CSCI 204 – Data Structures and Algorithms

Lab 10 – Lists & Iterators

1. Objectives

The objectives of this lab are to:

- Become familiar with iterators
- Write an iterator for a List class

Read the entire lab description before you begin working on the assignment.

2. Introduction

An iterator is a feature commonly found in those ADTs that allow a user to peek at any data in the ADT. Therefore, a Bag, a Dictionary, a List, and a Tree will have an iterator while a Stack and a Queue will not. An iterator allows the user to examine each datum in the ADT one at a time without needing to specify indices. This is how the for loop allows the user to examine the data in an array or the keys in a dictionary.

```
array = [1,3,5,7,9,2,4,6,8]
for x in array:
    print(x)

dict = {'Alice':3, 'Bob':23, 'Carol':7}
for name in dict:
    print(name, dict[name])
```

Iterators can be built in any language but Python provides built-in support for iterators in the form of a class method.

Take a minute to read more about the class method from the following link,
https://docs.python.org/3/reference/datamodel.html#object.__iter__

An iterator (in any language) comes with some basic functionality:

- `__init__(self, a_structure)` creates the iterator and gets it ready to return data. (Does not return any data yet.) The structure in the input is either the array or the head node holding data. Other inputs such as size and capacity may be passed in if needed.
- `__next__(self)` **Returns the next data in order in the ADT.**
- There are also a few that you *may* see in other languages, but are not in Python. (Skip ahead if

you'd like):

- `__prev__(self)` Returns the previous data in order in the ADT. **(Not in Python)**
- `__hasNext__(self)` Returns True if there is next data to be accessed in the ADT. **(Not in Python)**
- `__hasPrev__(self)` Returns True if there is previous data to be accessed in the ADT. **(Not in Python)**

In Python, the iterator is its own class that implements the above functionality and the ADT simply returns an instance of the iterator. See the following example.

```
class Bag:
    def __init__(self):
        self._head = None

    def __iter__(self):
        return _BagIterator(self._head)
```

Notice the underscore in front of the _BagIterator class name. This is because no other class should make a Bag iterator instance, that is, it's a *hidden* class.

```
class _BagIterator:
    def __init__(self, head):
        """ Create the iterator and get it ready to start when next
            is called. """

    def __next__(self):
        """ Return the next datum or raise a StopIteration
            exception. """

    def __iter__(self):
        """ return ourself so we can be used. """
        return self
```

We'll be able to use the iterator methods in the for loop. The loop knows when to stop because `__next__(self)` raises a `StopIteration` exception when there is no next data to return. Here is how our Bag iterator gets used:

```
for item in bag:
    print(item)
```

3. Getting started

Begin by making a working directory for this lab. Copy the test program **`main.py`** from

~csci204/student-labs/lab10/

You can also get the copy of the file directly from the course Moodle site.

After implementing the proper iterator, run the **`main.py`** program to test your implementation. Leave the **`main.py`** alone, put all your implementation in a file named **`linkedlist.py`**.

1. Implement a List class using (Singly) Linked-List nodes.

Create a `ListNode` class that contains the following attributes:

- `data`: contains any data that then node should represent.
- `next`: contains a pointer to the next item in the list.

The ListNode should also override a __str__() or a __repr__() method so that you can write a statement such as `print(my_node)` that can print the information of a `ListNode` object.

Create a `List` class (in the same file as the `ListNode` class) with the following methods & functionality:

- `__init__(self)`: creates an empty list. This mean creating attributes with attributes to keep track of the `head` of your list as well as the size of it, but with head pointing to None as there is nothing yet at the `head` of the list.
- `__len__(self)`: returns the size of the list.
- `insert(self, item, index)`: inserts an *item* at the given index where item is some data that must be placed into a `ListNode` object. If the index is too small, insert at index 0. If the index is too large (larger than your current size), insert it after the last element. If the index is occupied, put the item at the index and move the previously occupying item to the right.
- `append(self, item)`: appends an item to the end of the list.
- `pop(self, index)`: removes and returns the item at the given index. If the index is not provided, remove and return the last item in the list. If the index is illegal do nothing.
- `peek(self, index)`: returns the item at the given index. If the index is illegal raise a `ListException`.

2. Write a hidden `ListIterator` class (proceeding your class name with an underscore) in the same file as your list class. Your class must have the following functionality**:**
   - `__init__(self, a_structure)`: creates the iterator and gets it ready to return data. (Does not return any data yet). The structure in the input is either the array or the head node of your List. Other inputs such as size and capacity may be passed in if needed.
   - `__next__(self)`: returns the next data in order in the ADT. **If there is no next data, raise a StopIteration exception**.
   - `__iter__(self)`: return your ListIterator (just return `self`).

3. Give your `List` class functionality for an iterator by overriding the correct method. (If you can't remember the method to override reread the beginning of the lab directions!)

4. Run the **`main.py`** and save a copy of the output to a text file named `out.txt`.

Hand-in your List class with the Iterator class in the same file (`linkedlist.py`) and the test program (`main.py`) along with the output file `out.txt`.