Lab Exercise 7
Stacks and Queues

1 Introduction

This lab will introduce stacks and queues and allow you to have more experience for debugging. For this purpose, we will be evaluating ArrayStack and LinkedQueue discussed a little later. The provided implementation is buggy. You will use the debugger to evaluate the operation of the two classes to help to identify the bugs, and fix them.

Using Eclipse, import “Lab7”:

/afs/cs.uwm.edu/users/classes/cs351/401/pantherid/git/lab7.git

You are given two JUnit test suites to help you find the bugs and a driver to show how stacks and queues function.

2 Stack Implementation

A Stack works on a LIFO principle, in that the last added element is also the first to be removed. You can think of this as a tall container, where the only object you can get is the object at the top which is the last object added. An example is given below.

- push("Task 1")
- push("Task 2")
- push("Task 3")

(top)

__________ __________ __________
| Task 1 | | Task 2 | | Task 3 |
---------- ---------- ----------

- pop()

(top)

__________ __________
| Task 1 | | Task 2 |
---------- ----------

3 Queue Implementation

A Queue works on a FIFO principle, in that the first added element is also the first to be removed. You can think of this as a waiting line, where the first arrived person will be served first. An example is given below.

- add("Task 1")
- add("Task 2")
- add("Task 3")

(front)

__________ __________ __________
| Task 1 | | Task 2 | | Task 3 |
---------- ---------- ----------
• remove()

(front)

| Task 2 | | Task 3 |
|--------|--------|

4 Your Tasks

Read the sections above to familiarize yourself with each implementation’s operation. It should be noted that since the implementations are buggy, the unit tests at this moment will crash at one point or another. Try to run the tests and see what happens.

4.1 ArrayStack

You will be guided through ArrayStack to identify the bug. This stack was implemented with an non-dynamic array to simplify things. This means that the stack has a capacity (defaulted to 10) and should return false if the new element doesn’t fit in the stack. Read the code to understand the implementation.

1. Run TestArrayStack.java and notice that testAddMax fails with an ArrayIndexOutOfBoundsException, which is common with an array implementation.

2. Select the failed test and by looking at the Failure Trace, commonly found in the JUnit view, you can see the line that the exception was thrown on, in the push method.

3. Double-clicking the line on the Failure Trace. This should bring you to the exact line where the exception was thrown. Set a breakpoint there.

4. Since currently you are only interested in this one particular test, you can choose to just debug this one by right-clicking the test in the JUnit view and choose Debug. You should be prompted to change perspective to Debug.

5. The test should suspend on your breakpoint. Click step-over or press F6. You should be at the assertion of the invariant at the end of push.

6. Take a look at the Variables view, which should now have some variables highlighted. The highlighted items have changed since the last step. In this case you should notice that tail is now 1, and if you expand the data array, you should notice that the 0 index of the array is now populated with the element. Everything looks fine at the moment.

7. Since you didn’t notice anything abnormal about this run, you may click resume or F8 to resume the test until it runs into the breakpoint again (or another breakpoint).

8. Recall that the test failed with out of bounds exception with an index of 10. Repeating the above operations on the irrelevant iterations can be tedious. Instead, you can decide when to suspend at a breakpoint. Right-click on the breakpoint, and select Breakpoint Properties.... This should bring up a dialog with three check boxes, Enabled, Hit count, Conditional. As their names imply, the Enabled simply lets the Eclipse know to suspend at that breakpoint. Hit count is also intuitive: if you know the exact number of times until things get “interesting,” you can specify a number of times to skip the breakpoint. Conditional makes more sense at this point; you can specify a boolean expression that controls this breakpoint. The program will only suspend on this breakpoint if the boolean expression evaluates to true.

9. Check Conditional and in the text area below put
_tail == 10

Since you know that this is when the exception is thrown. Resume again, check that _tail is indeed 10 at this point.

10. Look at the variables of the stack. You notice that the stack is full, what is the expected behavior in this case? What will actually happen according to the current code?

11. Correct the code to pass the tests.

4.2 **LinkedQueue**

The queue is implemented using a singly-linked list with head and tail pointers. Read the code to familiarize yourself with *LinkedQueue*. Then debug the code, answer the questions below, and fix the code.

1. What line does the code fail on: ________________________________
2. What is the error: ________________________________
3. What is the case that the error happens?
4. What should happen in this case?

Once completed, see your TA to receive credit for this lab.