Lab 2

This lab provides an introduction to list implementations. In particular, you are asked to implement both a linked list and an "array" list.

1 List Abstract Data Type

The operations supported by a list can be discussed independent of the implementation. As such, a given list implementation can be used through the set of operations without programmatic concern for the details of the implementation. This allows one to vary the implementation based in the characteristics of the problem being solved.

Details of each operation are given below; you will implement these operations for a linked list implementation and for an "array" list implementation. You must verify, via test cases, that your implementations behave as expected (i.e., that they "work").

- `empty_list` This function takes no arguments and returns an empty list.
- `add` This function takes a list, an integer `index`, and another `value` (of any type) as arguments and places the `value` at `index` position in the list (zero-based indexing; any element at the given `index` before this operation will not immediately follow the new element). If the `index` is invalid (i.e., less than 0 or greater than the current length), then this operation should raise an `IndexError`. (Note that an `index` equal to the length is allowed and results in the new `value` being added to the end of the list.)

  This function *must* return the resulting list.
- `length` This function takes a list as an argument and returns the number of elements currently in the list.
- `get` This function takes a list and an integer `index` as arguments and returns the value at the `index` position in the list (zero-based indexing). If the `index` is invalid (i.e., it falls outside the bounds of the list), then this operation should raise an `IndexError`.
- `setitem` This function takes a list, an integer `index`, and another `value` (of any type) as arguments and replaces the element at `index` position in the list with the given `value`. If the `index` is invalid, then this operation should raise an `IndexError`.

  This function *must* return the resulting list.
- `remove` This function takes a list and an integer `index` as arguments and removes the element at the `index` position from the list. If the `index` is invalid (i.e., it falls outside the bounds of the list), then this operation should raise an `IndexError`.

  This function *must* return a 2-tuple of, in this order, the element previously at the specified `index` (i.e., the removed element) and the resulting list.

2 Linked List

In a file named `linked_list.py`, provide a data definition for a `LinkedList`, whose `Pair` class's `first` field can be any value. An empty list should be represented by the Python value `None`. Be sure to call your pair class `Pair`, so that my tests can create objects correctly.

Implement the functions listed above.

Place your test cases in a file named `linked_list_tests.py`.

Include type signatures and a docstring purpose statement as appropriate.

3 Array List

In a file named `array_list.py`, define the `ArrayList` class for an array list implementation and implement the aforementioned list operations. For this implementation, each element of the array represents one element of the list. This implementation must allow for your list to grow dynamically (i.e., you cannot assume a maximum size). To accommodate this, when your allocated "array" runs out of capacity, you should make a new "array" with double the capacity, and copy all of the values over.

Place your test cases in a file named `array_list_tests.py`

Include type signatures and a docstring purpose statement as appropriate.

You will use a Python list as the backing array for your array list implementation (Python lists, at least in the standard implementation, are backed by arrays).

Note: since this lab is a study of data structure implementation, you are prohibited from using almost all of Python's list operations in your array list implementation. The only list operations you may use are:

- initializing with a specific size (through the $*$ operator, e.g., `[None] * 100`), which will act as "allocating a new array", and
- indexing (e.g., `my_list[4]`)

Every other builtin Python operation dealing with lists is expressly forbidden.

This means that any copying required in your implementation must be done via loops to make the steps explicit (e.g., no slices allowed). This restriction only applies within this course when stated; when you use Python in the future and want an array-list-like data structure, you should certainly use the provided type and its operations.

4 Testing

As mentioned in the syllabus, your code is periodically graded prior to the deadline and you will receive automated feedback based on the results of my tests. *But*, in order to receive *any* feedback, your tests much provide 100% test coverage of the code you are submitting. If you do not have 100% coverage, this is the only feedback you will receive.

100% test coverage means that every line of your code is run at some point in some test. Imagine if that weren't the case. That means that you have a line of code (or multiple lines of code) that could do anything, and you'd never know. You never tested them!

Your tests will go in files named:

- `linked_list_tests.py`, and
- `array_list_tests.py`

5 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.