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

Date Due: June 15, 2017, 10pm **Total Marks: ??**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

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.

Question 1 (10 points):

Purpose: To practice file I/O, and the use of both Queues and Stacks ADT.

Degree of Difficulty: Easy

Phoenix Wright, ace attorney, takes a lot of court cases. There are two types of cases: urgent cases and non-urgent cases. Phoenix needs some help scheduling the order he should take his cases. A full list of all his court cases is on moodle (`cases.txt`). Each LINE in `cases.txt` is a different month, and each WORD on each line is a different case. A case beginning with **'URG'** is an urgent case, and is of the utmost priority.

Write a Python script that opens `cases.txt`, reads all the lines in the file, and for EACH MONTH/LINE in the text file, repeat the following:

- Display the number of the month (with the first line being month 1).
- Display all urgent cases, beginning with the most recent case (the rightmost urgent case on current line).
- Display all non-urgent cases in the order they appeared (starting with the leftmost non-urgent case on current line.

For our purposes here, a **case** is any text separated by one or more spaces, that is, exactly the strings you get when you use the string method `split()`. It is important to display the information for each month on the same line when printing to the console. You can do so by giving print the 'end' keyword parameter. Ex: `print("OBJECTION!", end=' ')`

Example output is below:

```
python a4q1.py cases.txt
Month 1: URG-dl6_incident URG-KG8_incident steel_samurai gramarye_murder airlines
Month 2: URG-SL9_incident URG-edgeworth reunion_turnabout larry_butz doug_swallow
Month 3: URG-kidnapping URG-mask*demasque turnabout_corner romein_letouse
```

Notice the following:

- All information from the current line/month are displayed on a single on the console.
- The most recent URGENT cases are printed out first (going right to left).
- The sequence of non-urgent cases are in order of left to right.

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.

For this program, you are required to use BOTH a stack and a queue for this question. Hint: there are two types of cases. Perhaps they should be stored using different data structures...

Using the Stack and Queue ADT's

Download the Stack and Queue ADT implementation 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** `TStack.py` and `TStack.py`. **You do not even need to look at them.** We will study simpler and more accessible implementations later in the course. You should focus on using the 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.

Important Note – Read this unless you want zero marks

The purpose of this program is to practice and achieve mastery of the Stack and Queue 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 to reverse the data.
- Your program uses a for look to step through a Python list or a Python string (except for initially reading the file and to add all values to a queue or stack).
- Your program does not use the Stack ADT AND Queue ADT in a way that demonstrates mastery of the ADT concept.

This is not a trick question. Just use both Stack and Queue ADT's.

What to Hand In

- Your implementation of the program: `a4q1.py`.
- Create one or two of your own custom versions of case.txt (make up your own case names!), and run your program on them (does it output what you think it should?). Copy/paste a few examples of your program as run from the command line, along with the contents of the file that you ran (so we know the output correct): `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 read.
- 2 marks: Your program displays output identical to the example given above (no you can't just copy apste the example output into your program).
- 2 marks: Your program uses the Stack ADT correctly.
- 2 marks: Your program uses the Queue ADT correctly.
- 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.

Question 2 (16 points):

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

Degree of Difficulty: Easy

The parentheses in an expression are balanced if an opening parenthesis has a corresponding closing parenthesis `()`, an opening curly brace has a corresponding closing one {}, and an opening square bracket has a closing one `[]`.

For example, `(a+b)` is balanced because it has both an opening and closing parenthesis. On the other hand, the expression `\{(x+y)/(z)` is not balanced because it does not have a closing curly brace. Similarly, `[a+b] * (x+y)]` is not balanced because it does not have an opening square bracket.

In addition to the above, every opening parenthesis, curly brace or square bracket must have a corresponding closing parenthesis, curly brace or square bracket to its right. For example, {a+b) * (c *d} is not balanced. Though there is one opening parenthesis and one closing parenthesis, this statement is not balanced because the closing parenthesis is not to the right of the opening parenthesis.

Furthermore, a parenthesis can only close when all parentheses opened after it are closed. For example `[( a+b ] * 5)` is not balanced because there has to be a closing parenthesis before the closing square bracket. Every parenthesis, curly brace or square bracket opened last should be closed first. The expression should be `[(a+b)*5]` to be balanced.

Your task is as follows: Using the Stack ADT, implement a function that accepts an expression and determines if the parentheses in the expression are balanced and prints to the screen if the expression is balanced or not.

For example, given the following expression, [(a + b { b + [c (d + e) - f] + g] the function will return False.

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.

What to Hand In

- Your working program in a file called `a4q2.py` containing the implementation of the parenthesis checking program described above
- A testing script called `a4q2-testing.py`. In this case, you are likely only testing your one evaluation function. As such, your tests should be thorough, with multiple test cases per equivalence class.
- A text-document called `a4q2-output.txt` that shows your program working on 2 examples of balanced parentheses and 2 examples of unbalanced parentheses. You may copy/paste to a text document from the console.
- Be sure to include your name, NSID, student number, and course number and at the top of all documents.

Evaluation

- 4 marks. The evaluation function correctly returns True or False depending on the string input.
- 2 marks: A stack is used to push characters onto.
- 2 marks: Different types of brackets/parenthesis are properly checked.
- 3 marks: Each type of bracket/parenthesis is tested.
- 3 marks: Tests include reasonable equivalence classes.
- 2 marks: demonstrations from your `a4q2-output.txt`.

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.

In this question you will be importing the previous question as a module, and using its parenthesis evaluation function. Do not attempt this question until you have finished the previous question.

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 PARENTHESIS CHECKER 9000.  Let's BRACKET!
> ( 3 + 4 )
is a balanced expression
> [( a+b ] * 5)
is an unbalanced expression
> quit
Thanks for using PARENTHESIS CHECKER 9000!
```

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. `is a balanced expression`) 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.

Question 4 (16 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.