Part I

Labs

Lab 1

Data Structures I

Lab Objective: *Learn to implement basic data structures.*

Introduction

Analyzing and manipulating data are essential skills in scientific computing. Storing, retrieving, and manipulating data take time. As a dataset grows, so does the amount of time it takes to access and manipulate it. The structure of how the data are stored determines how efficiently the data may be processed.

In this lab we will begin to study data structures. Data structures are objects for organizing data. There are diverse data structures, each with specific strengths and weaknesses. For example, some data structures take a long time to build, but once built their data are quickly accessible. Others are built quickly, but are not as efficiently accessible. Different applications will require different structures.

Nodes

Recall that some built-in data types for Python are booleans, strings, floats, and integers. Most data in applications will take one of these forms. However, as the size of a dataset increases, these types prove inefficient. Data structures will use *nodes* to overcome these inefficiencies.

Think of data as several types of objects that need to be stored in a warehouse. Then a node is a standard size box that can hold all the different types of objects. For example, suppose the warehouse needs to store lamps of various sizes. Rather than trying to stack lamps of different shapes on top of each other efficiently, it is preferable to put them in the boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier. This analogy extends to data structures.

A `Node` class is usually simple. In Python, the data in the Node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure. The data structure links the nodes together in a way that is efficient for its particular application.

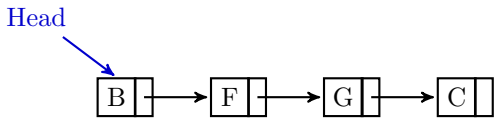

Figure 1.1: Singly-linked List

```
# Location: Node.py

class Node(object):
    """A Node class for storing data."""
    def __init__(self, data):
        """Construct a new node that stores some data."""
        self.data = data
```

```
# Import the Node class from Node.py
>>> from Node import Node

# Create some nodes.  Note that any data type may be stored.
>>> int_node = Node(1)
>>> str_node = Node('abc')
>>> lis_node = Node([1,'abc'])

# Access a nodes data.
>>> lis_node.data
[1, 'abc']
```

Problem 1. Add a new class to `Node.py` called `StrNode` that only accepts strings as data. Use inheritance and call the `Node` class constructor in the constructor for `StrNode` (Hint: Use the `type` built-in function).

Implement the `__str__` magic method in the `Node` class so that it returns a string representation of its data.

Linked Lists

A linked list is a simple data type that chains nodes together. Each node instance in a linked list stores a reference to the next link in the chain. A linked list class also stores a reference to the first node in the chain, called the head. See Figure 1.1.

```
# Location: Node.py

class LinkedListNode(Node):
    """A Node class for linked lists. Inherits from the 'Node' class.
    Contains a reference to the next node in the list.
    """
    def __init__(self, data):
        """Construct a Node and add an attribute for
```

Figure 1.2: Include a picture of the `add_node` method and describe it.

```
        the next node in the list.
        """
        Node.__init__(self, data)
        self.next = None
```

A basic implementation of a linked list will have a constructor and a method for adding new nodes to the end of the list. To get to the end of the list, start at the `head` of the list. Then traverse the list by going from node to node until the end is reached. Then, set the `next` attribute of the last node to be the new node. This is done in the following class. See Figure 1.2 for an illustration.

```
# Location: Lab4_Spec.py
from Node import LinkedListNode as Node

class LinkedList(object):
    """A class for creating linked list objects."""

    def __init__(self):
        """Creates a new linked list.
        Create the head attribute and set it
        to None since the list is empty
        """
        self.head = None

    def add_node(self, data):
        """Create a new Node containing the data
        and add it to the end of the list
        """

        new_node = Node(data)
        if self.head is None:
            # If the list is empty, point the head attribute to the new node.
            self.head = new_node
        else:
            # If the list is not empty, traverse the list and place the new_node ↩
                at the end.
            current_node = self.head
            while current_node.next is not None:
                # This moves the current node to the next node if it is not empty
                current_node = current_node.next

            current_node.next = new_node
```

Problem 2. Implement the `__str__` method for the `LinkedList` class so that when a `LinkedList` instance is printed, its output matches that of a Python list.

```
# Example of what printing a LinkedList object should like.
>>> from Lab4_spec import LinkedList
>>> my_list = LinkedList
```

Figure 1.3: Include a picture of the `remove_node` method and describe it.

Figure 1.4: Include a picture of the `insert_node` method and describe it.

```
>>> my_list.add_node(1)
>>> my_list.add_node(2)
>>> my_list.add_node(3)
>>> print(my_list)
[1, 2, 3]
```

In addition to adding new nodes to the end of a list, it is also useful to remove nodes and insert new nodes at specified locations. To delete a node, all references to the node must be removed. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to none.

```
# A naive node removal - Does not work.

def remove_node(self, data):

    # Find the node whose next node contains data
    current_node = self.head
    while current_node.next.data != data:
        current_node = current_node.next

    # Remove the next reference to the node that
    # Is to be deleted.

    current_node.next = None
```

Since the only reference to the node that is deleted is the previous node's next attribute, this will delete the node. However, since the only reference to the next node came from the deleted node, it also will deleted. This will continue to the end of the list. Thus, deleting one node in this manner deletes the remainder of the list. This can be remedied by pointing the previous node's next attribute to the node after the deleted node. Then there will be no reference to the removed node and it will be deleted. See Figure 1.3 for an illustration.

```
# A node removal that works

def remove_node(self, data):

    # First, check if the head is the node to be removed.
    # If so, then set the head to be the node after head.
    # This will remove the only reference to head, so it
    # will be deleted.
    if self.head.data == data:
        self.head = self.head.next
    else:
        current_node = self.head
        # Move current_node through the list until it points
        # To the node that precedes that target node.
```

```
        while current_node.next.data != data:
            current_node = current_node.next

        # Point the current node to the node after the
        # node that is to be deleted.

        new_next_node = current_node.next.next
        current_node.next = new_next_node
```

Problem 3. Though the above code works to remove specified nodes, it is not quite complete. Modify the `remove_node` method so that if the user tries to remove a node that is not in the list the method prints "Node not in list." The method should function as follows:

```
>>> my_list = LinkedList()
>>> my_list.add_node(1)
>>> my_list.add_node(2)
>>> print(my_list)
[1, 2]
>>> my_list.remove_node(2)
>>> print(my_list)
[1]
>>> my_list.remove_node(2)
Node not in list.
```

Problem 4. Inserting a node will use similar techniques to removing a node. Add an `insert_node` method to the `LinkedList` class that inserts a new node before the first node that contains the data specified by the user. This function will require two arguments: the data for the new node, and the data of the node before which the new node will be inserted. For example, the function should work as follows.

```
>>> my_list = LinkedList()
>>> my_list.add_node(1)
>>> my_list.add_node(3)
>>> print my_list
[1, 3]
>>> my_list.insert_node(2, 3)
>>> print my_list
[1, 2, 3]
```

See Figure 1.4 for an illustration of the `insert_node` method. Note that since `insert_node` inserts a node before a specified node, it is not possible to `insert_node` at the end of the list.

Doubly-Linked Lists

A doubly-linked list is a linked list where each node keeps track the node that precedes it as well as the node that follows. See Figure **??** for an illustration.

```
# location: Node.py

class DoublyLinkedListNode(Node):
    """A node for a doubly-linked list"""
    def __init__(self, data):
        """Set prev and next attributes"""
        Node.__init__(self, data)
        self.prev = None
        self.next = None
```

All of the methods for linked lists can be implemented for doubly-linked lists. See Figures A and B for illustrations of the insert and remove methods.

Problem 5. Implement a doubly-linked list class called `DoublyLinkedList` that builds a doubly-linked list of `DoublyLinkedListNode` instances. Add an attribute called `tail` that keeps track of the node at the end of the list. Inherit from the `LinkedList` class and overwrite the `add_node`, `remove_node`, and `insert_node` methods. Use the `tail` attribute to make the `add_node` method more efficient.

Problem 6. Implement a new data structure called a sorted linked list. A sorted linked list adds new nodes so that their data is in order. Inherit this class from `DoublyLinkedList`. Inherit the node class for this data structure from `StrNode`, and modify it so that it is compatible with doubly-linked lists. The only method that needs to be changed from `DoublyLinkedList` is `add_node`. When a new node is added, traverse the list until the data in the next node is greater than or equal to the data for the new node. Then insert the new node so that order is maintained. For example, the class should function as follows:

```
>>> from Lab4_spec import SortedLinkedList
>>> sorted_list = SortedLinkedList()
>>> sorted_list.add_node('aardvark')
>>> sorted_list.add_node('backgammon')
>>> print sorted_list
['aardvark', 'backgammon']

>>> sorted_list.add_node('zebra')
>>> sorted_list.add_node('year')
>>> print sorted_list
['aardvark', 'backgammon', 'year', 'zebra']
```

Import the `Lab4_data` module. This includes a method called `create_word_list` that returns a list of words that are out of order. Add these words to a sorted

linked list.