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 5

Stacks, Queues and Nodes

Date Due: 14 February 2020, 8pm **Total Marks: 78**

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. It might help you, but it adds an extra step for the markers.
- 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.

Version History

- **06/02/2020**: Release to students.

Question 0 (10 points):

Purpose: To force the use of Version Control in Assignment 5

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 5. This is a tool that we want you to become comfortable using in the future, so we'll require you to use it in simple ways first, even if those uses don't seem very useful. Do the following steps.

1. Create a new PyCharm project for Assignment 5.
2. Use `Enable Version Control Integration...` to **initialize** Git for your project.
3. Download the Python and text files provided for you on Moodle, 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 PyCharm's Terminal in your Assignment 5 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no').

Notes:

- No excuses. If you system does not have Git, or if you can't print the log as required, you will not get these marks.
- Git is installed on all Department computers (Windows and Linux). It's not a good idea to work on the same PyCharm project with Windows and with Linux. Pick one, and stick with it. Going back and forth can corrupt PyCharm's configuration files.
- If you do switch back and forth between Windows and Linux, you may have two local Git repositories, rather than just one. This is less than ideal for advanced use, but manageable for Assignment 4. Grab a git log from both systems, and submit as a single log (txt) file.
- 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 Python Terminal where the git app is.

What to Hand In

After completing and submitting your work for the remaining Questions, open a command-line window in your Assignment 5 project folder (or open the Terminal pane in PyCharm). 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 right 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 5. 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 2 commits per question, you're pretending.

Question 1 (10 points):

Purpose: To work with ADTs on the ADT programming side.

Degree of Difficulty: Easy. The implementation is not difficult if you have a clear understanding of FIFO and LIFO.

Task

In this question, you are provided with the implementation of a simple Stack and Queue (TStack.py and TQueue.py respectively). Your main task in this question is to implement a variant of Stack and Queue ADT, i.e., RStack and RQueue respectively.

Specifically, RStack ADT should have operations with the same names (and same LIFO behaviour) as any Stack ADT, namely: `create()`, `is_empty()`, `size()`, `push()`, `pop()`, `peek()`. However, the functions are to be implemented under the following constraints:

- Use only TQueue ADT to implement the functions.
- Modification of TQueue ADT is not allowed.
- No other ADTs (e.g., list) are allowed in this implementation.

In other words, given only the FIFO Queue operations (implemented in TQueue), implement the LIFO Stack operations in RStack.

Similarly, the RQueue ADT should have operations with the same names (and same FIFO behaviour) as any Queue ADT, namely: `create()`, `is_empty()`, `size()`, `enqueue()`, `dequeue()`, `peek()`. However, the functions are to be implemented under the following constraints:

- Use only TStack ADT to implement the functions.
- Modification of TStack ADT is not allowed.
- No other ADTs (e.g., list) are allowed in this implementation.

In other words, given only the LIFO Stack operations (implemented in TStack), implement the FIFO Queue operations in RQueue.

The implementation of the `create()` operation in both RStack and RQueue ADT has been implemented for you. You are to implement the rest of the operations. To get a better understanding on the first-in-first-out (FIFO) and last-in-first-out (LIFO) concept, please refer to chapter 8 of the readings.

Testing

We have provided two test scripts, which you can run to check your implementation. It will import *RStack.py* and *RQueue.py* and run a bunch of tests. The partial implementation will cause a number of failed tests, and a run-time error. You task is to implement the operations so that every test passes. You are encouraged to browse the test cases, and see how they were implemented, as you'll be doing similar testing in the future. You are not required to add more test cases to the script, but you are permitted to do so, especially if you're debugging.

The markers will apply a similar set of tests, and your grade will depend on how many tests pass.

What to Hand In

Hand in your implementation of *RStack.py* and *RQueue.py*. Don't rename it. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: Your implementation passes all tests.

Question 2 (10 points):

Purpose: To work with ADTs on the ADT programming side.

Degree of Difficulty: Moderate. The implementation is not difficult, but there may be some initial confusion about the difference between the Queue ADT as presented in Chapter 10, and the one you are working on here.

Set-up

The Queue ADT as we have studied allows storage of any number of data values: the Queue is always big enough to store any amount of data. While this is useful for most cases, it's not always realistic. If we're programming very small devices like a smart watch, or a microprocessor controlling a refrigerator, memory is not going to be as generous as with a notebook or desktop computer. Furthermore, we can imagine simulations (like the M/M/1 simulation in Chapter 9) where an infinite queue is really unrealistic. Some coffee shops operate in the real world, and fire regulations prohibit infinitely many customers waiting in line!

Task

In this question, we'll implement a variant of the Queue ADT with a given maximum capacity (we'll call it a *finite capacity queue*, or FCQueue). With 2 exceptions, the operations on FCQueue have the same effect as the operations on a normal Queue. There are 2 exceptions:

1. The create operation takes an argument, an integer which defines the capacity of the FCQueue. Once created, an FCQueue cannot store more items than the capacity.
2. The enqueue operation will work normally so long as the number of values stored is less than the capacity. Any attempt to enqueue a value to a full queue will result in the new value being dropped, and not stored in the queue at all. In the real world, this is like a customer walking away from the coffee shop if the queue is too long.

Start by downloading the FCQueue ADT module `FCQueue.py`. It's a partial implementation of all the queue operations; there are function definitions, with precise interface documentation, but trivial behaviour. Your task is to complete this ADT so that it behaves as if it had a finite capacity. The ADT has the following operations:

- The `create()` operation returns a new empty queue with a given capacity `cap`. The code is given to you:

```
1  def create(cap):
2      """
3      Purpose
4          creates an empty queue, with a given capacity
5      Return
6          an empty queue
7      """
8      b = dict()
9      b['storage'] = list()  # data goes here
10     b['capacity'] = cap    # remember the capacity
11     return b
```

  Notice that the value returned is a record (dictionary), with 2 key-value pairs. The data should be stored in the list associated with `'storage'`. Don't change the capacity at all!

- The `is_empty()` operation returns True if the queue is empty. You'll have to look at the storage to complete this operation.

- The `size()` operation returns the number of elements stored in the queue. You'll have to look at the storage to complete this operation.
- The `enqueue()` operation adds a new data element to the back of the queue. However, if the number of values already in the queue is equal to the capacity given to the create operation (above), the value should not be added; only add the value if the capacity would not be exceeded. You'll have to look at the capacity and the storage to complete this operation.
- The `dequeue()` operation removes a data element from the front of the queue. You'll have to look at the storage to complete this operation. Note that this operation should be designed so that, if called when the FCQueue is empty, a run-time error is caused. This is not something that requires additional programming! If you observe the implementation of Queue in Chapter 10, you'll notice that a runtime error is caused when dequeue is called on an empty Queue, and it's just the result of asking for a value from an empty list. That's good enough for us for now.
- The `peek()` operation returns a reference to the data element at the front of the queue. You'll have to look at the storage to complete this operation. Note that his operation should be designed so that, if called when the FCQueue is empty, a run-time error is caused. See the above clarification.

To implement these operations, you can refer to the implementation of the Queue ADT given in the readings, Chapter 10. But the value returned by the `create()` operation is a record (i.e., a dictionary), not a simple list, so you'll have to adapt and merge the ideas together. Understanding this is the only difficult part of this question, and once you get it, the rest will fall into place easily.

Testing

We have provided a test script, which you can run to check your implementation. It will import `FCQueue.py`, and run a bunch of unit and integration tests. The partial implementation will cause a number of failed tests, and a run-tie error. You task is to implement the operations so that every test passes. You are encouraged to browse the test cases, and see how they were implemented, as you'll be doing similar testing in the future. You are not required to add more test cases to the script, but you are permitted to do so, especially if you're debugging.

The markers will apply a similar set of tests, and your grade will depend on how many tests pass.

What to Hand In

Hand in your implementation of `FCQueue.py`. Don't rename it. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: Your implementation passes all tests.

Question 3 (24 points):

Purpose: To practice working with node chains created using the Node ADT.

Degree of Difficulty: Easy to Moderate.

In this question you'll write three functions for node-chains that are a little more challenging. On Moodle, you can find a *starter file* called `a5q3.py`, with all the functions and doc-strings in place, and your job is to write the bodies of the functions. You will also find a test script named `a5q3_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

Use `to_string()` provided in `to_string_checker.py` to help you test and debug your functions.

(a) (6 points) Implement the function `count_chain()`. The interface for the function is:

```
def count_chain(node_chain):
    """
    Purpose:
        Counts the number of nodes in the node chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
    Return:
        :return: The number of nodes in the node chain.
    """
```

Note carefully that the function is not to do any console output.
A demonstration of the application of the function is as follows:

```
empty_chain = None
chain = node.create(1, node.create(2, node.create(3)))

print('empty chain has', count_chain(empty_chain), 'elements')
print('chain has', count_chain(chain), 'elements')
```

The output from the demonstration is as follows:

```
empty chain has 0 elements
chain has 3 elements
```

(b) (6 points) Implement the function `contains_duplicates()`. The interface for the function is:

```
def contains_duplicates(node_chain):
    """
    Purpose:
        Returns whether or not the given node_chain contains one
        or more duplicate data values.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Return:
        :return: True if duplicate data value(s) were found,
        False otherwise
    """
```

For this question, you are NOT allowed to use a List to store data values, instead consider using two walkers and a nested loop. This operation should simply return True if we find ANY duplicate values, or False otherwise. Pseudocode for the algorithm follows on the next page.

```
start a walker to walk along the node chain
while walker has not reached the end
   start a checker to walk from walker to the end of the chain
   while checker has not reached the end
      if checker's data value is the same as walker's, return True
if walker reached the end of the chain, return False
```

A demonstration of the application of the function is as follows:

```
chain1 = node.create(1,
         node.create(2,
         node.create(3,
         node.create(4,
         node.create(5)))))
print('Duplicates?', contains_duplicates(chain1))
chain2 = node.create(1,
         node.create(2,
         node.create(3,
         node.create(4,
         node.create(1)))))
print('Duplicates?', contains_duplicates(chain2))
```

The output from the demonstration is as follows:

```
Duplicates? False
Duplicates? True
```

(c) (6 points) Implement the function `insert_at()`. The interface for the function is:

```
def insert_at(node_chain, value, index):
    """
    Purpose:
        Insert the given value into the node-chain so that
        it appears at the the given index.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
        :param value: a value to be inserted
        :param index: the index where the new value should appear
        Assumption:  0 <= index <= n
                     where n is the number of nodes in the chain
    Post-condition:
        The node-chain is modified to include a new node at the
        given index with the given value as data.
    Return
        :return: the node-chain with the new value in it
    """
```

Note carefully that the index is assumed to be in the range from 0 to n, where n is the length of the node-chain. Your function does not have to check this (but you may use an assertion here). Given an index of 0, the function puts the new value first, and given an index of n, it puts the new value last.

A demonstration of the application of the function is as follows:

```
empty_chain = None
one_node = node.create(5)
chain7 = node.create(5, node.create(7, node.create(11)))
```

```
print('Before:', to_string(empty_chain))
print('Before:', to_string(one_node))
print('Before:', to_string(chain7))

print('After:', to_string(insert_at(empty_chain, 'here', 0)))
print('After:', to_string(insert_at(one_node, 'here', 0)))
print('After:', to_string(insert_at(chain7, 'here', 1)))
print('After:', to_string(insert_at(chain7, 'again', 4)))
```

The output from the demonstration is as follows:

```
Before: EMPTY
Before: [ 5 | / ]
Before: [ 5 | *-]-->[ 7 | *-]-->[ 11 | / ]
After: [ here | / ]
After: [ here | *-]-->[ 5 | / ]
After: [ 5 | *-]-->[ here | *-]-->[ 7 | *-]-->[ 11 | / ]
After: [ 5 | *-]-->[ here | *-]-->[ 7 | *-]-->[ 11 | *-]-->[ again | / ]
```

Note that the variable `chain7` above refers to a node-chain that has had two values inserted into it.

(d) (6 points) Before you submit your work, review it, and edit it for programming style. Make sure your variables are named well, and that you have appropriate (not excessive) internal documentation (do not change the doc-string, which we have given you).

What to Hand In

A file named `a5q3.py` with the definitions of the three functions. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 6 marks: Your function `count_chain()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to return the number of nodes in the chain correctly.
  - Works on node-chains of any length.
- 6 marks: Your function `contains_duplicates()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT (and does NOT use lists) to check if the node-chain contains any duplicate data values.
  - Works on node-chains of any length.
- 6 marks: Your function `insert_at()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to insert a new node with the given value so that it appears at the given index.
  - Works on node-chains of any length.
- 6 marks: Overall, you used good programming style, including:
  - Good variable names
  - Appropriate internal comments (outside of the given doc-strings)