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 6

Implementing the Linked List ADT

Date Due: July 12, 2019, 11pm **Total Marks: 50**

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

- **07/05/2019**: released to students

Question 0 (8 points):

Purpose: To force the use of Version Control in Assignment 6

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 6. 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 6.
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 6 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 A6 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.

What to Hand In

After completing and submitting your work for Question 1, open a command-line window in your Assignment 6 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 `a6-git.log`, or `a6-git.txt`

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

- 8 marks: The log file shows that you used Git as part of your work for Assignment 6. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per function for a total of at least 24 commits.

Question 1 (42 points):

Purpose: To build programming skills by implementing the linked list ADT. To learn to implement an ADT according to a description. To learn to use testing effectively. To master the concept of reference. To master the concept of node-chains. To gain experience thinking about different problem cases.

Degree of Difficulty: Moderate There are a few tricky bits here, but the major difficulty is the length of the assignment. Do not wait to get started! This question will take 10-12 hours to complete.

1. On Moodle you will find four Python scripts:
   - `node.py`
   - `LList.py`
   - `llist_unit_test.py`
   - `llist_integration_test.py`

   Download them and add them to your project for Assignment 6 (and to Git). See below for a section on how to use the test scripts.
2. It is your task to implement all the operations in the `LList.py` script. Currently, the operations are *stubs*, meaning that the functions are defined but do nothing useful yet. They return trivial values, and you'll have to modify them.
3. The Linked List operations are described in the course readings, in lecture, and below.

The linked list ADT

The Linked List ADT is very much like the node-based Queue ADT we studied in class. The `create()` operation is as follows:

```
def create():
    """
    Purpose
        creates an empty list
    Return
        :return an empty list
    """
    llist = {}
    llist['size'] = 0       # how many elements in the list
    llist['head'] = None    # the node chain starts here; initially empty
    llist['tail'] = None    # the last node in the chain
    return llist
```

A linked list is a dictionary with the following keys:

size This keeps track of how many values are in the list.

head This is a reference to the first node in the node chain. An empty Linked List has no node chain, which we represent with None.

tail This is a reference to the last node in the chain. If the list is empty, this is `None`.

The Linked List ADT operations

When you open the `LList.py` document, you will find a complete description of all the operations you have to implement. Here is a brief list of them, with a few hints:

create() Creates an empty Linked List data structure. This is already complete!

is_empty(alist) Checks if the given Linked List `alist` has no data in it.
Hint: Stack and Queue have this one.

size(alist) Returns the number of data values in the given Linked List `alist`.
Hint: Stack and Queue have this one.

add_to_front(alist, val) Adds the data value `val` into the Linked List `alist` at the front.
Hint: This is similar to Stack's `push()`.

add_to_back(alist, val) Adds the data value `val` into the Linked List `alist` at the end.
Hint: This is similar to Queue's `enqueue`.

value_is_in(alist, val) Check if the given value `val` is in the given Linked List `alist`.
Hint: Just walk along the chain starting from the head of the chain until you find it, or reach the end of the chain.

get_index_of_value(alist, val) Return the index of the given `val` in the given Linked List `alist`.
Hint: Like `value_is_in(alist, val)`, but if you find it, return the count of how many steps you took in the chain.

retrieve_data_at_index(alist, idx) Return the value stored in Linked List `alist` at the index `idx`. Walk along the chain, counting the nodes, and return the data stored at `idx` steps.
Hint: Start counting at index zero, of course!

set_data_at_index(alist, idx, val) Store `val` into Linked List `alist` at the index `idx`.
Hint: Walk along the chain, counting the nodes, and store the new value as data at the node `idx` steps in. Start counting at index zero, of course!

remove_from_front(alist) Removes and returns the first value in Linked List `alist`.
Hint: This is similar to Queue's `dequeue`.

remove_from_back(alist) Removes and returns the last value in Linked List `alist`. This is not similar to the Queue or Stack operations!

Hints: Break the problem into 3 parts, and get each part working and tested before you go on to the next part.

- First, deal with trying to remove from an empty list.
- Second, deal with removing the last value in a list if size is exactly 1, and no bigger.
- Third, deal with the general case. The last value is easy to obtain, but the node in front of it has to become the new end of the chain. You have to walk down the chain to do that! Draw a diagram, and convince yourself that you have to walk along the chain to the end of it to do this properly.

insert_value_at_index(alist, val, idx) Insert the data value `val` into Linked List `alist` at index `idx`. The new value is in the chain at `idx` steps from the front of the chain. The node that used to be at `idx` comes after the new node.
Hints: Break the problem into parts.

- Deal with cases where `idx` has an invalid value: too big, or too small.
- You can call `add_to_front()` if `idx` is 0. There is a similar use for `add_to_back()`.
- Deal with the general case. You have to walk down the chain until you find the correct place, and connect the new node into the chain. Draw a diagram, and convince yourself that you have to walk along the chain to the end of it to do this properly.

Note: If the given idx is the same as the size of the Linked List, at the value to the end of the list. This is most easily done using `add_to_back()`.

delete_item_at_index(alist, idx) Delete the value at index `idx` in Linked List `alist`. Here you unhook the node from the chain.
Hints: Break the problem into 3 parts, and get each part working and tested before you go on to the next part.

- Deal with cases where `idx` has an invalid value: too big, or too small.
- You can call `remove_from_front()` if `idx` is 0. There is a similar use for `remove_from_back()`.
- Deal with the general case. You have to walk down the chain until you find the correct place, and disconnect the appropriate node from the chain. Draw a diagram!

Because of the similarity between the Linked List ADT and the node-based Queue and Stack ADTs, some of the operations will be similar, if not exactly the same. You may borrow from Queue and Stack (the node-based implementations), being careful to realize that copy/paste may only be the start of your work, not the end!

How to FAIL to complete this question

Write all your code all at once, without any testing as you go. Then run the test scripts to find that you've got a lot of errors to fix. Good luck!

How to complete this question

Write one function at a time, starting with a simple node-chain version (like A5). Test your node-chain version, and when you are sure it works, adapt it for the linked list operation. Then run the unit test script, with a limit on the number of failures reported. Debug each operation one at a time, then when the unit test script reports no errors in the operation, move on to the next.

Testing using the given test scripts

We've provided two test scripts, one for unit testing, and one for integration testing. They are fairly well documented, so you can open them up and have a look. These scripts consist of definitions of simple Python functions, all of which have the same basic task:

1. First, create a context for the test. This usually involves creating a LList record somehow. As we're doing unit testing on the LList ADT, we sometimes create node and LList records without using their `create()` methods. This is valid in a test script, but not in an application; we can violate the ADT Principle in testing the ADT.
2. Second, check how one other operation works in the given context. In our scripts, we use assertions, which check for something that should be true, and which alert us when that thing is not true.

For example, here's a unit test:

```
def test_create_key_size():
    # each test function sets up an independent context for the test
    # here, we use the ADT to create a LList

    allist = List.create()

    # then the test uses assert, giving the test condition and the reason
    # here we check that the record returned has one of the necessary fields

    assert 'size' in allist, "create(): new LList record; 'size' MISSING!"
```

It's a simple test of the `create()` operation. A list is created, and then an assertion is used to check if the key `'size'` is one of the keys in the LList record (dictionary). Notice how the assertion replaces the familiar if-statement that we would expect to see in the testing we have done so far. An assertion like this only reports a problem if the condition is false, and the string indicates the reason for the test (check the key) and the problem (key is missing). This particular test will pass using the script `LList.py` that we've given you; it may fail if you tinker with the `create()` operation.

Here's a test from the integration test script:

```
def test_integration_add_to_back_is_empty():
    # an integration test tests how operations work together
    # first set up a list with a bunch of nodes in the node chain

    thellist = List.create()
    stuff = 'HEY STOPSIGN THANK-YOU TURN-AROUND DOING-DOING HORSESHOE TURTLE'.split()
    for word in stuff:
        List.add_to_back(thellist, word)

    # now check if a single aspect worked properly
    result = List.is_empty(thellist)
    assert result is False, "checking is_empty() after add_to_back(); returned True!"
```

This test creates a list, adds a bunch of strings to the list, then checks `is_empty()`.

The test scripts have no code that calls any of the functions, so if you run the script as a normal Python program, no testing is done. These scripts are intended to be conducted using a popular Python library called `pytest`, which is installed on all computer science systems, and should be part of the default installation when you installed Anaconda. It's primarily a command-line tool, meaning that we can run tests from the command line. The pytest application works as follows:

1. Running pytest on the command-line with no command-line arguments:

   ```
   UNIX$ pytest
   ```

   - reads all files in the current folder with the word test in the name
   - calls any function with the word test in the function name
   - watches for assertions, counts the ones that pass and the ones that fail
   - each failed assertion generates a full report, which displays the entire function, and the location of the error. See below for hints about how to manage the output.
   - unlike normal Python, pytest records an assertion failure, but does not stop
   - ends with a summary, e.g,
     `== 51 failed, 28 passed in 0.90 seconds ==`

2. Running pytest with a script named on the command-line

```
UNIX$ pytest llist_unit_test.py
```

   - reads only the named file, then does the above only for this file
   - you can put as many filenames as you like, but they should be test scripts.

3. Running pytest, using command-line arguments to limit the output:

```
UNIX$ pytest  --tb=no  llist_unit_test.py
```

   - this commandline argument NEEDS 2 dashes! It won't work if you use just one
   - eliminates almost all of the reporting, just reports progress, and final result
   - runs all test scripts if no script given on command line

4. Running pytest to limit the testing:

```
UNIX$ pytest  --maxfail=10  llist_unit_test.py
```

   - this commandline argument NEEDS 2 dashes! It won't work if you use just one
   - keeps working until 10 test failures are accumulated, then stops
   - saves you time when you only have completed some of the operations
   - you can change the value to anything you
   - when maxfail=1, you just get the first failure.

5. **You can use pytest in PyCharm, too**. Just do the following:
   - Open the Settings/Preferences | Tools | Python Integrated Tools settings dialog
   - In the Default test runner field select "pytest" It may appear as "py.test"
   - Click OK to save the settings.

   Now you can choose to run this script using pytest from PyCharm's Run menu! PyCharm will organize the test results in a new interface panel at the bottom of the window. You can click on any of the tests, and see the report for that test. PyCharm's interface allows you to choose to hide or show tests that pass, and you can even choose to let PyCharm run our tests automatically as you type.

When you run the unit test script on the file as given on Moodle, you should see that 54 of the tests pass (and about 100 tests fail) before you have added any code to `LList.py`. That's because the function stubs we've written give the correct answer for those 54 test cases. Likewise, 17 of the 79 integration tests should pass before you've added any code. The tests that are passing by default may fail if you add code that does not work properly.

The pytest application generates a lot of output, and if you run it on the command-line you'll probably only see the end of the output. Limiting the output while you work through unit testing is certainly a good idea.

You may write your own test scripts (using the simple format we've used before, or you can try your hand at pytest). Feel free to use A5Q1 (`to_string()`) if you need visual assistance (we advise you don't display a LList without any formatting; it probably won't be too helpful to you). When you think you've got a working operation completed, run the unit test script. When you think you have all the operations working and passing the unit tests, run the integration test script.

You will know you are done when you see something like this:

```
UNIX$ pytest --tb=no
...
=== 233 passed in 0.41 seconds ===
```

Be careful! It's fairly easy to write a loop that never terminates, and you might think `pytest` stopped early, when it's stuck half way through. If you don't see a final report, pytest is still running.

What to Hand In

Hand in your `LList.py` program, named `LList.py`. It should contain only the Linked List ADT operations, and nothing else (no test code, no extra functions).

Do not submit `llist_unit_test.py`, `llist_integration_test.py` or `node.py`.

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

Note: If you submit a file that is not named `LList.py` exactly, you may receive zero marks. Our scoring script will use `import LList as List`, and if your submission is named something else, the import will fail and we will not try to figure why.

Evaluation

- Your solution to this question must be named `LList.py`, or you will receive a zero on this question.
- 12 marks: You completed an implementation for all 12 operations. One mark per operation. The implementation does not need to be correct, but it should be relevant, and it should be code written with an effort to demonstrate good style and internal documentation. The mark will be deducted if there is no implementation, or if the function does not demonstrate good programming style, or if your function violates the Node ADT.
- 30 marks: When our scoring script runs using your implementation of `LList.py`, no errors are reported. Partial credit will be allocated as follows:

Number of tests passed	Marks	Comment
0-71	0	the given LList script already passes 71 tests total
72-100	5	
101-125	10	using the hints about Stack and Queue gets you here
126-150	15	
151-175	20	passing all unit tests gets you here at least
176-200	24	
201-232	27	this is actually pretty good
233	30	

For example, if your implementation passes 209 tests, you'll get 27/30. If your implementation passes 156 tests, you'll get 20/30. Our test script is based on the test scripts distributed with the assignment, but may not be exactly the same.