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

Linked Lists

Date Due: Sunday July 1o, 11:59pm **Total Marks: 42**

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.
- This assignment is homework assigned to students and will be graded. This assignment shall not be distributed, in whole or in part, 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 assignment 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.**
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to com-pare every pair of student submissions. Plagiarism can include: copying answers from a web page, or from a classmate, or from solutions published in previous semesters. Basically, if you cannot delete your whole assignment and do it again yourself (given adequate time), it's not your work, so don't try to claim credit for it. Your success in this course depends on what you can do, not on what you can hand in.
- 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.
- Read the purpose of each question. Read the Evaluation section of each question.
- 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 Canvas.** If you are not sure, talk to a Lab TA about how to do this.
- **Canvas 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. **Do not send late assignment submissions to your instructors, lab TAs, or markers. If you require an extension, request it in advance using email.**

Version History

- **07/08/2021**: An updated score script was released, with some typos corrected, a few minor formatting tweaks, and some variable renaming to avoid hyper-active IDEs from making a mess in your edit window. To ensure that the assignment description matches what you see when you run the script, the example output was revised. Also, a slight revision was made to the note on the last page.
- **07/08/2021**: Revised the hints for `set_data_at_index()` and added hints for `remove_from_back()`.
- **07/07/2021**: released to students

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 Canvas 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.

The node data structure

The starter file defines the node class, so that we do not need to import the node class. Let's have a look:

```
class node(object):

    def __init__(self, data, next=None):
        """
        Create a new node for the given data.
        Pre-conditions:
            data:  Any data value to be stored in the node
            next:  Another node (or None, by default)
        """
        self.data = data
        self.next = next

    # Note: use the attributes directly; no setters or getters!
```

The attributes for this version of the node class are *public*! Therefore we can use the attributes directly without needing to call getters and setters like `get_data()`. In fact, this version of the node class does not define any getters and setters. Because this class has no protected or private data, and because there are no methods, there is no abstraction protecting the node objects, and therefore we call this class a *data structure*; it's not strictly speaking, an ADT, but no one will be too upset if you call it an ADT casually.

This version of node makes implementing the LList operations a little easier. First, we no longer need to use the setters and getters from the version of node in Chapter 14, in which the attributes were private. Second, we don't need to import the node module, and use dot-notation to use the node class constructor. For example, we can write code like this within the LList class:

```
anode = node(1)
anode.data = 2
bnode = node(3)
bnode.next = anode
```

It's less cluttered, which helps us a bit. We'll re-use this simplification in future chapters as well.

The linked list ADT

Lecture 17 introduced the LList ADT, and we're going to spend this assignment implementing the operations. The LList ADT is not a replacement for Python lists. The purpose here is to get good programming practice with nodes, and creating ADTs that are more than a few lines of code. The experience doing this assignment is very valuable. The skills you build are applicable to future chapters and future courses. Every first year student in every university does something like this (the language might be different, but linked lists are universal).

Let's review. The `__init__()` operation is as follows:

```
class LList(object):
    def __init__(self):
        """
        Purpose
            creates an empty LList object
        """
        self._size = 0      # how many elements in the stack
        self._head = None   # the node chain starts here; initially empty
        self._tail = None
```

A linked list is an object with the following attributes:

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`.

Notice that the LList attributes are *protected* in the given starter file, but private in the textbook. This helps us write test cases that would not be possible with private attributes.

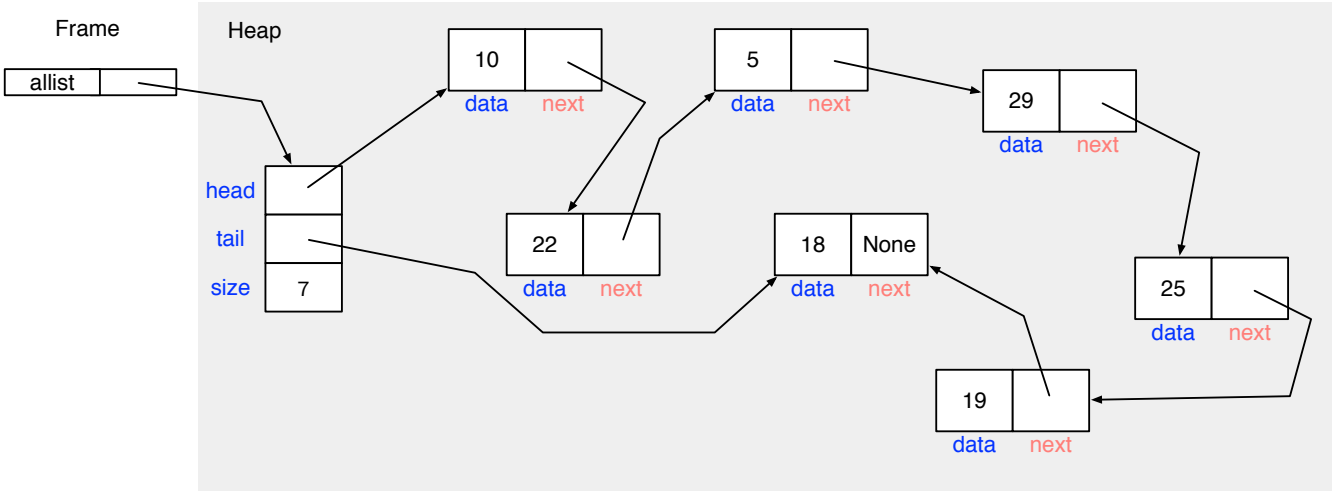

Figure 1: A frame diagram for a typical example of a non-empty linked list.

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.

Advice and notes

The main task in this assignment is to do all the programming for the LList ADT operations. However, another aspect of this assignment is the testing. A node-based implementation of 12 operations that can be used in arbitrary combinations in applications requires significant testing. Each method needs to be tested on its own, and in combination with other methods.

You will not get your implementation correct the first time, and you will have to debug. You will encounter faults that confuse you, and you will have trouble figuring it out. **All this is part of the exercise. You have to work thorough this, and learn from it. If you give up, or worse, you will not be prepared for what comes after this assignment.**

How to FAIL to complete this assignment

Implement all the operations 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. If you do this, you are setting yourself up for massive, epic failure.

How to complete this assignment

Implement one operation at a time, in the order of the description on a previous page. Then run the given score script (see below for more information). Debug the operation before moving on to the next one.

Testing using the given score script

The score script includes unit testing, and integration testing. It's a very large script, with more than 250 test cases. You do not need to read every test case, but it is instructive to browse the test cases.

Note: The first 200 lines of the score script is not examinable in CMPT 145; this part is very clearly marked. It uses threads, which you will study lightly in CMPT 270, and other techniques that you will learn in later courses. You need only the lightest superficial understanding of this code, so that you will be confident in the use of the script.

The score script does some nice things.

- It can show a lot of detail, or just a minimal amount, depending on what you want.
- It can run all the tests, or just a few until failures start piling up.
- It tries to advise you about what the problem is. It might be a run-time error in your code, or a test case failure, or an infinite loop. Yes, it will interrupt your infinite loops! More information about these three problems appears below.
- It tells you your "correctness" score on the assignment so far (see **Evaluation** below).

You can control the behaviour of the script by changing two variables in the score script:

```
verbose = True  # True turns on the verbose behaviour.
failure_limit = 0   # How many test failures will cause the script to stop trying
```

The variable `verbose` controls how much output there is. The variable `failure_limit` controls how many failures can accumulate before the script stops trying more. If you put a zero here, it will run them all. If you use the value 3, the script will run as many tests as it can, but when there are 3 failures, it will stop. You will want to edit these values to modify the behaviour of your script as you work. A useful configuration to start with is `verbose = True` and `failure_limit = 1`.

There are 257 test cases implemented in the script. Each LList operation is carefully tested. There are unit test cases, where each operation is tested with limited dependence on other operations. There are also integration tests as well, which attempt to use operations in combination. Each test case has the following:

1. Its own function. This prevents accidental dependencies between test cases. Everything needed by a test case is defined in the function.
2. A mention of the operation being tested (`test_item`).
3. A reason for the test, expressed as a string (`reason`) that explains what was tested.
4. A set up for the test, called a *test harness*. For this assignment, this involves creating a LList object somehow. For unit tests on the LList ADT, we sometimes create node and LList objects without using their methods. This is valid in a test script, but not in an application, of course. For integration tests, we construct LList objects using the LList operations that we've already tested with unit testing.
5. A method call for the item being tested, whose output is recorded using the variable `result`.
6. An expression for the expected answer, recorded using the variable `expected`.
7. An assertion to check if the result is equal to what was expected.

For example, here's a unit test:

```
def test_001():
    test_item = "LList()"
    reason = "new LList object _size should be 0"
    a_llist = L.LList()
    result = a_llist._size
    expected = 0
    assert result == expected, assert_report.format(test_item, reason, result, expected)
```

It's a simple test of the `LList()` operation (the class constuctor, which calls `__init__()`). A LList object is created, and then an assertion is used to check if the size is equal to zero. 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. This particular test will pass using the script `LList.py` that we've given you. The only reason that this test would fail is if you modify (badly) the `__init__()` operation.

Note: the variable `assert_report` refers to a string in the score file:

```
assert_report = "{}: {}; found '{}', expected '{}'"
```

All test cases use it, to provide a standard look to all test case failure output. The assertion statement in all the test cases is exactly the same. It compares some result to an expected value.
Here's an integration test:

```
def test_177():
    test_item = "multiple add_to_back() followed by is_empty()"
    reason = "LList should not be empty"

    # an integration test tests how operations work together
    # first set up a LList with a bunch of nodes in the node chain
    the_llist = L.LList()
    stuff = 'HEY STOP SIGN THANK YOU TURN AROUND DOING HORSE SHOE TURTLE'.split()
    for word in stuff:
        the_llist.add_to_back(word)

    # now check if a single operation works properly
    result = the_llist.is_empty()
    expected = False
    assert result == expected, assert_report.format(test_item, reason, result, expected)
```

This test creates a list, adds a bunch of strings to the LList object, then checks `is_empty()`. After a bunch of things have been added, `is_empty()`should return `False`, as the reason implies. Sometimes the reason is implicit about the test case.

When you run the test script on the starter file as given on Moodle, you should see that 42 of the tests pass before you have added any code to `LList.py`. That's because the function stubs coincidentally give

the correct answer for those 42 test cases. These test cases are included in the score script because it is possible for an implementation to make things worse!

You may also write your own test scripts (using the simple format we've used before, or you can try your hand at using the style here). When you think you've got a working operation completed, run the test script.

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

```
...
----------------- Summary ------------------
LList score: 30/30
Tests passed: 257/257
--------------------------------------------
```

If you have the script in verbose mode, this information may be crowded by other output, but it will be near the end of the output.

Understanding the score script output

The script tells you when you have problems, but not how to fix them. The first step is understanding the problem.

Test case failures One of the things that the score script could tell you is that a test case failed. It will look like this:

```
Progress: 1 tests passed, out of 1 attempted.
Progress: 2 tests passed, out of 2 attempted.
Progress: 3 tests passed, out of 3 attempted.
***Test failure*** in test case: test_004
---the test case failed
---your function did not do the right thing (incorrect return value or improper post-condition)
LList() + is_empty(): new LList object should be empty; found 'None', expected 'True'
----------------- Summary ------------------
LList score: 0/30
Tests passed: 3/4
--------------------------------------------
-*-*-Halted after 1 failures.  Partial score reported above!-*-*-
```

For this example, the script was configured to stop after the first error. Most of this text comes from the score script. It tells you the following information:

- Three test cases passed before the first failure.
- The test case that failed is `test_004()`. You can find this function in the score script.
- The test item was `LList() + is_empty()`, meaning that a new LList was created, and then `is_empty()` was called.
- This test case is checking the output from `is_empty()`. It should return `True`, but it actually returned `None`. In the starter file, all the operations return None, initially.

When you see `***Test failure***`, it almost certainly means that the assertion in the test case failed, and that is almost certainly caused because your function did not return the expected answer.

Now you have to look at your code and figure out why `is_empty()` is returning `None`. For more advanced test cases, you will need to read the test case, to figure out the inputs and the expected output. It will take a tiny bit of practice to read the test cases, but test cases have a lot of similarity. The time you invest understanding one test case will pay off, since probably many test cases use a similar set up.

Note: The test cases were tested using a complete implementation of the LList ADT. However, it's possible that there are errors in the score script that have not been detected yet. If you think you have found one, you may report it to your instructor. Be sure to mention the test function number, and some explanation of the problem you found.

Run-time errors If your function causes a run-time error, it's counted as a problem, but not displayed as a test case failure. A run-time error occurs if your function crashes, and can't return. Normally, Python would display the run-time error, and you'd recognize it. However, the score script interferes with Python's normal behaviour. To make up for this interference, the script tries to pass on some of the information Python would normally tell you.

For example, suppose we implemented `is_empty()` as follows:

```
    def is_empty(self):
        return self.data + 1
```

This makes no sense, because a LList object does not have a `data` attribute. So this implementation will cause a runtime error if the function is called. The score script's output looks a little different:

```
Progress: 1 tests passed, out of 1 attempted.
Progress: 2 tests passed, out of 2 attempted.
Progress: 3 tests passed, out of 3 attempted.
***Runtime error (Exception) in test case: test_004
---the test case did not complete
---Probable cause: one of your functions called in this test crashed before it could return an
answer
---Possible alternative: one of your functions called in this test returned a value of the wrong
type
Traceback (most recent call last):
  File ".../score.py", line 163, in runemall
    runnable()
  File ".../score.py", line 103, in inner
    result = fn(*args, **kwargs)
  File ".../score.py", line 268, in test_004
    result = the_llist.is_empty()
  File ".../LList.py", line 57, in is_empty
    return self.data + 1
AttributeError: 'LList' object has no attribute 'data'

----------------- Summary ------------------
LList score: 0/30
Tests passed: 3/4
--------------------------------------------
-*-*-Halted after 1 failures.  Partial score reported above!-*-*-
```

Again we see that 3 test cases succeeded, and the 4th one failed. But this time, it's not a test case failure, but a runtime error `AttributeError`. The score script displays Python's normal information about the run-time error. It points to a problem on line 268 in the score script, which identifies which test case is involved (`test_004()`), but also to line 57 in the LList implementation, which is the method `is_empty()`.

Added Note: The report mentions two functions called `runemall()` and `error()`. These functions were part of execution that lead up to the error, but they probably didn't cause it. When Python encounters an error, there may be several functions that have been called. The function calls (and the frames they create) really are organized as a stack, and the output from a runtime error reflects that. You want to focus your attention on the bottom of the stack! In some cases, the bottom of the stack might be the test case function, or it might be a LList operation you are working on. The functions at the bottom of the stack are also at the bottom of the report! In this case, the the expression `return self.data + 1` that is at the bottom. The problem we created above was that there is no data attribute, right? **In any case, do not try to debug any of the code in the first 200 lines of the score script. Even if these functions are included in the error report, it's almost certain that this code is not causing your errors. If you think you've found a situation that suggests a bug in this code, let the instructor know.**

Interrupting infinite loops If you happen to create an infinite loop in one or your methods, you will get a different report:

```
Progress: 1 tests passed, out of 1 attempted.
Progress: 2 tests passed, out of 2 attempted.
Progress: 3 tests passed, out of 3 attempted.
test_004 took too long
---Had to terminate a function used in test case: test_004
------took too long to finish
------Probable cause: one of your functions called in this test has an
infinite loop
Total tests: 4 Tests passed: 3
-----------------------------------------
---LList score: 0/30
Tests passed: 3/4

-*-*Halted after 1 failures.  Partial score reported above!*-*-
```

The score script does not try to figure out where the infinite loop is, except to point to the test case that was interrupted (namely `test_004()`). Use wolf-fencing within the methods used in the test case to try to identify the problematic method.

The code used by the score script to interrupt a long-running test case is not a general solution to detecting infinite loops. It only works in this assignment because each test case should require no more than a few milliseconds to complete. So it's pretty safe to assume that a test case requiring more than 2 seconds has an infinite loop somewhere. However, there are lots and lots of computational tasks that require more than 2 seconds. The problem with stopping a function after a given, fixed amount of time is that it is generally impossible to know if the function would have returned an answer after the time limit, or if it is actually in an infinite loop. You'll learn more about this in later courses.

Strategies for debugging

The score script can tell you there is a problem, but not what the problem is. It is useful to think about how to use the score script to help you debug your code.

1. Read the code (function) for the failed test case. Look at the code that sets up the data, and be clear about what the input to the operation is, and what the expected output is. The test case code is written using CMPT 145 material, and you are expected to be able to read each test case function.
2. Use a break point in the test case function. You can start stepping through the test case, without stepping through the whole score script.
3. You can isolate certain test cases by commenting out test cases. The best way to do this is by jumping to the end of the score script. You will find a very long list with the names of all the test case functions in it. Comment out the test cases functions you don't want to run.
4. You can create informal test cases of your own design. Do this in a separate Python script, in which you import LList. There is value in informal testing, but not to the exclusion of formal testing.
5. Draw diagrams of node chains and LList objects, as shown in Figure 1. Draw the LList object before the operation, and after the operation. Use the diagrams to help you figure out how to design the algorithms for the operations. Return to these diagrams to step through your own code, by hand.
6. Copy the problematic test case to another script (don't forget the `assert_report` string). You can import LList, and call the test function directly. This will apply the test case directly, giving you back all of Python's normal behaviour, but you will lose the ability of the score script to interrupt infinite loops.