1 Linked Lists

Lab Objective: *One of the fundamental problems in programming is knowing which data structures to use to optimize code. The type of data structure used determines how quickly data is accessed and modified, which affects the overall speed of a program. In this lab we introduce a basic data structure called a linked list and create a class to implement it.*

A *linked list* is a data structure that chains data together. Every linked list needs a reference to the first item in the chain, called the `head`. A reference to the last item in the chain, called the `tail`, is also often included. Each item in the list stores a piece of data, plus at least one reference to another item in the list. The items in the list are called *nodes*.

Nodes

Think of data as several types of objects that need to be stored in a warehouse. A *node* is like a standard size box that can hold all the different types of objects. For example, suppose a particular warehouse stores lamps of various sizes. Rather than trying to carefully stack lamps of different shapes on top of each other, it is preferable to first put them in boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier. A *data structure* is like the warehouse, which specifies where and how the different boxes are stored.

A node class is usually simple. The data in the node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure.

Problem 1. Consider the following generic node class.

```
class Node:
    """A basic node class for storing data."""
    def __init__(self, data):
        """Store the data in the value attribute."""
        self.value = data
```

Modify the constructor so that it only accepts data of type `int`, `float`, or `str`. If another type of data is given, raise a `TypeError` with an appropriate error message. Modify the constructor docstring to document these restrictions.

The nodes of a *singly linked list* have a single reference to the next node in the list (see Figure 1.1), while the nodes of a *doubly linked list* have two references: one for the previous node, and one for the next node (see Figure 1.2). This allows for a doubly linked list to be traversed in both directions, whereas a singly linked list can only be traversed in one direction.

```
class LinkedListNode(Node):
    """A node class for doubly linked lists. Inherits from the Node class.
    Contains references to the next and previous nodes in the linked list.
    """
    def __init__(self, data):
        """Store the data in the value attribute and initialize
        attributes for the next and previous nodes in the list.
        """
        Node.__init__(self, data)       # Use inheritance to set self.value.
        self.next = None                # Reference to the next node.
        self.prev = None                # Reference to the previous node.
```

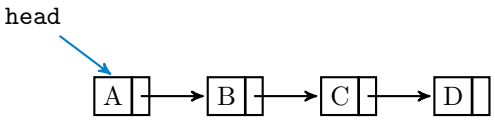

Figure 1.1: A singly linked list. Each node has a reference to the next node in the list. The head attribute is always assigned to the first node.

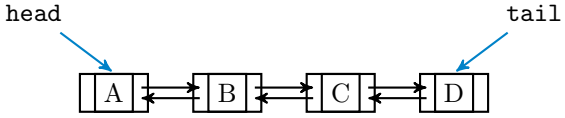

Figure 1.2: A doubly linked list. Each node has a reference to the node before it and a reference to the node after it. In addition to the head attribute, this list has a tail attribute that is always assigned to the last node.

The following `LinkedList` class chains `LinkedListNode` instances together by modifying each node's `next` and `prev` attributes. The list is empty initially, so the `head` and `tail` attributes are assigned the placeholder value `None` in the constructor. The `append()` method makes a new node and adds it to the very end of the list (see Figure 1.3). There are two cases for appending that must be considered separately in the implementation: either the list is empty, or the list is nonempty.

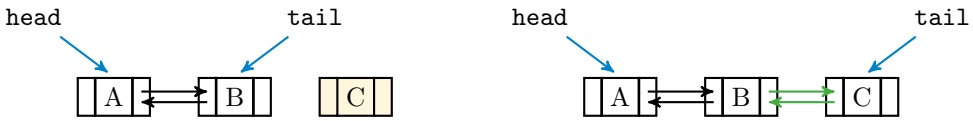

Figure 1.3: Appending a new node to the end of a nonempty doubly linked list. The green arrows are the new connections. Note that the `tail` attribute is reassigned from B to C.

```
class LinkedList:
    """Doubly linked list data structure class.

    Attributes:
        head (LinkedListNode): the first node in the list.
        tail (LinkedListNode): the last node in the list.
    """
    def __init__(self):
        """Initialize the head and tail attributes by setting
        them to None, since the list is empty initially.
        """
        self.head = None
        self.tail = None

    def append(self, data):
        """Append a new node containing the data to the end of the list."""
        # Create a new node to store the input data.
        new_node = LinkedListNode(data)
        if self.head is None:
            # If the list is empty, assign the head and tail attributes to
            # new_node, since it becomes the first and last node in the list.
            self.head = new_node
            self.tail = new_node
        else:
            # If the list is not empty, place new_node after the tail.
            self.tail.next = new_node               # tail --> new_node
            new_node.prev = self.tail               # tail <-- new_node
            # Now the last node in the list is new_node, so reassign the tail.
            self.tail = new_node
```

Achtung!

The `is` operator is **not** the same as the `==` operator. While `==` checks for numerical equality, `is` evaluates whether or not two objects are the same by checking their location in memory.

```
>>> 7 == 7.0            # True since the numerical values are the same.
True

# 7 is an int and 7.0 is a float, so they cannot be stored at the same
# location in memory. Therefore 7 "is not" 7.0.
>>> 7 is 7.0
False
```

For numerical comparisons, always use `==`. When comparing to built-in Python constants such as `None`, `True`, `False`, or `NotImplemented`, use `is` instead.

Locating Nodes

The `LinkedList` class only explicitly keeps track of the first and last nodes in the list via the `head` and `tail` attributes. To access any other node, use each successive node's `next` and `prev` attributes.

```
>>> my_list = LinkedList()
>>> for data in (2, 4, 6):
...     my_list.append(data)
...
# To access each value, use the head attribute of the LinkedList
# and the next and value attributes of each node in the list.
>>> my_list.head.value
2
>>> my_list.head.next.value                    # 2 --> 4
4
>>> my_list.head.next.next is my_list.tail     # 2 --> 4 --> 6
True
```

Problem 2. Add the following methods to the `LinkedList` class.

1. `find()`: Accept a piece of data and return the first node in the list containing that data (return the actual `LinkedListNode` object, not its `value`). If no such node exists, or if the list is empty, raise a `ValueError` with an appropriate error message.
   (Hint: if `n` is assigned to one of the nodes the list, what does `n = n.next` do?)

2. `get()`: Accept an integer i and return the ith node in the list. If i is negative or greater than or equal to the number of nodes in the list, raise an `IndexError`.
   (Hint: add an attribute that tracks the current size of the list. Update it every time a node is successfully added or removed, such as at the end of the `append()` method.)

Magic Methods

Endowing data structures with magic methods makes them much more intuitive to use. Consider, for example, how a Python list responds to built-in functions like `len()` and `print()`. At the bare minimum, the `LinkedList` class should have the same functionality.

Problem 3. Add the following magic methods to the `LinkedList` class.

1. Write the `__len__()` method so that the length of a `LinkedList` instance is equal to the number of nodes in the list.

2. Write the `__str__()` method so that when a `LinkedList` instance is printed, its output matches that of a Python list. Entries are separated by a comma and one space; strings are surrounded by single quotes, or by double quotes if the string itself has a single quote.
   (Hint: use `repr()` to deal with quotes easily.)

Removal

To delete a node, all references to the node must be removed. Python automatically deletes the object once there is no way for the user to access it. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to `None`. However, there is a problem with this approach.

```
class LinkedList:
    # ...
    def remove(self, data):
        """Attempt to remove the first node containing the specified data.
        This method incorrectly removes additional nodes.
        """
        # Find the target node and sever the links pointing to it.
        target = self.find(data)
        target.prev.next = None                    # -/-> target
        target.next.prev = None                    # target <-/-
```

Removing all references to the target node deletes the node (see Figure 1.4). Unfortunately, the nodes before and after the target node are no longer linked.

```
>>> my_list = LinkedList()
>>> for i in range(10):
...     my_list.append(i)
...
>>> print(my_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> my_list.remove(4)              # Removing a node improperly results in
>>> print(my_list)                 # the rest of the chain being lost.
[0, 1, 2, 3]                       # Should be [0, 1, 2, 3, 5, 6, 7, 8, 9].
```

Figure 1.4: Naïve removal for doubly linked Lists. Deleting all references pointing to C deletes the node, but it also separates nodes A and B from node D.

This can be remedied by pointing the previous node's `next` attribute to the node after the deleted node, and similarly changing that node's `prev` attribute. Then there will be no reference to the removed node and it will be deleted, but the chain will still be connected.

Figure 1.5: Correct removal for doubly linked Lists. To avoid gaps in the chain, nodes B and D must be linked together.

Problem 4. Modify the `remove()` method given above so that it correctly removes the first node in the list containing the specified data. Also account for the special cases of removing the first, last, or only node, in which `head` and/or `tail` must be reassigned. Raise a `ValueError` if there is no node in the list that contains the data.
(Hint: use the `find()` method from Problem 2 to locate the target node.)

ACHTUNG!

Python keeps track of the variables in use and automatically deletes a variable (freeing up the memory that stored the object) if there is no access to it. This feature is called *garbage collection*. In many other languages, leaving a reference to an object without explicitly deleting it can lead to a serious memory leak. See `https://docs.python.org/3/library/gc.html` for more information on Python's garbage collection system.

Insertion

The `append()` method can add new nodes to the end of the list, but not to the middle. To do this, get references to the nodes before and after where the new node should be, then adjust their `next` and `prev` attributes. Be careful not to disconnect the nodes in a way that accidentally deletes nodes like in Figure 1.4.

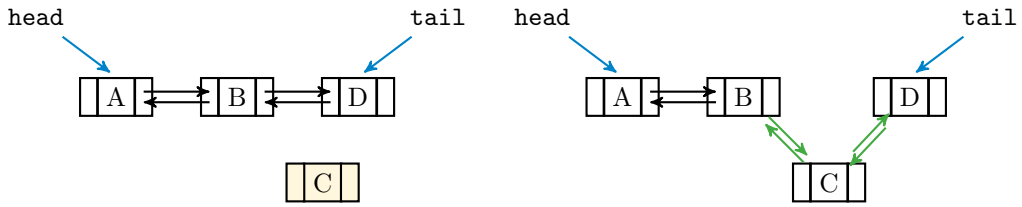

Figure 1.6: Insertion for doubly linked lists.

Problem 5. Add an `insert()` method to the `LinkedList` class that accepts an integer `index` and data to add to the list. Insert a new node containing the data immediately **before** the node in the list currently at position `index`. After the insertion, the new node should be at position `index`. For example, Figure 1.6 places a new node containing C at index 2. Carefully account for the special case of inserting before the first node, which requires `head` to be reassigned.
(Hint: except when inserting before the head, get references to the nodes that should be immediately before and after the new node following the insertion. Consider using the `get()` method from Problem 2 to locate one of these nodes.)

If `index` is equal to the number of nodes in the list, append the node to the end of the list by calling `append()`. If `index` is negative or strictly greater than the number of nodes in the list, raise an `IndexError`.

Note

The temporal complexity for inserting to the beginning or end of a linked list is $O(1)$, but inserting anywhere else is $O(n)$, where n is the number of nodes in the list. This is quite slow compared other data structures. In the next lab we turn our attention to *trees*, special kinds of linked lists that allow for much quicker sorting and data retrieval.

Restricted-Access Lists

It is sometimes wise to restrict the user's access to some of the data within a structure. In particular, because insertion, removal, and lookup are $O(n)$ for data in the middle of a linked list, cutting off access to the middle of the list forces the user to only use $O(1)$ operations at the front and end of the list. The three most common and basic restricted-access structures that implement this idea are *stacks*, *queues*, and *deques*. Each structure restricts the user's access differently, making them ideal for different situations.

- **Stack**: *Last In, First Out* (LIFO). Only the last item that was inserted can be accessed. A stack is like a pile of plates: the last plate put on the pile is (or should be) the first one to be taken off. Stacks usually have two main methods: `push()`, to insert new data, and `pop()`, to remove and return the last piece of data inserted.
- **Queue** (pronounced "cue"): *First In, First Out* (FIFO). New nodes are added to the end of the queue, but an existing node can only be removed or accessed if it is at the front of the queue. A queue is like a polite line at the bank: the person at the front of the line is served first, while newcomers add themselves to the back of the line. Queues also usually have a `push()` and a `pop()` method, but `push()` inserts data to the end of the queue while `pop()` removes and returns the data at the front of the queue. The `push()` and `pop()` operations are sometimes called `enqueue()` and `dequeue()`, respectively.
- **Deque** (pronounced "deck"): a double-ended queue. Data can be inserted or removed from either end, but data in the middle is inaccessible. A deque is like a deck of cards, where only the top and bottom cards are readily accessible. A deque has two methods for insertion and two for removal, usually called `append()`, `appendleft()`, `pop()`, and `popleft()`.

  A deque can act as a queue by using only `append()` and `popleft()` (or `appendleft()` and `pop()`), or as a stack by using only `append()` and `pop()` (or `appendleft()` and `popleft()`).

Problem 6. Write a `Deque` class that inherits from `LinkedList`.

1. Write the following methods. Since they all involve data at the endpoints, avoid iterating through the list so the resulting operations are $O(1)$.
   - `pop()`: Remove the last node in the list and return its data. Account for the special case of removing the only node in the list. Raise a `ValueError` if the list is empty.
   - `popleft()`: Remove the first node in the list and return its data. Raise a `ValueError` if the list is empty.
     (Hint: use inheritance and the `remove()` method of `LinkedList`.)

- `appendleft()`: Insert a new node at the beginning of the list.
  (Hint: use inheritance and the `insert()` method of `LinkedList`.)

Note that the `LinkedList` class already implements `append()`.

2. Override the `remove()` method with the following code.

```
def remove(*args, **kwargs):
    raise NotImplementedError("Use pop() or popleft() for removal")
```

This effectively disables `remove()` for the `Deque` class, preventing the user from removing a node from the middle of the list.

3. Disable `insert()` as well.

Note

The `*args` argument allows the `remove()` method to receive any number of positional arguments without raising a `TypeError`, and the `**kwargs` argument allows it to receive any number of keyword arguments. This is the most general form of a function signature.

Python lists have `append()` and `pop()` methods, so they can be used as stacks. However, data access and removal from the front is much slower than from the end, as Python lists are implemented as dynamic arrays and not linked lists.

The `collections` module in the standard library has a `deque` object that is implemented as a doubly linked list. This is an excellent object to use in practice instead of a Python list when speed is of the essence and data only needs to be accessed from the ends of the list. Both lists and deques are slow to modify elements in the middle, but lists can access middle elements quickly. Table 1.1 describes the complexity for common operations on lists v. deques in Python.

Operation	List Complexity	Deque Complexity
Append/Remove from the end	$O(1)$	$O(1)$
Append/Remove from the start	$O(n)$	$O(1)$
Insert/Delete in the middle	$O(n)$	$O(n)$
Access element at the start/end	$O(1)$	$O(1)$
Access element in the middle	$O(1)$	$O(n)$

Table 1.1: Complexity of operations on lists and deques.

Problem 7. Write a function that accepts the name of a file to be read and a file to write to. Read the first file, adding each line of text to a stack. After reading the entire file, pop each entry off of the stack one at a time, writing the result to the second file.

For example, if the file to be read has the following list of words on the left, the resulting file should have the list of words on the right.

```
My homework is too hard for me.          I am a mathematician.
I do not believe that                    Programming is hard, but
I can solve these problems.              I can solve these problems.
Programming is hard, but                 I do not believe that
I am a mathematician.                    My homework is too hard for me.
```

You may use a Python list, your `Deque` class, or `collections.deque` for the stack. Test your function on the file `english.txt`, which contains a list of over 58,000 English words in alphabetical order.

Additional Material

Possible Improvements to the LinkedList Class

The following are some ideas for expanding the functionality of the `LinkedList` class.

1. Add a keyword argument to the constructor so that if an iterable is provided, each element of the iterable is immediately added to the list. This makes it possible to cast an iterable as a `LinkedList` the same way that an iterable can be cast as one of Python's standard data structures.

```
>>> my_list = [1, 2, 3, 4, 5]
>>> my_linked_list = LinkedList(my_list)  # Cast my_list as a LinkedList.
>>> print(my_linked_list)
[1, 2, 3, 4, 5]
>>> type(my_linked_list)
LinkedList
```

2. Add the following methods.
   - `count()`: return the number of occurrences of a specified value.
   - `reverse()`: reverse the ordering of the nodes (in place).
   - `roll()`: shift the nodes a given number of steps to the right or left (in place).
   - `sort()`: sort the nodes by their data (in place).
3. Implement more magic methods.
   - `__add__()`: concatenate two lists.
   - `__getitem__()` and `__setitem__()`: enable standard bracket indexing. Try to allow for negative indexing as well.
   - `__iter__()`: support `for` loop iteration, the `iter()` built-in function, and the `in` statement.

Other Kinds of Linked Lists

The `LinkedList` class can also be used as the backbone for more specialized data structures.

1. A *sorted list* adds new nodes strategically so that the data is always kept in order. Therefore, a `SortedLinkedList` class should have an `add()` method that receives some `data` and inserts a new node containing `data` before the first node in the list that has a `value` that is greater or equal to `data` (thereby preserving the ordering). Other methods for adding nodes should be disabled. Note however, that a linked list is **not** an ideal implementation for a sorted list because each insertion is $O(n)$ (try sorting `english.txt`).
2. In a *circular linked list*, the "last" node connects back to the "first" node. Thus a reference to the tail is unnecessary. The `roll()` method mentioned above is used often so the `head` attribute is at an "active" part of the list where nodes are inserted, removed, or accessed often. This data structure can therefore decrease the average insertion or removal time for certain data sets.