Lab 1

Data Structures I: Lists

Lab Objective: *Implement linked lists as an introduction to 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.

Data structures are specialized objects for organizing data. There are many kinds, 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 require different structures for optimal performance.

Nodes

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

If we thought of data as several types of objects that need to be stored in a warehouse, a node would be a standard size box that can hold all the different types of objects. Suppose a 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.

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.

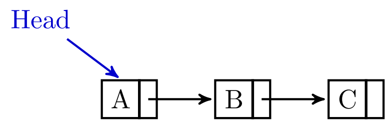

Figure 1.1: A Singly-linked List. The head attribute tracks the first node.

```python
# LinkedLists.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
```

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

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

# Access a node's data.
>>> list_node.data
[1, 'abc']
```

Problem 1. Add extra functionality to the `Node` class by implementing the `__str__` magic method so that it returns a string representation of its data. Also implement the `__lt__`, `__eq__`, and `__gt__` comparison magic methods so that the data stored inside of two Nodes is compared. For example, a Node `x` is less than a Node `y` if and only if the data contained in `x` is less than the data contained in `y`.

NOTE

Often the data stored in a node is actually a *key* value. The key could be a pointer, a dictionary key, or the index of an array where the true desired information resides. However, for simplicity, in this and the following lab we store actual data in node objects, not references to data located elsewhere.

Linked Lists

A linked list is a 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.

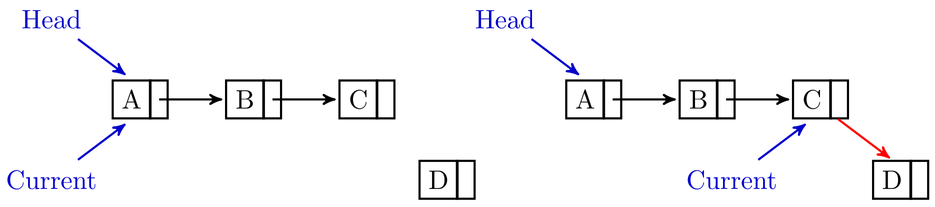

Figure 1.2: To add a new node to the end of the list, create a new node and start at the head. Iterate to the last node in the list and connect it to the new node.

```python
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 initialize an
        attribute for 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.

```python
class LinkedList(object):
    """Singly-linked list data structure class.
    The first node in the list is referenced to by 'head'.
    """
    def __init__(self):
        """Create a new empty linked list. Create the head
        attribute and set it to None since the list is empty.
        """
        self.head = None

    def add(self, data):
        """Add a new Node containing 'data' to the end of the list."""
        new_node = LinkedListNode(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:
                # Move current_node to the next node if it is nonempty.
                current_node = current_node.next
            # current_node now points to the last node in the list.
            current_node.next = new_node
```

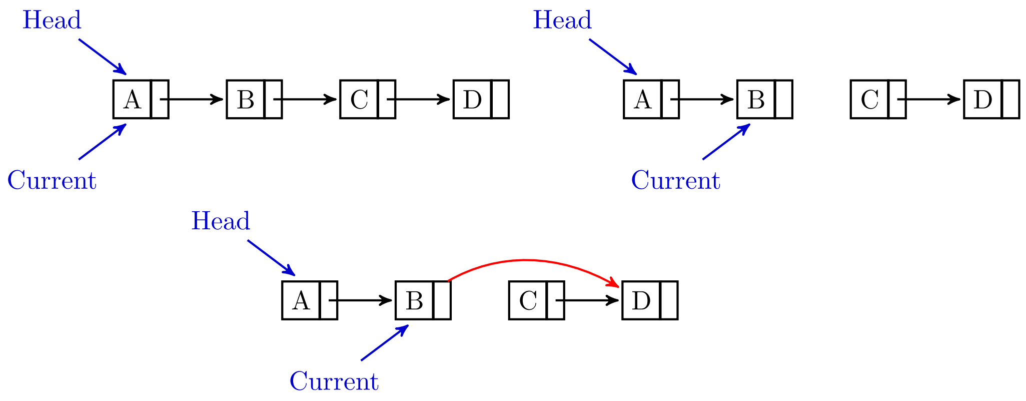

Figure 1.3: By disconnecting B from C, C and D are deleted since nothing points to C. To keep D from being lost, connect B to D first. Then only C is deleted.

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

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. Then Python will automatically delete the object, since 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.

```
class LinkedList(object):
    def __init__(self):
        # ...
    def add(self, data):
        # ...

    def remove(self, data):
        """Remove the Node containing '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 target node
        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.

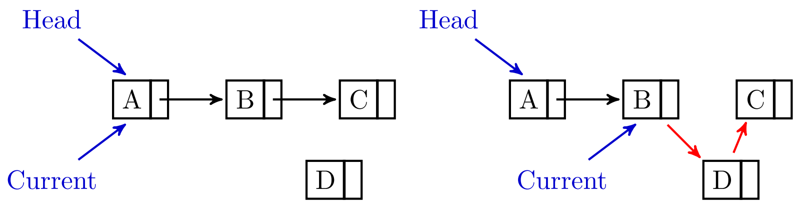

Figure 1.4: To insert D before C, find the node before C and set the connections.

```python
class LinkedList(object):
    def __init__(self):
        # ...
    def add(self, data):
        # ...

    def remove(self, data):
        """Remove the Node containing 'data'."""
        # First, check if the head is the node to be removed. If so, set the
        # new head to be the first node after the old head. This removes
        # the only reference to the old head, so it is then 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 the target node.
            while current_node.next.data != data:
                current_node = current_node.next

            # Point current_node to the node after the target node.
            new_next_node = current_node.next.next
            current_node.next = new_next_node
```

WARNING

Python keeps track of the variables in use and automatically deletes a variable if there is no access to it. In many other languages, leaving a reference to an object without explicitly deleting it could cause a serious memory leak. See here for more information on python's auto-cleanup system.

Problem 3. Though the above code works to remove specified nodes, it is not quite complete. Modify the `remove` method to account for possible errors: if the list is empty, or if the target node is not in the list, raise a `ValueError` with error message "`data` is not in the list."

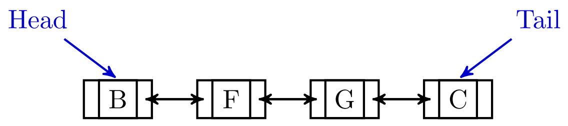

Figure 1.5: A Doubly-linked List. The tail attribute tracks the last node.

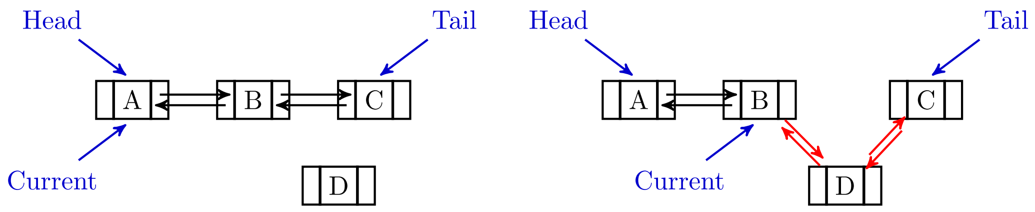

Figure 1.6: Insertion for Doubly-linked Lists.

Problem 4. Add an `insert` method to the `LinkedList` class that inserts a new node before the first node in the list that contains the data specified by the user. Accept data for the new node (`data`) and data for the node before which the new node will be inserted (`place`). If the list is empty, or there is no node containing `place` in the list, raise a `ValueError` with error message "`place` is not in the list."

See Figure 1.4 for an illustration of the `insert` method. Note that since `insert` inserts a node before a specified node that is already in the list, it is not possible to `insert` at the end of the list or to an empty 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. The end of the list is also typically kept track of with a `tail` attribute. See Figure 1.5 for an illustration.

```python
# LinkedLists.py

class DoublyLinkedListNode(LinkedListNode):
    """A Node class for doubly-linked lists. Inherits from the 'Node' class.
    Contains references to the next and previous nodes in the list.
    """
    def __init__(self,data):
        """Set the next and prev attributes."""
        Node.__init__(self,data)
        self.next = None
        self.prev = None
```

All of the methods for linked lists can be implemented for doubly-linked lists. See Figures 1.6 and 1.7 for illustrations of the insert and remove methods.

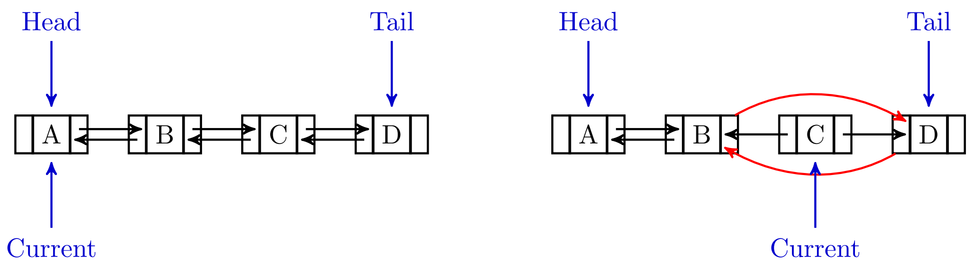

Figure 1.7: Removal for Doubly-linked Lists.

Problem 5. Implement a `DoublyLinkedList` class that inherits from `LinkedList` and uses `DoublyLinkedListNode` instances to build the list. Add a `tail` attribute that keeps track of the node at the end of the list. Overwrite `add`, `remove`, and `insert`. Raise the same exceptions as before.

Problem 6. Implement a sorted linked list. This data structure adds new nodes strategically so that the data is always kept in order. Inherit this class from `DoublyLinkedList`, and override the `add` and `insert` methods. 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, thereby preserving the ordering. Also override the `insert` method with the following:

```
def insert(self, *args):
    raise ValueError("insert() has been disabled for this class.")
```

This effectively disables `insert` for the `SortedLinkedList` class and prevents the user from accidentally inserting a node in a location that would disrupt the ordering. The `*args` argument allows `insert` to receive any number of arguments without raising a `TypeError` exception.

To test this data structure, import the provided `WordList` module. This includes a method called `create_word_list` that reads each line of text from a file and returns it as a randomly-ordered list of strings. Write a function called `sort_words` that sorts the list generated by `create_word_list` by adding them to a `SortedLinkedList` object. Then return the object.

Warning

`English.txt`, the default source file for `create_word_list`, contains over 58,000 English words. Sorting the entire data set should take about 15 minutes. Test your data structure on small data sets first.

Note

Python has many quick sorting methods. Even on the seemingly large data set of over 58,000 words used in the preceding problem, the `sort` method for Python lists is almost instantaneous. 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

Often it is wise to restrict the user's access to the some of the data within a structure. The three most common and basic restricted-access structures are stacks, queues, and deques. Each structure restricts the user's access differently, making them ideal for different situations. These structures will reappear in many future labs and applications.

- A *stack* is a *Last In, First Out* structure (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 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.

- A *queue* (pronounced "cue") is a *First In, First Out* structure (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 line at the bank: the person at the front of the line is served next, 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 (`push` and `pop` for queues are sometimes called `enqueue` and `dequeue`, respectively).

- A *deque* (pronounced "deck") is 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`.

  In practice, a deque can also be used as a stack or a queue. If we restrict our usage to `append` and `pop` (or to `appendleft` and `popleft`), we effectively have a stack. Similarly, if we are restricted to `append` and `popleft` (or to `appendleft` and `pop`), we effectively have a queue.

The `collections` module in the Python standard library has a `deque` object, 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.

Problem 7. (Optional) Write `Stack`, `Queue`, and `Deque` classes.

The `Deque` class should inherit from the `DoublyLinkedList` class. Use inheritance to implement the `append`, `appendleft`, `pop`, and `popleft` methods as described in the preceding section. The `append` and `appendleft` methods should accept a single parameter (the data to be added) and return nothing, while the `pop` and `popleft` methods should accept no parameters and return a single value (the data removed). Disable all other methods to restrict data access.

The `Stack` and `Queue` classes should inherit from the `Deque` class. Add a `push` method and overload the `pop` method in each class to match the behaviors described in the preceding section. Disable any other methods.