University of Saskatchewan
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephone: (306) 966-4886, Facimile: (306) 966-4884

CMPT 145
Winter 2018
Principles of Computer Science

Assignment 6

Implementing the Linked List ADT

Date Due: March 9, 2018, 10pm **Total Marks: 50**

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`.

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. And frankly, if you only have 12 commits, you're pretending.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 4 marks for good commit messages; 2 marks for anything else.
- Students were told at least 24 commits (2 per function).
- I don't want you counting commits very carefully. More than 10 commits is worth 4 marks, and 2 marks if there are not even 10 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 three documents: `node.py`, `LList.py`, and `score_llist.py`. Download them and make a new Python project for Assignment 5.
2. It is your task to implement all the operations in the `LList.py` document. Currently, the operations are "stubs" meaning that the functions are defined but do nothing useful yet.
3. Immediately run `score_llist.py`, and notice the output. There is a lot of output, about 200 lines, because the Linked List ADT is only partially implemented. It's a kind of test script, but it may not be as useful to you as a test script as your own scripts. For one thing, the error messages displayed try to point out where a problem occurred, but it doesn't use the formal test case format we've been suggesting that you use.
   When you run the score script, you should see that 64 of the 216 tests pass before you have added any code to `LList.py`. That's because the function stubs we've written give the correct answer for those 64 tests. The tests that are passing by accident may fail if you add code that does not work properly.
4. 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 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!

Assignment 5 asked you to write functions on node-chains. It also gives you a good strategy for completing this question. Write a node-chain version of the operation, and when that works, adapt it for the Linked List ADT.

Write your own test scripts, and use A5Q1 (`to_{string()`) if you need visual assistance. When you think you've got a working operation completed, run the `score_llist.py` script. You will know when you are done when the `score_llist.py` reports 216 of 216 tests passed, i.e., reporting no error.

Be careful! It's fairly easy to write a loop that never terminates, and you might think `score_llist.py` stopped early, when it's stuck half way through. The score script tries its best to perform all the tests, but it is certainly possible for it to crash half way through, or get caught in one of your infinite loops.

How to FAIL to complete this question

Write all your code all at once, without any testing as you go. Then run `score_llist.py` to find that you've got a lot of errors to fix. Good luck debugging that.

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. Run a few tests that you wrote yourself, to catch simple problems. Then run `score_llist.py`.

The score script has a counter ADT within it, which can allow you to set a limit on the number of errors. It might be helpful at the start of your work to set the limit to 10 errors, so that the score script will halt when 10 errors are found. Near the end of your work, you can set the limit back to 0 (meaning to report all errors).

What to Hand In

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

Do not submit `score_llist.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 List ADT, or~~ 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
0-64	0
65-100	5
101-130	10
131-175	15
176-190	19
191-200	24
201-217	27
218	30

  For example, if your implementation passes 209 tests, you'll get 27/30. If your implementation passes 156 tests, you'll get 15/30. Our test script is based on `score_llist.py`, 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.
- If the operation violates the Node ADT, deduct the mark.
- If the operation misuses LList ADT (for example, if the list is sent to a Node operation), deduct the mark.
- Run the script `score_llist.py`. The script should output a total number of errors, and a total number of tests, e.g.

  ```
  *** 64 of 216 tests passed
  Final Count for Linked List ADT: 64 of 216 tests passed
  ```

  The first number is the one you should look at, e.g., 64. Some tests are conditional, which means that the total number of tests might be 216 or 218.
- It's possible that a runtime error halts the program abnormally, or that some list operation runs into an infinite loop. Use the numbers on the last line that gets produced by the script.

Extra work for no credit

If you have time, implement the following functions:

clear(alist) (Easy) Removes all the values from the Linked List `alist`.

extend(alist, blist) (Easy) Extends the Linked List `alist` by adding all of the elements in Linked List `blist` at the end.

slice(alist, start, end, step) (Moderate) Create a new Linked List from the given Linked List `alist`, by including the values starting from index `start` going up to but not including index `end` with a given step-size `step`. Assume that `end` is not before `start`, and that `step` is positive.

sorted(alist) (Tricky) Rearranges the data in Linked List `alist` into increasing order.