easy for the tutors to check your work at each of the milestones.

2 Queues

The queue is an important data structure which is used in a variety of algorithms. It is a dynamic data structure that exhibits so-called *First In First Out (FIFO)* behaviour. What does that mean?

First of all, queues provide the following three main operations (all defined within the IQueue interface provided):

- `enqueue(E obj)`
- `E dequeue()`
- `boolean isEmpty()`

If you enqueue $x_1, \ldots, x_n$ in some order, then you will dequeue them in the same order. So the first object enqueued will be the first to be dequeued. This is the *First In First Out* (FIFO) behaviour.

As long as your data structure provides the previously mentioned operations and they respect the *FIFO* principle, then it is considered a queue.

Notice that implementation details do not determine whether a data structure is a queue or not. It is only the behaviour that matters.

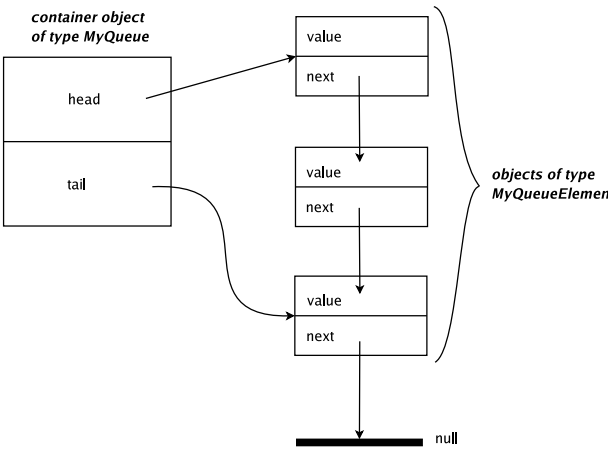

Figure 1: A queue with three elements

2.1 Implementation

We will implement a queue using a linked list. First we will show how to implement a queue from scratch, later on you will have to do it yourself, using the provided `DoublyLinkedList` class.

We will be dealing with two types of objects:

`MyQueue` This class implements the IQueue interface and specifies the container object that manages a queue. All operations which can be performed on a queue will be grouped in this class.

`ListElement` This class specifies the data carrier object that will be used to store the data. These objects will form a singly linked list, where the head of the list is the first object to be dequeued and new objects are added to the tail of the list. The container object will be responsible for managing these. They will not be visible to the user of `MyQueue`.

Remember:

- Each class is defined in a separate file. The file name is the same as the class name.
- In order to call a method on some object it must be defined in that object's class file.

Exercise 1 *Open the files which have the definitions of the classes in an editor. Describe the operations one can perform on these objects.*

Exercise 2 *Figure 1 depicts a queue with three elements. Draw pictures of queues with zero elements and after enqueueing the strings "queue" and "de-*

queue". Annotate the diagram with numbers to show which order the different steps occurred in.

Exercise 3 *Draw another diagram of the queue after one call to the dequeue method. Which value is returned?*

Exercise 4 *Create a* `toString()` *method which returns a* `String` *representation of the queue.*

Exercise 5 *Create a method* `modifyHead()`*, which takes an argument of generic type E and updates the current head. Which files will you have to modify? Where will you place the new method?* Notice that this operation is not typical for queues and is not defined in the `IQueue` interface.

Exercise 6 *Test both of these methods by creating a queue in a main method, adding items to it and calling* `modifyHead()`*. Use your* `toString()` *method to print out the queue at various stages of this test.*

Milestone 1) You have now completed *Milestone 1*, and your lab tutor should sign it off.

Exercise 7 *Create a class* `LinkedListQueue`*, which will provide the standard queue operations, but will be using the provided* `DoublyLinkedList` *class to store and manage the values stored. In the files* `DoublyLinkedList.java` *and* `ListElement.java` *you are provided with an implementation for a doubly-linked list. A doubly-linked list is one in which list elements contain pointers to both the next and previous items in the list, and both the head and the tail of the list are stored. This allows the list to be iterated over both forwards and backwards, and also allows data to be added and removed from both ends of the list in an efficient manner.*

3 Stacks

A stack is a data structure in which elements are added and removed in a Last In First Out (LIFO) order. Normally we refer to the adding of elements as *pushing* onto the stack, and removing of elements as *popping* from the stack. We are going to implement the Stack class by again using the existing `DoublyLinkedList` class.

Exercise 8) You should inspect the code in `DoublyLinkedList.java` and `ListElement.java` to understand how the doubly linked list works.

Look at the MyStack.java files, you will find the following source code:

```
public class MyStack<E> implements IStack<E> {

        // INCOMPLETE.
        public void push(E val) {
                // TODO: implement pushing
        }

        // INCOMPLETE.
        public E pop() {
                // TODO: implement popping
                return null;
        }

        // INCOMPLETE
        public boolean isEmpty() {
                // TODO: check whether list is empty
                return false;
        }
}
```

In order to implement these methods you will need to call some of the methods from `DoublyLinkedList` class.

Exercise 9) Implement the 3 methods, `push`, `pop` and `isEmpty()`.

Milestone 2) You have accomplished **Milestone 2**, your lab tutor should sign you off.

We are now going to demonstrate the usefulness of the stack structure by implementing a simple calculator for expressions in the postfix notation. We typically use 'infix' expressions, where operators are put between operands. For example $1 + 2$ has its addition symbol between 1 and 2. This is not the only way one can write expressions, however.

In the postfix notation the operator comes after its arguments so the above example would be written as 12+. Postfix notation is confusing at first, but allows expressions to be written unambiguously without brackets. The expression $(1 + 2) * 3$ can be written as $12 + 3*$ as the order in which operators apply is encoded in the order they are written. Evaluating such expressions by hand can sometimes be confusing, but by using the notion of a stack we can easily implement a program to do this for us.

You should now look at Calculator.java, this class currently takes input on stan-

dard in and prints out whether a value or an operator was input. You should experiment with this code so that you understand what is happening. The calls are made in the five methods that are all of the form `newValueToken`. There is a method for each operator.

The calculator is capable of accepting single-digit numbers and the operators `+, -, *, /` and `=` . When you reach the `=` operator you should evaluate the expression preceeding it, and print some output on standard out. In order to do this you need to implement the `evaluate` method. The general procedure is to scan through the string, recording any numbers seen. When an operator is encountered, the last two values seen should be retrieved and the operation performed on them and the result stored. If you have any questions about postfix expressions then please ask your lab tutor.

Exercise 8 *Design your program and decide how you are going to evaluate the input and draw the stack at different phases of the evaluation of the expression* `1 2 + 3 * =`.

Exercise 9 *Implement the methods which are called when new tokens are entered.*

Milestone 3) You have now completed **Milestone 3** and your lab tutor should check it off.

If you have reached this stage, you have achieved full marks for Lab 3. There are optional assignments for people who are ahead of schedule.

Exercise 10 *One can also consider prefix notation, in which operators come before their operands. For example* $1 + 2$ *is represented as* $+12$. *Alter your system to evaluate such expressions.*

Exercise 11 *If you have a calculator that takes prefix notation and your input is in postfix notation, think about how you could rearrange the input so that your calculator could evaluate such expressions.*