Lab 4: Queues & doubly linked lists

F28SG – Introduction to Data Structures and Algorithms (6 marks)

Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

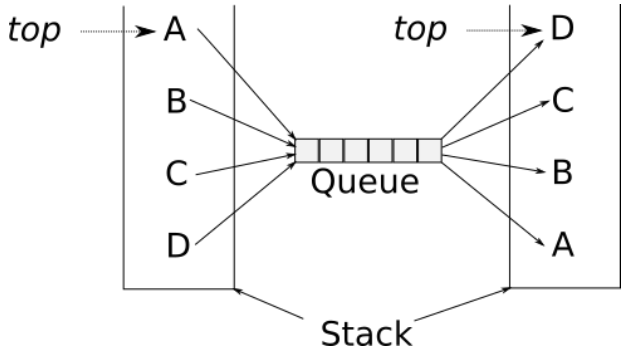
Project Structure

The project is organized as follows:

- The src directory contains the following source files
  - **ReverseStack.java** needs completing for Q1
  - **Queue.java** is an implementation of a queue to use for Q1
  - **Stack.java** is an implementation of a stack to use for Q1
  - **LQeue.java** is the start of a queue implementation using a linked list for Q3
  - **DLinkedList.java** is an implementation of a doubly linked list to complete for Q4
- The test directory contains the unit tests for the project.

Q1) Reverse a stack using a queue (1 point)

Your first task is to **reverse a stack using only a queue** provided by the Queue.java file.



To do this, complete the implementation of the method in **ReverseStack.java**

```
public static void reverseStack(Stack st){

}
```

This should repopulate the **existing** stack `st`, rather than creating a new Stack instance inside this method. That is, this method should update "in place" the data in the stack.

Two tests are provided in **ReverseStackTest.java**. Both should pass when you have correctly implemented this `reverseStack` method.

State the Big-Oh complexity as a comment above the `reverseStack` method.

This question introduces a new concept in this course: Java Generics. Notice that previously we used `Stack`, but we are now using `Stack<String>`. Review the code in **src/Stack.java** , the class definition has changed from:

```
public class Stack
```

Which we used in previous labs, to:

```
public class Stack<T>
```

This solves a key limitation of the old version, which was highlighted in Q3 of lab 1 for the Polish calculator. The `Stack` class in lab 1 stored instances of Java's most general type: the `Object` class, and you had to cast to an `Integer` every time you popped from the stack.

The improved version of Stack, which uses Java Generics, means you can specify the type of the elements that will be in the data structure, by replacing the `T` with your chosen type (or "class", because to construct a new user defined type in Java you create a class for it). This is a form of a programming language type feature called: *parametric polymorphism,* where "*poly*" means "*many*". Java borrowed this type system feature from functional programming languages. See ***Hint2!*** for a reference to a book about Java Generics.

Using the old version of Stack means writing code like:

```java
@Test
public void reverseStackTest() {
    Stack st = new Stack(5);
    st.push("A");
    st.push("B");
    st.push("C");
    ReverseStack.reverseStack(st);
    assertEquals("A", (String) st.pop());
    assertEquals("B", (String) st.pop());
    assertEquals("C", (String) st.pop());
}
```

But now we can write:

```java
@Test
public void reverseStackTest() {
    Stack<String> st = new Stack<>(5);
    st.push("A");
    st.push("B");
    st.push("C");
    ReverseStack.reverseStack(st);
    assertEquals("A", st.pop());
    assertEquals("B", st.pop());
    assertEquals("C", st.pop());
}
```

The benefits of using Java Generics in this way are:

1. The implementation of the Stack class is more reusable, because it works for strings, integers, etc., and any user defined class.
2. The use of the improved `Stack<T>` class is that our code is more type safe because
    a. There is no casting between types in our own code. Instead, we rely on casting performed internally inside the improved `Stack` class.
    b. We would get a type error if we tried to push or pop an element with the wrong type, e.g. you can't push the Integer value 34 into a stack of Strings when using the `Stack<String>`.
3. Your code becomes a lot more readable in the absence of type casting.

Please consider using Java Generics more in your code in labs 4-8. You will see them again soon in labs 7 and 8 when sorting Linked Lists.

Q2) Testing a linked list based queue implementation (1 point)

In the test directory there is a file called **LQueueTest.java** with a set of empty test methods you need to implement. This should test the operations of a queue, which you will implement using a linked list in Q3. Write suitable test methods for each of them. You should be able to adapt some of the test methods you have previously used for stacks but remember that a queue is a FIFO and not a LIFO structure.

For the tests for the `isEmpty` method, rather than using `assertEquals` you might find `assertTrue` and `assertFalse` more useful for brevity.

When documenting your LQueue tests write one sentence above each test method, with Javadoc syntax if you wish, explaining what the test is checking. For example: *"Populates a queue with 3 values then checks that the first of these values added to the queue is at the front of the queue".*

Q3) Implement a queue using a linked list (2 points)

In **LQueue.java**, a start has been made on implementing a queue using a linked list. Your task is to complete this implementation. Your implementation should preserve the FIFO properties of a queue. All your tests from Q2 should pass once you have implemented these methods.

State the Big-Oh complexity as a comment above the `isEmpty, size, enqueue, dequeue` and `front` methods.

Q4) Reverse doubly linked list [challenging] (1 point)

DLinkedList.java is an implementation of a doubly linked list. Your task is to implement the method which reverses this list:

```
public void reverse(){

}
```

You must do this ***without using the Stack class*** and ***without copying the list's values into another temporary structure (e.g an array)*** to later reinsert back into the list. Instead, you should rearrange the object pointers that connect the list nodes together, leaving the data inside each node untouched. See ***Hint1!*** if you need help.

Unit tests for this method are provided in **DLinkedListTest.java**. All the test methods should pass when you have implemented this method.

Q5) Code Quality (1 point)

Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).

An additional mark is awarded if your code is deemed to be of high quality:

- **Big-Oh**
  - Has Big-Oh complexity been documented for all methods where requested?
- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**
  - Are you using comments ***inside your implementation methods and test methods*** to provide an algorithmic commentary about what the code is doing.

Here is a **good** comment:

```
// creates unidirectional connection from the predecessor node to the new node
prevNode.nextNode = newNode;
```

Here is a **less useful** comment:

```
// sets preNode.nextNode to newNode
prevNode.nextNode = newNode;
```

  - Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS, the shortcut is: ***⌘ + Alt + j***
- **Conventions**
  - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

Hints

Hint1! Every Node object has two object references **nextNode** and **prevNode**. Currently the direction of these arrows between every pair of adjacent nodes dictates the order of the list, starting from the head node. Rather than removing elements from the list and adding them back in a different order, you could instead change the direction of these object reference arrows. Remember also to update what **headNode** and **tailNode** points to.

Hint2! Java Generics and Collections, Maurice Naftalin, Philip Wadler, 2006, O'Reilly Media, Inc., ISBN: 9780596527754, https://www.oreilly.com/library/view/java-generics-and/0596527756/ .