CS 120 (Fall 21): Introduction to Computer Programming II

Short Project #5 - Intro to Linked Lists

due at 5pm, Thu 23 Sep 2021

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

NEW RULE: In this class, using `try/except` to avoid having to check for `None` is **banned.**

1 Overview

This project is your first one to work with Linked Lists. You will be writing a handful of functions that manipulate lists; you will also be writing one function which implements a shape, to match a reference diagram that I've given you.

And yes, every "list" in this project refers to a linked list, **not** an array.

2 Basic Requirements

You will need to submit two Python files. The first, named `linked_list_short.py`, will contain several functions: `list_to_array()`, `array_to_list()`, `list_length()`, `is_sorted()`, and `accordion()` . The second, named `too_many_aliases.py`, will contain a single function, `too_many_aliases()` .

2.1 Rule: No Creating or Changing `ListNode` Objects

Except in `array_to_list()`, you must never create a new `ListNode` object, or modify its value. Instead, your code should be focused on changing the arrangement of the nodes - that is, changing the `next` pointers.

2.2 Rule: No Temporary Arrays

Sometimes, students are tempted to solve linked list problems by building a temporary array to hold their nodes (or the values). They then solve the problem using arrays, and finally convert things back to linked lists.

While there's nothing wrong with doing this in other situations, this practice is **banned** for this project. I want you to practice with **working with lists.** It's a critical skill!

Of course, `list_to_array()` converts a linked list into an array, and `array_to_list()` does the opposite. But you can't use this trick in the othter functions!

2.3 Rule: List Class

All of the code in this project must import the file `list_node.py` , which I have provided. You **must not** modify this file. All of the list node objects that I provide you will be of the `ListNode` class, which is defined in that file.

Likewise, if you want to create any linked list objects, you must create them with ListNode. You can do so by calling the constructor, and passing it a value:

```
new_node = ListNode("abc123")
```

Students sometimes copy-paste the code for `ListNode` into their own file. This is **forbidden,** and will likely result in your failing testcases. Instead, `import` the code from `list_node.py` .

2.4 Other Notes

As always in this class, we will represent a linked list using a reference to the head node. Empty lists are allowed; they are represented by a reference to `None`.

3 Design and Debug Hints

Struggling how to get started with the code? Struggling how to debug a nasty problem? Check out these hints!

- **Use a Reference Diagram.**

  Don't waste time trying to write code - particularly, code that modifies a list - if you cannot draw the reference diagram first. Like we did in class, start by drawing a specific picture; that is, give yourself a simple input, and draw exactly what the code ought to do.

  But then, make sure to write a "generic" reference diagram, too! Show exactly what the code will look like, many iterations into the loop. What variables will you need? What can you assume about the state of the variables, and the input? What small change can you make, to make just a tiny bit of progress toward the solution?

- **Print Your List**

  Print out the contents of every list you have - and all the temporary variables - on every pass of the loop. Draw a reference diagram of the "before" and "after" states, on each iteration of the loop. Is the function doing what you expect it to?

- **Breaking the List**

  Sometimes, your code will "remove" a node from the list; you change some next pointers (or, you advanced the "head" pointer past it). But often, you keep the old node around, stored in a temporary variable, because you plan to use it later.

When you do this, it isn't necessary to strictly "remove" it from the list, but it's often useful: try setting the `next` pointer in the removed node, so that it's no longer connected to rest of the list.

If you don't do this, then (usually) it doesn't break anything, but sometimes you get confused - for instance, if you try to print the old node, you'll end up printing the entire list - even though this node has been "removed."

Worse, it's easy to make mistakes with such a node, and accidentally create a cycle. Can you find the bug in the code below?

```
head = ListNode(10)
head.next = ListNode(20)
head.next.next = ListNode(30)
# as of this point, the list has 3 elements: 10,20,30

# remove the head
tmp = head
head = head.next

# move the head back to the end of the list
head.next.next = tmp
```

Draw this up carefully in a reference diagram. What is the bug?

(spec continues on the next page)

4 `list_to_array(head)`

This function converts a linked list to an array containing the same values. The parameter is a linked list (perhaps empty); the return value must be an array, with the same values, in the same order.

Do not change the list that you were passed.

5 `array_to_list(data)`

This function converts an array to a linked list. The parameter is an array of values (perhaps zero ength); it must return a linked list, containing the same nodes, in the same order.

This is the only function (as part of this Short Project) where you will create new nodes; remember, you can create a new `ListNode` object like this:

```
obj = ListNode(val)
```

where the parameter is the value that will be stored inside the node.

Do not change the array that you were passed.

6 `list_length(head)`

This function returns the length of a linked list (which might be empty).

7 `is_sorted(head)`

This function takes a single parameter, which is a list, and returns `True` or `False`, indicating whether or not the values in the list are sorted (in ascending order). Note that duplicates may exist in the list; these count as being in order.

An empty list should return `True`, as should a list of length 1.

(spec continues on the next page)

8 `accordion(old_head)`

This function takes an old list, removes every other node, and then returns the updated list. Old nodes, which are removed from the list, may simply become garbage; you don't need to return them in any way.

When removing nodes from the list, you must remove the old head, the node 2 after the old head, and the node 2 after that, out to the end of the list. For instance, if the input list is like this:

```
10 -> 20 -> -1 -> "asdf" -> 13 -> "qwerty" -> 1024
```

then the list that you return must look like this:

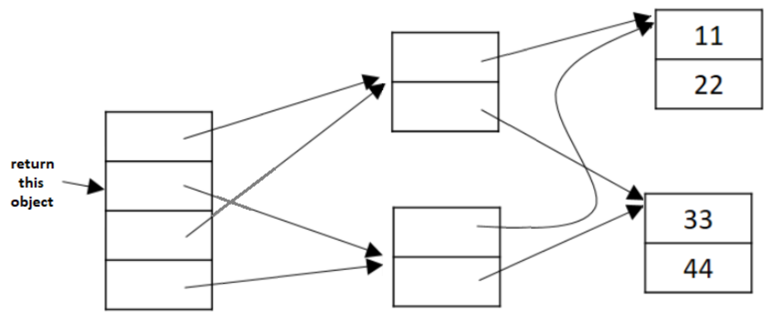
```
20 -> "asdf" -> "qwerty"
```

If the old list contained only a single node, or was empty, return an empty list.

9 Shape Function : `too_many_aliases()`

NOTE: All of the rectangles in these pictures represent Python lists (that is, arrays). Do not attempt to use tuples, classes, or any other type. (But there might be other types **inside** the boxes!)

This Short Project only requires that you implement a single shape function, named `too_many_aliases()` . Place it in a file named `too_many_aliases.py` . It should return the following shape:



10 Turning in Your Solution

You must turn in your code using GradeScope.