Assignment 6 – Solutions and Grading

Linked Lists

Date Due: Wednesday, July 14, 11:59pm **Total Marks: 42**

Question 1 (42 points):

Purpose: Students will practice the following skills:

- 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 special 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 5-8 hours to complete. If you are unlucky, and get stuck, it could be longer than that.

References: You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes
- Chapter 16: Node-based Stacks and Queues
- Chapter 17: Linked Lists

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task Overview

1. On Moodle you will find two Python scripts:
   - `LList.py`: A starter file for the LList ADT mentioned in Chapter 17.
   - `score.py`: A script with a large number of test cases.

   Download them and add them to your project for Assignment 6. See below for a section on how to use the score script.
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. You do not need to add methods. The Linked List operations are described in the course readings, in lecture, and below.
3. Your grade on this assignment will largely depend on how many test cases your script passes. See below for more details.

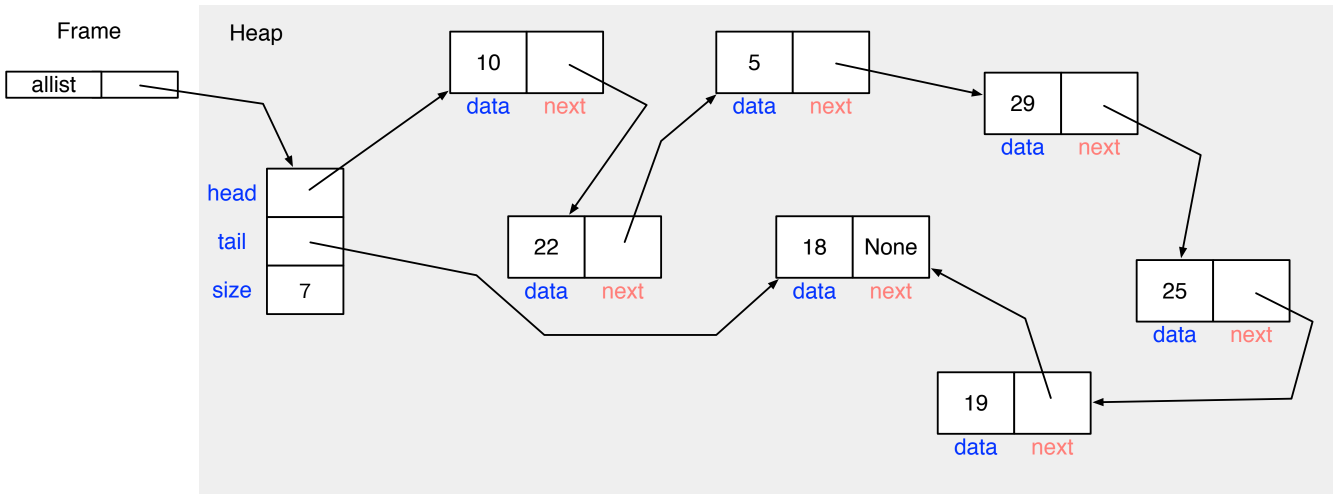

Figure 1: A frame diagram for a typical example of a non-empty linked list.

The Linked List ADT operations

The LList ADT is described int he textbook, but we will review the operations here. 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:

__init__() Creates an empty Linked Listobject.
Hint: This is already complete! You don't need to change it.

is_empty() Checks if the Linked List object has no data in it.
Hint: Stack and Queue from Chapter 16 have this one. Yes, you can use it.

size() Returns the number of data values in the Linked List object.
Hint: Stack and Queue from Chapter 16 have this one. Yes, you can use it.

add_to_front(value) Adds the data value `value` into the Linked List object at the front.
Hint: This is similar to Stack's `push()`. Yes, you can use it, but you will have to make some modifications.

add_to_back(value) Adds the data value `value` into the Linked List object at the end.
Hint: This is similar to Queue's `enqueue()`. Yes, you can use it, but you will have to make some modifications.

remove_from_front() Removes the first node from the node chain in the Linked List object. Returns a tuple `(True, value)`, where `value` is the data value stored in the removed node. If the Linked List is empty, it should return `(False, None)`.

- For example, calling `allist.remove_from_front()` in Figure 1 would remove the node containing 10 from the node chain, and return the tuple `(True, 10)`.

Hint: This method is similar to Queue's `dequeue` (from Chapter 16), except for a few details. Yes, you can use it, but you will have to make some modifications.

get_index_of_value(value) Returns a tuple `(True, index)`, where `index` is the index of `value` in the Linked List object. If `value` is not found, it should return `(False, None)`.
Hint: Just walk along the chain starting from the head of the chain until you find it, or reach the end of the chain. If you find it, return the count of how many steps you took in the chain. The index of the first value in the list is 0, just as in normal Python lists.

value_is_in(value) Check if the given value `value` is in the Linked List object.
Hint: It's very similar to `get_index_of_value(value)`.

retrieve_data_at_index(index) Returns a tuple `(True, value)`, where `value` is the data value stored at index `index`. If the index is not valid (negative, or too large), it should return `(False, None)`

- For example, calling `allist.retrieve_data(2)` in Figure 1 would return the tuple `(True, 5)`. The node chain is not changed.

Hint: Walk along the chain, counting the nodes, and return the data stored at `index` steps. Start counting at index zero, of course!

set_data_at_index(index, value) Store `value` into the Linked List object at the index `index`. This method does not change the structure of the list, but will change the data value stored at a node. Returns `True` if the index is valid (and the data value set), and `False` otherwise.

- For example, calling `allist.set_data_at_index(5, 24)` in Figure 1 would change the data value in the sixth node from 19 to 24. The node chain is not changed.

Hints: Break the problem into 2 parts.

- First, deal with trying to modify an empty list.
- Second, deal with the general case. Walk along the chain, counting the nodes, and store the new value as data at the node `index` steps in. Start counting at index zero, of course!

Errandum: (08/07/2021) A previous version gave misleading hints for `set_data_at_index()`.

remove_from_back() Removes the last node from the node chain in the Linked List object. Returns a tuple `(True, value)`, where `value` is the data value stored in the removed node. If the Linked List is empty, it should return `(False, None)`.

- For example, calling `allist.remove_from_back()` in Figure 1 would remove the node containing 18 from the node chain, and return the tuple `(True, 18)`.

Hints: Break the problem into 3 parts.

- 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, a node chain has more than one node. It's easy to remove the last node, but it's harder to set the attribute `_tail` correctly. You have to walk along the chain and find the node that will become the new tail.

Errandum: (08/07/2021) This version added hints for `remove_from_back()`.

insert_value_at_index(value, index) Insert the data value `value` into the Linked List object at index `index`. Returns `True` if the index is valid (and the data value set), and `False` otherwise.

- For example, calling `allist.insert_value_at_index(3, 12)` in Figure 1 would add a new node with data value 12 between the nodes containing 5 and 29.

Hints: Break the problem into parts.

- Deal with cases where `index` has an invalid value: too big, or too small.
- You can call `add_to_front()` if `index` 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. The new value is in the chain at `index` steps from the front of the chain. The node that used to be at `index` comes after the new node.

Note: If the given index is the same as the size of the Linked List, add the value to the end of the node-chain. This is most easily done using `add_to_back()`.

delete_item_at_index(index) Delete the node at index `index` in the Linked List object. Returns `True` if the index is valid (and the node deleted), and `False` otherwise.

- For example, calling `allist.delete_item_at_index(4)` in Figure 1 would remove the node containing the value 25; the node-chain would point from the node containing 29 to the node containing 19.

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 `index` has an invalid value: too big, or too small.
- You can call `remove_from_front()` if `index` 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.

What to Hand In

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

Do not submit `score.py`. Do not import any modules in your LList ADT. It has to be totally independent, except for the node class, which is defined in `LList.py` already.

Be sure to include your name, NSID, and student number at the top of `LList.py`. For your submitted version, you can replace the "copyright" notice with your personal identification information.

Evaluation

- Your solution to this question must be named `LList.py`, or you will receive a zero on this assignment.
- 12 marks: **Programming Style**. You completed an implementation for all 12 operations. One mark per operation. The implementation is not being graded for correctness, but it should be relevant, and it should be code written with an effort to demonstrate good style and internal documentation. Marks will be deducted if there is no implementation, or if the function does not demonstrate good programming style.
- 30 marks: **Correctness**. When our scoring script runs using your implementation of `LList.py`, we'll run using verbose mode turned off, and all test cases attempted. We'll score the correctness of your implementation using the following table.

Number of tests passed	Marks	Comment
0-42	0	the given LList script already passes 42 tests total
43-77	5	
78-102	10	If you only take code from Stack and Queue, you get here
103-152	15	
153-202	20	
203-242	24	
243-252	27	this is really good!
253-257	30	close enough to perfect!

For example, if your implementation passes 249 tests, you'll get 27/30. If your implementation passes 172 tests, you'll get 20/30. The score script you have already tells you this information, so you will know your mark for correctness when you hand it in.

The marker's test script is based on the score script distributed with the assignment, but may not be exactly the same.

Solution: A model solution appears in the file: `LList_solution.py`.

Marking guidelines:

- Each operation should have an implementation. The initial file had trivial operation implementations, so check that something was added to each.
- Each of the operations gets a mark. The mark is for good style, not correctness.
  - Browse the Readings Chapter 14 to see the kinds of things to look for.
- If the operation misuses LList ADT (for example, if the list is sent to a Node operation), deduct the mark.
- To grade the implementation:
  - Copy the `score.py` script provided to the student's submission folder downloaded from Moodle.
  - If a score script already exists, replace it. Don't use any score script submitted by the students.
  - Run `python` as follows:

```
UNIX$ python score.py
----------------- Summary ------------------
LList score: 30/30
Tests passed: 257/257
--------------------------------------------
```

  - Simply transcribe the `LList score:` to the Moodle grading.
  - Because Moodle rubrics are not very pleasant with more than 5 grade levels, I have opted to implement the grading in "tiers".
  - Example 1: if the student code passed 57 tests:
    - Click "43-77 test cases passed" in Level 1.
    - Click "< 153 test cases passed" in Level 2.
  - Example 2: if the student code passed 157 tests:
    - Click "> 103 test cases passed" in Level 1.
    - Click "153-202 test cases passed" in Level 2.
  - More detail about the behaviour of the score script:
    - The script will detect if the submission file is present or not. If the student did not name the script `LList.py`, or if the script was not submitted, the score will be 0/30.
    - If the script is properly named, the script should count the number of tests passed, and output a score, as above.
    - If the student's program has an infinite loop, the script will (hopefully) terminate the operation, and move to the next test.
    - Run-time errors in the students' code will cause a test failure, but the test script will keep trying the rest of the tests.
    - The score script has some controls that turn on and off a lot of output. Hopefully you will not need to explore those. The "marking" settings are:

```
verbose = False # True turns on the verbose behaviour.
failure_limit = 0  # 0 causes all tests to run
```