UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 4

Stacks and Queues – Solutions and Grading

Date Due: Feb 9, 2017, 10pm **Total Marks: 54**

Question 0 (10 points):

Purpose: To force the use of Version Control in Assignment 4

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 4. This is a tool that you need to be required to use, even when you don't need it, so that when you do need it, you are already familiar with it. Do the following steps.

1. Create a new PyCharm project for Assignment 4.
2. Use `Enable Version Control Integration...` to initialize Git for your project.
3. Download the Python and text files provided for you with the Assignment, and add them to your project.
4. Before you do any coding or start any other questions, make an initial commit.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open the terminal in your Assignment 4 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm, locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.

   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A3 folder, and run `git --no-pager log` there. If you did all your work in this folder, git will be able to see it even if you did your work on Windows. Git's information is out of sight, but in your folder.

   Note: If you are working at home on Windows, Google for how to make git available on your command-line window. You basically have to tell the command-line app where the git app is.

You may need to work in the lab for this; Git is installed there. Not having Git installed is not really an excuse. It's like driving a car without wearing a seatbelt. It's not an excuse to say "My car doesn't have a seatbelt."

What to Hand In

After completing and submitting your work for Questions 1-4, open a command-line window in your Assignment 4 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/-paste this into a text file named `a4-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the way to use git, but it is the way students work on assignments.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: The log file shows that you used Git as part of your work for Assignment 4. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per question for a total of at least 6 commits. And frankly, if you only have 6 commits, you're pretending.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 5 marks for good commit messages; 4 marks for anything else.
- Students were told at least eight commits (2 per question).
- Give 5 marks for 8 commits or more, and 2 marks for anything less than 8.

Question 1 (10 points):

Purpose: To practice file I/O, and the use of the Stack ADT.

Degree of Difficulty: Easy

Write a Python script that opens a text file, reads all the lines in the file, and displays it to the console as follows:

- The lines are displayed in reverse order; the first line in the file is the last line displayed.
- The words in the line are in reverse order; the first word in a line from the file is the last word displayed on a line.
- The characters in each word are not reversed; a word in the file is displayed without change.

For our purposes here, a **word** is any text separated by one or more spaces, that is, exactly the strings you get when you use the string method `split()`.

For example, suppose you have a text file named `months.txt` with the following three lines:

```
January February March April
May June July August
September October November December
```

Running your script on the console should produce the following output:

```
UNIX$ python3 a4q1.py months.txt
December November October September
August July June May
April March February January
```

Notice the following:

- The sequence of the lines displayed is reversed compared to the file.
- The sequence of words displayed on a line is reversed compared to the file.
- The sequence of characters in the words are not reversed.

You can test your script informally; you will not need to hand in any formal testing beyond a few example runs. But you should run your script on a few different files. There are also some example text files on the A4 Moodle page that you can use to try your program out.

Using the Stack ADT

Download the Stack ADT implementation named `TStack.py` from the Assignment 4 page on Moodle. To help you avoid errors, this implementation of the Stack ADT does not allow you to violate the ADT in a careless way. **You do not need to understand the code in the file `TStack.py`. You do not even need to look at it.** We will study simpler and more accessible implementations later in the course. You should focus on using the Stack operations correctly. `TStack.py` has the same Stack ADT interface, and has been thoroughly tested. If your script does not use the Stack ADT correctly, or if you violate the ADT Principle, a runtime error will be caused.

Important Note – Read this unless you want zero marks

The purpose of this program is to practice and achieve mastery of the Stack ADT. Your program must use the Stack ADT for this.

You will get **zero marks** if any of the following are true for your code:

- Your program uses the `reverse()` method for lists.
- Your program uses the extended slice syntax for lists or strings to reverse the data.
- Your program uses a for look to step through a Python list or a Python string.
- Your program does not use the Stack ADT in a way that demonstrates mastery of the stack concept.

This is not a trick question. Just use the Stack ADT.

What to Hand In

- Your implementation of the program: `a4q1.py`.
- Copy/paste a few examples of your program as run from the command line: `a4q1_output.txt`.

Be sure to include your name, NSID, student number, course number and lecture section at the top of all documents.

Evaluation

- 2 marks: Your program uses a command-line argument to obtain the name of a file to be reversed.
- 2 marks: Your program displays the reversed contents of the file to the console.
- 2 marks: Your program uses the Stack ADT to reverse the order of the lines in the file.
- 2 marks: Your program uses the Stack ADT to reverse the order of the words on a line of text (string).
- 2 marks: Your output file demonstrates the program working on at least 3 examples.

Note: Use of `reverse()`, or extended slices, or any other technique to avoid using Stacks will result in a grade of zero for this question.

Solution: A model solution is found in the script `a4q1.py`. The solution uses two stacks:

1. A stack to contain the individual lines of the named file. The model solution stores strings, for simplicity, but it is acceptable to store lists of words (i.e., a line that was broken into words by the string method `split()`) or even individual stacks (one stack for each line).
2. A stack for the words in a single line. In the model solution, this stack is created in the function `print_rev_lines()`, but a global script with exactly one such stack re-used repeatedly is fine too.

I had lots of questions about how to get all the words of an input line to appear on the console in a single line. The use of `print(word, end='')` followed by `print()` is a very common technique that's worth remembering.

Notes for markers:

- The purpose of the question is to practice using the Stack ADT. Any solution that does not attempt to use the Stack ADT operations, but instead uses some other Python technique, should be given a mark of zero.
- Watch for violations of the ADT principle: accessing the data in a Stack apart from the Stack ADT operations is a violation. If you find violations of the ADT principle, deduct all 3 marks allocated to the use of the Stack ADT.

- My solution uses a function and 2 Stacks. It is completely acceptable to submit a solution that does not define any functions, and uses only 2 stacks. Queues are optional.
- Watch for solutions that reverse the characters of a whole line. If you find any of those, let me know by email.
- My solution checks the number of command-line arguments, but that's just for good example; it's not required.

Question 2 (16 points):

Purpose: To work with a real application for stacks

Degree of Difficulty: Moderate

In class, we saw how to evaluate numerical expressions expressed in *post-fix* (also known as *reverse Polish notation*). The code for that is available on the course Moodle. It turns out to be fairly easy to write a similar program to evaluate expressions using normal mathematical notation.

Input

The input will be a mathematical expression in the form of a string, using at least the four arithmetic operators ($+, -, \times, /$) as well as pairs of brackets. To avoid problems that are not of interest to our work right now, we'll also use lots of space where we normally wouldn't. We'll use spaces between operators, numbers and brackets. Here's a list of expressions for example:

```
example1 = '( 1 + 1 )'              # should evaluate to 2
example2 = '( ( 11 + 12 ) * 13 )'   # should evaluate to 299
```

Notice particularly that we are using brackets explicitly for every operator. In every-day math, brackets are sometimes left out, but in this is not an option here. Every operation must be bracketed! The brackets and the spacing eliminate programming problems that are not of interest to us right now.

Hint: You will find it useful to split the string into sub-strings using split(), and even to put all the substrings in a Queue, as we did for the PostFix program.

Algorithm

We will use two stacks: one for numeric values, as in the PostFix program, and a second stack just for the operators. We take each symbol from the input one at a time, and then decide what to do with it:

- If the symbol is `'('`, ignore it.
- If the symbol is a string that represents a numeric value (use the function `isfloat()`, provided in the script `isfloat.py`), convert to a floating point value and push it on the numeric value stack.
- If the symbol is an operator, push the operator on the operator stack.
- If the symbol is `')'`, pop the operator stack, and two values from the numeric value stack, do the appropriate computation, and push the result back on the numeric value stack.

You should write a function to do all this processing.

Since the objective here is to practice using stacks, you must use the Stack ADT provided for this assignment. Also, you may assume that the input will be syntactically correct, for full marks. For this question your program does not need to be robust against syntactically incorrect expressions.

Using the Stack ADT

Download the Stack ADT implementation named `TStack.py` from the Assignment 4 page on Moodle. To help you avoid errors, this implementation of the Stack ADT does not allow you to violate the ADT in a careless way. **You do not need to understand the code in the file `TStack.py`. You do not even need to look at it.** We will study simpler and more accessible implementations later in the course. You should focus on using the Stack ADT operations correctly. `TStack.py` has the same Stack ADT interface, and has been thoroughly tested. If your script does not use the Stack ADT correctly, or if you violate the ADT Principle, a runtime error will be caused.

Testing

Write a test script that checks each operations in simple use, and then a small number of more interesting cases using more complicated expressions. Your test script should do unit testing of your function to evaluate expressions. You can add tests of any other functions you write, but focus on testing the expressions.

What to Hand In

- Your function, written in Python, in a file named `a4q2.py`
- Your test-script for the function, in a file called `a4q2_test.py`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: Your evaluation function correctly evaluates expressions of the form given above. It uses two Stacks.
  - 2 marks: The evaluation function correctly handles numeric data by pushing onto a numbers stack.
  - 2 marks. The evaluation function correctly handles operators by pushing on an operator stack.
  - 4 marks. The evaluation function correctly evaluates expressions when a `')'` is encountered. Two values are popped from the numbers stack, and an operator is popped from the operator stack. The correct operation is performed, and the result is pushed back on the numbers stack.
  - 2 marks. When there is nothing left to evaluate, the result is popped from the numbers stack.
- 6 marks: Your test script covers all the operations individually once, and a few tests where some operations are used in combination.
  - 1 mark each: Every operator is tested.
  - 2 marks: Testing includes at least one expression with several nested operations.

Solution: A model solution is posted in the file `a4q1.py`. An example test script is in the file `a4q1-test.py`.

Notes for markers:

- The important aspect of the question is the use of Stacks. For full marks, the Stack ADT must be used. It literally does not matter which implementation is used!
- Watch for violations of the Stack ADT.
- The input should be a string, with plenty of spaces between operators. The model solution splits the string and enqueues all the elements in a Queue. The use of the Queue is not required. A for-loop through the list is fine.
- The model solution has a very rudimentary error condition for an unknown operator. This is entirely optional. The question did not require robust handling of problematic input.
- The testing script doesn't have to be long, and doesn't have to check all the things the model solution checks. If every operator is in the test script, and if at least one expression uses several nested expressions, that's enough.

Question 3 (4 points):

Purpose: To introduce students to the concept of a REPL. Complete this if you have time.

Degree of Difficulty: Easy if Question 2 is working correctly.

The term "REPL" is an acronym for "read-eval-print loop". It is the basis for many software tools, for example, the UNIX command-line is essentially a REPL, and the Python interactive environment is a very sophisticated REPL, and literally hundreds of other useful applications: R, MATLAB, Mathematica, Maple, to name a few.

A REPL is basically the following loop:

```
while True:
   prompt for and Read a command string from the console
   Evaluate the command string
   Print the resulting value to the console
```

In this question, you'll implement a REPL for your evaluation function from Question 2. You could make it slightly more sophisticated by allowing the user to type something like "quit" which will avoid evaluation and terminate the loop. Implement your REPL in a separate script, and import your evaluation function as a module.

An example run for your script might be as follows:

```
Welcome to Calculator.  Let's calculate!
> ( 3 + 4 )
7.0
> ( ( 3 + 4 ) * 5 )
35.0
> quit
Thanks for using Calculator!
```

This is a script you can run rom PyCharm, or on the command line. The first line and last line are not part of the loop, and are just present to create a friendly context. The `'>'` are the prompts displayed by the REPL. The expressions (e.g., `'( 3 + 4 )'`) are typed by the user. The results (e.g. `7.0`) are displayed by the REPL.

What to Hand In

- Your REPL, written in Python, in a file named `a4q3.py`
- A file named `a4q3-output.txt` containing a demonstration of your REPL working, using copy/paste from the PyCharm console.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 2 marks: Your script `a4q3.py` contains a REPL that uses `a4q2.py` to evaluate simple arithmetic expressions.
- 2 marks: You demonstrated your REPL in action. It doesn't have to be cool.

Solution: A model solution is posted in the file `a4q3.py`. An example test script is in the file `a4q3-output.py`.

Notes for markers:

- This exercise is intended to reveal the concept of REPL.
- Marking can be be generous here.

Question 4 (14 points):

Purpose: To work with ADTs on the ADT programming side. To learn that multiple implementations can have an identical interface.

Degree of Difficulty: Moderate to Tricky. A substantial number of marks can be obtained in this question without solving it completely. The implementation of QueueTwo might be a little confusing, but it is not actually difficult.

In class, we saw ADTs for Stacks and Queues implemented as Python lists. The code for these is available on Moodle (but not on the A4 page). The Queue implementation used Python list methods to realize the behaviour of the Queue: we used `append()` inside the `enqueue()` operation, and `pop(0)` inside the `dequeue()` operation. The ADT interface hides the details of the implementation.

In this question, you'll be building an alternate implementation of the Queue ADT, called QueueTwo. The ADT interface for QueueTwo is exactly the same as Queue, but the implementations behind the interface are quite different. The lesson to learn here, among other things, is that there may be several ways to implement an ADT's operations.

From the outside, a QueueTwo looks exactly like a Queue. But on the inside, the data structure for the QueueTwo implementation will be a dictionary containing two Stacks. Let's call them E-stack and D-stack.

The create operation is as follows:

```
def create():
    """
    Purpose
        creates an empty queue
    Return
        an empty queue
    """
    q2 = {}
    q2['e-stack'] = Stack.create()
    q2['d-stack'] = Stack.create()
    return q2
```

The two stacks will be hidden behind the ADT interface, and the QueueTwo operations will use these two stacks in a clever way, so that, to an applications programmer, the queue behaviour is still FIFO. The ADT programmer, however, will see that the two LIFO stacks, working together, get the FIFO job done.

But how? That's the big question. In general, an enqueue operation will place the new value on the top of the E-stack (the E comes from enqueue). The dequeue operation will remove the value from the top of the D-stack (the D comes from dequeue). But we have to do this carefully, as follows.

The create operation above starts with two empty stacks. New data enqueued should be pushed on the E-stack. The enqueued data will pile up on the E-stack as long as there have been no dequeue operations. However, when there is a first dequeue request, the value to be dequeued has to be the first item enqueued (FIFO, remember), and in this situation, it will be at the bottom of the E-stack! Therefore, before you can dequeue it, you have to pop all elements from the E-stack and push them on to the D-stack. Now the value to be dequeued is at the top of the D-stack, and can be popped!

In general, you may see enqueue and dequeue requests in any order. The following rules indicate what to do:

- If there is an *dequeue* operation when the **E-stack is not empty**, we pop every element off the E-stack and push it on to the D-stack first, then pop the top element of the D-stack.
- If there is an *enqueue* operation when the **D-stack is not empty**, we pop every element off the D-stack and push it on to the E stack first, then push the new element on top of the E-stack.

We have provided a "starter-file" for the implementation, `QueueTwo.py`, which is available from the A4 Moodle page. It contains the beginnings of your solution, and your work in this question will be to complete them. The operations are as follows:

1. `create()` creates an empty queue, by hiding 2 Stacks in a dictionary data structure.
2. `is_empty(queue)` This is true if the queue is empty. You have to check both Stacks.
3. `size(queue)` This returns the total number of elements in the queue. You have to check both Stacks.
4. `enqueue(queue, value)` Add the value to the queue at the back. The back is the top of the E-stack when the D-stack is empty. Keep the above rules in mind!
5. `dequeue(queue)` Remove and return the value at the front of the queue. The front is the top of the D-stack when the E-stack is empty. Keep the above rules in mind!
6. `peek(queue)` Return (without removing) the value at the front of the queue. The front is the top of the D-stack when the E-stack is empty. Keep the above rules in mind!

Remember that from the outside, the QueueTwo looks and seems exactly like a Queue; it's only on the inside that the differences are apparent. Any test script that works for Queue should also work for a correct implementation of QueueTwo.

Remember also that the purpose of this question is to work with stacks and queues. We're practicing thinking about how stacks work here. After that's it's just a bit of programming.

Suggestions: How to complete this question

1. First, obtain the Stack and Queue ADTs from Moodle. You can find the files `TQueue.py` and `TStack.py` on the Moodle page for A4. Don't use the list-based implementations from Chapter 10, because it's too easy to get away with violating the ADT principle.
2. Write a test script called `a4q4_test.py` for the **Queue ADT**. Make sure your test script tests all the operations (black-box testing is fine), including `size()` and `is_empty()`. At this stage, your test script should have the following line

   ```
   import TQueue as Queue
   ```

   and the `TQueue.py` implementation should pass all your tests. Your test scripts should be written to check the operations without looking inside the Queue data structure.
3. Download the starter-file `QueueTwo.py` from the A4 Moodle page. This file contains the implementation of `create()` shown above, but non-functional but documented ADT operations.
4. Fill in correct implementations for the operations, as described above. You will have to do it all at once. It would be difficult to do this step incrementally.
5. Change the import line in your test script `a4q4_test.py`:

   ```
   import QueueTwo as Queue
   ```

   Then run your test-script (see the value in the use of `as Name`?) Detect faults, find your errors, fix them, and repeat.

Using the Stack and Queue ADTs

Download the Stack and Queue ADT implementations named `TStack.py` and `TQueue.py` from the Assignment 4 page on Moodle. To help you avoid errors, these implementations do not allow you to violate the ADT in a careless way. **You do not need to understand the code in the files. You do not even need to look**

at it. We will study simpler and more accessible implementations later in the course. You should focus on using the Stack and Queue operations correctly. `TStack.py` has the same Stack ADT interface, and has been thoroughly tested; `TQueue.py` has the same Queue ADT interface, and has been thoroughly tested. If your script does not use the Stack or Queue ADT correctly, or if you violate the ADT Principle, a runtime error will be caused.

What to Hand In

- Your file `QueueTwo.py` containing the new implementation of the Queue ADT.
- Your test-script in a file called `a4q4_test.py`.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 4 marks. Your test script tests all the ADT operations using black-box testing, of course!
- 4 marks: The Queue ADT interface has not changed.
- 1 mark: Your `size()` operation correctly returns the number of items in the queue.
- 1 mark: Your `is_empty()` operation correctly returns True if there are no items in the queue, and False otherwise.
- 2 marks: Your `enqueue()` operation correctly uses the two stacks to add a data value in FIFO order.
- 2 marks: Your `dequeue()` operation correctly uses the two stacks to remove a data value in FIFO order.
- 2 marks: Your `peek()` operation correctly uses the two stacks to return a data value (without removing it) in FIFO order.

Solution: A model test-script can be found in the file `test_queue.py`. A model implementation of QueueTwo can be found in the file `QueueTwo.py`.

The test script uses the Queue ADT only. The first few tests check for consistency between `create()`, `size()` and `is_empty()`. After that, the tests do some enqueueing, and later, some dequeueing. The strategy is to build integration tests up by adding one more ADT operation to the tests, so that if there is an error, we have some kind of indication where the error might be.

Rethinking the question, it might have been better to require unit testing separate data structures (Queue, and QueueTwo), but have only one test script for integration testing.

Notes for markers:

- The test script has to run for both queues, so it cannot look inside the data structure.
- The test script should call all the operations at least once each, but enqueue and dequeue several times.
- The 4 marks for not changing the interface is all or nothing. Almost everyone should get these marks. The purpose is to give marks to prevent something students shouldn't even have thought of doing anyway.
- The implementation must use two stacks, and should use the Stack interface only!
- Watch for violations of the ADT principle. If any operation violates the ADT protocol, deduct all the marks for that operation.