Module 4 Lab: Linked List

Create the fundamental `Node` and `LinkedList` classes using test driven development (TDD): Red, Green, Refactor.

Part 1 - `class Node`

Create two files: `linkedlist.py` and `test_linkedlist.py`. Implement unittests and functionality for a `Node` class as described below.

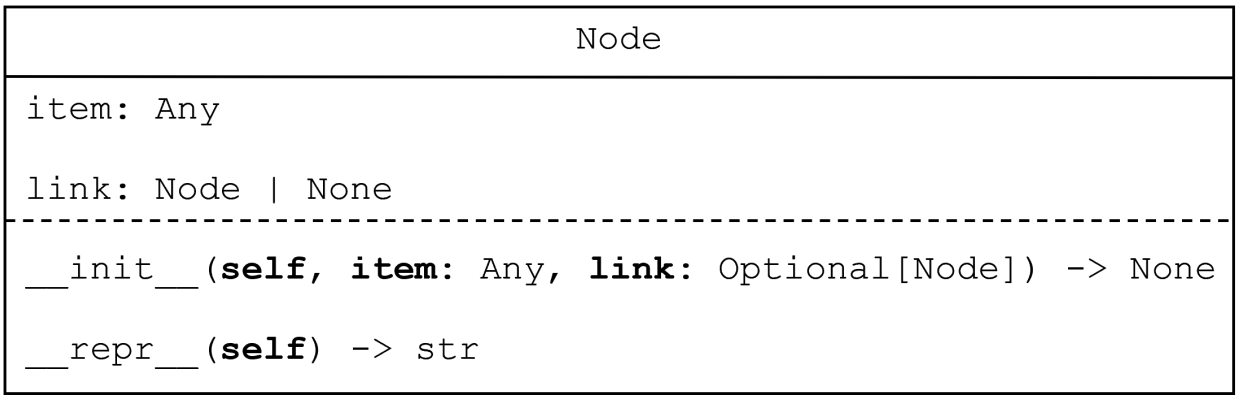

Figure 1: Class diagram for `Node`. Note the use of expected types after colons. `Node | None` denotes it can be a `Node` object or the `None` object. `Optional[Node]` means there is an optional `Node` parameter that defaults to `None`.

- `init(self, item:Any, link:Optional[Node]) -> None`
  - Creates a new Node object. `link` is either another `Node` or the `None` object
- `repr(self) -> str`
  - Returns a string representaiton of the object, e.g. `Node(Jake)`
  - This is a dunder method: `__repr__()`

TDD Flow:

1) `init`
   - Red: `test_linkedlist.py`, `TestNode` class:
     - Create a node, and assert it has the correct attributes, e.g.

     ```
     node1 = Node(...)
     self.assertEqual(node1.item, ...)
     self.assertEqual(node1.link, ...)
     ```

   - Green: `linkedlist.py`, `Node` class: implement `__init__`
2) `repr`
   - Red: `test_linkedlist.py`, `TestNode` class:
     - Create a node, and assert it has the correct `repr()` return, e.g.

     ```
     node1 = Node(...)
     self.assertEqual(repr(node1), "Node(...)")
     ```

   - Green: `linkedlist.py`, `Node` class: implement `__repr__`

Readability Note

You should (almost) **always** call dunder methods using their "magic" syntax. E.g.:

- `repr(node1)` instead of `node1.__repr__()`
- `x + y` instead of `x.__add__(y)`
- `object1 == object2` instead of `object1.__eq__(object2)`
- `len(collection)` instead of `collection.__len__()` or `collection._len`

This applies to test cases *and* functionality within a class. The magic syntax is preferred because it improves readability. The only common exception is when calling `super().__init__`.

Part 2 - `class LinkedList`

In the same files as above, but in different classes (`LinkedList` and `TestLinkedList`), implement the `LinkedList` class as described below.

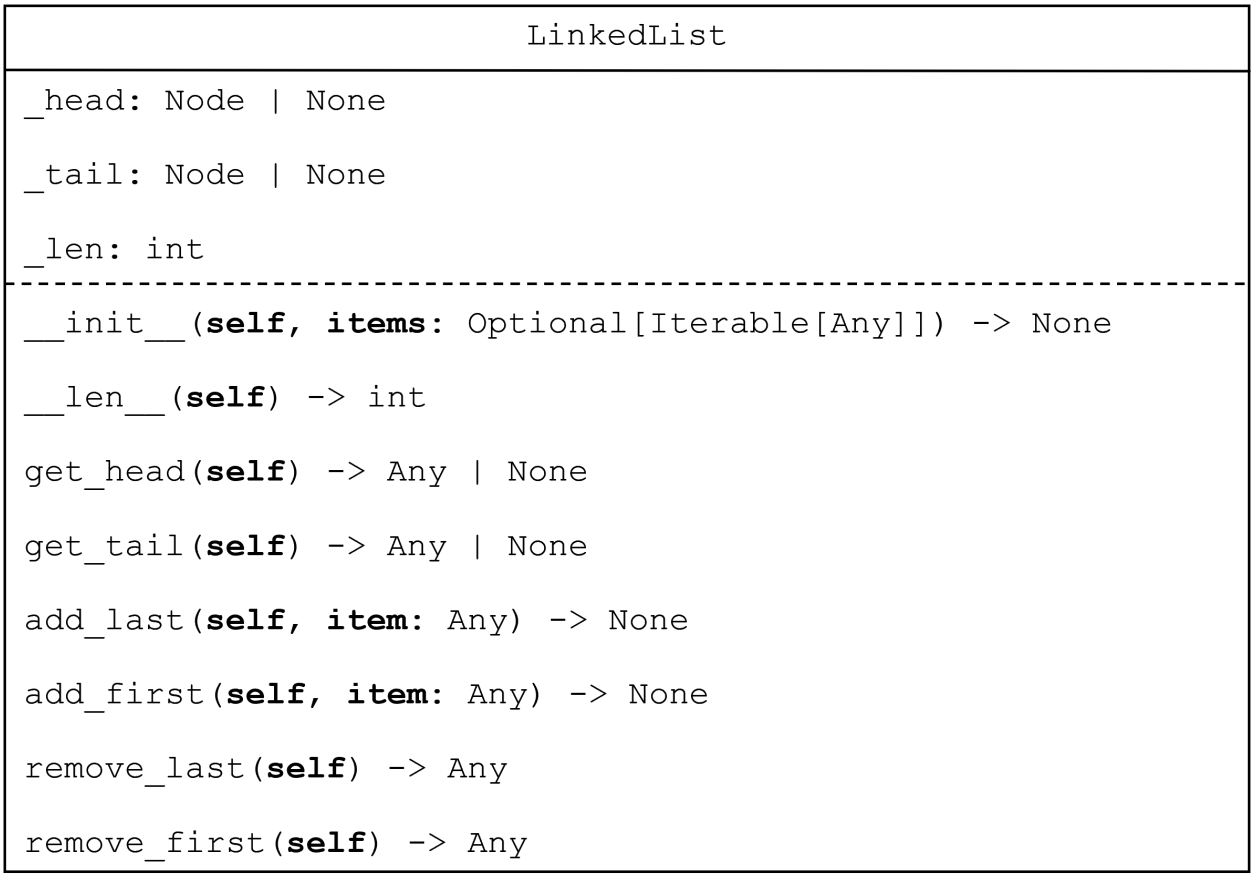

Figure 2: Class diagram for `LinkedList`. Note that `items` in `LinkedList.__init__` is an optional paramter (it defaults to `None`, and it can be any iterable if included. This means lists, tuples, sets, dictionaries, strings, or even the `range` object are fair game.)

- `init(self, items: Optional[Iterable[Any]) -> None`
  - optional collection `items` are sequentially added to LinkedList if included
  - use `add_last()` to add items in correct order
- `get_head(self) -> Any | None`

  - Returns `item` stored in `self._head`, or `None` for an empty `LinkedList`
- `get_tail(self) -> Any | None`
  - Returns `item` stored in `self._tail`, or `None` for an empty `LinkedList`
- `add_first(self, item: Any) -> None`
  - adds a node to the front of the `LinkedList` with `item`
- `add_last(self, item: Any) -> None`
  - adds a node to the end of the `LinkedList` with `item`
- `remove_first(self) -> Any`
  - removes first node from `LinkedList` and returns its item
  - raises a `RuntimeError` if `LinkedList` is empty when called
- `remove_last(self) --> Any`
  - removes last node from `LinkedList` and returns its item
  - raises a `RuntimeError` if `LinkedList` is empty when called

Flow:

1) Empty initialization
   - Red: `test_linkedlist.py`, `TestLinkedList` class:
     - Create empty `LinkedList` e.g. `LL1 = LinkedList()`
     - Assert length is correct (should be 0)
     - Assert `get_head()` and `get_tail()` return correct values (should be `None`)
   - Green: `linkedlist.py`, `LinkedList` class:
     - Implement `__init__`, `__len__`, `get_head()`, and `get_tail()`
2) `add_last`
   - Red: `test_linkedlist.py`, `TestLinkedList` class:
     - Create an empty `LinkedList`
     - Using a `for` loop, sequentially add items to end of `LinkedList`. With each add, assert that you get the correct values from:
       * `len()`
       * `get_head()`
       * `get_tail()`
   -
     - Green: `linkedlist.py`, `LinkedList` class:
       * Implement `add_last`
3) Non-empty initialization
   - Red: `test_linkedlist.py`, `TestLinkedList` class:
     - initialize a new linked list with some iterable, e.g.:
       * `LL1 = LinkedList(['a', 'b', 'c'])`
       * `LL1 = LinkedList(range(10))`
     - Test you get correct values from `len`, `get_head()`, and `get_tail()`
   -
     - Green: `linkedlist.py`, `LinkedList` class:
       * Finish implementing `__init__`
       * See note on default parameters below
4) `add_first`
   - Test similar to `add_last`
   - Implement
5) `remove_first`
   - Red: `test_linkedlist.py`, `TestLinkedList` class:
     - Build up a linked list (either create a non-empty list, or use `add_first` or `add_last`)

- Iteratively call `remove_first()` until the `LinkedList` is empty. With every `remove_first()` call, assert you get correct values from:
  - `remove_first()` (should return item in first node)
  - `len()`
  - `get_head()`
  - `get_tail()`

- Green: `linkedlist.py`, `LinkedList` class:
  - implement `remove_first()`
- Red: `test_linkedlist.py`, `TestLinkedList` class:
  - Assert that you get a `RuntimeError` when removing from an empty `LinkedList`.
- Green : `linkedlist.py`, `LinkedList` class:
  - Implement that error in `remove_first()`

6) `remove_last`
   - `test_linkedlist.py`: similar to `remove_first()` above
   - `linkedlist.py`: similar to `remove_first()`

Note on default parameters

As we begin to make our own data structures, we will often want to give users the option to pass in a starting collection. It is tempting to use an empty list as a default value to allow this, but that is bad practice. Python only initializes default arguments for a method once, so mutable default arguments end up being shared across *all instances of a class*:

```
>>> class MyListWrapper:
...     def __init__(self, L=[]): # Empty list is a BAD default, because it is mutable
...             self.L = L
...
>>> x = MyListWrapper() # x.L is the default empty list
>>> y = MyListWrapper() # y.L is the SAME default list
>>> x.L.append(3)
>>> x.L
[3]
>>> y.L
[3]
>>>
```

If we want to make a custom collection with an optional collection of arguments, we should use an immutable like `None` for our default list, and create an empty list on the fly *inside* of our constructor method `init`:

```
>>> class BetterListWrapper:
...     def __init__(self, L=None):
...             if L is None:
...                     self.L = [] # new empty list created for every object
...             else:
...                     self.L = L # whatever the user passed in
...
>>> x = BetterListWrapper()
>>> y = BetterListWrapper()
>>> x.L.append(3)
```

```
>>> x.L
[3]
>>> y.L
[]
```

You should use a similar pattern for your `LinkedList.__init__()` method, e.g.:

```
class LinkedList:
    def __init__(self, items=None):
        # ...initialize LinkedList attributes

        # if user passed in a collection, add them one at a time
        if items is not None:
            for item in items:
                self.add_last(item)
```

Submitting

At a minimum, submit the following files:

- `linkedlist.py`
- `test_linkedlist.py`

Students must submit **individually** by the due date (typically, Friday at 11:59 pm EST) to receive credit.

We are unfortunately not able to autograde your unittests - this means we cannot give you instant feedback on whether your tests are correct or not. Make good use of your lab time by asking your classmates and lab instructor to look over your tests. Learning to write good tests is one of the best ways to reduce the difficulty of coding.