

General linked list	N	Uni	6	General linked list	~100	Implement singly linked list: size/(all insert)/(all delete)/access
	N	Uni	7		~100	Implement doubly linked list: size/(all insert)/(all delete)/access
	N	Uni	8		~100	Implement circular linked list: size/(all insert)/(all delete)/access
400	N	Uni	9		~100	Sample application of general linked list

Topics (original plan now outdated)

1. What is a Data Structure?
 1. Basic data structures (list, set, tuples, dictionary)
 2. Advanced data structures (**memory usage? Yes but keep it simple**)
 3. Linear Lists
2. What is a Linked List?
 1. Linked List Basics - nodes, list head, linking
 2. Benefits of using Linked Lists
3. Singly Linked List
 1. Structure - Node, Size, Links
 2. Storing/accessing data - Add, delete Access
4. Doubly Linked List
 1. Structure - Node, Size, Links
 2. Storing/accessing data - Add, delete Access
5. Circular Linked List
 1. Structure - Node, Size, Links
 2. Storing/accessing data - Add, delete Access
6. Linked List Applications
 1. Storing test results (storing printing and statistics)
 2. Factory Production Line (Linked list conveyor belt - produce, process, ship)
 3. Multi - Player D&D style battle game (make and then store a linked list of characters and enemies and have turn based gameplay)

1. What is a Data Structure?

Data structures are used to organize data and store it in a single structure. They allow related data to be grouped together and make the data easier to access. Without data structures, you would be forced to store every piece of information inside its own variable, which would become extremely unorganized and difficult to manage!

There are many examples of when to use data structures, some of which you have seen already. For example, if you want to store a list of names, then you might use a list. Examples

for when to use data structures are when you need to store a list of names or when you want to keep a sequence of points in a graph.

There are many different types of data structures, but they can all be grouped under two main categories which we'll learn about next; basic data structures and advanced data structures.

1.1 Basic Data Structures

Remember the **Time and Space Complexity** concepts we learned in the last chapter? These concepts are essential for understanding how efficiently data structures and algorithms handle large datasets, such as those in social networks like Facebook or X.

In the digital world, especially on social networks, computers manage massive amounts of data. Imagine billions of people, each with their own circle of friends, posts, likes, and comments!

Both **Time and Space Complexity** are crucial when designing these systems that can handle the vast, interconnected datasets typical in social networks.

Careful consideration is needed to ensure that applications remain responsive and scalable, even as the number of users and the volume of data grow exponentially.

- **Time Complexity:** Indicates the computational time an operation, such as adding or removing an element is performed.
- **Space Complexity:** Represents the amount of memory a data structure or algorithm consumes.

This brings us to the importance of **Data Structures**.

Unlike **Algorithms**, which define how operations such as searching, sorting, or modifying data are carried out, **Data Structures** define how the data is stored.

The most basic **data structures** in Python are the built-in arrays you have seen before in previous Python lessons. These include:

List

Lists are versatile data structures in Python, capable of **holding elements of different types**. They are **mutable**, so elements can be changed, added, or removed, and they are **ordered**, meaning the elements are in a specific order and can **contain duplicate values**.

Below is a quick review of common list related statements and methods:

- Declaration: `my_list = [1, 2, 3]`
- Access: `element = my_list[0]`
- Modification: `my_list[1] = 'new value'`

Method	Description	Example
append(x)	Adds an item x to the end of the list.	<pre>my_list = [1, 2, 3] my_list.append(4) print(my_list) Output: [1, 2, 3, 4]</pre>
insert(index, x)	Inserts an item x at a given position. All elements after the position index are shifted.	<pre>my_list = [1, 2, 3] my_list.insert(1, 'a') print(my_list) Output: [1, 'a', 2, 3]</pre>
del list[index]	Deletes the element at a specific index.	<pre>my_list = [1, 2, 3] del my_list[1] print(my_list) Output: [1, 3]</pre>
remove(x)	Removes the first occurrence of the value x from the list.	<pre>my_list = [1, 2, 3, 2] my_list.remove(2) print(my_list) Output: [1, 3, 2]</pre>
pop(index) & pop()	Removes and returns the item at the given index. If no index is specified, it removes and returns the last index .	<pre># pop() with index my_list = [1, 2, 3] popped_item = my_list.pop(1) print(popped_item) print(my_list) Output: 2 [1, 3] # pop() without index popped_item = my_list.pop() print(popped_item) print(my_list) Output: 3 [1]</pre>
list[index]	Accesses the element at the index.	my_list = [1, 2, 3]

		print(my_list[1]) Output: 2
index(x)	Returns the index of the first occurrence of x in the list.	my_list = [1, 2, 3] print(my_list.index(2)) Output: 1

Set

Sets are **mutable, unordered** collections of unique (**no duplicates**) items. They are most commonly used for mathematical operations like union, intersection, and difference.

Below is a quick review of common set related statements and methods:

- Declaration: my_set = {1, 2, 3}
- Check Membership: exists = 3 in my_set

Method	Description	Example
add(x)	Adds an element x to the set.	my_set = {1, 2, 3} my_set.add(4) print(my_set) Output: {1, 2, 3, 4}
remove(x)	Removes an element x from the set. Raises a KeyError if the element is not present .	my_set = {1, 2, 3} my_set.remove(2) print(my_set) Output: {1, 3}
discard(x)	Removes an element x from the set if it is present. Does nothing if the element is not present .	my_set = {1, 2, 3} my_set.discard(2) print(my_set) Output: {1, 3} # Does nothing if the element is not present my_set.discard(5) print(my_set) Output: {1, 3}
pop()	Removes and returns an arbitrary element from the	my_set = {1, 2, 3} popped_element =

	set. Raises a KeyError if the set is empty.	<pre>my_set.pop() print(popped_element)</pre> <p># Output: 1 (or 2 or 3, since sets are unordered)</p> <pre>print(my_set)</pre> <p># Output: Remaining elements in the set</p>
--	--	--

Tuple

Tuples are **ordered, immutable** collections, often useful for scenarios where data shouldn't be modified and **are allowed to contain duplicate values**.

Below is a quick review of common tuple related statements and methods:

- Declaration: `my_tuple = (1, 2, 3)`
- Access: `element = my_tuple[0]`
- Unpacking: `a, b, c = my_tuple`

Method	Description	Example
<code>tuple[index]</code>	Access an element at a specific index.	<pre>my_tuple = (1, 2, 3, 4) print(my_tuple[1])</pre> <p># Output: 2</p>
<code>tuple[start:end]</code>	Create a sub-tuple from a given range of indices (inclusive of start, exclusive of end).	<pre>my_tuple = (1, 2, 3, 4, 5) print(my_tuple[1:3])</pre> <p># Output: (2, 3)</p>
<code>len(tuple)</code>	Get the number of elements in the tuple.	<pre>my_tuple = (1, 2, 3, 4) print(len(my_tuple))</pre> <p># Output: 4</p>
<code>tuple.count(x)</code>	Count the occurrences of a value.	<pre>my_tuple = (1, 2, 3, 2, 4) print(my_tuple.count(2))</pre> <p># Output: 2</p>
<code>x in tuple</code>	Check if an element exists in the tuple.	<pre>my_tuple = (1, 2, 3, 4) print(2 in my_tuple)</pre> <p># Output: True</p>

		print(5 in my_tuple) # Output: False
--	--	---

Dictionary

Dictionaries are **ordered** collections that use **key-value pairs** for **efficient data retrieval**. Keys must be **unique** and **immutable** (cannot be changed, e.g., strings, numbers, or tuples); values can be of any type.

Below is a quick review of common dictionary related statements and methods:

- Declaration: `my_dict = {'key1': 'value1', 'key2': 'value2'}`
- Addition/Modification: `my_dict['key'] = 'new_value'`

Method	Description	Example
<code>del dict[key]</code>	Remove the key-value pair.	<pre>my_dict = {'a': 1, 'b': 2, 'c': 3} del my_dict['b'] print(my_dict) # Output: {'a': 1, 'c': 3}</pre>
<code>len(dict)</code>	Get the number of key-value pairs in the dictionary.	<pre>my_dict = {'a': 1, 'b': 2, 'c': 3} print(len(my_dict)) # Output: 3</pre>
<code>dict.get(key)</code>	Access the value associated with a key, with a default value if the key is not found. Unlike direct indexing, it does not raise a <code>KeyError</code> .	<pre>my_dict = {'a': 1, 'b': 2, 'c': 3} print(my_dict.get('b')) # Output: 2 print(my_dict.get('d', 'Not Found')) # Output: 'Not Found'</pre>
<code>dict.pop(key)</code>	Remove and return the value associated with a key.	<pre>my_dict = {'a': 1, 'b': 2, 'c': 3} value = my_dict.pop('b') print(value) # Output: 2</pre>

		print(my_dict) # Output: {'a': 1, 'c': 3}
--	--	---

COMPARISON TABLE OF LIST, TUPLE, SET, AND DICTIONARY

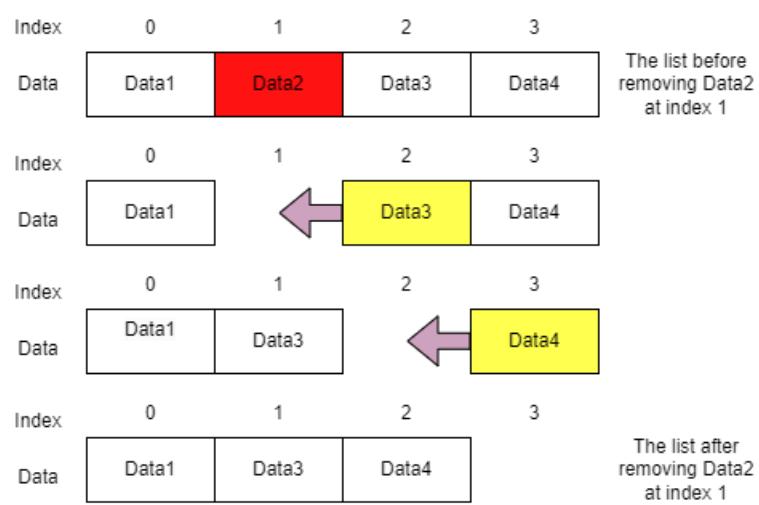
	List	Tuple	Set	Dictionary
Indexing	Indexed	Indexed	Not indexed	Keyed
Mutability	Mutable	Immutable	Mutable	Mutable
Order	Ordered	Ordered	Unordered	Ordered
Duplicates Allowed	Yes	Yes	No	Keys: No Values: Yes

These structures are easy to use and will form the basis of most data management in Python programs. Here is a list of the advantages of using these data structures:

- Modification of data structures is easy since it allows for flexibility in adapting to changing requirements.
- Designed to optimize time efficiency for common operations like insertion, deletion, and retrieval.
- Representation of data is easy since it provides a clear and intuitive way to express information.

However, since they are designed to be simple and easy to use for a wide variety of scenarios, they are not necessarily the most efficient for every occasion, especially when the scenario becomes more specialized and specific.

An example of inefficiency is the removal of data from a list as we mentioned in the last lesson. When an element is removed from a list, all elements that came after that element need to be individually shifted into different locations in the computer's memory to maintain the correct order.



Removing the “Data2” at index 1 requires that “Data3” and “Data4” be moved to the left by 1 index. We can see that it would be more efficient if they didn't need to be moved.

The inefficiencies and issues with the use of the basic inbuilt data structures when attempting to write programs necessitate the creation and use of more complex data structures that, through their design, solve problems efficiently.

Advanced data structures, while not being used in simple applications, are invaluable when creating complex algorithms for processes such as sorting data and searching for specific data in large lists of data.

Exercises 1 - 5 (T/F)

Exercise 1

Lists in Python can only contain elements of the same data type. Is this statement true or false?

- True
 False

Explanation: Lists in Python can contain elements of different data types. For example, a list can have integers, strings, and even other lists as elements.

Exercise 2

Sets in Python are ordered collections of items. Is this statement true or false?

- True
 False

Explanation: Sets in Python are unordered collections of unique items. The order of elements in a set is not guaranteed and may change.

Exercise 3

Tuples are mutable data structures in Python, which means their elements can be changed after creation. Is this statement true or false?

- True
- False

Explanation: Tuples are immutable data structures in Python. Once a tuple is created, its elements cannot be changed, added, or removed.

Exercise 4

Dictionary keys in Python are immutable. Is this statement true or false?

- True
- False

Explanation: Dictionary keys in Python must be immutable types such as strings, numbers, or tuples. Mutable types like lists cannot be used as keys.

Exercise 5

The space complexity of a data structure refers to the amount of disk storage it requires. Is this statement true or false?

- True
- False

Explanation: Space complexity typically refers to the amount of memory (RAM) that a data structure or algorithm uses during its execution, not disk storage.

Exercises 6 - 8 (MCQ)

Exercise 6

Which method can be used to add an element to a set in Python? Select all that apply.

- append()
- ~~add()~~
- insert()
- extend()

Explanation: The add() method is correct as it adds a single element to a set, and won't add if the element already exists in the set. The other answers are all used for lists not sets:

- The append() method adds elements to lists.
- The insert() method adds elements at a particular index in a list.

- The extend() method adds elements from another iterable to the end of the list.

Exercise 7

In Python, which of the following are valid ways to remove an element from a list? Select all that apply.

- `pop(index)`
- `remove(value)`
- `delete(value)`
- `del list[index]`

Exercise 8

Which of the following are NOT characteristics of a dictionary in Python? Select all that apply.

- Unordered
- Mutable
- Allows duplicate keys
- Stores data in key-value pairs

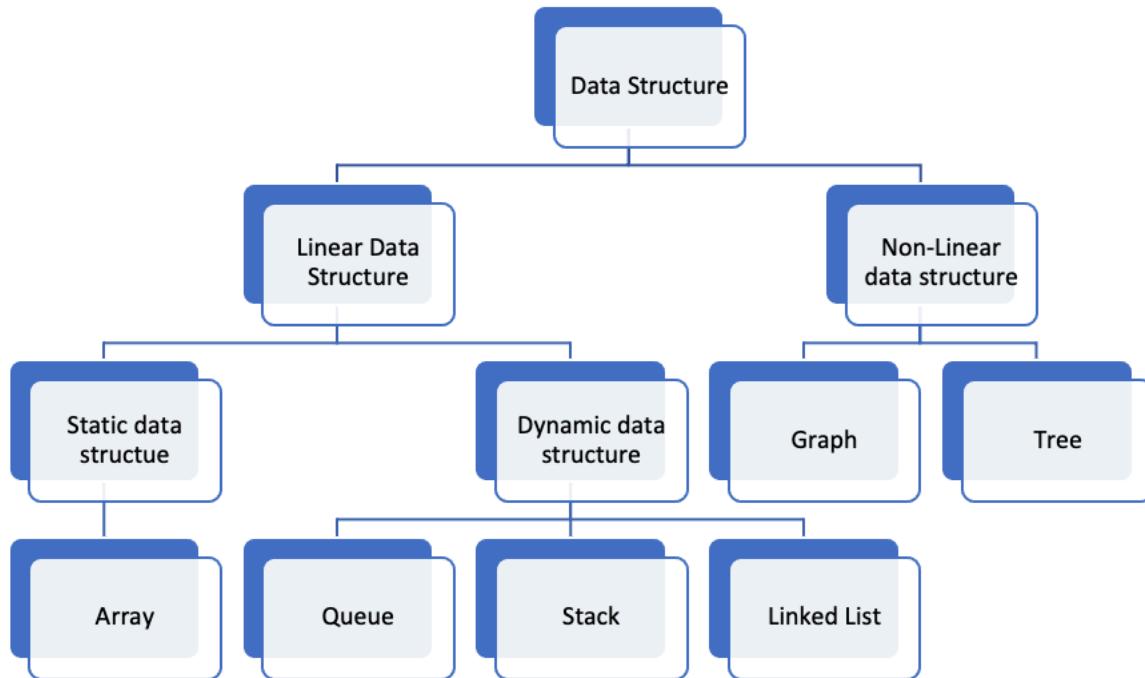
Explanation:

1. **Unordered:** This is **true before Python 3.7**. Dictionaries in Python are unordered collections (before Python 3.7). From Python 3.7 onward, dictionaries maintain insertion order as an implementation detail.
2. **Mutable:** This is true. Dictionaries are mutable, meaning you can change, add, or remove items after the dictionary is created.
3. **Allows duplicate keys:** This is **false**. Dictionaries do **not** allow duplicate keys. If you try to add a duplicate key, the value associated with that key will be updated to the new value.
4. **Stores data in key-value pairs:** This is true. Dictionaries store data as key-value pairs.

Exercises 9 - 20 ()

1.2 Advanced Data Structures

Python provides classes and other object-oriented programming concepts to implement and manage many advanced data structures. Below is a diagram of some of the most used data structures.



You will notice that there are 2 distinct kinds of data structures:

- Linear
- Non-Linear

Linear data structures such as arrays, queues, stacks, and linked lists, arrange elements sequentially, ensuring each element connects to the next in a defined order.

In contrast, non-linear data structures, such as graphs and trees, organize data hierarchically or through complex relationships, where elements may connect to multiple others.

For now, we will only introduce these data structures briefly, but later we will dive deeper into their specific implementations.

Since there are so many, choosing the right data structure is crucial when designing and implementing complex algorithms.

Each structure offers **unique benefits** and **trade-offs** in terms of efficiency and usability which depend on the specific requirements of the task at hand.

Linear data structures are collections of data elements arranged in a **specific order**, where each element is connected to the next one in a sequence. This allows traversal through elements one by one. Each element typically connects to its adjacent elements, forming a

chain-like structure.

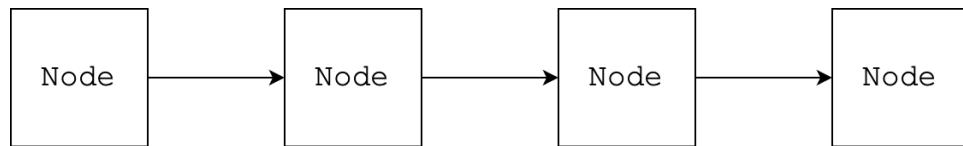
Think about a line of people trying to buy something at a shop - every person has one person in front of them and one person behind them (unless they are at the front or end of the line). This fixed order makes it easy to traverse the structure sequentially.

In linear lists, there is always a **clear beginning** and an **end**. This makes it simpler to go through each element in the list one by one; a process we call **traversal** or **iteration**.

For some types of linear lists, there are different ways to implement them.

A common approach is using a “linked” data structure where each element (node) points to the next, creating a chain-like structure.

Below is a generalized diagram which shows this “linked” structure:



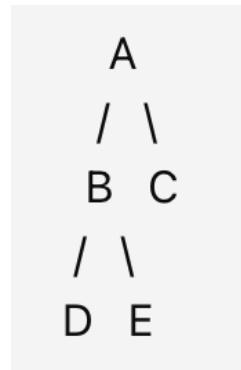
It's like reading a book from the first page to the last, or walking through a line of people, starting with the first person and ending with the last.

As mentioned before, linear data structures include Arrays, Linked Lists, Queues and Stacks.

Each is structured linearly, but they differ in element accessibility. Because of this, each has its own pros and cons in various use cases.

Non-linear data structures differ from linear lists because of the non-linear arrangement of their nodes. A common example of a non-linear data structure is the tree structure, in which each node of the tree can be connected to any number of other nodes.

The tree structure can be depicted as:



These data structures will be explained in a later chapter.

While linear lists are made up of objects linked together with at most two adjacent objects in a chain, non-linear structures contain linking structures that can have any number of links (≥ 0) to any other object in the structure.

This means that these structures have **no limit on the patterns of linking** that can be created, but it also causes their structure to become more **complex** and **challenging** to manage, as performing **traversals** of the data structure **becomes difficult** when it is non-linear.

We will discuss some linear data structures in more detail in this chapter, in particular the classic linked-list. However, in future chapters we will break down all kinds of linear and non-linear data structures.

Exercises 1 - 4 (MCQ)

Exercise 1

What is the primary feature of a linear list?

- Its elements can only be linked to those with the same value.
- Its elements can only be linked to two adjacent elements, one before and one after.
- Its elements can be linked to any other.
- Its elements are ordered by size.

Exercise 2

Select all dynamic linear data structures.

- Queue
- Stack
- Linked List
- Graph

Explanation: Graphs are non-linear kinds of data structure made up of nodes/vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

Exercise 3

What is the primary feature of a non-linear data structure?

- Its objects can be linked to any other to create structure
- Its objects are linked to all others

- Adjacent objects cannot be linked
- A unique ID is given to each object

Exercise 4

Select all non-linear data structures.

- Queue
- Tree
- Linked List
- Graph

2. What Is a Linked List

2.1 Memory Address

Before we go into what a Linked List is and how it can be used and created, let's have a look at what memory addresses are.

A memory address is a **unique location** in the computer's memory where data is stored.

Think of memory addresses like unique **street addresses** for houses in a neighborhood. **Each house (variable)** has its **distinct address**, making it easy for **mail (data)** to be **delivered** to the right place.

In our case, the computer uses memory addresses to ensure that data is stored and retrieved accurately.

When a variable is **created**, the computer will **allocate** a memory address to it to ensure it can be located when needed.

The memory address allocated to the variable is **different** each time the program is executed.

For now we don't need to know how the computer allocates these addresses, we just need to know that **every variable has a unique memory address**. Here is an example of how multiple variables are stored in the memory.

Adresses	0x0001	0x0002	0x0003	0x0043	0x0044
Data	"Cat"	Object("Anna", female, 13)	51.32	...	132	Object("john", male, 15)

In the diagram above, you can see that every string ("Cat"), number (51.32, 132) and object is stored in its own memory block.

Each data variable will have a unique memory address, for example the memory address of the string "Cat" is '0x0001'.

"0x" is a prefix that indicates that the following number is a **hexadecimal**; Hexadecimal is a numeral system that uses 16 symbols (0-9 and A-F, where A represents 10, B represents 11, and so on up to F representing 15).

We won't be going into more detail than what we have covered here and the specific computer memory structure will not be discussed in this lesson. But what is important to take away is that as long as we keep track of these addresses that point to the data we want, then we don't need to worry about where the data is really stored.

Note: We won't use memory addresses for coding, but it is important to know how the

computer works behind the scenes.

Exercises 1 - 2 (MCQ)

Exercise 1

What is a memory address?

- A place where the program is executed
- A place where all the variables with same data type are stored
- A unique location in the computer memory
- A place where the computer stores the program

Explanation:

The correct answer is ‘A unique location in the computer memory’ because memory addresses uniquely identify where each piece of data is stored, allowing efficient retrieval.

Exercise 2

Which of the following is true? Choose all that apply.

- The computer will randomly allocate memory addresses to each variable
- The memory address allocated to each variable is always the same for each execution
- Variables with the same data can share the same memory address
- Each time the program is executed each variable will be allocated a new memory address

Explanation:

Option 1: **False**. Memory allocation is not random. The computer uses specific memory management techniques and algorithms to allocate addresses based on available memory and program execution needs.

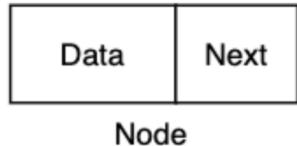
Option 2: **False**. Memory addresses for variables can change with each execution of a program because the program might be loaded at different memory locations by the operating system.

Option 3: **True**, but with clarification. In Python, certain immutable objects (like small integers or strings) may have the same memory address due to interning or caching mechanisms. For instance, small integers between -5 and 256 and string literals are often stored in the same memory location.

Option 4: **True**. When a program runs, the memory allocation for its variables typically occurs in a new memory space, so the variables are likely to have different memory addresses in each execution.

2.2 Nodes

For advanced data structures, we don't use the term elements like in basic data structures. Instead, we use the term **nodes** because each node needs to store the **data and links** to other nodes. Let's take a closer look at a linked list's **node** structure. A node typically contains two fields:



- **Data:** This is where the **actual value or information** is stored. It can be as simple as a variable storing a string, a more complex object or even a full data structure.
- **Next:** It holds the **memory address** of the next node. It is used to **point to the next node** in the list.

Creating the Node Class

Each node is created as an object that stores data and the reference to the next node. Thus, we will need to create a class that contains two parts:

1. **Data** - Can be any type of data (e.g., integers, strings, or objects).
2. **Next** - Stores a reference to the next node or `None` if it is the last node.

Example

```
class Node:  
    def __init__(self, data):  
        self.data = data      # Data stored in the node  
        self.next = None       # Reference to the next node in the  
list
```

This `Node` class serves as the blueprint for creating nodes in a linked list. The `__init__` method initializes a `Node` with its required data.

Initially the `next` attribute is set to `None` as there is only one `Node` in the linked list.

Later, when `Nodes` are linked, the `next` attribute will be updated to point to the subsequent `Node` in the list.

Currently, a single node alone is no more useful than a single variable and needs to be linked to other nodes to populate the data structure.

Linking is the key characteristic of a linked list, as it is what gives it **structure**.

Linking is implemented by saving a reference to a node as data in another node, allowing the list to be structured as a chain of linked nodes, each connected to another.

In summary, each node in a linked list acts as a container that stores data and a reference to the next node. This structure allows nodes to be connected in sequence, forming a chain. By linking nodes together, we create a flexible structure where data can be added, removed, or traversed efficiently.

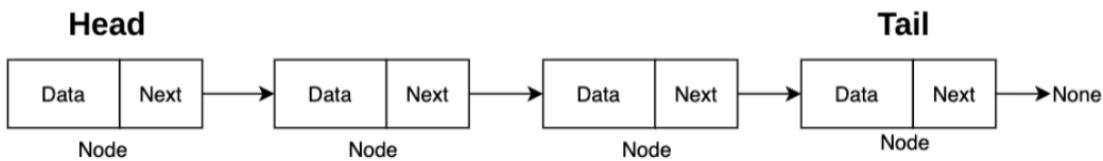
In a linked list, we keep track of the first node, known as the **head node**. The head node serves as the entry point to access the entire list.

By knowing the head node, we can traverse the list and perform various operations, using the head node as the starting point for traversal. By following the references (or 'next' pointers) from one node to the next, we can move through the entire list until we reach the last node where the 'next' pointer is `None`.

In addition to the head node, another important concept in linked lists is the **tail node**. The tail node is the last node in the linked list, and its next reference points to `None`, indicating the end of the list. By setting the tail node's next pointer to `None`, we ensure a clear endpoint for the list. This is crucial for preventing errors during traversal, such as infinite loops, where the list might be mistakenly traversed indefinitely if a clear endpoint is not set.

Tracking the tail node also allows for efficient appending of new nodes at the end of this list.

Below is a visual representation of the structure of a Linked List and its head and tail nodes:



Example - Linked Nodes

```
# Create nodes
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)

# Link nodes
node1.next = node2
node2.next = node3

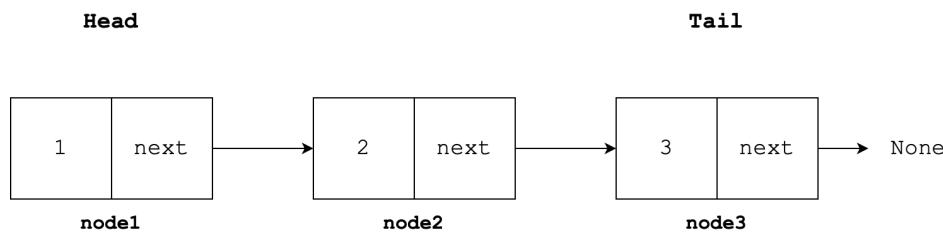
# The last node's next attribute points to None
node3.next = None
```

In this example, we create three nodes (`node1`, `node2`, and `node3`) with data values 1, 2, and 3, respectively.

We then link the nodes together by setting the `next` attribute of `node1` to point to `node2`, and the `next` attribute of `node2` to point to `node3`.

Finally, the `next` attribute of `node3` is set to `None` to signify the end of the list.

Below visualizes the resulting linked list from the above steps:



Why is tail's next field set to None?

Because:

- It signals the end of the list, meaning it is the last node in this linked list and there are no next Nodes.
- Ensure safe traversal — when a node's pointer is `None`, it's the end. The absence of a clear endpoint can result in errors or infinite loops during traversal operations.

Now, let's proceed with writing some Python code examples to illustrate the creation of a single `Node` and a simple linked list.

```
# Define the Node class
class Node:
    def __init__(self, data):
        self.data = data # Data stored in the node
        self.next = None # Reference to the next node in the list

# Function to print the linked list starting from a given node
def print_linked_list(node):
    while node:
        print(node.data, end=" -> ")
        node = node.next
    print("None")

# Create a single Node
node1 = Node(1) # Create the first node with data 1
```

```

# Print the data of the single Node
print("Data of Node 1:", node1.data) # Output: Data of Node 1: 1

# Print the next pointer of the single Node
print("Next pointer of Node 1:", node1.next) # Output: Next pointer
of Node 1: None

# Create a simple linked list
node2 = Node(2) # Create the second node with data 2
node3 = Node(3) # Create the third node with data 3

# Link the nodes
node1.next = node2 # Link node1 to node2
node2.next = node3 # Link node2 to node3

# Print the linked list starting from the first node
print_linked_list(node1) # Output: 1 -> 2 -> 3 -> None

```

Expected Output:

Data of Node 1: 1
 Next pointer of Node 1: None
 1 -> 2 -> 3 -> None

The above code puts everything we have learnt so far together!

First, we define our `Node` class with the required fields then we create our list! `print_linked_list()` and the other `print()` statements are in place to provide us with a visual representation of our list and are not required for linked lists to function.

Exercises 1 - 11 (T/F)

Exercise 1

A linked list is a linear data structure where elements are stored in nodes.

- True
 False

Exercise 2

In a linked list, nodes are connected in a non-linear structure.

- True

False

Exercise 3

The next pointer of the last node in a linked list points to the head node.

True
 False

Exercise 4

Each node in a linked list contains a reference (link) to the previous element in the list only.

True
 False

Exercise 5

Traversing a linked list starts from the tail node and moves towards the head node.

True
 False

Exercise 6

In a linked list, nodes are stored in contiguous memory locations.

True
 False

Explanation: In a linked list, nodes are not stored in contiguous memory locations. Each node contains a reference (pointer) to the next node, which may be located anywhere in memory. This allows for flexibility in memory usage, unlike arrays that require contiguous memory allocation.

Exercise 7

A node in a linked list typically consists of a data value and a pointer to the next node.

True
 False

Exercise 8

The tail node in a linked list is used to efficiently insert new nodes at the beginning of the list.

True

False

Exercise 9

Deleting a node from a linked list **always** requires traversing the entire list.

True
 False

Explanation: Normally we have access to the head of a linked list, and when we want to delete the first element in a linked list, we can discard the first node and assign the head to the next node.

Exercise 10

Adding a new node at the end of a linked list requires traversing the entire list if there is no tail pointer.

True
 False

Explanation: To add a node at the end of the linked list, you need to traverse the list until you reach the last node, then update its next pointer to point to the new node.

Exercise 11

The elements in a linked list are stored in nodes which can be located anywhere in memory.

True
 False

Explanation:

One difference between a list and a linked list is how elements are stored in memory. In a list, elements must be stored contiguously within an assigned area of memory. In contrast, in a linked list, each node can be stored anywhere in memory, with each node containing a reference to the next node in the sequence.

Exercises 12 - 28 (MCQ)

Exercise 12

What are the basic building blocks of a linked list?

Data
 Variables
 Nodes

Links

Exercise 13

Which of the following statements about linked lists is **false**?

- Each element in a linked list is called a node.
- A linked list is a linear data structure.
- ~~Nodes in a linked list must be stored in contiguous memory locations.~~
- Nodes contain data and a reference (link) to the next node.

Exercise 14

How is the structure of a linked list created?

- ~~By linking nodes to each other manually~~
- Through the naming of variables
- By allocating a continuous block of memory for all nodes
- By creating a standard Python list where each element is a node

Exercise 15

What should the links of nodes be initialized to by default?

- 0
- ~~None~~
- "End"
- head

Exercise 16

What are the two essential components of a node in a linked list?

- Data and index
- ~~Data and reference (pointer)~~
- Index and reference (pointer)
- Index and memory address

Exercise 17

What type of data can be stored in a node? Select all correct answers.

- ~~Integers~~
- ~~Strings~~
- ~~Objects~~
- ~~Booleans~~

Exercise 18

What structure does a basic linked list have?

- Square
- Chain
- Tree
- Spiral

Exercise 19

What is the purpose of the head node?

- To act as the initial access point for traversing the linked list
- To store data about the linked list
- To be the only node that can be accessed
- To prevent nodes from being added at the wrong end

Exercise 20

Within the linked list structure a node can only be linked with nodes that are?

- Larger
- Adjacent
- At the end of the list
- At the start of the list

Exercise 21

What does the `next` attribute of a node in a linked list represent?

- The data stored in the next node
- The reference (pointer) to the next node
- The index of the next node
- The memory address of the current node

Exercise 23

Which of the following statements are true about the `next` attribute of the last node in a linked list? Select all correct answers.

- Numeric
- It points to None

- It points to the previous node
- ~~It is used to signify the end of the list~~

Exercise 24

A linked list is a _____ data structure?

- Numeric
- Linear
- Static
- Non-linear

Exercise 25

In a singly linked list, if you want to access the n-th node from the head node, what operation is required?

- Directly access through the array index
- Directly jump to the n-th node
- ~~Sequentially traverse the linked list until the n-th node~~
- None of the above

Exercise 27

What does the tail node in a linked list point to?

- The head node
- The previous node
- ~~None (or null)~~
- The tail node itself

Exercise 28

What is the most correct definition of a linked list?.

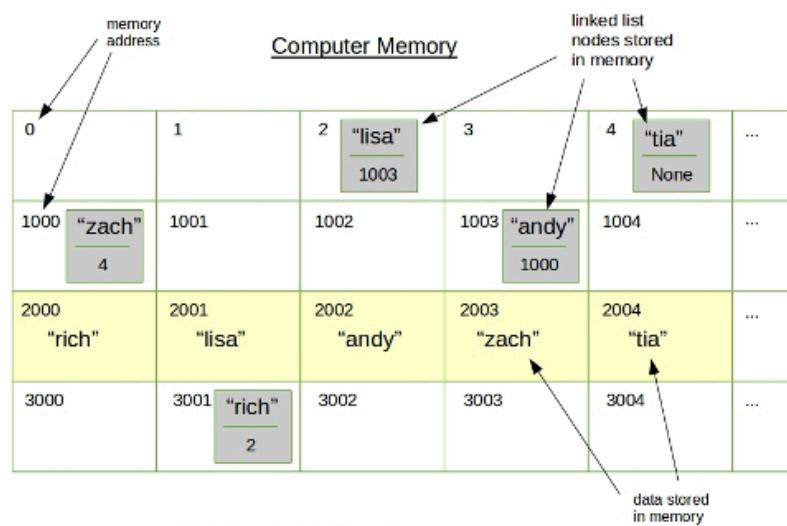
- A linear data structure where elements are stored in an array.
- ~~A linear data structure where elements are stored in nodes and each node points to the next node.~~
- A linear data structure where elements are stored in contiguous memory locations.
- A non-linear data structure where elements are connected in a hierarchical manner.

2.3 Linked List Uses (20 Exercises)

It is important to understand why a linked list might be preferable to Python's built-in data structures.

Key Characteristics of Linked List:

- **Dynamic size:** Linked lists can grow or shrink dynamically during runtime. Nodes can be added or removed without resizing or shifting elements, unlike arrays or Python lists.
- **Time Complexity:** Insertion at the beginning of the list takes $O(1)$ time, while insertion at the end or in the middle of the list takes $O(n)$ time in the worst case due to the need to traverse the list.
- **Non-contiguous Memory:** Unlike arrays, which store elements in contiguous memory locations, nodes in a linked list can be scattered throughout the memory. Each node is independently allocated and linked to the next node through its next pointer, enabling flexibility in memory allocation.



For example, in list L (in yellow) it stores each element next to each other, but linked lists (in gray) store each node arbitrarily.

3001 (Head node): "rich" -> 2: "lisa" -> 1003: "andy" -> 1000: "Zach" -> 4 (Tail node): tia" -> None

- **Time Complexity:** Memory allocation time is dependent on the underlying memory

manager but generally $O(1)$ time is taken for each allocation.

- **Efficient insertion and deletion:** Inserting or deleting a node in a linked list can be done efficiently, especially at the beginning of the list. Insertion or deletion in the middle or end of the list requires traversing to the desired position. Sometimes insertion and deletion at the end of the list can be performed in $O(1)$ time as long as we have the reference of the end node.
- Time Complexity:
 - Insertion at the beginning of the linked list: $O(1)$
 - Insertion at the end of the linked list: $O(n)$ (If having a tail-referencing variable then $O(1)$)
 - Deletion at the beginning of the linked list: $O(1)$
 - Deletion at the end of the linked list: $O(n)$
 - Deletion (given a value): $O(n)$
- **No random access:** Linked lists do not support direct (random) access to elements based on their index. To access an element, traversal must start from the head node and follow the next pointers until the desired node is reached.
- Time Complexity: Accessing an element takes $O(n)$ time in the worst case.

Let's summarize what we just covered:

Function	Array	Python List	Linked List
Size	Fixed	Dynamic	Dynamic
Memory Allocation	Contiguous	Contiguous	Non-contiguous
Access Time	$O(1)$	$O(1)$	$O(n)$ (requires traversal)
Random Access (using indices)	Yes	Yes	No
Insertion/Deletion	$O(n)$, need to shift elements forward, need to copy all the elements and store in a new array since fixed size.	$O(1)$ for append and pop at the end, $O(n)$ for others	$O(1)$ at known positions (e.g., head or tail); $O(n)$ for traversal when position is unknown.

The basic data structures are usually the most simple to implement and use when creating basic programs.

However, there are several key scenarios when a linked list will be of greater utility to the programmer.

For example, imagine you have a leaderboard that shows the top players' scores. When a new score is added, you need to find the right place for it so that the leaderboard remains in order.

If you use Python lists to implement this:

- Storage: In Python lists, all the scores are stored in a line, one after another.
- Insertion: If a new player gets a score that fits in the middle of the leaderboard, you need to slide all the lower scores down to make room for the new one. This is like moving every book on a shelf one spot to the right to make space for a new book.
- Deletion: If a player wants to delete their score, you remove it and then move all the lower scores up one spot to close the gap.

If you use Linked Lists to implement this:

- Storage: Each score is like a box with a note inside it. The note tells you where the next score's box is, but the boxes aren't necessarily in a line.
- Insertion: When a new score comes in, you just find the right spot, place the new box there, and adjust the nodes in the neighboring boxes to point to this new box. There's no need to move all the other scores; you just change the **links between nodes**.
- Deletion: If a score needs to be deleted, you don't shift all the other scores. You just change the node in the previous box to **point to the one after the deleted score**.

In conclusion, if you're **frequently inserting or deleting** scores, a linked list makes these tasks easier and more efficient, as you don't need to move everything around each time.

However, if your leaderboard **doesn't change frequently** and you often need to **read** scores in order, a Python list might be a simpler and quicker choice for that purpose.

Insertion and deletion of a node in a linked list

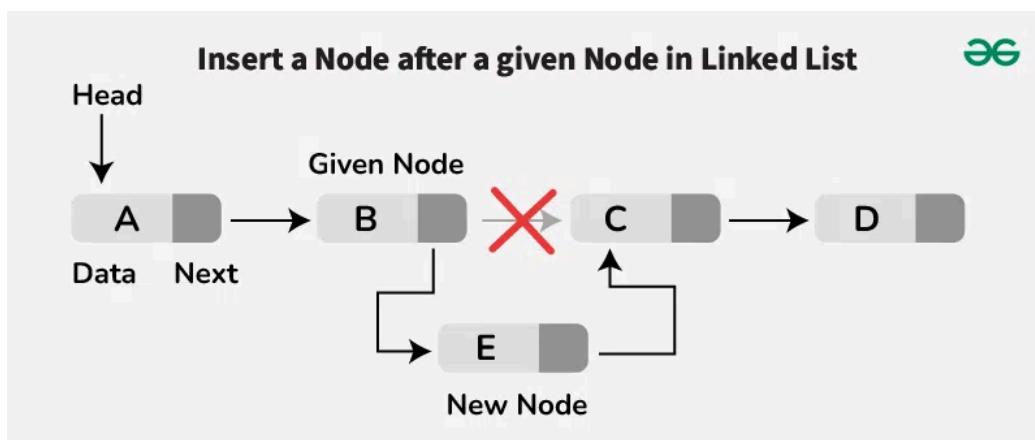
The insertion and deletion of a node within a linked list involves a very simple process, in which a node has its next link modified to reflect the intended changes.

For insertion

1. Find the position in the list where you intend to insert the element. This is achieved by passing through the head node of the list, and recursively repeating this process down the linked list.
2. Copy the link of the current node's next to the new node's next.
3. Update the previous node's next with the current node's memory address.

This effectively updates the pointers of the nodes in the list, maintaining the linked lists structure, but including the new node.

A visual depiction of this process may be seen below.

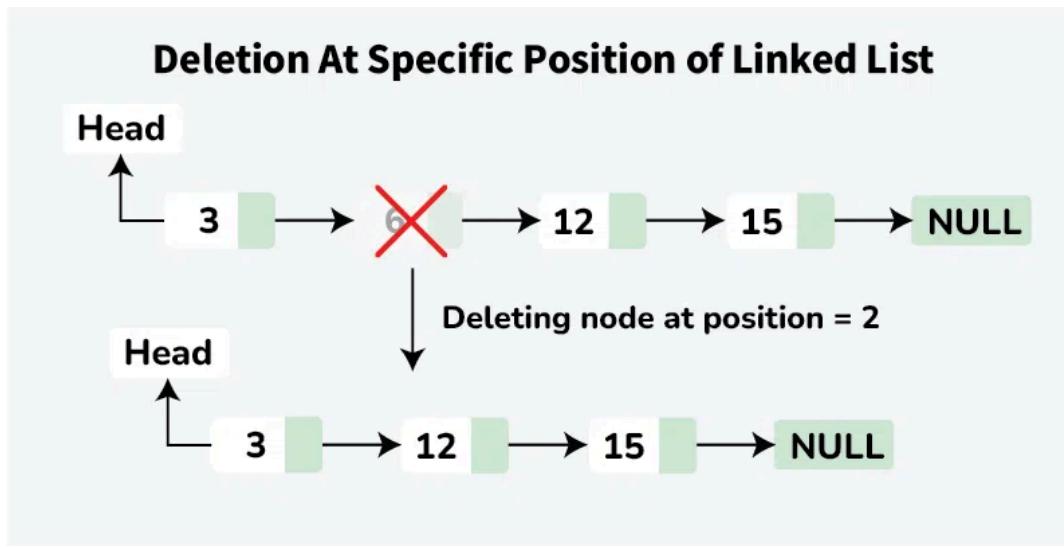


For deletion

1. Find the position in the list where you intend to delete the element. This is achieved through the same process as insertion.
2. Copy the link of the current node's next and save this memory address.
3. Update the previous node's next with the saved memory address.

This effectively removes the node from the list, as the memory address storing the deleted node is lost.

A visual depiction of this process may be seen below.



Exercises 1-10 (T/F)

Exercise 2

Linked lists offer more efficiency than basic data structures like arrays or Python lists, particularly when considering operations like insertion and deletion.

- True
- False

Explanation: Linked lists are faster for insertions and deletions ($O(1)$) at the head or insertions at the end (if having a tail-referencing variable) because they only require updating pointers, while arrays need to shift elements, which takes more time ($O(n)$).

Exercise 3

Linked lists can grow or shrink dynamically during runtime.

- True
- False

Explanation: Linked lists can grow or shrink as needed by adding or removing nodes, without the need for resizing like arrays.

Exercise 4

Insertion and deletion operations in linked lists can be performed by updating the references.

- True
- False

Explanation: In linked lists, nodes are inserted or deleted by updating the references (pointers) to connect the surrounding nodes.

Exercise 5

Insertion at the beginning of a linked list takes $O(n)$ time.

- True
- False

Explanation: It takes $O(1)$ time

Exercise 6

Linked lists support random access to elements based on their index.

- True
- False

Explanation: Linked lists do **not** support random access to elements based on their index. To access an element at a specific index in a linked list, you must traverse the list sequentially from the head node to the desired index.

Exercise 7

Accessing an element in a linked list takes $O(1)$ time in the worst case.

- True
- False

Explanation: It takes $O(n)$ time in the worst case because of the need of traversing every single node to access the element .

Exercise 8

In a leaderboard scenario, when a new score is added to a leaderboard implemented as a linked list, all lower scores need to be shifted down.

- True
- False

Explanation: If the leaderboard is implemented as a linked list, you can simply adjust the pointers.

Exercise 9

Linked lists are more efficient than Python lists when frequently inserting or deleting elements in the middle of the list, assuming the specific node or position for the operation is already known

- True
- False

Explanation:

In a linked list, insertion or deletion of an element in the middle can be done in $O(1)$ time if you already have a reference to the node where the operation needs to be performed. This is because you only need to adjust the next (and potentially prev in a doubly linked list) pointers.

In a Python list (Array), inserting or deleting an element in the middle requires shifting all subsequent elements to maintain order, which takes $O(n)$ time in the worst case.

Exercise 10

Insertion at the end of a linked list always takes O(1) time.

- True
- False

Explanation: It takes O(n) time in the worst case if there is no tail reference due to the need to traverse the list.

Exercises 11 - 23 (MCQ)

Exercise 11

What are the benefits of using linked lists? Select all correct answers.

- Efficiency of deletion and insertion
- Compression
- Prediction
- Adaptability in size

Explanation: Insertion and deletion in linked lists is O(1) if the node reference is known. They can grow and shrink dynamically without requiring resizing, unlike arrays.

Exercise 12

What actions are more efficient for a linked list than a list? Select all correct answers.

- Creation of the list
- Deletion of data
- Accessing of data
- Insertion of data

Explanation: Linked lists are more efficient for **deletion** and **insertion** when the position is known, requiring only pointer updates (**O(1)**). However, they are less efficient for **creation** and **accessing data**, as traversal takes **O(n)** compared to Python lists' direct access (**O(1)**).

Exercise 13

Standard Lists in Python become more inefficient when _____. Choose the correct option to complete the statement.

- More elements are added
- The elements in the list take up more memory
- Elements are removed from the list
- A list is used over time

Explanation: One might think that removing the element from the list is inefficient due to shifting of data needed in it. In fact, sometimes removing the data may improve the efficiency

by freeing up memory and reducing the overall size. However, as elements are added to a list, performing any operation on the list can become **less efficient in terms of time and space complexity**.

Exercise 14

As more elements are added, what happens to a linked list?

- It becomes more efficient
- ~~It maintains its efficiency for deletion and insertion~~
- It becomes less efficient
- It stops working

Explanation: As more elements are added, a linked list maintains its efficiency for **insertion and deletion at the head or tail (assuming tail reference available)** ($O(1)$). For **middle** insertions or deletions, if the node is **known**, the operation is still $O(1)$. If the node isn't known, finding it requires $O(n)$ time. Unlike arrays or Python lists, linked lists **don't require resizing or shifting**, so their efficiency **remains stable as the list grows**.

Exercise 15

Linked lists can be used as ____.

- The basis for basic data structures
- The basis of a method
- The building blocks of classes
- ~~The building blocks of advanced data structures~~

Explanation: Linked lists are fundamental for constructing **more complex** data structures like **stacks, queues, and graphs**. They provide a **flexible** structure where elements can be easily inserted or removed **without the need for contiguous** memory allocation.

Exercise 16

Unlike built in data structures, linked lists can be ____.

- Used to store data
- Managed using methods
- ~~Customized for special purposes~~
- Accessed using methods

Explanation: Linked lists allow for customization, such as creating **doubly linked lists** or **circular linked lists**, depending on the requirements. They are more flexible than built-in data structures like Python lists, which have predefined behaviors and features.

Exercise 17

What is the drawback to the linked lists compared to other data structures like arrays?

- ~~It uses more space in memory~~
- It takes longer to process data
- You can only use one per program
- It slows down the rest of the program

Explanation:

Linked list needs to store both the value and a reference to the next node, which requires more memory than a list, as a list only needs to store the value.

Exercise 18

How are the nodes of a linked list stored in memory compared to the elements of an array?

- Contiguously
- ~~Non-contiguously~~
- In a stack
- In a queue

Exercise 19

Imagine you are building a task scheduling system for a team. Tasks can be added to the list (inserting) and completed tasks need to be removed (deleting) frequently. Which data structure is more efficient in this task scheduling system?

- Array
- Python list
- ~~Linked list~~
- Stack

Explanation: In a task scheduling system where tasks are **frequently** added and removed, a linked list is more efficient than a Python list. In a Python list, insertion and deletion both require shifting subsequent tasks, taking $O(n)$ time. On the other hand, in a linked list, insertion and deletion are more efficient, adjusting pointers without shifting elements, taking $O(1)$ time if the position is known.

Exercise 20

Why do linked lists not support random access to elements?

- Because nodes are stored in contiguous memory.
- ~~Because each node points to the next node and traversal is required.~~
- Because the nodes are not linked together in any way.
- Because linked lists are stored in a stack.

Explanation: Unlike arrays, where elements are stored **contiguously** in memory and can be accessed directly using an index, linked lists store nodes in a **scattered manner**, requiring

traversal from the head to access specific nodes.

Exercise 21

Which of the following operations is generally more efficient in a linked list compared to an array?

- Sorting the elements.
- Accessing an element by index.
- Inserting an element at the beginning.
- Accessing the last element.

Explanation: Inserting at the beginning of a linked list is an O(1) operation since you only need to **adjust the head pointer**. In contrast, arrays require shifting all elements after the insertion point, making it slower (O(n)).

Exercise 22

Which of the following is a disadvantage of linked lists compared to arrays?

- More time needed for insertion and deletion of elements.
- Dynamic size.
- Higher memory usage due to storing references.
- Non-contiguous memory allocation.

Explanation: Linked lists require **extra memory** to store references (or pointers) to the next node in addition to the data, whereas arrays only store the data itself.

Exercise 23

Which of the following is the correct statement about linked lists?

- They have fixed size like arrays.
- Each element is directly accessed by its index.
- Each node contains data and a reference to the next node.
- They use contiguous memory allocation.

Explanation: Linked lists consist of nodes, each containing data and a reference (or pointer) to the next node, enabling traversal. This is in contrast to arrays, which use contiguous memory allocation and can access elements directly by index.

Recap (Lessons 1 and 2)

Data Structures

While this chapter is about linked list data structures, you have actually gotten a lot of experience with all sorts of data structures before this.

Built-in python data structures include:

- Lists
- Sets
- Tuples
- Dictionaries

So when we talk about data structures, just remember that we are specifically referring to containers of data, they are not some abstract thing you have never seen before.

Benefits of built-in data structures

- Efficiency has been optimized by the maintainers of the language.
- They are relatively general and easy to visualize.
- Easy to use.

Downsides of built-in data structures

The major downside of just relying on python's in-built data structures is that they are relatively general. However, this generality often comes at the cost of performance in specific scenarios.

Therefore we need to be able to create data structures from scratch that python hasn't implemented, or maybe even recreate the data structures python has already implemented to match our specific needs.

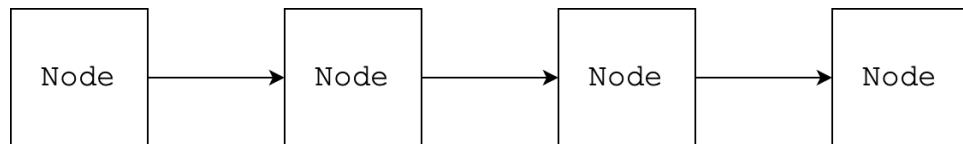
Types of data structures:

There are 2 primary types of data structures, these include:

- Linear - Arrays, Queues, Stacks, Linked Lists, etc.
- Non-Linear - Graphs, Trees

Linear Structures

Linear structures are often visualized as a line of nodes.



The linked list that we are going to discuss is a good example of this. Linked Lists showcase how there is generally a straightforward ordering toward linear structures.

Non-Linear structures

While these structures will typically use nodes like Linked lists do, they do not only operate with a single branch, in fact they may branch off in many different directions.

While they may not appear as straightforward as linear structures are, it is important to note that there is still a reason behind the structure of non-linear structures, otherwise it would be hard to insert and retrieve information.

Memory Addresses

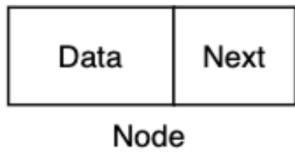
In memory, rather than storing the data address to a variable, we store the addresses of the data they point to in variables.

This means that data structures can access other data structures if they have knowledge about the address that points to the other data structure.

Nodes

Nodes are a popular data structure that contain data and addresses of other nodes that they are connected to.

As you can see, this node holds data, which can be anything, and a next pointer that could be None or another memory address which points toward another node.



This structure is the most common structure for nodes seen in a linked list.

Linked List

Linked lists are a collection of nodes that are connected by the fact that nodes can know the location of another node.

Typically, the first node in the list is called the head, while the last node in the list is called the tail. The tail will typically point to None.

This means that when you traverse the linked list, you know you have reached the end when the next value points to None.

To get to any node in the list, you cannot just index immediately into it like you can with a list. Instead you must traverse the linked list until you can reach the node you wish to work with.

Insertion and deletion can both be handled without having to shift every other item around, like it does in a list. However, if the insertion or deletion point is toward the end of the list, it will still result in an O(n) operation as it is required to traverse the entire linked list.

Advantages of Linked Lists

- **Dynamic Size:** Unlike arrays, linked lists can grow or shrink in size without requiring resizing or reallocation.
- **Efficient Insertions/Deletions:** Adding or removing nodes (when the position is known) is $O(1)$ in linked lists, as no shifting of elements is required.
- **Efficient Memory Utilization:** They utilize memory efficiently when the size of the data structure is unpredictable or changes frequently.
- **Implementation of Advanced Data Structures:** Linked lists are the foundation for more complex data structures like stacks, queues, graphs, and trees.

Disadvantages of Linked Lists

- **Sequential Access:** Linked lists do not support random access; traversing the list to find a specific item takes $O(n)$ time.
- **Higher Memory Usage:** Each node requires additional memory to store a pointer (or reference) to the next node, making them less memory-efficient than arrays.
- **Complexity in Implementation:** Managing pointers or references and ensuring proper insertion or deletion can be error-prone and more complex than using arrays or Python's built-in lists.

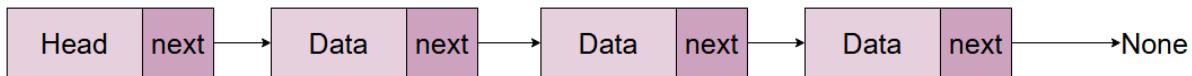
3. Singly Linked List

3.1 Introduction to Singly Linked List

In this lesson, we'll focus on understanding the singly linked list.

Feature	Array	Single Linked List
Memory Allocation	Contiguous	Non-contiguous
Access Time	$O(1)$ (Direct Indexing)	$O(n)$ (Traversal Required)
Insertion/Deletion	Expensive (Shifting Required)	Efficient ($O(1)$ at head) When not at head $O(n)$ (Traversal required)
Memory Usage	Fixed	Dynamic, with pointers

A Singly Linked List is a type of linked list where each node contains a data element and a reference (pointer) to the next node.



A singly linked list is characterized by unidirectional traversal. In a singly linked list, traversal can only be done in **one** direction, from the head node to the tail node, as each node only has a reference to the next node and **no reference to the previous node**.

Traversing the entire list takes $O(n)$ time, where n is the number of nodes in the list. For example, traversing a list with 10 nodes requires visiting each node once.

Using the following Node class we created in the last lesson, we will create the LinkedList class to implement the functionality of the list:

Example - Node Class

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

The time complexity for creating a node instance is $O(1)$.

Example - Linked List Class

```
class LinkedList:  
    def __init__(self):  
        self.head = None # The head node is None initially  
        self.size = 0 # The default linked list is empty
```

The time complexity for initializing a LinkedList is O(1)

The LinkedList class contains:

- a 'head' variable storing a reference to the node at the head of the list.
- a 'size' variable used to store an integer representing the number of nodes in the list.

Now we can write some code to implement the basic functionality of a linked list such as:

1. Creating a list object (by the Constructor of LinkedList above)

2. Appending an object

- a. Instantiating a node object
- b. Adding this node to the list
- c. Updating the size of the list

3. Accessing the data within the list

The following code sets up the general functionality of the linked list in its most basic implementation:

```
# 1.Create a new instance of the LinkedList class called "list1"  
list1 = LinkedList()  
  
# 2.aCreates a new instance of the Node class called "item1" with  
# the data value "soup"  
item1 = Node("soup")  
  
# 2.b.Sets the head of "list1" to point to "item1".  
list1.head = item1  
  
# 2.c.Increments the size attribute of "list1" by 1. The size  
# becomes 1 because we added one node to the list.  
list1.size += 1  
  
# 3.Prints the data value of the head node of "list1". It will  
# print "soup" since that's the data value of the only node in the  
# list.  
print(list1.head.data)
```

Output:

soup

After creating the list object and adding a node to the list, further nodes can be added and

accessed as below:

```
# Creates a new instance of the Node class called "item2" with the
# data value "carrots".
item2 = Node("carrots")

# Set the next attribute of the current head node "list1" to point
# to "item2". Now, the linked list has two nodes, "item1" and "item2"
list1.head.next = item2

# Increment the size attribute of "list1" by 1. The size becomes 2
# since we added another node to the list.
list1.size += 1

# Print the data value of the second node in "list1". It will print
# "carrots" since that's the data value of the second node in the
# list.
print(list1.head.next.data)
```

Output:

soup
carrots

Key Characteristics of a Singly Linked List:

1. **Unidirectional traversal:** In a singly linked list, traversal can only be done in one direction, from the head node to the tail node. Each node only has a reference to the next node, and there is no reference to the previous node. Once you move past a node, unless you save its reference elsewhere, the node cannot be accessed again.
 - Time Complexity: Traversing the entire list takes $O(n)$ time, where n is the number of nodes in the list.
2. **Dynamic size:** Singly linked lists can grow or shrink dynamically during runtime. Nodes can be easily added or removed from the list without the need for resizing or shifting elements, as opposed to arrays with fixed sizes.
 - Time Complexity: Insertion at the beginning of the list takes $O(1)$ time, while insertion at the end or in the middle of the list takes $O(n)$ time in the worst case due to the need to traverse the list.
3. **Non-contiguous memory allocation:** Unlike arrays, which store elements in contiguous memory locations, nodes in a singly linked list can be scattered throughout the memory. Each node is independently allocated and linked to the next node through its next pointer.
 - Time Complexity: Memory allocation time is dependent on the underlying memory manager but generally considered $O(1)$ for each allocation.

Use Singly-Linked Lists when:

- Frequent insertions and deletions are needed at the beginning or middle of the list.

- Memory availability is fragmented, and contiguous allocation (as in arrays) is not possible.
- Implementing stacks, queues, or adjacency lists in graphs.

We have just covered what is called a “Singly-Linked List”, but linked lists can come in **several varieties**, each with distinct features that make them favorable over others for specific cases. We will introduce them in detail in the subsequent chapters. Here are some other commonly used types of linked lists:

- **Doubly Linked List:** Unlike a singly linked list, which only has a **next** field, a doubly linked list allows each node to store both **next** and **previous** fields. The **next** field points to the next node (as in a singly linked list), and the **previous** field points to the preceding node..
- **Circular Linked List:** Rather than the tail node’s next field pointing to none, it points back to the head of the list creating a circular structure.

Exercises 1 - 6 (MCQ)

Exercise 1

What is a characteristic of a singly-linked list?

- Bidirectional traversal
- Fixed size
- Dynamic size
- Contiguous memory allocation

Explanation:

A singly linked list can grow or shrink in size as needed, allowing for dynamic memory allocation. This makes it flexible compared to arrays that have a fixed size.

Exercise 2

A singly-linked list node is linked to?

- The linked list
- The next node in the list or None
- The next or previous node in the list or None
- The head node or None

Exercise 3

What is the main advantage of using a singly-linked list over an array?

- Faster random access to elements
- Efficient insertion and deletion of elements at any position

- Efficient insertion and deletion of elements at the beginning or end
- Ability to store elements in contiguous memory locations

Explanation:

Option 1: This is false. Singly linked lists do not support random access, so accessing an element by index requires $O(n)$ time due to traversal. Arrays provide $O(1)$ random access.

Option 2: This is true, particularly for insertions and deletions at the beginning or when you have a reference to the node at a specific position. Unlike arrays, where inserting or deleting an element requires shifting other elements (which takes $O(n)$ time), linked lists only need pointer updates, which can be done in $O(1)$ time if the node reference is known.

Option 3: This is partially true. Singly linked lists can efficiently handle insertions and deletions at the beginning $O(1)$, but for insertions at the end, traversal is needed unless a tail pointer is maintained.

Option 4: This is false. Singly linked lists do not store elements in contiguous memory; they use nodes that are scattered in memory and connected by pointers.

Exercise 4

What is next initialized as?

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = _____
```

- None
- Null
- 0
- False

Exercise 5

A linked list's size variable is used to _____. Select the correct option to complete the sentence.

- Track the size of the data in the list
- Track the number of nodes in the list
- Track the amount of modifications made to the list
- Track the memory used by the linked list

Exercise 6

In a linked list, the head of the singly linked list is the object before the first node and acts as a

starting point.

- True
- False

Explanation: The head is the first node of the linked list.

Exercises 6 - 15 (Fill in the Blank)

Exercise 6

Complete the code to create the Node class.

Given code

```
class Node:  
    def __init__(self, [Fill in answer here]):  
        self.[Fill in answer here] = [Fill in answer here]  
        self.next = None
```

Expected input

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

Exercise 7

Complete the code to create the Linked List class.

Given code

```
class LinkedList:  
    def __init__(self):  
        self.[Fill in answer here] = None  
        self.size = [Fill in answer here]
```

Expected input

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0
```

Exercise 8

Complete the code below to create a linked list.

Given code

```
class LinkedList:
```

```
def __init__(self):
    self.head = None
    self.size = 0

food_list = [Fill in answer here]
```

Expected input

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
```

Exercise 9

Using the Node class, create a new instance of the Node class with the data "grapes" and assign it to the variable item1.

Given code

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

item1 = [Fill in answer here]
```

Expected input

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

item1 = Node("grapes")
```

Exercise 10

Complete the code below to assign the created node to the head of the linked list.

Given code

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
```

```
food_list = LinkedList()
item1 = Node("grapes")
[Fill in answer here]
```

Expected answer

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
```

Exercise 11

Complete the code below to print the data in the head node.

Given code

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
food_list = LinkedList()  
item1 = Node("grapes")  
food_list.head = item1  
print([Fill in answer here])
```

Expected answer

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
food_list = LinkedList()  
item1 = Node("grapes")  
food_list.head = item1  
print(food_list.head.data)
```

Expected Output:

grapes

Exercise 12

Complete the code below to add a second node to the list storing "milk".

Given code

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None
```

```

        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
item2 = [Fill in answer here]
food_list[Fill in answer here] = item2

```

Expected answer

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
item2 = Node("milk")
food_list.head.next = item2

```

Exercise 13

Complete the code below to increment the size of the list once the new node is added.

Given code

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.[Fill in answer here]

```

Expected answer

```

class Node:

```

```

def __init__(self, data):
    self.data = data
    self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.size += 1

```

Exercise 14

Complete the code below to assign 1 more node storing "onions" to the linked list incrementing the size as added .

Given code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.size += 1
item3 = [Fill in answer here]
[Fill in answer here] = item3
[Fill in answer here] += 1

```

Expected answer

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

```

```

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.size += 1
item3 = Node("onions")
food_list.head.next.next = item3
food_list.size += 1

```

Exercise 15

Complete the code below to print out the three nodes saved in food_list.

Given code

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.size += 1
item3 = Node("onions")
food_list.head.next.next = item3
food_list.size += 1
print([Fill in answer here])
print([Fill in answer here])
print([Fill in answer here])

```

Expected answer

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

```

```

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

food_list = LinkedList()
item1 = Node("grapes")
food_list.head = item1
food_list.size += 1
item2 = Node("milk")
food_list.head.next = item2
food_list.size += 1
item3 = Node("onions")
food_list.head.next.next = item3
food_list.size += 1
print(food_list.head.data)
print(food_list.head.next.data)
print(food_list.head.next.next.data)

```

Expected output

grapes
milk
onions

Note: This is a very inefficient way of accessing data from a Linked List and a much more efficient way will be discussed later.

Exercises 16 - 20 (Coding)

Story:

Cooper is a highschool student currently studying statistics in his mathematics class. He has chosen to do his assignment on his friend's favourite numbers and believes it will be easier for him to perform mathematics on data in a code format.

Cooper wants to store his data set in a linked list.

It is your job to assist him in making this solution!

Exercise 16

Story:

First we must start with creating the classes for Node and LinkedList. Cooper knows how to write a class definition but is not sure how to appropriately write the constructor. Can you help him?

Write Node and LinkedList classes. Requirements:

- Node class must have 2 attributes: data (to store the data) and next (to store the next node).
- The LinkedList class must keep track of the head (first node) and size (number of nodes) of the linked list.

You will be using these classes for the upcoming exercises in this section.

Given code:

```
class Node:  
    # Your code here  
  
class LinkedList:  
    # Your code here
```

Expected input

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__():  
        self.head = None  
        self.size = 0
```

Exercise 17

Story:

Now that we have set up the data structure, we can now start declaring it in our code and fill it with data. Cooper interviews his friend Laura and she says that her favourite number is 25.

Initialize a LinkedList called best_numbers and add a head node containing the number 25. Print the data of that node from the list after storing. Remember to increment list size.

Expected input

```
best_numbers = LinkedList()  
node1 = Node(25)  
best_numbers.head = node1  
best_numbers.size += 1  
print(best_numbers.head.data)
```

Expected output

25

Explanation:

A Node containing the number 25 is created and assigned as the head of the LinkedList.

Exercise 18

Story:

Cooper interviews his two other friends Amanda and Liana and they tell him that their

favourite numbers are 522 and 2555 respectively. Let's help Cooper add these numbers to the linked list we have built together.

Add nodes containing the numbers 522 and 2555 to the best_numbers linked list and print its size.

Expected input

```
node2 = Node(522)
best_numbers.head.next = node2
best_numbers.size += 1
node3 = Node(2555)
best_numbers.head.next.next = node3
best_numbers.size += 1
print(best_numbers.size)
```

Expected output

3

Exercise 19

Story:

Cooper wants to see what the average is for the numbers gathered from his surveying. This will help him understand if people like smaller or larger numbers better.

Print the sum of the data values in the three nodes stored in the best_numbers linked list.

Expected input

```
print((best_numbers.head.data + best_numbers.head.next.data +
best_numbers.head.next.next.data) / best_numbers.size)
```

Expected output

3102 # $25+522+2555=3102$

Exercise 20

Story:

After we have done all this work, Cooper's friends decide that their favourite numbers have changed to 15, 511 and 1555 respectively. Let's help Cooper change the data in his list.

Update the data in the best_numbers list with the following values for each node: set the first node's data to 15, the second node's data to 511, and the third node's data to 1555.

Expected input

```
best_numbers.head.data = 15
best_numbers.head.next.data = 511
best_numbers.head.next.next.data = 1555
```

3.2 Inserting Data in Singly Linked List

In the previous section, we showed the operation of the most basic form of a linked list.

Manually managing links in a linked list can become error-prone and inconvenient as the list grows.

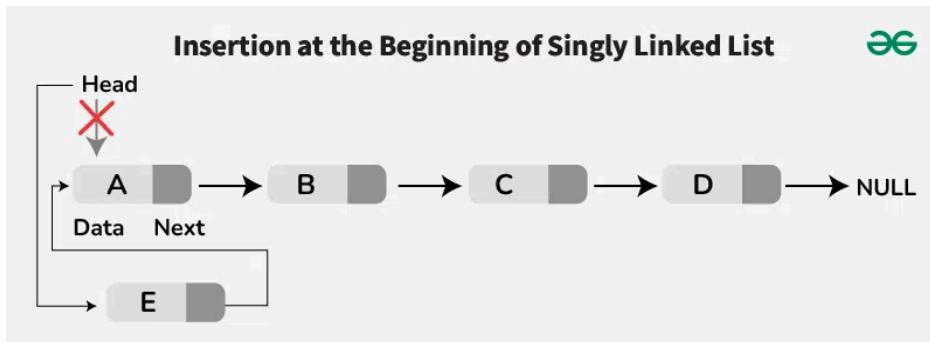
For instance, inserting a node in the middle of the list requires careful updates to the links to avoid breaking the existing structure.

As such, we need to develop specialized methods for the linked list to improve efficiency and usability when storing and accessing data.

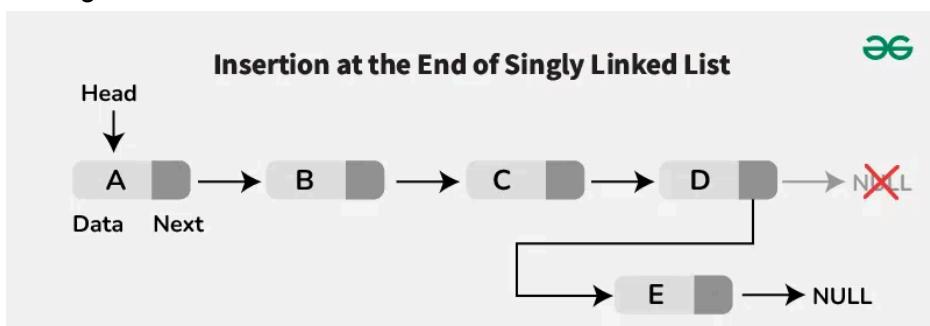
In a singly linked list, **inserting data** involves **creating a new node** and **connecting it to the existing nodes in the list**. Inserting data is a fundamental operation in linked lists and is one of the key advantages of using linked lists over arrays.

Now, let's consider the different scenarios for inserting data in a singly linked list:

- **Inserting at the Beginning of the List - `prepend()`**: Add a new node at the start of the list, making it the new head.

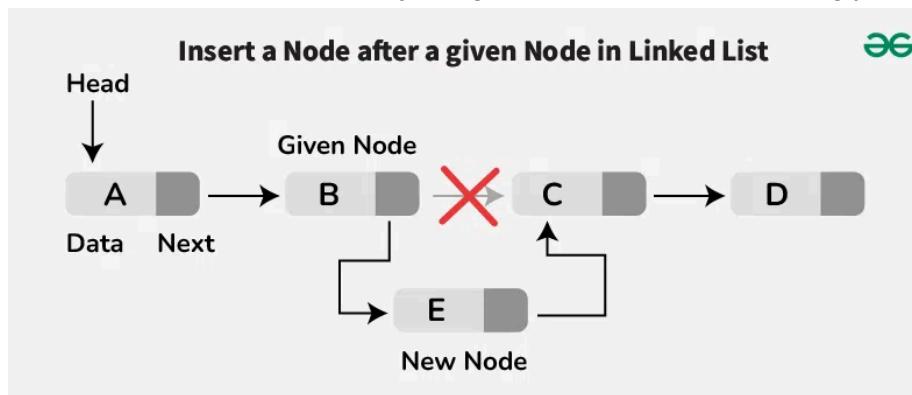


- **Inserting at the End of the List - `append()`**: Add a new node at the end of the list, making it the new tail.



- **Inserting at a Specific Position in the List - `insert_at()`**: Add a new node at a

specified position in the list, adjusting the next pointers accordingly.



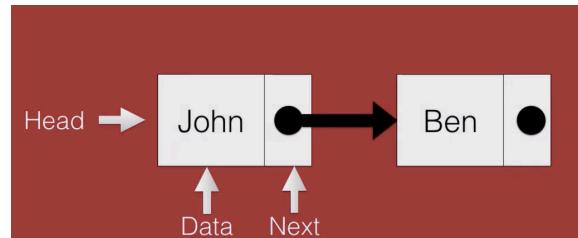
In this lesson, we will look into these scenarios.

3.2.1 `prepend()` Method

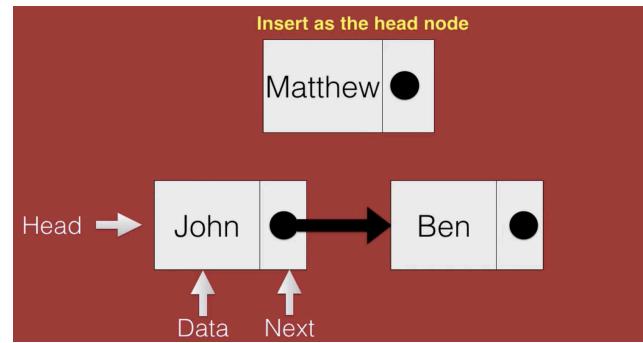
Inserting at the **beginning** of the list is a common scenario when working with singly linked lists. It involves **creating a new node** and **making it the new head** of the list.

The process can be broken down into a few simple steps, let's go through with an example:

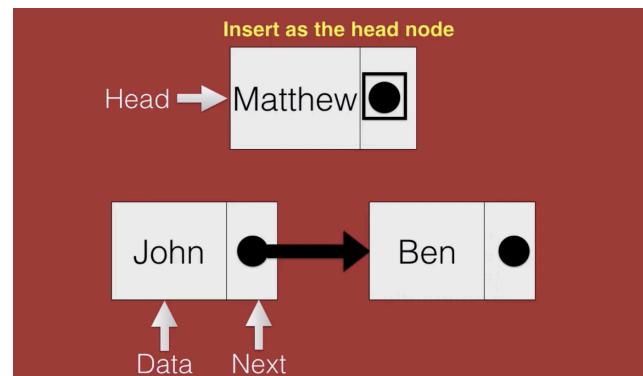
Here is a singly linked list with two nodes as shown below. The first node has the data "John" and the second node has the data "Ben".



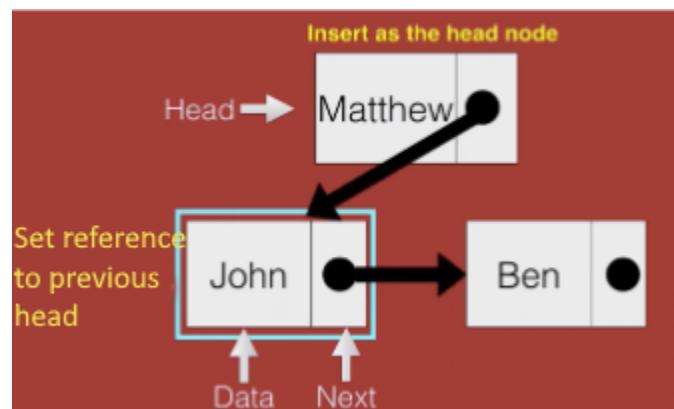
Now, let's say we want to insert a new node into this list with the data "Matthew", and the requirement is for "Matthew" to be inserted as the head node. One possible solution might be to remove "John" as the head node and make "Matthew" the new head node. But what problem could this cause?



By removing the existing head node, we will lose access to the rest of the list. If the new node "Matthew" does not have a reference to the next node, we won't know how to point back to the original head node "John". This can break the connection between the nodes, which we want to avoid.



So now, all that the next pointer of "Matthew" needs to do is point to the previous head node and that completes the linked list. Now, the *next* of "Matthew" points to "John" and the *next* of "John" still points to "Ben".



Steps to Insert “Matthew” as the New Head Node

1. **Create the New Node:** Create a new node with the data “Matthew”.
2. **Update the New Node’s Pointer (Next Reference):** Set the new node’s (Matthew) next reference to point to the reference stored in the head (John).
3. **Update the Head Pointer:** Finally, update the head of the list to point to the new node (Matthew), making Matthew the new head instead of John.
4. **Increment size:** Finally, increase the size of the list by 1 to reflect the addition of the new node.

By following these steps, the next reference of “Matthew” will point to “John”, and the next reference of “John” will continue to point to “Ben”, preserving the linked list structure.

Now, let's implement this in Python code.

The new method “**prepend**” in the *LinkedList* class adds a new node to the head of the list relinking the original head node and setting the new node to the head node:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def prepend(self, data):  
        # Preserve the new head node in a temporary node  
        new_node = Node(data)          # Point the new node to the  
current head  
        new_node.next = self.head      # Update the head to the  
new node  
        self.head = new_node  
        self.size += 1  
  
    def print_list(self):  
        # Traverse and print the linked list  
        current = self.head  
        while current is not None:  
            print(current.data, end=" ")  
            current = current.next  
        print()  
  
# Example usage  
linked_list = LinkedList()
```

```

linked_list.prepend(1)
linked_list.prepend(2)
linked_list.prepend(3)

print("Linked List:")
linked_list.print_list()
print("Size:", linked_list.size)

```

Output:

Linked List:

3 2 1

Size: 3

Now, do you remember the time complexity? It's good for us to think about time complexity when we learn the data structure. So, can you consider the time complexity for this prepend method?

The prepend operation has a time complexity of **O(1)**, which means it takes constant time regardless of the size of the linked list.

Let's break down the steps involved:

1. Creating a new node: **O(1)**
 - a. Allocating memory for a new node and assigning the data to it takes constant time.
2. Updating the next pointer of the new node: **O(1)**
 - a. Setting the next pointer of the new node to the temporary node (previous head) is a simple pointer assignment and takes constant time.
3. Updating the head pointer: **O(1)**
 - a. Updating the head pointer to point to the new node is a simple pointer assignment and takes constant time.

Therefore, the total time complexity of the prepend operation is $O(1) + O(1) + O(1) = O(1)$ (constant growth rate).

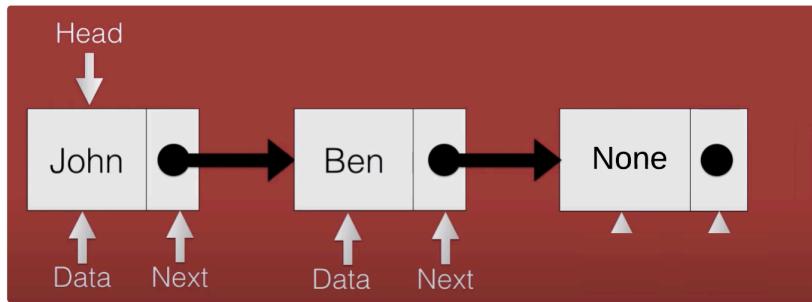
The space complexity of the prepend operation is also **O(1)** because it only requires a constant amount of extra space to create a new node. Unlike arrays, no additional memory is needed for resizing or shifting elements.

3.2.2 append() Method

Let's move on to the next insertion scenario!

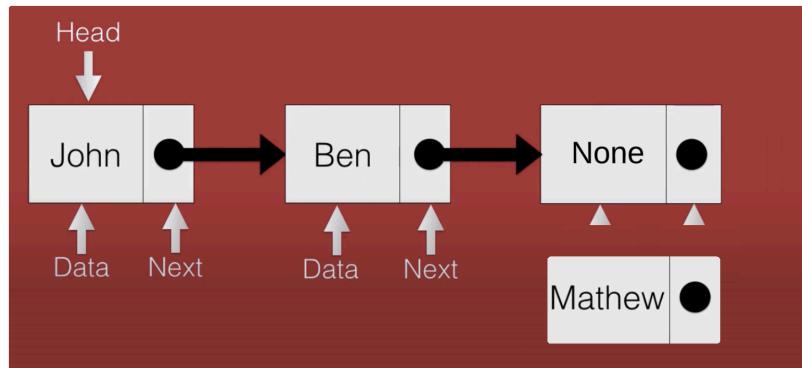
Appending a new node to the end of the list involves traversing the list to find the last node and updating its next pointer to point to the new node. Here's how the append operation works:

Consider a singly linked list with two nodes as shown below. The first node has the data "John", the second node has the data "Ben".



Now, we want to append a new node with the data "Mathew" at the end of this list. Compared with the previous "prepend" operation, we don't need to store any nodes in a temporary variable. This is because we are not changing the head of the list; instead, we are simply **updating the next pointer of the last node to point to the new node**.

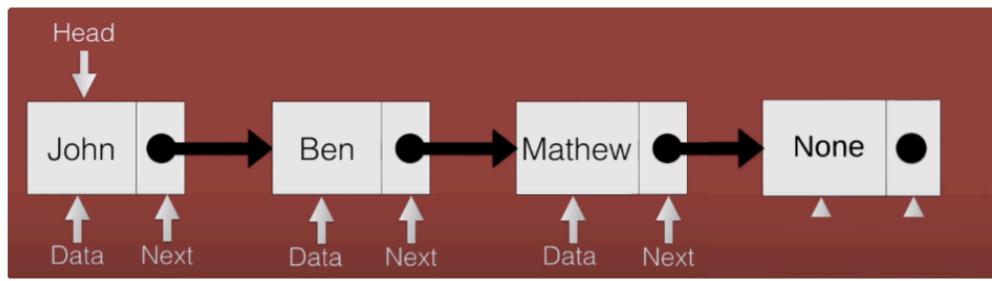
Firstly, we need to create a new node and assign the data to the new node.



Secondly, we need to check if the list is empty. If the list is **empty** (i.e., the head is None), **make the new node the head of the list**.

If it is **not empty**, we need to traverse the list to find the last node. Start from the head of the list and traverse the list by following the next pointers, continue traversing until we **reach the last node** (i.e., the node whose next pointer is None).

Then update the next pointer of the last node: **Update the next pointer of the last node to point to the new node**, effectively appending the new node to the end of the list, we do not need to set the `next` of the new tail to `None` because the constructor function has already done that.



Finally, increase the size of the list by 1 to reflect the addition of the new node.

Before implementing this operation in python code, let's write down the program logic first:

- Create a new node, assign the data to the new node and set the next pointer of the new node to None.
- Check if the list is empty:
 - If the head of the list is None, it means the list is empty.
 - In this case, make the new node the head of the list.
 - Update the size of the list to 1.
- If the list is not empty:
 - Initialize a variable (e.g. current) to store the current node during traversal.
 - Set current to the head of the list.
- Traverse the list to find the last node:
 - Traverse the list using a loop until current.next is None, indicating the last node.
 - This condition ensures that current will stop at the last node.
- Update the next pointer of the last node:
 - Update the next pointer of the last node (current.next) to the new node.
 - This effectively appends the new node to the end of the list.
- Increment the size of the list:
 - Increase the size of the list by 1 to reflect the addition of the new node.

The new method “**append**” in the `LinkedList` class adds a new node to the end of the list, updating the next pointer of the current last node to the new node.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

```

```

def __init__(self):
    self.head = None
    self.size = 0

def append(self, data):
    # Create a new node
    new_node = Node(data)

    # Check if the list is empty
    if self.head is None:
        self.head = new_node
    else:
        # Initialize the pointer "current" to the head of the
list
        current = self.head

        # Traverse the list to find the last node
        while current.next is not None:
            current = current.next

        # Update the next pointer of the last node
        current.next = new_node

    # Increment the size of the list
    self.size += 1

def print_list(self):
    current = self.head
    while current is not None:
        print(current.data, end=" ")
        current = current.next
    print()

# Example usage
linked_list = LinkedList()

linked_list.append(1)
linked_list.append(2)
linked_list.append(3)
linked_list.append(4)

print("Linked List:")
linked_list.print_list()
print("Size:", linked_list.size)

```

Output:

Linked List:

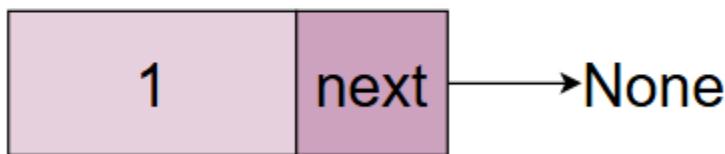
1 2 3 4

Size: 4

Here is the visualization of the linked list appending process of the example above:

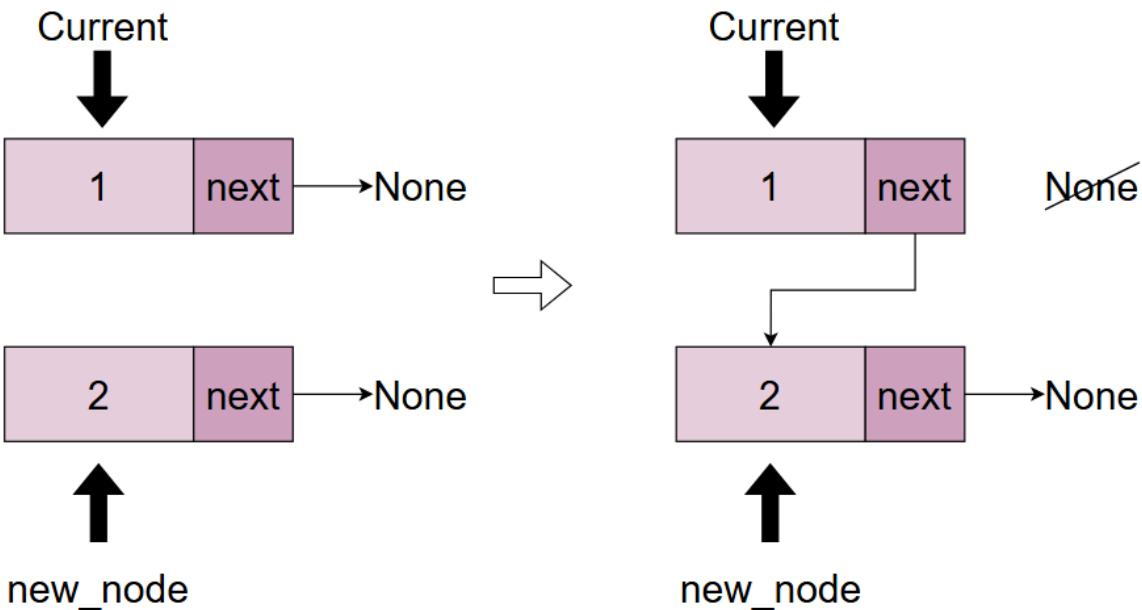
`linked_list.append(1):`

linked_list.append(1)

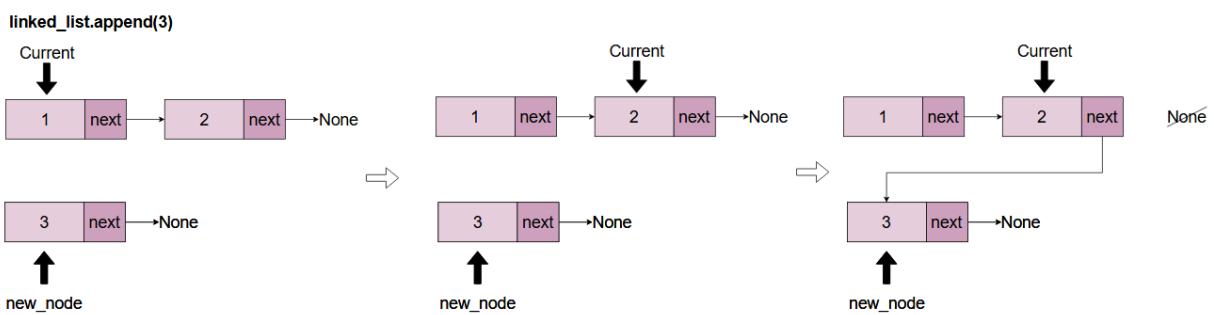


`linked_list.append(2):`

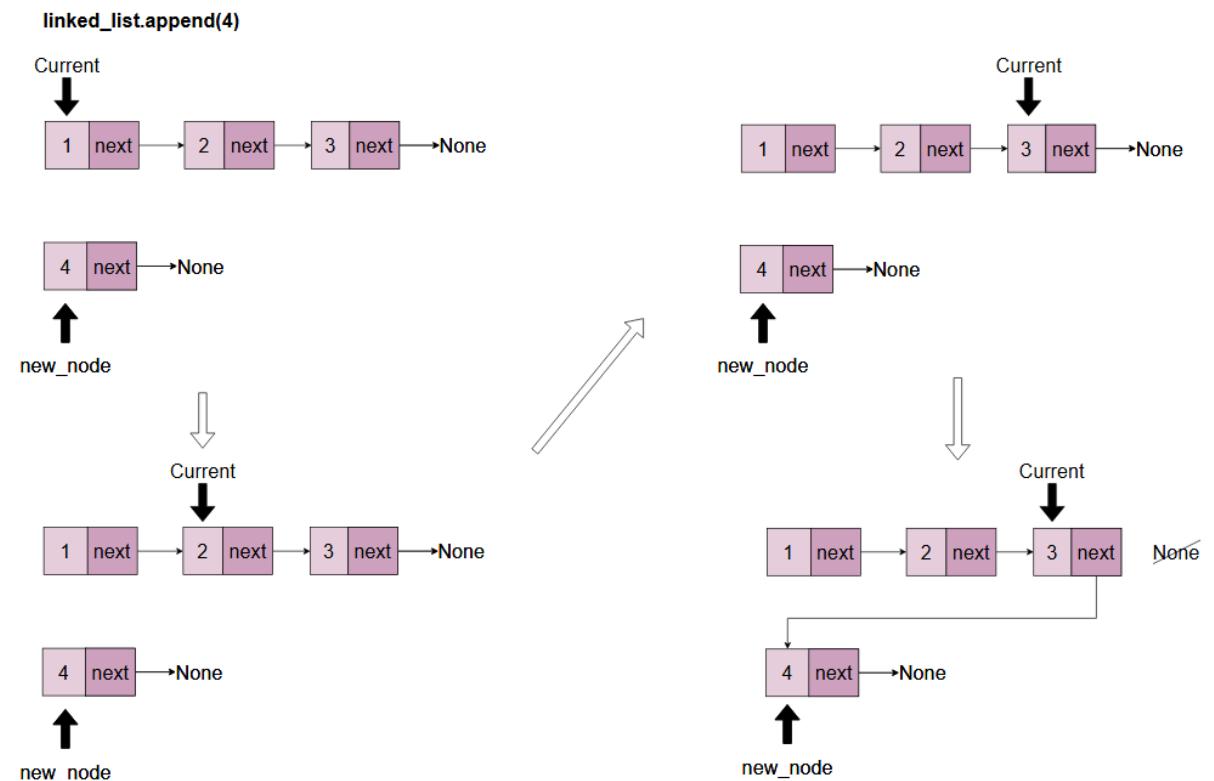
linked_list.append(2)



`linked_list.append(3):`



`linked_list.append(4):`



Essentially, in all cases you must traverse the list until the `next` field returns a value of `None`. Once you have arrived at the node that meets this condition, you need to set the `next` field to the `new_node` you'd like to append.

Now let's consider the time & space complexity for this method.

The time complexity of the append operation **depends on the size of the linked list**.

Let's analyze the steps involved:

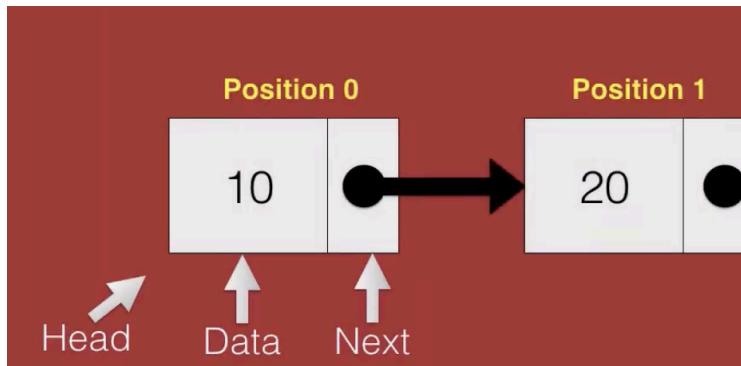
1. Creating a new node: **O(1)**
 - a. Allocating memory for a new node and assigning the data to it takes constant time.

<p>2. Checking if the list is empty: O(1)</p> <ul style="list-style-type: none"> a. Checking if the head is None is a simple comparison operation and takes constant time.
<p>3. Traversing the list to find the last node: O(n)</p> <ul style="list-style-type: none"> a. In the worst case, where the new node is being appended to the end of the list, we need to traverse the entire list to reach the last node. b. The traversal requires visiting each node in the list once, resulting in a linear time complexity proportional to the size of the list.
<p>4. Updating the next pointer of the last node: O(1)</p> <ul style="list-style-type: none"> a. Setting the next pointer of the last node to the new node is a simple pointer assignment and takes constant time.
<p>5. Incrementing the size of the list: O(1)</p> <ul style="list-style-type: none"> a. Incrementing the size variable takes constant time.
<p>Therefore, the overall time complexity of the append operation is O(n), where n is the length of the linked list and means the time complexity of this operation is linearly proportional to the length of the linked list.</p>
<p>The space complexity of the append operation is O(1) because it only requires a constant amount of extra space to create a new node, disproportionate to the size of the linked list.</p>
<p>It's important to note that while the append operation has a linear time complexity, it is still efficient for adding elements to the end of the list. The time complexity is directly proportional to the size of the list, as we need to traverse the entire list to reach the last node.</p>

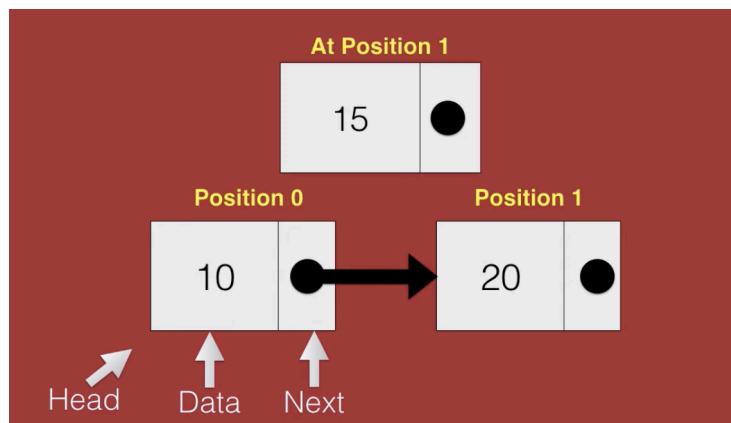
3.2.3 `insert_at()` Method

Inserting a node at a specific position in a singly linked list **involves adding a new node at a given index or position**. The process requires **traversing** the list to **reach the desired position** and updating the next pointers accordingly. Let's go through the steps involved in the "insert_at" operation.

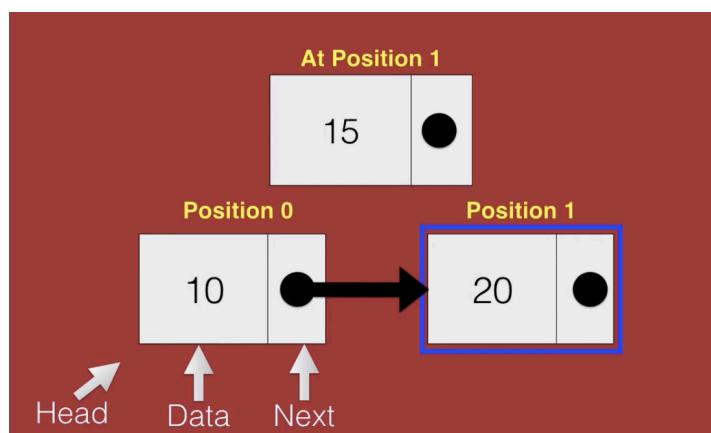
Consider we have a list with two nodes - the head node with the data 10 and the second node with the data 20 as shown below. We have the head node at position 0 and second node at position 1.



What we're trying to do here is to insert a new node with the data 15 at position 1 in between 10 and 20. So the next of 10 needs to point to 15 and the next of 15 needs to point to 20.



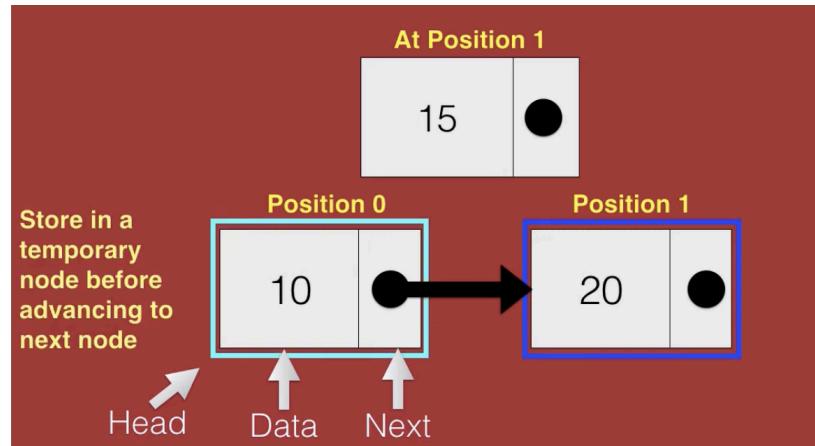
We first need to traverse the list until **one node before** the location where we want to insert our node. In our case, we want to insert at position 1, so we need to travel to position 0.



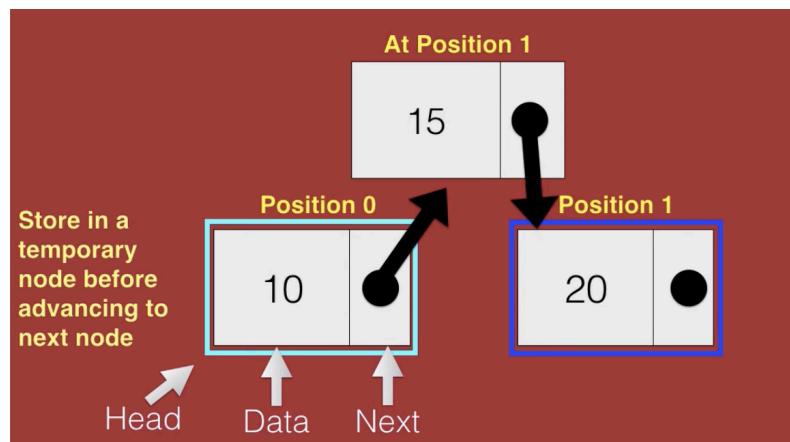
However, in this example, we assumed the given position is valid. In truth, we always need to **check if the given position is valid**. If the position is **less than 0** or **greater than the size of the list**, **return an error or handle it appropriately**. If the position is 0, then it comes to

insert at the beginning.

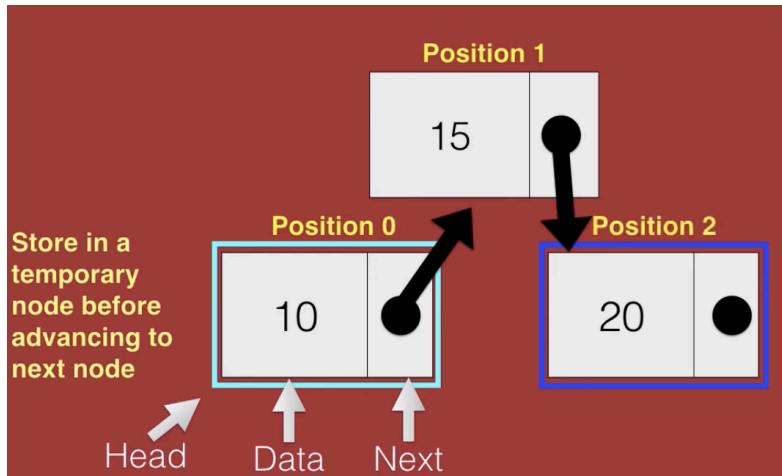
Once we reach the node at position 1, we cannot go back to the node position 0, so before we advance to the node at position 1, we need to **store the details of the node at position 0 in a temporary node**.



Once we do that, we can **remove the connection from 10 to 20**, and then establish this connection from **10 to 15**, and the **next of 15 now points to the node at position 1** and that completes the connection.



Finally, update the next pointers - set the next pointer of the new node 15 to the node 20 at position 1, and then the next pointer of node 10 at position 0 to the new node 15, and increment the size of the list.



Before implementing this operation in python code, let's write down the program logic first:

- **Create a new node** -> Assign the data to the new node.
- **Check if the given position is valid:**
 - If the position is less than 0 or greater than the size of the list, return an error or handle it appropriately.
- **If the position is 0** (inserting at the beginning):
 - Set the next pointer of the new node to the current head of the list.
 - Update the head to point to the new node.
 - Increment the size of the list.
- **If the position is greater than 0:**
 - Initialize a **temporary** variable (e.g., current) to store the current node during traversal.
 - Set current to the head of the list.
 - Move through the list to find the node immediately before the insertion point. Use a loop to move current to the next node **until the position is reached or the end of the list is encountered.**
 - **Update the next pointers:**
 - Set the next pointer of the new node to the node after current.
 - Set the next pointer of current to the new node.
 - Increment the size of the list.

The new method “`insert_at`” in the `LinkedList` class adds a new node **at a specified index in the list, allowing for nodes to be inserted into the middle of the list.**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def insert_at(self, position, data):
        # Create a new node
        new_node = Node(data)

        # Check if the position is valid
        if position < 0 or position > self.size:
            return "Invalid position"

        # If the position is 0 (inserting at the beginning)
        if position == 0:
            new_node.next = self.head
            self.head = new_node
        else:
            # Initialize current to the head of the list
            current = self.head
            # Traverse the list to reach the node just before the
            desired position
            for _ in range(position - 1):
                current = current.next
                if current is None:
                    return "Invalid position"
            # Update the next pointers
            new_node.next = current.next
            current.next = new_node

        # Increment the size of the list
        self.size += 1

    def print_list(self):
        current = self.head
        while current is not None:
            print(current.data, end=" ")
            current = current.next
        print()

# Example usage
linked_list = LinkedList()

```

```

linked_list.insert_at(0, 1) # Insert 1 at position 0
linked_list.insert_at(1, 2) # Insert 2 at position 1
linked_list.insert_at(1, 3) # Insert 3 at position 1
linked_list.insert_at(3, 4) # Insert 4 at position 3

print("Linked List:")
linked_list.print_list()
print("Size:", linked_list.size)

```

Output:

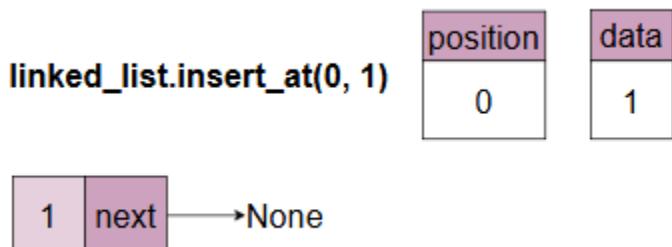
Linked List:

1 3 2 4

Size: 4

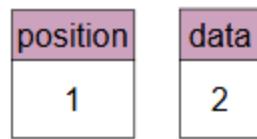
Here is the visualization of the linked list inserting process of the example above:

linked_list.insert_at(0,1):

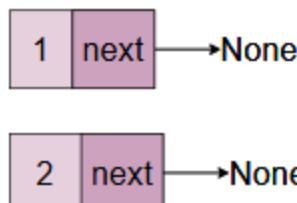


linked_list.insert_at(1,2):

`linked_list.insert_at(1, 2)`

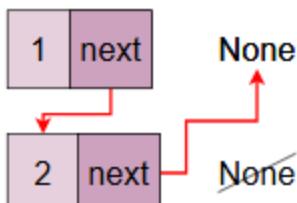


Current



`new_node`

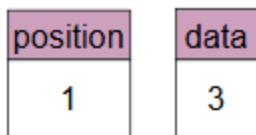
Current



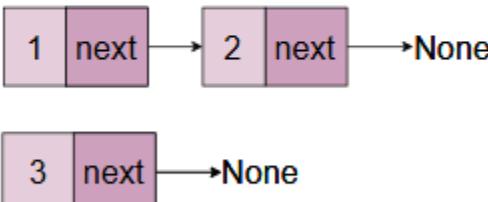
`new_node`

`linked_list.insert_at(1,3):`

`linked_list.insert_at(1, 3)`

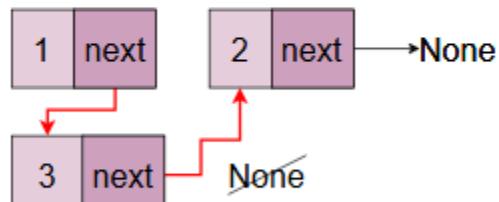


Current



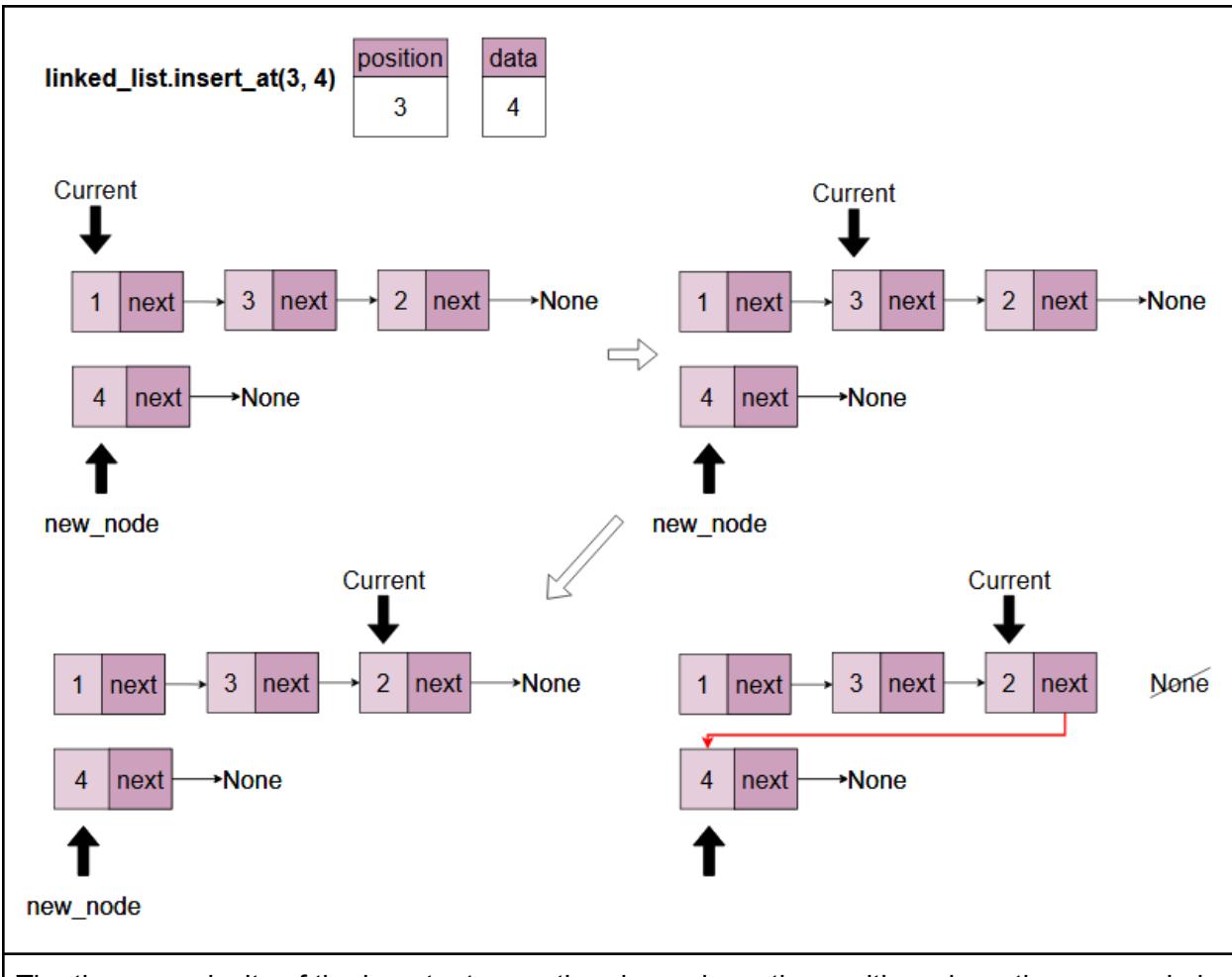
`new_node`

Current



`new_node`

`linked_list.insert_at(3,4):`



The time complexity of the `insert_at` operation depends on the position where the new node is being inserted.

Let's analyze the steps involved:

1. Creating a new node: **O(1)**
 - a. Allocating memory for a new node and assigning the data to it takes constant time.
2. Checking if the position is valid: **O(1)**
 - a. Comparing the position with 0 and the size of the list takes constant time.
3. Inserting at the beginning (position 0): **O(1)**
 - a. If the position is 0, updating the next pointer of the new node and the head takes constant time.
4. Traversing the list to reach the desired position: **O(n)**
 - a. In the worst case, where the position is near the end of the list, we need to traverse almost the entire list to reach the desired position.
 - b. The traversal requires visiting each node from the head until the node just

before the desired position is reached.

- c. The time complexity of this step is proportional to the position, which can be at most the size of the list minus one.

5. Updating the next pointers: $O(1)$

- a. Setting the next pointer of the new node to the next node of the current node and updating the next pointer of the current node takes constant time.

6. Incrementing the size of the list: $O(1)$

- a. Incrementing the size variable takes constant time.

Therefore, the overall time complexity of the `insert_at` operation is $O(n)$ in the **worst** case, where n represents that the time complexity of this operation is linearly proportional to the length of the linked list.

However, it's important to note that the time complexity can be lower in certain cases:

- If the position is 0 (inserting at the beginning), the time complexity is $O(1)$ since no traversal is required.
- If the position is close to the beginning of the list, the traversal step will be shorter, resulting in a lower time complexity.

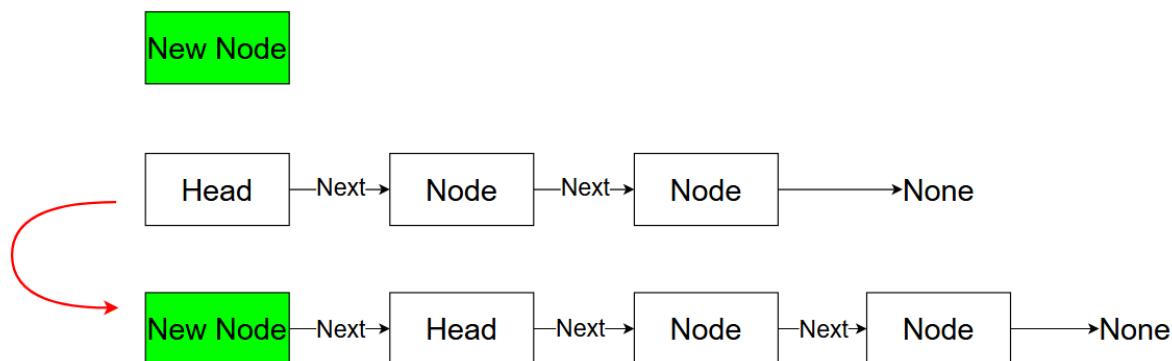
Space Complexity:

The space complexity of the `insert_at` operation is $O(1)$ because it only requires a constant amount of extra space to create a new node, regardless of the size of the linked list.

Exercises 1 - 5 (Images)

Exercise 1

What Linked List operation is performed in the image below?



- `prepend(new_node)`
- `append (new_node)`

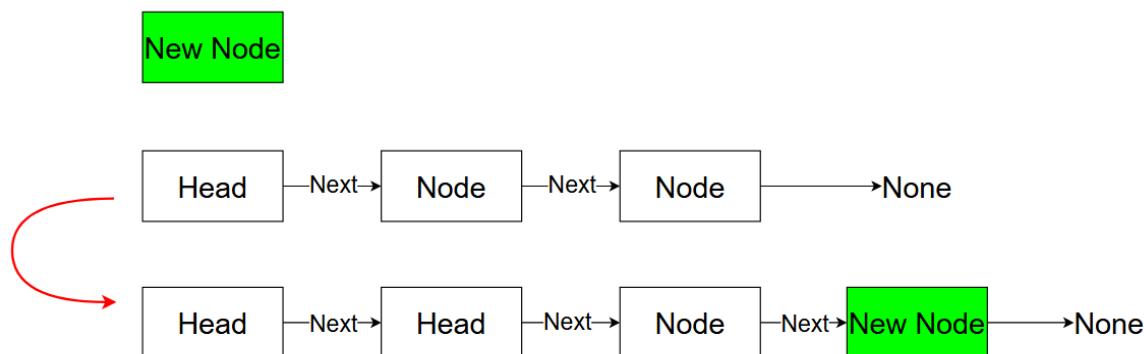
- insert_at(2, new_node)
- index(new_node)

Explanation:

The **new node** has its *next field* set to the **head**, and the **head** has its *next field* set to the **new node**. This describes the insertion of a node at the start of a linked list, which is the `prepend()` method.

Exercise 2

What Linked List operation is performed in the below image?



- prepend(new_node)
- append(new_node)
- insert_at(0, new_node)
- get(new_node)

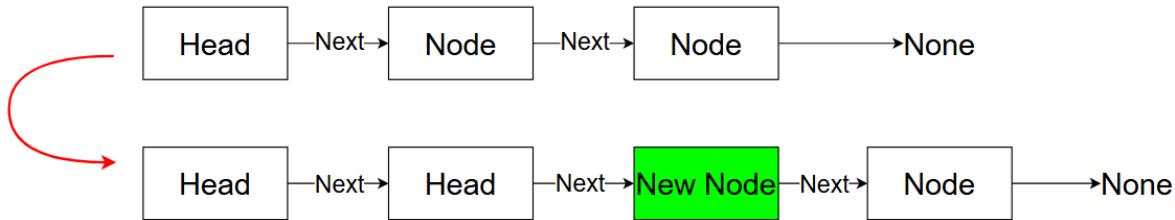
Explanation:

The **last node** has its *next field* set to the **new node**, this describes the insertion of a node at the end of a linked list, which is the `append()` method.

Exercise 3

What Linked List method is shown in the below image?

New Node



- prepend(new_node)
- append(new_node)
- ~~insert_at(2, new_node)~~
- insert_at(3, new_node)

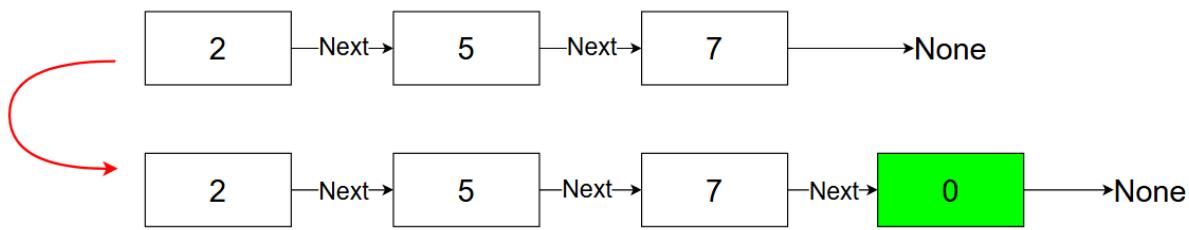
Explanation:

Neither the head or tail node are updated, meaning this insertion is somewhere in the middle of the linked list. This describes the insertion of a node at a given position, which is the `insert_at()` method.

Exercise 4

What Linked List method is shown in the below image?

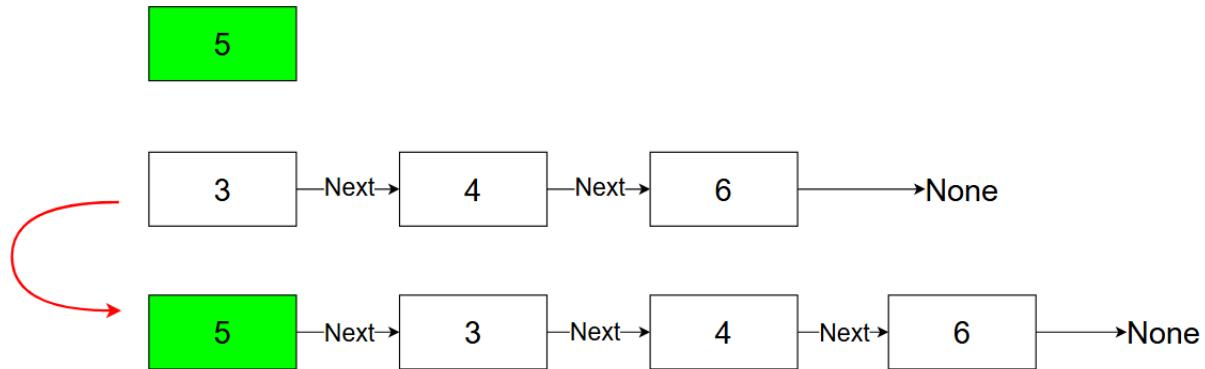
0



- prepend(0)
- ~~append(0)~~
- ~~insert_at(3, 0)~~
- insert_at(2, 0)

Exercise 5

What Linked List method is shown in the below image?



- prepend(5)
- append(5)
- insert_at(4, 5)
- insert_at(1, 5)

Exercises 6 - 11 (MCQ)

Ex 6

What are the benefits of using class methods for implementing linked list functionality? Select all correct answers

- It reduces the amount of code that needs to be written
- It makes linked lists use less storage
- It prevents errors by simplifying linked list usage
- It prevents linked lists from being misused

Explanation: Classes generally avoid having to retype the same code, and the defined structure prevents errors and misuse. But the storage size is determined by the size of the list.

Ex 7

What does the append() method do?

- Adds a node to the start of the list
- Replaces the node at the end of the list

- | |
|---|
| <input checked="" type="checkbox"/> Adds a node to the end of the list |
| <input type="checkbox"/> Replaces the data in the node at the end of the list |

Ex 8

What does the prepend() method do?

- | |
|---|
| <input type="checkbox"/> Adds a node to the end of the list |
| <input checked="" type="checkbox"/> Adds a node to the beginning of the list |
| <input type="checkbox"/> Removes the node at the start of the list |
| <input type="checkbox"/> Replaces the data in the node at the start of the list |

Ex 9

What does the insert_at() method do?

- | |
|--|
| <input type="checkbox"/> Adds a node at a specified index and removes the rest of the list |
| <input type="checkbox"/> Adds a new node after the specified index |
| <input type="checkbox"/> Changes the data at a specified index |
| <input checked="" type="checkbox"/> Adds a new node at a specified index at the list |

Ex 10

What will the following code do?

```
list3 = LinkedList()
list3.prepend("Hello")
```

- | |
|--|
| <input type="checkbox"/> Add a node storing the string "Hello" to the start of the list |
| <input checked="" type="checkbox"/> Add the string "Hello" to the head variable in the linked list |
| <input type="checkbox"/> Add a node storing the string "Hello" to the end of the list |
| <input type="checkbox"/> Remove the node storing the string "Hello" |

Explanation: the second option may be confusing

1: Correct: `prepend ("Hello")` adds a new node with the string "Hello" at the start of the list.

2: Incorrect: The code does not add the string “Hello” to the head variable, the string is added to a new node which is then added to the start of the linked list to be the new head.

Ex 11

What will the following code do?

```
i = 0  
count = 5  
data = 10  
list5 = LinkedList()  
while i < count:  
    list5.append(data)  
    i += 1
```

- Starting at index 5 it will print out every list element
- It will print out every list element
- ~~it will append the data 10 to the linked list 5 times~~
- It will replace every list element

Exercises 12 - 14 (Fill in the Blank)

Ex 12

Complete the code below to create the prepend() method.

```
def prepend(self, data):  
    new_node = Node(data)  
    new_node.next = [Fill in answer here]  
    [Fill in answer here] = new_node  
    self.size += 1
```

Expected answer:

```
def prepend(self, data):  
    new_node = Node(data)  
    # Making new_node to be a new head of the linked list.  
    new_node.next = self.head  
    self.head = new_node  
    self.size += 1
```

Ex 13

Complete the code below to create the append() method.

```
def append(self, data):
    new_node = Node(data)
    if self.head is [Fill in answer here]:
        self.head = new_node
        self.size += 1
        return
    last_node = self.head
    while last_node.next:
        last_node = [Fill in answer here]
    last_node.next = new_node
    self.size += 1
```

Expected answer:

```
def append(self, data):
    new_node = Node(data) # Initialize a new node with data
    "data"
    if self.head is None: # If the list is empty, set the new
    node as the head node.
        self.head = new_node # Making the new node as head
    node.
        self.size += 1
        return
    last_node = self.head # Start at the beginning of the
linked list to find the last item.
    while last_node.next: #Traverse the linked list to find
the last item.
        last_node = last_node.next
    last_node.next = new_node # Attach the new node as the
next item of the last node.
    self.size += 1
```

Ex 14

Complete the code below to create the insert_at() method.

```
def insert_at(self, index, data):
    if index < 0 or index > self.size: # Check for invalid
    indices
        return "Invalid Index"
    if index == [Fill in answer here]:
        self.prepend(data)
        return
    new_node = Node(data)
    count = 0
```

```

cur_node = self.head
while cur_node:
    if count == index - 1:
        [Fill in answer here]
        cur_node.next = new_node
        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

```

Expected answer:

```

def insert_at(self, index, data):
    if index < 0 or index > self.size: # Check for invalid
    indices
        return "Invalid Index"
    if index == 0: # If the node is to be added at the start
    of the list
        self.prepend(data) # Add at the start of the linked
    list function
        return
    new_node = Node(data)
    count = 0
    cur_node = self.head
    while cur_node: # Loop runs until stop at the None
        if count == index - 1: # Set a condition to insert
    at "index"
            new_node.next = cur_node.next
            cur_node.next = new_node
            self.size += 1
            return
    cur_node = cur_node.next
    count += 1

```

Explanation:

The `insert_at()` method inserts a new node at a specified index in the linked list.

1. **If the index is 0:**
 - o The method calls `self.prepend(data)` to insert the new node at the beginning of the list.
2. **If the index is greater than 0:**
 - o The code iterates through the list to find the node at `index - 1`.
 - o Once it finds the node, it sets the `next` pointer of the new node to point to the next node (`cur_node.next`), and then sets the `next` pointer of the current node (`cur_node`) to point to the new node.
 - o The list size is incremented to reflect the addition of the new node.

Exercises 15 - 25 (Coding)

Story:

Stephen is a system administrator for Super City's University. It is his job to organise the usernames of users that need access into the University's network. He is requesting help from you, a programmer, to build a data structure to record these usernames.

Ex 15

Create a basic Node class for a singly linked list that will store the names of computer users as strings.

The Node class should have an `__init__` method that takes a parameter name (representing the name of a computer user).

Inside the `__init__` method, initialize the name attribute with the provided parameter and set the next attribute to None.

Given Code:

```
class Node:  
    # Your code here
```

Expected input:

```
class Node:  
    def __init__(self, name):  
        self.name = name  
        self.next = None
```

Ex 16

Create a linked list class for a singly linked list and define its `__init__` constructor.

Implement the `__init__` method within the `LinkedList` class.

Inside the `__init__` method, initialize two attributes:

- head: Represents the starting point of the linked list; initially set it to None as the list is empty.
- size: Indicates the number of nodes in the linked list; initially set it to 0.

Given Code:

```
class LinkedList:  
    # Your code here
```

Expected input:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0
```

Ex 17

Story:

Steven believes an append() method would be an appropriate method to add to the LinkedList class. Can you assist him by making this method for him? He has detailed how to make the method below:

1. Create a new node with the data to be appended inside it.
2. Check if the list is empty:
 - a. If the list is empty then set the node as the head of the list and increase the list's size.
3. While there are no nodes in the list, traverse the list until the end is reached.
4. Set the last node's next value to the new node
5. Increase the size of the list.

Write code for the append() method in the LinkedList class that adds a new node containing the given data at the end of the list.

Given Code:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        # Your code here
```

Expected input:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.size += 1  
            return  
        last_node = self.head  
        while last_node.next:  
            last_node = last_node.next  
        last_node.next = new_node  
        self.size += 1
```

Ex 18

Story:

Steven believes an prepend() method would be an appropriate method to add to the LinkedList class. Can you assist him by making this method for him? He has detailed how to make the method below:

Write code for the prepend() method in the LinkedList class.

Given code:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.size += 1  
            return  
        last_node = self.head  
        while last_node.next:  
            last_node = last_node.next  
        last_node.next = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        # Your code here
```

Hint:

To add a new node at the beginning of the linked list:

- Create a new node with the given data.
- Set the **next** pointer of the new node to point to the current head of the list.
- Update the head of the list to be the new node.
- Don't forget to increment the size of the list.

Expected input:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:
```

```

        self.head = new_node
        self.size += 1
        return
    last_node = self.head
    while last_node.next:
        last_node = last_node.next
    last_node.next = new_node
    self.size += 1

def prepend(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
    self.size += 1

```

Ex 19

Story:

Steven believes an `insert_at()` method would be an appropriate method to add to the `LinkedList` class. Can you assist him by making this method for him? He has detailed how to make the method below:

Write code for the `insert_at()` method in the `LinkedList` class.

Given code:

```

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.size += 1
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def insert_at(self, index, data):
        # Your code here

```

Hint:

To insert a new node at a specific index:

- If the index is `0`, you can reuse the `prepend()` method to insert the node at the beginning.
- If the index is greater than `0`, iterate through the list until you reach the node at `index - 1`.
- Link the new node to the next node and update the `next` pointer of the current node to point to the new node.
- Make sure to update the size of the list after insertion.

Expected input:

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.size += 1  
            return  
        last_node = self.head  
        while last_node.next:  
            last_node = last_node.next  
        last_node.next = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def insert_at(self, index, data):  
        if index == 0:  
            self.prepend(data)  
            return  
        new_node = Node(data)  
        count = 0  
        cur_node = self.head  
        while cur_node:  
            if count == index - 1:  
                new_node.next = cur_node.next
```

```

        cur_node.next = new_node
        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

```

You have completed the implementation of the Node and LinkedList, well done! The final code is below. Now, use this code for exercises 20-26 to create nodes in the linked list and print it.

```

class Node:
    def __init__(self, name):
        self.name = name
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.size += 1
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def insert_at(self, index, data):
        if index == 0:
            self.prepend(data)
            return
        new_node = Node(data)
        count = 0
        cur_node = self.head
        while cur_node:
            if count == index - 1:
                new_node.next = cur_node.next
                cur_node.next = new_node
            count += 1

```

```

        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

def print_list(self):
    current = self.head
    while current is not None:
        print(f"{current.name} -> ", end="")
        current = current.next
    print("None")

```

Ex 20

Now that we have created the appropriate data structure, Steve needs to start filling the list with patrons of the University's network. Using the defined methods, create a linked list called `user_list` and add the username "Phil_Anthropist" to the list.

```

user_list = LinkedList()
# Your code here

user_list.print_list()

```

Expected Input:

```

user_list = LinkedList()
user_list.append("Phil_Anthropist") OR
user_list.prepend("Phil_Anthropist")

```

```
user_list.print_list()
```

Current user list:

`Phil_Anthropist -> None`

Ex 21

Using the defined methods, append the following names to the linked list 'user_list' in the shown order.

- Anita_Catcher1
- EllaVader1997

```

user_list = LinkedList()
user_list.append("Phil_Anthropist")
# Your code here

user_list.print_list()

```

Expected Input:

```

user_list = LinkedList()
user_list.append("Phil_Anthropist")

```

```
user_list.append("Anita_Catcher1")
user_list.append("EllaVader1997")

user_list.print_list()
```

Current user list:

Phil_Anthropist -> Anita_Catcher1 -> EllaVader1997 -> None

Ex 22

Using the defined methods, prepend the following names to the linked list 'user_list' in the shown order.

- RoxanneMinerals
- JustinCase

```
user_list = LinkedList()
user_list.append("Phil_Anthropist")
user_list.append("Anita_Catcher1")
user_list.append("EllaVader1997")
# Your code here

user_list.print_list()
```

Expected input:

```
user_list.prepend("RoxanneMinerals")
user_list.prepend("JustinCase")
```

Current user list:

JustinCase -> RoxanneMinerals -> Phil_Anthropist -> Anita_Catcher1 -> EllaVader1997 -> None

Ex 23

Using the defined methods add the name "RonENose2" to the end of the user_list.

```
user_list = LinkedList()
user_list.append("Phil_Anthropist")
user_list.append("Anita_Catcher1")
user_list.append("EllaVader1997")
user_list.prepend("RoxanneMinerals")
user_list.prepend("JustinCase")
# Your code here

user_list.print_list()
```

Expected input:

```
user_list.append("RonENose2")
```

Current user list:

RonENose2 -> JustinCase -> RoxanneMinerals -> Phil_Anthropist -> Anita_Catcher1 ->
EllaVader1997 -> None

Ex 24

Using the defined methods add the name "SeymourClearly1954" at index 3 of the user_list.

```
user_list = LinkedList()
user_list.append("Phil_Anthropist")
user_list.append("Anita_Catcher1")
user_list.append("EllaVader1997")
user_list.prepend("RoxanneMinerals")
user_list.prepend("JustinCase")

user_list.prepend("RonENose2")
# Your code here

user_list.print_list()
```

Expected input:

```
user_list.insert_at(3, "SeymourClearly1954")
```

Current user list:

RonENose2 -> JustinCase -> RoxanneMinerals -> SeymourClearly1954 -> Phil_Anthropist ->
Anita_Catcher1 -> EllaVader1997 -> None

Ex 25

Create a method print_list in the linked list class to print all of the names in the list. Use this to print out user_list.

Expected input:

```
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.name)
        cur_node = cur_node.next

user_list.print_list()
```

Expected output:

RonENose2
JustinCase
RoxanneMinerals
SeymourClearly1954
Phil_Anthropist

Anita_Catcher1
EllaVader1997

3.3 Accessing a Node

In the previous section, we discussed the basic operations of a linked list, including inserting nodes. While insertion is crucial, efficiently accessing nodes in a linked list is equally important.

As more nodes are added, using statements like

`list1.head.next.next.next.next.next.data` become increasingly cumbersome and impractical.

To overcome these challenges, we need methods to efficiently access nodes in a linked list.

Now, let's consider the different methods for accessing data in a singly linked list:

- **Accessing a Node by Index - `get()`**: Retrieve the data of a node at a specified position in the list without manually traversing the list each time.
- **Finding the Index of a Node by Data - `index()`**: Search through the linked list to find the index of the first node with matching data.

In other words, the `get()` method uses an index parameter to find the data of the node, and the `index()` method uses a data parameter to find the index of the node. In the following lessons, we will look into these methods.

3.3.1 `get()` Method

The `get()` method is fundamental for accessing data stored in a linked list. It allows you to retrieve the data of a node at a specified position without manually traversing the list step by step each time using `.next` repeatedly, significantly simplifying the access process.

Steps:

1. **Validate Index**: Check if the provided index is negative or out of the linked list size. If so, return -1 to indicate an invalid index.
2. **Initialize Current Node**: Set the current node (`cur_node`) to the head of the linked list.
3. **Initialize Counter**: Use a counter (`count`) to track the current position of the node.
4. **Traverse the List**: Use a while loop to iterate through the linked list.
5. **Check Index Match**: During each iteration, check if the current node is at the specified index.
 - a. If Match Found: Return the data stored in the current node.
 - b. If No Match: Move to the next node in the linked list and increment the counter.
6. **Step 6: If the loop completes without finding the specified node, return -1.**

The `get()` method provides a way to access data in a linked list based on the specified index. It checks if the index is valid and if so, traverses the list until reaching the node at the specified index.

If found, it returns the data stored in that node; otherwise, it returns -1 to indicate that the index is invalid.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def get(self, index):
        # Check if the index is negative
        if index < 0 or index >= self.size:
            return -1 # Indicates that `index` is an invalid index

        # Initialize the current node to the head of the linked list
        cur_node = self.head
        count = 0 # Counter to track the position of the current
node

        while cur_node: # Traverse the linked list
            if count == index: # Check if the current node is at the
specified index
                return cur_node.data # Return the data stored in the
current node
            cur_node = cur_node.next # Move to the next node in the
```

```

linked list
    count += 1 # Increment the counter
    return -1 # If the loop completes without finding the node,
return -1

```

Example

```

linked_list = LinkedList()
linked_list.append("John")
linked_list.append("Ben")
linked_list.append("Matthew")

# Accessing elements using the get() method
print(linked_list.get(0))
print(linked_list.get(1))
print(linked_list.get(2))
print(linked_list.get(3))

```

Output:

John
Ben
Matthew
-1

The time complexity of the get operation depends on the position of the target node we are looking for.

Let's analyze the steps involved:

1. Checking if the index is valid: O(1)
 - a. Comparing the index with 0 and the size of the list takes constant time.
2. Assigning the head node as the current node: O(1)
 - a. The current node denotes the node we are currently traversing.
3. Setting the count to 0 initially: O(1)
 - a. The count acts as a running counter to keep track of the index/position of the current node.
4. Traversing the list to reach the desired position: O(n)
 - a. In the worst case, where the position is near the end of the list, we need to traverse almost the entire list to reach the desired position.
 - b. The traversal requires visiting each node from the head until the node just before the desired position is reached.
 - c. The time complexity of this step is proportional to the position, which can be at most the size of the list minus one.

Therefore, the overall time complexity of the get operation is $O(n)$ in the worst case, where n represents the linear relationship between the time complexity and the input size (the length of the linked list).

However, it's important to note that the time complexity can be lower in certain cases:

- If the position is 0 (inserting at the beginning), the time complexity is $O(1)$ since no traversal is required.
- If the position is close to the beginning of the list, the traversal step will be shorter, resulting in a lower time complexity.

Space Complexity:

The space complexity of the get operation is $O(1)$ because it only requires a constant amount of extra space to create a counter, and storing the current node we are traversing, regardless of the size of the linked list.

3.3.2 `index()` Method

The `index()` method is essential for finding the position of a node with specified data in a linked list. It traverses the list to locate the first occurrence of the given data and returns its index.

This method simplifies the process of searching for elements in the list.

Steps:

1. **Initialize the Current Node:** Start from the head of the linked list.
2. **Initialize a Counter:** Use a counter to keep track of the current index.
3. **Traverse the Linked List:** Use a while loop to go through each node.
4. **Check for Matching Data:** Compare the current node's data with the search data.
5. **Return the Index:** If the data matches, return the current index.
6. **Move to the Next Node:** If the data does not match, move to the next node and increment the counter.
7. **Return -1 if Not Found:** If the loop completes without finding the data, return -1 to indicate the data is not in the list.

The `index()` method searches through the linked list to find the first node with data matching the specified search data. It returns the index (position) of that node if found, and -1 if no node contains the specified data.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def index(self, data):
        cur_node = self.head # Starts from the head of the linked
list
        count = 0 # Initialize a counter to keep track of index

        while cur_node:
            if cur_node.data == data: # Check if the current node's
data matches the search data
                return count # If found, return its current index
(count)

            cur_node = cur_node.next # Move to the next node
            count += 1 # Increment the index for the next node
        return -1 # If the data is not found, return -1

# Example usage
linked_list = LinkedList()
linked_list.append("John")
linked_list.append("Ben")
linked_list.append("Matthew")

print(linked_list.index("John"))
print(linked_list.index("Ben"))

```

```
print(linked_list.index("Matthew"))
print(linked_list.index("David"))
```

Output:

0
1
2
-1

The time complexity of the index operation depends on the position of the target node we are looking for.

Let's analyze the steps involved:

1. Assigning the head node as the current node: O(1)
 - a. The current node denotes the node we are currently traversing.
2. Setting the count to 0 initially: O(1)
 - a. The count acts as a running counter to keep track of the index/position of the current node.
3. Traversing the list to reach the desired position: O(n)
 - a. In the worst case, where the position is near the end of the list, we need to traverse almost the entire list to reach the desired position.
 - b. The traversal requires visiting each node from the head until the node just before the desired position is reached.
 - c. The time complexity of this step is proportional to the position, which can be at most the size of the list minus one.

Therefore, the overall time complexity of the index operation is O(n) in the worst case, where n represents the linear relationship between the time complexity and the input size (the length of the linked list).

However, it's important to note that the time complexity can be lower in certain cases:

- If the position is 0 (accessing the first node), the time complexity is O(1) since no traversal is required.
- If the position is close to the beginning of the list, the traversal step will be shorter, resulting in a lower time complexity.

Space Complexity:

The space complexity of the get operation is O(1) because it only requires a constant amount of extra space to create a counter, and storing the current node we are traversing, regardless of the size of the linked list.

Exercises 1 - 2 (Fill the gap)

Story:

Stephen is thrilled with the code we made last lesson and is hoping to commission your help once more to add functionality to the current code. He is hoping to add the get() and index() method to his Linked List and has written some of it, but has gotten stuck. Can you help him fill in the blanks?

Ex 1

Complete the code below to create the get() method.

```
def get(self, index):
    if index < 0:
        return -1
    cur_node = [Fill in answer here]
    count = 0
    while cur_node:
        if count == index:
            return [Fill in answer here]
        cur_node = cur_node.next
        count += 1
    return -1
```

Expected answer:

```
def get(self, index):
    if index < 0:
        return -1
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1
```

Ex 2

Complete the code below to create the index() method.

```
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
```

```

        return [Fill in answer here]
    cur_node = cur_node.next
    count += [Fill in answer here]
return -1

```

Expected answer:

```

def index(self, data):
    cur_node = self.head #Starts from the head of the linkedList
    count = 0 # Initialize a counter to keep track of index
    while cur_node:
        if cur_node.data == data: # Check if the current node's data
matches the search data
            return count # If found , return its current index
(count)
        cur_node = cur_node.next
        count += 1 # Increment the index for the next node
return -1

```

Exercises 3 - 5 (Coding)

Ex 3

Write code for the `get()` method in the `LinkedList` class.

Given code:

```

def get(self, index):
    # Your code here

```

Hint:

- Base case: check whether the index is less than 0 or larger than list size and return -1 if so.
- Start with the head node and set a counter to track the index.
- Traverse through the list until you find the node that matches the given index.
- If the node is found, return its data; if not, return -1 indicating the index is invalid.

Expected input

```

def get(self, index):
    if index < 0 or index >= self.size:
        return -1
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1 # Handle unexpected scenario

```

Ex 4

Write code for the `index()` method in the `LinkedList` class.

Given Code:

```
def index(self, data):  
    # Your code here
```

Expected input

```
def index(self, data):  
    cur_node = self.head  
    count = 0  
    while cur_node:  
        if cur_node.data == data:  
            return count  
        cur_node = cur_node.next  
        count += 1  
    return -1 # Not found data in the list
```

Ex 5

Write a method to search for a node in a linked list that receives a parameter `key` as input and return True if found otherwise False. You may use the `get` method as a helper method.

Given Code:

```
def search(self, key):  
    # Your code here
```

Expected input

```
def search(self, key):  
    count = 0  
    while count < self.size:  
        if self.get(count) == key:  
            return True  
        count += 1  
    return False
```

Exercise 6 - 9**Story:**

Now that we have added the necessary methods to our program. Stephen wants to make sure that it is working correctly. He has come to you to help him code some implementations of the linked list that he is likely to need when the system comes live.

Refer to the following code for exercises 6-12:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.size += 1  
            return  
        last_node = self.head  
        while last_node.next:  
            last_node = last_node.next  
        last_node.next = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def insert_at(self, index, data):  
        if index == 0:  
            self.prepend(data)  
            return  
        new_node = Node(data)  
        count = 0  
        cur_node = self.head  
        while cur_node:  
            if count == index - 1:  
                new_node.next = cur_node.next  
                cur_node.next = new_node  
                self.size += 1  
                return  
            cur_node = cur_node.next  
            count += 1  
  
    def get(self, index):  
        cur_node = self.head  
        count = 0  
        while cur_node:  
            if count == index:
```

```

        return cur_node.data
    cur_node = cur_node.next
    count += 1
return None

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

user_list = LinkedList()
user_list.append("Phil_Anthropist")
user_list.append("Anita_Catcher1")
user_list.append("EllaVader1997")
user_list.prepend("RoxanneMinerals")
user_list.prepend("JustinCase")

```

Using the get() method, retrieve and print the values of the nodes at index 1 and index 2 in the user_list linked list. Get and print the values of the nodes at index 1 and 2

Note: There's a given method get() in the LinkedList class.

Ex 6

Stephen knows that he will need to retrieve specific usernames in future development so he has asked you to ensure that the get() method works.

Using the get() method, retrieve and print the values of the nodes at index 1 and index 2 in the user_list linked list. Get and print the values of the nodes at index 1 and 2.

Note: There's a given method get() in the LinkedList class.

Expected input:

```
print(user_list.get(1))
print(user_list.get(2))
```

Expected output:

RoxanneMinerals
Phil_Anthropist

Ex 7

Stephen will need to make sure that University patrons are using usernames of sufficient

length to ensure security. He has asked you to craft a solution that he can use later.

Write a program to print all of the names with fifteen or more letters in user_list.

Note: You may use the given methods get() or index() directly.

Expected input:

```
count = 0
while count < user_list.size:
    if len(user_list.get(count)) >= 15:
        print(user_list.get(count))
    count += 1
```

Expected output:

Phil_Anthropist
RoxanneMinerals

Ex 8

Stephen needs to implement a login system now that we have a working username database.

Create a specialised function to check if a user has username 'name' in the linked list. Return True if it is in the list.

Note: You may use the given methods get() or index().

Expected input:

```
def has_name(name):
    if user_list.index(name) != -1:
        return True
    return False
```

OR

```
def has_name(name):
    count = 0
    while count < user_list.size:
        if user_list.get(count) == name:
            return True
        count += 1
    return False

# Example usage:
print(has_name("Phil")) # True
print(has_name("Lucy")) # False
```

Ex 9

Now that we are working on a user verification program, create a program where a user enters a username when prompted "What is your name? " and is either greeted "Welcome back" or is told that "You do not have an account".

Hint: use the has_name() function made in the previous exercise.

Expected input:

```
name = input("What is your name? ")
if has_name(name):
    print("Welcome back")
else:
    print("You do not have an account")
```

Exercises 10-12

Story:

Now we have created a username manager, we now need to implement a password manager. Stephen has asked you for help once again:

Ex 10

We should store the passwords in a separate linked list structure. Create another linked list called password_list to store the user's passwords.

Expected input:

```
password_list = LinkedList()
```

Ex 11

Add the password "passw0rd" to the list password_list.

Expected input:

```
password_list.append("passw0rd") OR
password_list.prepend("passw0rd")
```

Ex 12

Modify the login program to incorporate password verification. Make a has_password function that is similar to the has_name function. Then prompt the user for their name and password. Check if the entered name is in the user_list and if the corresponding password matches the one stored in the password_list. If the credentials are correct, print "Welcome back"; otherwise, print "Incorrect username or password".

Expected input:

```
def has_password(password):  
    if password_list.index(password) != -1:  
        return True  
    return False
```

OR

```
def has_password(password):  
    count = 0  
    while count < password_list.size:  
        if password_list.get(count) == password:  
            return True  
        count += 1  
    return False  
  
name = input("What is your name? ")  
password = input("what is your password?")  
if (has_name(name) and has_password(password)):  
    print("Welcome back")  
else:  
    print("Incorrect username or password")
```

3.4 Delete Data in Singly Linked List

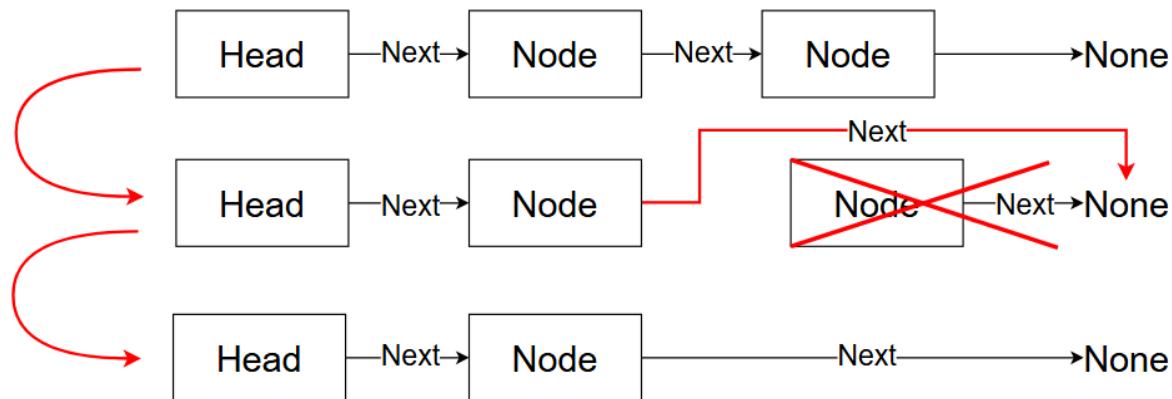
Now that we have methods that can handle the addition of nodes and the accessing of data from a linked list, we should come up with a way to remove this data from the list.

Removing unnecessary nodes helps to maintain the usability of the linked list while also enhancing the efficiency by freeing up memory.

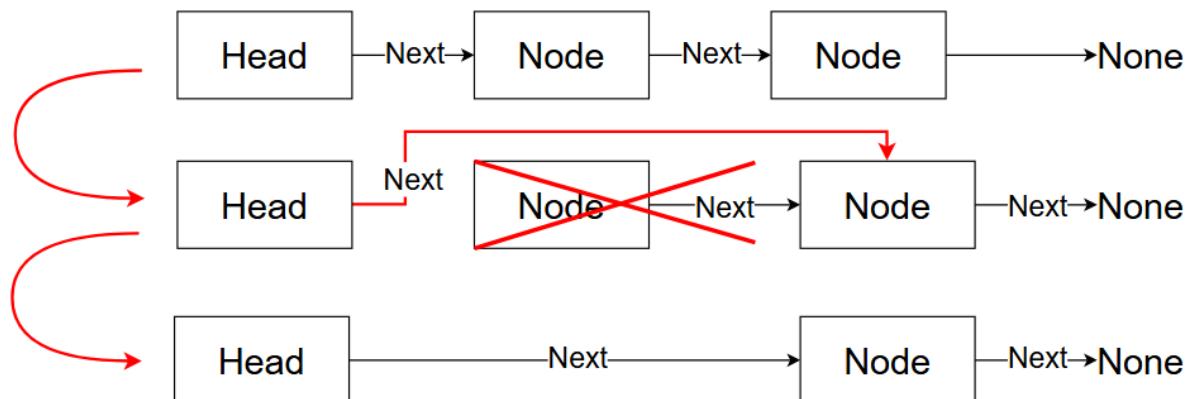
Thus, implementing methods to delete nodes is essential for managing a linked list effectively.

Deleting data from a singly linked list involves removing a node from the list and updating the next pointers accordingly. There are three common scenarios for deleting data:

- **Deleting the Last Node - `delete_last()`**: Remove the last node in the list by updating the next pointer of the second-to-last node to **None**.



- **Deleting a Node at a Specific Position - `delete_at()`**: Traverse the list to reach the node just before the desired position and update its next pointer to skip the node to be deleted. This operation can be used to remove the head node if the given index is 0.

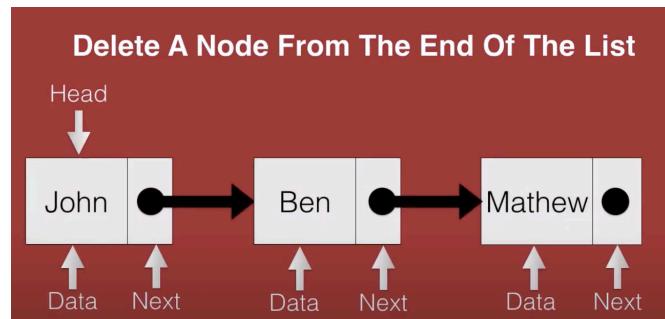


- **Deleting a node with specific value - `delete()`**: Remove the node that contains a specific value.

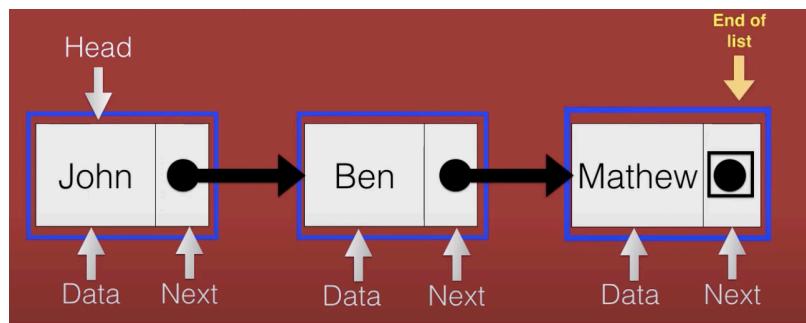
Let's explore each scenario in detail!

3.4.1 `delete_last()` Method

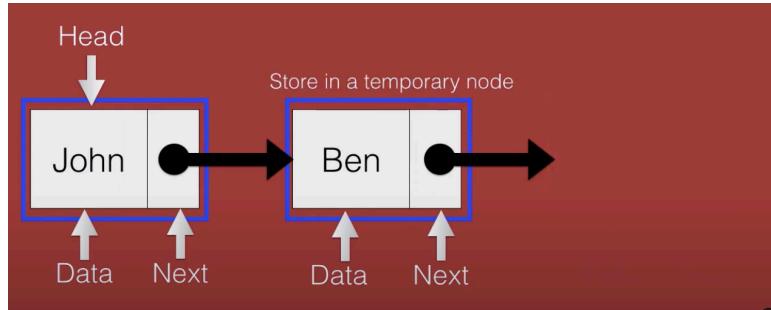
Let's take an example of a singly linked list with three nodes. The first node is our head node with the data "John", the second node with the data "Ben", and the third node with the data "Mathew". So now, "Mathew" is the node which we want to delete.



We start from the head node "John" and move to the second node "Ben" using the next pointer of "John." Then, using the next pointer of "Ben," we move to the third node "Mathew." We can see that the next pointer of "Mathew" is **None**, indicating that "Mathew" is the last node.



Before we delete the node "Mathew", we need to store the second last node in a temporary node, so that we can update its next field to reflect the deletion of the node "Mathew". In this case, we set the next field of the node "Ben" to none. This completes the deletion process.



So in summary, to delete this node "Matthew," we need to follow these steps:

1. **Traverse to the second last node:** Start from the head and move to the **second-last** node.
2. **Preserve the Penultimate Node:** Store the second-to-last node (in this case, "Ben") in a temporary variable.
3. **Delete the Last Node:** Remove the last node ("Matthew").
4. **Update the Next Pointer:** Make the next pointer of the second-to-last node (Ben) point to None.

Now, let's write this operation in Python code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def delete_last(self):
        # Check if the list is empty
        if self.head is None:
            return

        # If the list has only one node
        if self.head.next is None:
            self.head = None
            self.size = 0
        else:
            # Traverse the list to find the second-to-last node
```

```

        current = self.head
        while current.next.next is not None:
            current = current.next

            # Update the next pointer of the second-to-last node
            current.next = None
            self.size -= 1

    def print_list(self):
        current = self.head
        while current is not None:
            print(current.data, end=" ")
            current = current.next
        print()

```

```

# Example usage
linked_list = LinkedList()

node1 = Node("John")
node2 = Node("Ben")
node3 = Node("Mathew")

linked_list.head = node1
node1.next = node2
node2.next = node3

print("Original Linked List:")
linked_list.print_list()

linked_list.delete_last()

print("Linked List after deleting the last node:")
linked_list.print_list()

```

Output:

Original Linked List:

John Ben Mathew

Linked List after deleting the last node:

John Ben

So in summary, deleting the last node in a singly linked list involves traversing the list to find the second-to-last node and updating its next pointer to None. Let's analyze its time complexity!

The steps involved:

1. Checking if the list is empty: **O(1)**
 - a. Checking if the head is None takes constant time.
2. Checking if the list has only one node: **O(1)**
 - a. Checking if the next pointer of the head is None takes constant time.
3. Traversing the list to find the second-to-last node: **O(n)**
 - a. In order to delete the last node, we need to find the second-to-last node.
 - b. This requires traversing the list from the head until we reach the node whose next pointer points to the last node.
 - c. The time complexity of this step is proportional to the size of the list minus one.
4. Updating the next pointer of the second-to-last node: **O(1)**
 - a. Once we have the second-to-last node, updating its next pointer to None takes constant time.
5. Decrementing the size of the list: **O(1)**
 - a. Decrementing the size variable takes constant time.

Therefore, the overall time complexity of deleting the last node is **O(n)**, because finding the second-to-last node involves traversing the list.

However, there are a few special cases to consider:

- If the list is empty, the operation takes constant time **O(1)** since no traversal is required.
- If the list has only one node, the operation takes constant time **O(1)** as we only need to update the head to None.

Space Complexity:

The space complexity of deleting the last node is **O(1)** because it only requires a constant amount of extra space to store the temporary variables, regardless of the size of the linked list.

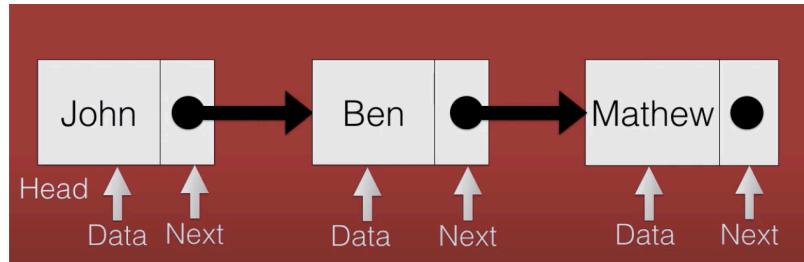
3.4.2 `delete_at()` Method

Now, let's go to the next scenario: deleting a node at a specific position.

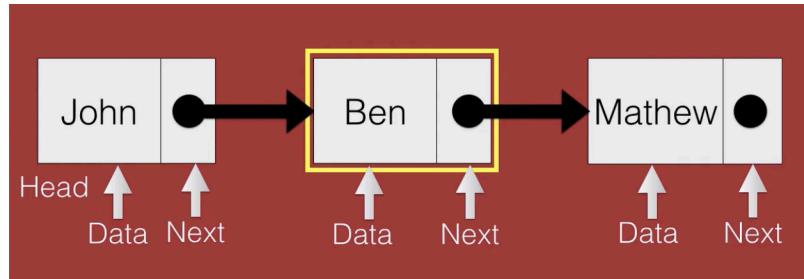
Deleting a node at a specific position requires traversing the list to reach the node just before the desired position and updating its next pointer to skip the node to be deleted.

Let's look at one example:

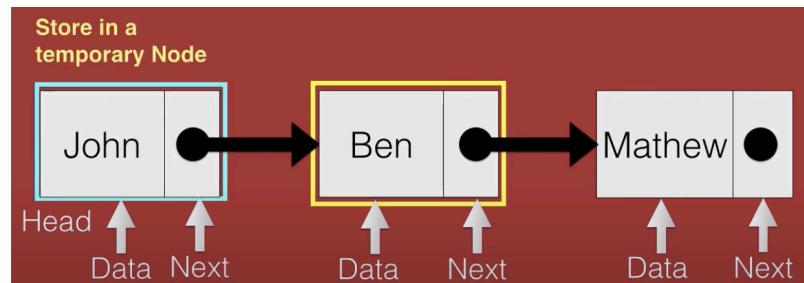
Suppose there is a singly linked list with three nodes: “John”, “Ben”, and “Matthew”. We want to delete the node containing the data “Ben”.



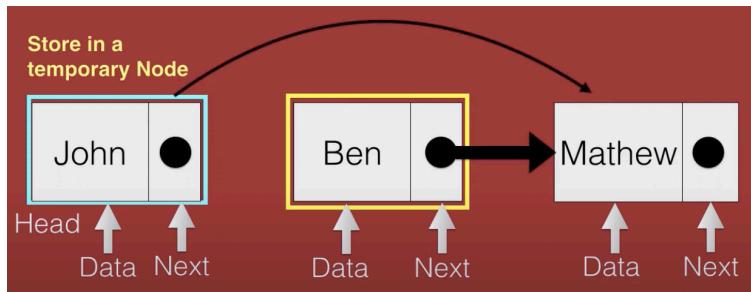
Once “Ben” is deleted, “John” needs to point to “Matthew”. So, we first traverse the list until we find the node before “Ben”, which in our case is “John”. Once we reach “John”, we update the next pointer of “John” to point to the next field pointer of “Ben” (which is “Matthew”).



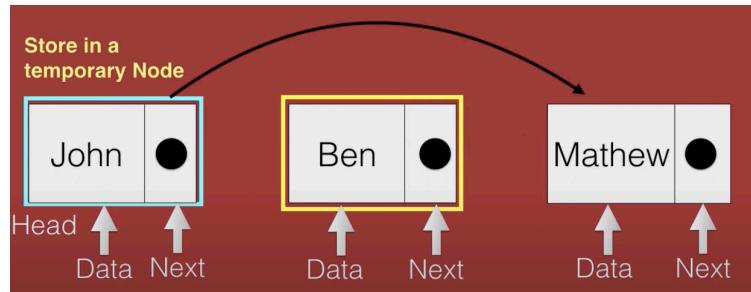
Similar to inserting a node, before we traverse to “Ben”, we need to store the reference to “John” in a temporary node to maintain the connection.



Once we have the reference to “John” stored, we can remove the connection from “John” to “Ben”. Then, using the next field pointer of “Ben”, we can establish a connection from “John” to “Matthew”.



After this connection is established, we can remove the connection from "Ben" to "Matthew" to complete the process.



So in summary, to delete the node at position 1 ("Ben"), we need to follow these steps:

- Traverse to the Target Position - 1:** Start from the head and move to the node just before, in this case, is "Ben".
- Preserve the Previous Node:** Store the reference to the previous node (in this case, position 0 which is "John") in a temporary variable.
- Delete the Target Node:** Remove the node at position 1 containing "Ben".
- Update the Next Pointer:** Make the next pointer of the previous node at position 0 ("John") point to the node after "Ben" at position 2 ("Matthew").

Now, let's write this operation in Python code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
```

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node
    self.size += 1

def delete_at(self, position):
    # Check if the given position is valid
    if position < 0 or position >= self.size:
        return "Invalid position"

    # If the position is 0 (deleting the head)
    if position == 0:
        self.head = self.head.next
        self.size -= 1
    else:
        # Traverse the list to reach the node just before the
        # desired position
        current = self.head
        for _ in range(position - 1):
            current = current.next
        if current is None:
            return "Invalid position"

        # Update the next pointer of the previous node
        current.next = current.next.next
        self.size -= 1

def print_list(self):
    current = self.head
    while current is not None:
        print(current.data, end=" ")
        current = current.next
    print()

# Example usage
linked_list = LinkedList()

linked_list.append("John")

```

```

linked_list.append("Ben")
linked_list.append("Mathew")

print("Original Linked List:")
linked_list.print_list()

linked_list.delete_at(1)

print("Linked List after deleting the node at position 1:")
linked_list.print_list()

```

Output:

Original Linked List:

John Ben Mathew

Linked List after deleting the node at position 1:

John Mathew

Now let's analyze this method's time complexity.

Deleting a node at a specific position involves traversing the list to find the node at the given position and updating the next pointers accordingly.

Let's analyze the steps involved:

1. Checking if the position is valid: **O(1)**
 - a. Comparing the position with 0 and the size of the list takes constant time.
2. Deleting the head node (position 0): **O(1)**
 - a. If the position is 0, updating the head to point to the next node takes constant time.
3. Traversing the list to find the node at the given position: **O(n)**
 - a. To delete a node at a specific position, we need to traverse the list to find the node at that position.
 - b. In the worst case, where the position is near the end of the list, and we need to traverse almost the entire list.
 - c. The traversal requires visiting each node from the head until the node just before the desired position is reached.
 - d. The time complexity of this step is linearly proportional to the position, which can be at most the size of the list minus one.
4. Updating the next pointers: **O(1)**
 - a. Once we have the node just before the node to be deleted, updating its next pointer to skip the node to be deleted takes constant time.
5. Decrementing the size of the list: **O(1)**

- a. Decrementing the size variable takes constant time.

Therefore, the overall time complexity of deleting a node at a specific position is **O(n)** in the worst case, where **n** represents the linear relationship between the number of nodes in the linked list and the time complexity of the operation.

This is because the traversal step dominates the time complexity, and in the worst case, we need to traverse almost the entire list to find the node at the given position.

However, it's important to note that the time complexity can be lower in certain cases:

- If the position is 0 (deleting the head node), the time complexity is **O(1)** since no traversal is required.
- If the position is close to the beginning of the list, the traversal step will be shorter, resulting in a lower time complexity.

Space Complexity:

The space complexity of deleting a node at a specific position is **O(1)** because it only requires a constant amount of extra space to store the temporary variables, regardless of the size of the linked list.

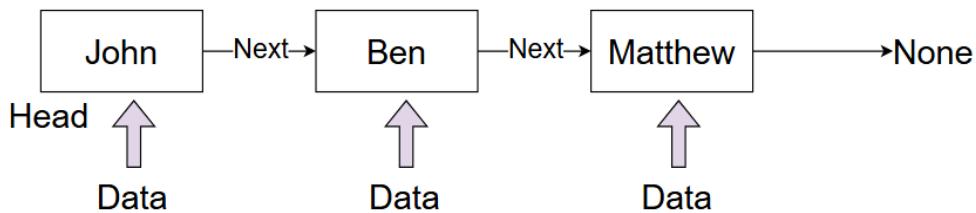
3.4.3 `delete()` Method

Now, let's go to the next scenario: deleting a node with a specific value.

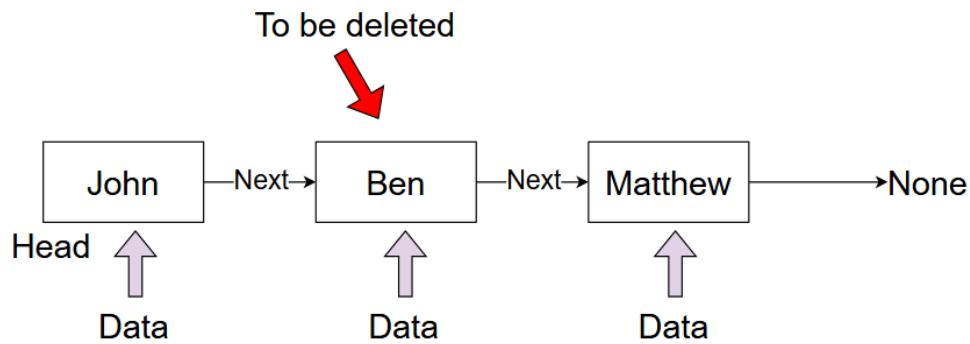
Deleting a node with a specific value requires traversing the list to reach the node just before the node with the desired value and updating its next pointer to skip the node to be deleted.

Let's look at one example:

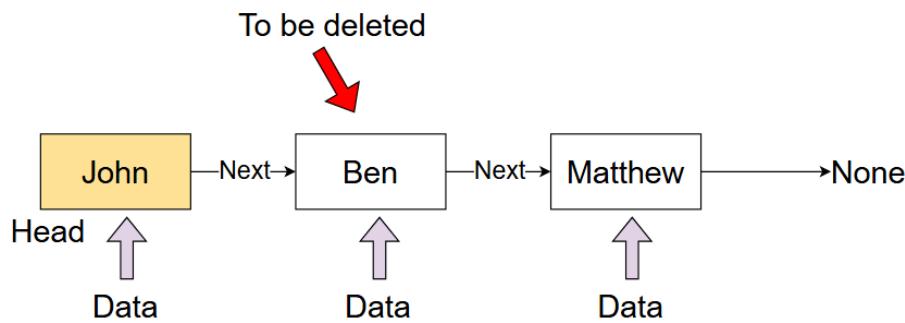
Suppose there is a singly linked list with three nodes: "John", "Ben", and "Matthew". We want to delete the node containing the data "Ben".



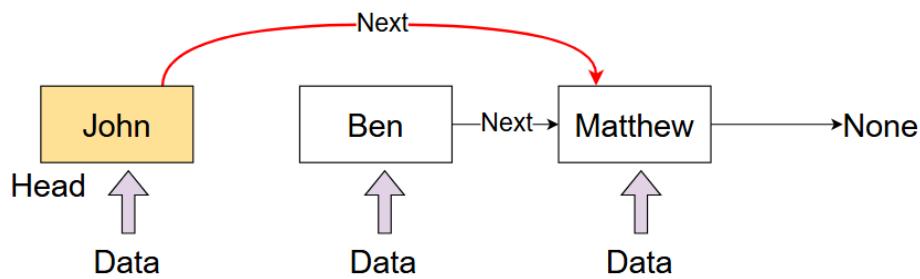
Once "Ben" is deleted, "John" needs to point to "Matthew". So, we first traverse the list until we reach "Ben". Once we reach "Ben", we update the next pointer of "John" to point to the next node after "Ben" (which is "Matthew").



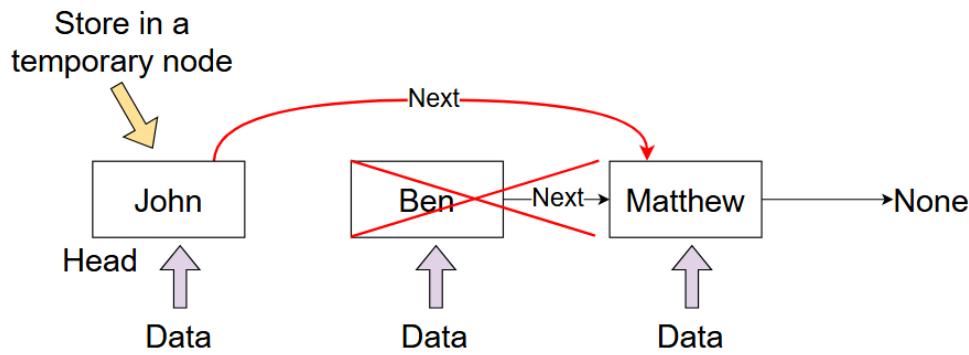
Similar to inserting a node, we traverse to the node where its next node is “Ben”.



Once we find this node, we can simply remove the node “Ben” by assigning the node “Matthew” as the next node of “John” (establish a connection between “John” and “Matthew”).



After this connection is established, we can remove the connection from “Ben” to “Matthew” to complete the process.



So in summary, to delete the node "Ben," we need to follow these steps:

- 1. Traverse to the Target Node:** Start from the head and move to the node where its next node is "Ben".
- 2. Connect the current node with the next node of the desired node (Update the Pointer):** assign the next node of the node "Ben" as the next node of the current node "John". In this case, it would be `current_node.next = current_node.next.next`.
- 3. Delete the Node:** the node "Ben" is now disconnected and removed from the linked list.

Now, let's write this operation in Python code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1
```

```

def delete(self, value):
    # Check if the list is empty
    if self.head is None:
        return
    # Check if the head itself needs to be deleted
    if self.head.data == value:
        self.head = self.head.next
        self.size -= 1
        return
    # Traverse the list to reach the node just before the
    # desired position
    current = self.head
    while current.next is not None:
        if current.next.data == value:
            # Update the next pointer of the previous node
            current.next = current.next.next
            # Update the length of the linked list
            self.size -= 1
            return
        # Update the current node and prepare to proceed to the
        # next node
        current = current.next

    def print_list(self):
        current = self.head
        while current is not None:
            print(current.data, end=" ")
            current = current.next
        print()

# Example usage
linked_list = LinkedList()

linked_list.append("John")
linked_list.append("Ben")
linked_list.append("Mathew")

print("Original Linked List:")
linked_list.print_list()

linked_list.delete("Ben")

print("Linked List after deleting the node 'Ben':")

```

```
linked_list.print_list()
```

Output:

Original Linked List:

John Ben Mathew

Linked List after deleting the node 'Ben':

John Mathew

Now let's analyze this method's time complexity.

Deleting a node with a specific value involves traversing the list to find a node with the given value and updating the next pointers accordingly.

Let's analyze the steps involved:

1. Checking if the list is empty: **O(1)**
 - a. Checking if the list's head is None takes constant time.
2. Deleting the head node: **O(1)**
 - a. Checking if the head's data equals the value to delete takes constant time.
3. Traversing the list to find the node with a specific value: **O(n)**
 - a. To delete a node with a specific value, we need to traverse the list to find the node.
 - b. In the worst case, where the node is near the end of the list, we need to traverse almost the entire list.
 - c. The traversal requires visiting each node from the head until the node just before the desired value is reached.
 - d. The time complexity of this step is linearly proportional to the position, which can be at most the size of the list minus one.
- Updating the next pointers: **O(1)**
 - a. Once we have the node just before the node to be deleted, updating its next pointer to skip the node to be deleted takes constant time.
5. Decrementing the size of the list: **O(1)**
 - a. Decrementing the size variable takes constant time.

Therefore, the overall time complexity of deleting a node with a specific value is **O(n)** in the worst case, where **n** represents the linear relationship between the number of nodes in the linked list and the time complexity of the operation.

This is because the traversal step dominates the time complexity, and in the worst case, we need to traverse almost the entire list to find the node with the given value.

However, it's important to note that the time complexity can be lower in certain cases:

- If the position of the desired node is 0 (deleting the head node), the time complexity is

O(1) since no traversal is required.

- If the position is close to the beginning of the list, the traversal step will be shorter, resulting in a lower time complexity.

Space Complexity:

The space complexity of deleting a node with the given value is **O(1)** because it only requires a constant amount of extra space to store the temporary variables, regardless of the size of the linked list.

Exercises 1 - 5 (MCQ)

Ex 1

What is the purpose of deleting a node? Select all correct answers.

- | |
|---|
| <input checked="" type="checkbox"/> Freeing up space in memory |
| <input type="checkbox"/> Adding the node to a new linked list |
| <input checked="" type="checkbox"/> Removing unwanted data from the linked list |
| <input type="checkbox"/> Preventing data from being used elsewhere |

Ex 2

What does the delete method do?

- | |
|---|
| <input type="checkbox"/> Removes a node from the start of the list |
| <input type="checkbox"/> Replaces the node at the end of the list |
| <input checked="" type="checkbox"/> Removes the first node with matching data |
| <input type="checkbox"/> Removes a node at the specified index |

Ex 3

What does the delete_at method do?

- | |
|---|
| <input type="checkbox"/> Removes a node from the start of the list |
| <input type="checkbox"/> Replaces the node at the end of the list |
| <input type="checkbox"/> Removes the first node with matching data |
| <input checked="" type="checkbox"/> Removes a node at the specified index |

Ex 4

Once a node has been deleted, the size needs to be _____.
Select the correct option to complete the sentence.

Incremented

Decrementated

Printed

Set to None

Ex 5

Once deleted, a node's data can still be retrieved from the list.

True

False

Exercises 6 - 10 (Fill in the Blank)**Ex 6**

Complete the code below to create the delete() method.

```
def delete(self, value):
    # Check if the list is empty
    if self.head is [Fill in answer here]
        return
    # Check if the head itself needs to be deleted
    if self.head.data == [Fill in answer here]
        self.head = self.head.next
        self.size -= 1
        return
    # Traverse the list starting from the head
    current = [Fill in answer here]
    while current.next is not None:
        if current.next.data == value:
            current.next = current.next.next
            self.size -= 1
            return
        # Update the current node
        current = [Fill in answer here]
```

Expected answer:

```
def delete(self, value):
```

```

# Check if the list is empty
if self.head is None:
    return
# Check if the head itself needs to be deleted
if self.head.data == value:
    self.head = self.head.next
    self.size -= 1
    return
# Traverse the list starting from the head
current = self.head
while current.next is not None:
    if current.next.data == value:
        current.next = current.next.next
        self.size -= 1
        return
    # Update the current node
    current = current.next

```

Ex 7

Complete the code below to create the delete_at() method.

```

def delete_at(self, [Fill in answer here]):
    # Check if the given position is valid
    if position < 0 or position >= self.size:
        return

    # If the position is 0 (deleting the head)
    if position == [Fill in answer here]
        self.head = self.head.next
        self.size -= 1
    else:
        # Traverse the list
        current = [Fill in answer here]
        for _ in range(position - 1):
            current = current.next
            if current is None:
                return

    # Update the next pointer of the previous node
    current.next = [Fill in answer here]
    self.size -= 1

```

Expected answer:

```

def delete_at(self, position):
    # Check if the given position is valid

```

```

if position < 0 or position >= self.size:
    return "Invalid position"

# If the position is 0 (deleting the head)
if position == 0:
    self.head = self.head.next
    self.size -= 1
else:
    # Traverse the list
    current = self.head
    for _ in range(position - 1):
        current = current.next
        if current is None:
            return "Invalid position"

    # Update the next pointer of the previous node
    current.next = current.next.next
    self.size -= 1

```

Ex 8

Complete the code below to delete a node storing the number 4.

```

list = LinkedList()
list.append(1)
list.append(3)
list.append(2)
list.append(4)
list.[Fill in answer here]

```

Expected answer:

```

list = LinkedList()
list.append(1)
list.append(3)
list.append(2)
list.append(4)
list.delete(4)

```

Ex 9

Complete the code below to delete the node at index 4.

```

numbers_list.append(7)
numbers_list.[Fill in answer here]

```

Expected answer:

```
numbers_list.append(7)
numbers_list.delete_at(4)
```

Ex 10

Complete the code below to remove all nodes storing "hello" in a linked list called 'greetings' (assume there's a linked list class with all necessary methods).

```
finished = False
while not finished:
    greetings.[Fill in answer here]("hello")
    if [Fill in answer here].index([Fill in answer here]) == -1:
        finished = True
```

Expected answer:

```
finished = False
while not finished:
    greetings.delete("hello")
    if greetings.index("hello") == -1:
        finished = True
```

Explanation: This code will delete all nodes with "hello" in the linked list greetings, using a variable finished to know when to stop (True: stop, false: continue). The variable finished will switch to True when there's no node with "hello" left in the list.

Exercises 11 - 19 (Coding)

Ex 11

Write the delete() method for the LinkedList class. This method should remove the first occurrence of a node containing the specified value from the list.

Hint:

1. Check if the list is empty first and return if so.
2. Check if the head is being deleted. Update the head and decrement the size.
3. Traverse the list from the head until the next node is the target node. Update the current's next pointer and decrement the size.

Given code:

```
def delete(self, value):
    # Your code here
```

Expected input:

```
def delete(self, value):
    if self.head is None:
        return
    if self.head.data == value:
```

```

        self.head = self.head.next
        self.size -= 1
        return
    current = self.head
    while current.next is not None:
        if current.next.data == value:
            current.next = current.next.next
            self.size -= 1
            return
        current = current.next

```

Ex 12

Write the delete_at() method for the linked list. You do not need to return anything if the specified index is out of range or if the list is empty.

Hint:

1. Check if the given position is valid and return if so.
2. Check if the position is 0. Update the head and decrement the size.
3. Traverse the list from the head until one before the node to be deleted. Update the current's next pointer and decrement the size.

Given code:

```
def delete_at(self, position):
    # Your code here
```

Expected input:

```

def delete_at(self, position):
    # Check if the given position is valid
    if position < 0 or position >= self.size:
        return

    # If the position is 0 (deleting the head)
    if position == 0:
        self.head = self.head.next
        self.size -= 1
    else:
        # Traverse the list
        current = self.head
        for _ in range(position - 1):
            current = current.next
            if current is None:
                return

    # Update the next pointer of the previous node
    current.next = current.next.next
    self.size -= 1

```

Ex 13

Given a linked list 'numbers' that contains the following numbers:

- 3
- 2
- 4
- 6
- 3
- 3
- 3

Write one line of code to remove the first occurrence of number 3.

Expected input:

numbers.delete(3)

Explanation: This is more correct than numbers.delete_at(0) because we just want to delete the first occurrence of the number 3, rather than knowing exactly what index to delete.

Ex 14

Given a linked list 'numbers' that contains the following numbers:

- 3
- 2
- 4
- 6
- 3
- 3
- 3

Remove the first number 4 in the 'numbers' Linked List.

Expected input:

numbers.delete(4)

Ex 15

Given a linked list 'numbers' that contains the following numbers:

- 3
- 2
- 4
- 6
- 3
- 3
- 3

Remove the node at index 4 in the numbers linked list.

Expected input:

```
numbers.delete_at(4)
```

Ex 16

Given a linked list 'club_members' that contains the following strings:

- "Phil"
- "George"
- "Mary"
- "Lee"
- "John"
- "Kelvin"
- "Patrick"
- "Jane"

Write one line of code to delete "George" from the list.

Expected input:

```
club_members.delete("George")
```

Ex 17

Given a linked list 'club_members' that contains the following strings:

- "Phil"
- "George"
- "Mary"
- "Lee"
- "John"
- "Kelvin"
- "Patrick"
- "Jane"

Remove the member "Lee" from the 'club_members' Linked List.

Expected input:

```
club_members.delete("Lee")
```

Ex 18

Given a linked list 'club_members' that contains the following strings:

- "Phil"
- "George"
- "Mary"
- "Lee"
- "John"
- "Kelvin"
- "Patrick"
- "Jane"

Remove the 3rd member from the 'club_members' Linked List.

Expected input:

```
club_members.delete_at(2)
```

Ex 19

Given a linked list 'club_members' that contains the following strings:

- "Phil"
- "George"
- "Mary"
- "Lee"
- "John"
- "Kelvin"
- "Patrick"
- "Jane"

Write code to remove every member from the 'club_members' Linked List. You can use a while loop to repeatedly delete the head node until the list is empty.

Expected input:

```
while club_members.size > 0:  
    club_members.delete_at(0)
```

Alternative solution:

```
while club_members.size > 0:  
    club_members.delete_last()
```

While both solutions work, `delete_at(0)` is typically more efficient for singly linked lists because it avoids traversal. `delete_last()` would require traversal to the end of the list for each deletion, making it less efficient ($O(n^2)$ time complexity for clearing the entire list).

3.5 Time Complexity Comparison with Lists

Now that we've seen how methods for linked lists work, it's important to compare them to other data structures so that we understand when it is most effective to use them.

We will compare the time complexity of our linked list functions with equivalent methods from the List data structure.

Adding new values:

Adding to the beginning:

- For Lists: $O(n)$ time complexity. This is because every value is shifted up one index to create space for the new value in the list.
- For Linked Lists: $O(1)$ time complexity. We can simply attach a node into the chain.

Adding to the end:

- For Lists: $O(1)$ time complexity. We simply append it onto the end without needing to shift elements.
- For Linked Lists: $O(n)$ time complexity. This is because we need to traverse the entire chain to reach the end and then insert the node. **However, this can be optimized to $O(1)$ by maintaining a tail pointer that directly references the last node.**

Adding in the middle:

- For Lists: $O(n)$ time complexity. We will on average require $n/2$ shifts, which simplifies to $O(n)$ time.
- For Linked Lists: $O(n)$ time complexity. Likewise, we will on average need to traverse $n/2$ nodes to reach the desired location.

Accessing values:

Accessing by index:

- For Lists: $O(1)$ time complexity. Lists provide random access, allowing for $O(1)$ access times when retrieving data from a particular index.
- For Linked Lists: $O(n)$ time complexity. Nodes in Linked Lists can not be accessed directly like elements in Lists. Linked Lists need to be traversed to get to it.

Accessing by value:

- For Lists: $O(n)$ time complexity. This is because we must traverse the entire list and compare each value in the list with the value we are looking for.
- For Linked Lists: $O(n)$ time complexity, for the same reason as Lists.

Deleting values:

Deleting from the beginning:

- For Lists: $O(n)$ time complexity. We need to shift every single value in the List down one index to fill in the gap created at the start of the List.
- For Linked Lists: $O(1)$ time complexity. We just delete the node, we don't need to shift the other elements.

Deleting from the end:

- For Lists: $O(1)$ time complexity. This is because it doesn't require any shifts.
- For Linked Lists: $O(n)$ time complexity. This is because we need to traverse the entire

chain to reach the end and update the pointers.

Deleting from the middle:

- For Lists: $O(n)$ time complexity. We require an average of $n/2$ shifts. This means that deleting a value from the middle of a List has $O(n)$ time complexity.
- For Linked Lists: $O(n)$ time complexity. We need to traverse an average of $n/2$ nodes, resulting in a time complexity of $O(n)$.

The table below summarizes the time complexities of the various operations we covered:

Functionality	List Time Complexity	Linked List Time Complexity
Adding to the beginning	$O(n)$	$O(1)$
Adding to the end	$O(1)$	$O(n)$ (or $O(1)$ with tail)
Adding in the middle	$O(n)$	$O(n)$
Accessing by index	$O(1)$	$O(n)$
Accessing by value	$O(n)$	$O(n)$
Deleting from the beginning	$O(n)$	$O(1)$
Deleting from the end	$O(1)$	$O(n)$ (or $O(1)$ with tail)
Deleting from the middle	$O(n)$	$O(n)$

As can be seen in the table above, **Linked Lists excel** when we are **frequently adding and deleting data from the start** of the Linked List. On the other hand, **Lists** are much more **preferable** when we want to **frequently add and delete data from the end**, and **access data at specific indices**.

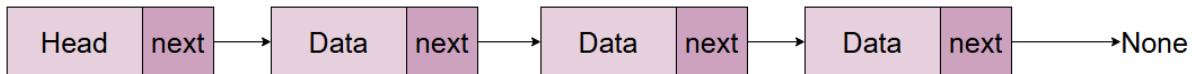
However, Linked Lists can also achieve $O(1)$ complexity for **adding to the end and removing the last node** if they maintain a **tail-referencing variable**. This optimization eliminates the need to traverse the entire list to access the last node, making these operations as efficient as appending to a list.

Recap (Lesson 3)

Singly linked list

Singly linked lists are typically how the idea of linked lists are most commonly explained.

The defining feature of the singly linked list is that the list contains a single direction of travel that typically travels from the head node to the tail node.



Traveling a linked list takes $O(n)$ time to reach the final tail node. This means that many of its methods are within the $O(n)$ range for time complexity.

Singly linked list node

The node of a singly linked list contains its data and the value to the next node.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

The time complexity for creating a node is $O(1)$.

Initializing singly linked lists

All singly linked lists initialize with a `None` head node, and a size of zero.

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0
```

The time complexity for initializing a `LinkedList` is $O(1)$

Prepend

The prepend method adds a node to the start of a list. This is an $O(1)$ method because it happens at the head node.

```
def prepend(self, data):  
    new_node = Node(data)  
    new_node.next = self.head  
    self.head = new_node  
    self.size += 1
```

Append

The append method adds a node to the tail of a list. This is an O(n) method because it happens at the tail node which must be traversed to.

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node
    self.size += 1
```

Insert at

The insert_at method adds a node anywhere into the list. It has a O(n) complexity because the index that may be added could be the tail node, which requires a full list traversal.

```
def insert_at(self, position, data):
    new_node = Node(data)
    if position < 0 or position > self.size:
        return "Invalid position"
    if position == 0:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        for _ in range(position - 1):
            current = current.next
            if current is None:
                return "Invalid position"
        new_node.next = current.next
        current.next = new_node
    self.size += 1
```

Delete at

The delete at method removes a node based on any valid index. This is an O(n) method because the node to delete may be the final one.

```
def delete_at(self, position):
    if position < 0 or position >= self.size:
        return "Invalid position"
    if position == 0:
        self.head = self.head.next
```

```

        self.size -= 1
    else:
        current = self.head
        for _ in range(position - 1):
            current = current.next
            if current is None:
                return "Invalid position"
        current.next = current.next.next
        self.size -= 1

```

You can also have methods that delete the last node, delete by a node's value OR delete the first node.

Getting values

The get method can retrieve a value at a specified index. Again it is an O(n) method because it may be the tail node.

```

def get(self, index):
    if index < 0:
        return -1
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1

```

It is also possible to get the index by the value from traversing in a similar manner.

Complexity summary

A summary of the complexities and comparisons to the typical python list can be found in the following table:

Functionality	List Time Complexity	Linked List Time Complexity
Adding to the beginning	O(n)	O(1)
Adding to the end	O(1)	O(n)
Adding in the middle	O(n)	O(n)
Accessing by index	O(1)	O(n)

Accessing by value	$O(n)$	$O(n)$
Deleting from the beginning	$O(n)$	$O(1)$
Deleting from the end	$O(1)$	$O(n)$
Deleting from the middle	$O(n)$	$O(n)$

4. Doubly Linked List

4.1 Structure

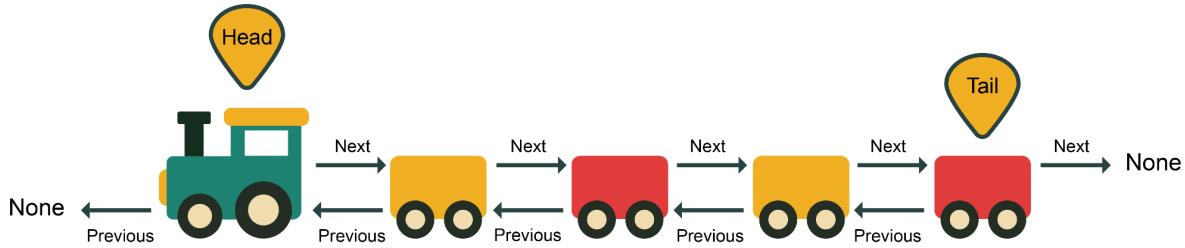
The doubly linked list varies from the standard singly linked list in that each node contains a reference to both the next and the previous nodes.

This allows for **bidirectional traversal** of the list, meaning we can navigate both forward and backward through the list.

A doubly linked list uses **more memory** than a singly linked list because it stores an additional reference to the previous node.

However, this extra reference allows for more efficient traversal and manipulation of the list.

Imagine a train composed of multiple cars connected together. In a doubly linked list, each train car represents a **node**, and the connections between the cars represent the references to the **previous** and **next** nodes.



- **Head Node:** The first node in the list. The **previous** pointer of the head node is **None** because there is no node before it.
- **Tail Node:** The last node in the list. The **next** pointer of the tail node is **None** because there is no node after it.
- **Previous Pointer:** Each node contains a pointer to the previous node in the list, enabling traversal in the backward direction.
- **Next Pointer:** Each node contains a pointer to the next node in the list. This allows traversal in the forward direction.

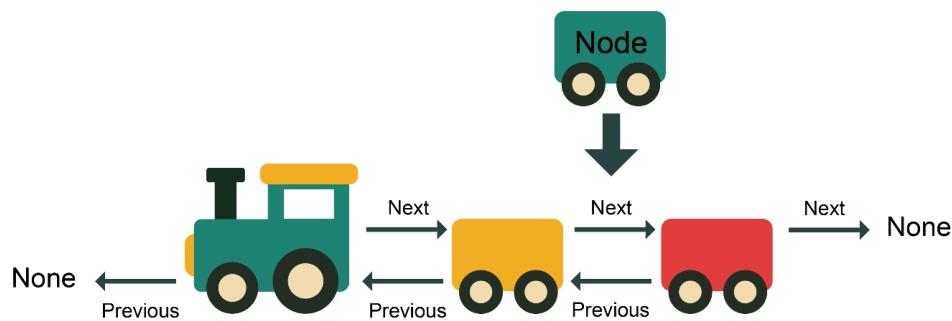
Now, let's say you're the train conductor, and you need to add or remove a car from the middle of the train. With a doubly linked list, it's like having each car equipped with **two walkie-talkies**:



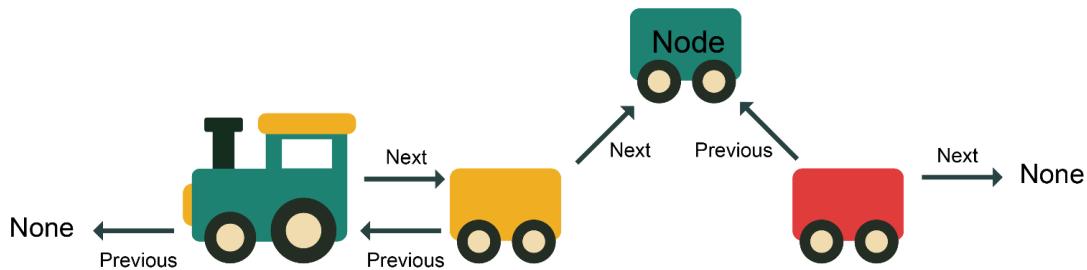
- One to communicate with the car in **front**
- One to communicate with the car **behind**

This two-way communication system allows for efficient updates and traversal.

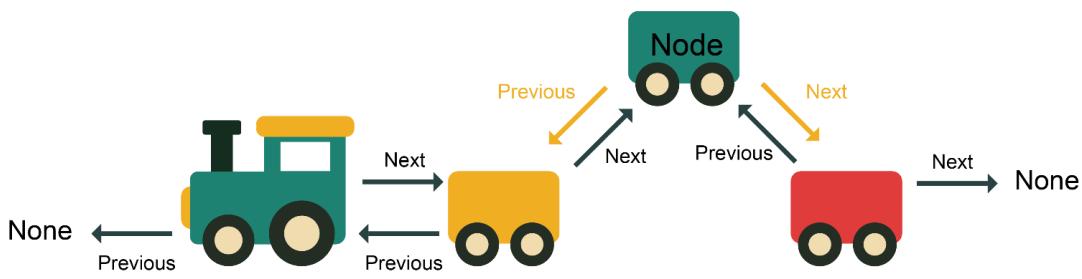
Adding a New Car:



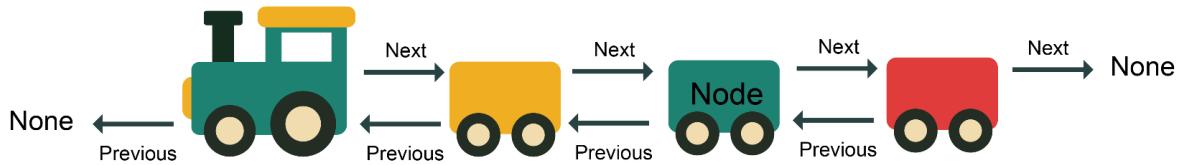
To add a new car, you can easily locate the spot where you want to insert it. You communicate with the cars on either side of that spot and tell them, "Hey, I'm adding a new car between you two."



The cars then update their walkie-talkies to establish a connection with the new car.

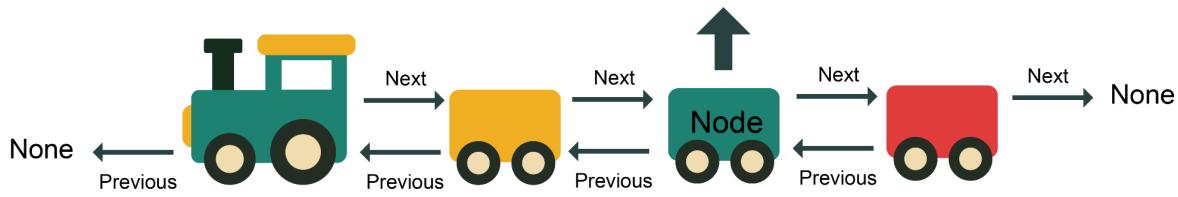


The new car sets its walkie-talkies to connect with the cars in front and behind it.

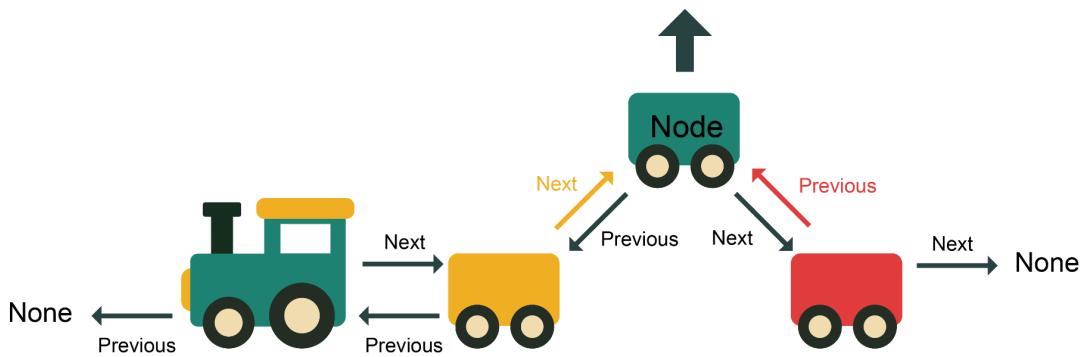


This way, the train remains connected, and you can efficiently add the new car without disrupting the entire train.

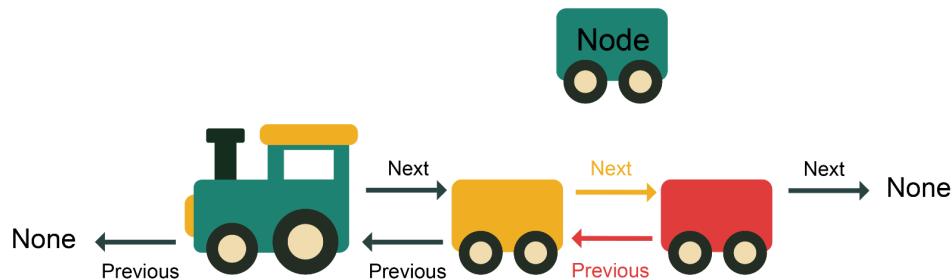
Removing a Car:



Similarly, if you need to remove a car from the middle of the train, you can communicate with the cars on either side of the one you want to remove.



You tell them, "I'm removing the car between you two. Please connect your walkie-talkies directly to each other."



The cars then update their walkie-talkies to bypass the removed car, maintaining the connection of the train.

Comparison with Singly Linked List:

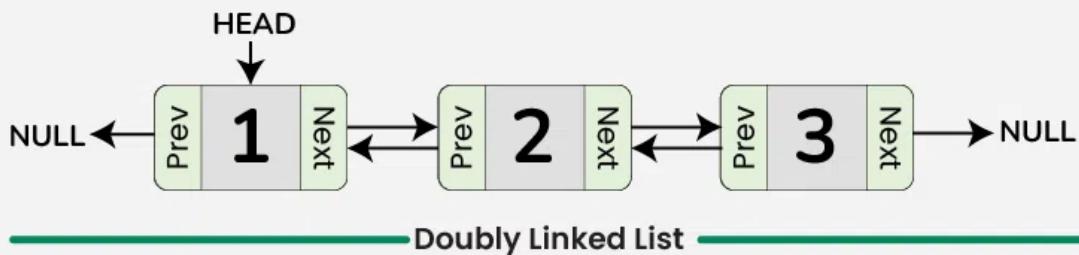
In contrast, if the train had a singly linked list structure, it would be like each car only having a walkie-talkie that could communicate with the car in front of it.

To add or remove a car in the middle, you would need to start from the engine and communicate with each car one by one until you reach the desired spot. This process would be more time-consuming and less efficient.

So, while a doubly linked list requires more resources (like having two walkie-talkies per car), it provides faster and more flexible traversal of the list, especially when adding or removing nodes in the middle as it is more straightforward.

It's like having a **more advanced communication system** that allows for **quick and efficient** changes to the train's structure.

Examples of Double Linked List:



Advantages

- **Bidirectional Traversal:** Nodes can be easily accessed in both directions, making certain operations (e.g., reversing the list) more efficient.
- **Efficient Insertions/Deletions:** Adding or removing nodes in the middle of the list is faster because both prev and next pointers are available.
- **Enhanced Flexibility:** Useful for scenarios requiring frequent updates to the list, such as undo operations or managing caches.

Disadvantages

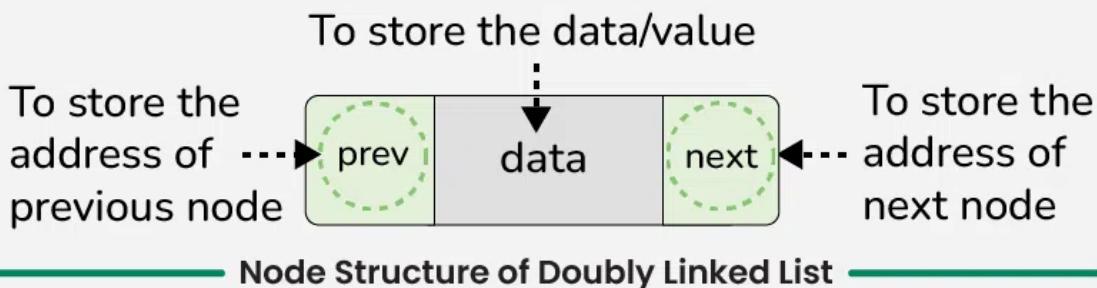
- **Higher Memory Usage:** Each node stores an additional pointer, increasing the memory overhead.
- **Increased Complexity:** Maintaining both prev and next pointers requires more careful implementation, increasing the chance of bugs.
- **Performance Overhead:** Slightly slower than singly linked lists due to the need for maintaining two pointers during insertions and deletions.

We can summarize the comparison as follows:

Feature	Singly Linked List	Doubly Linked List
Memory Usage	Lower (one pointer per node)	Higher (two pointers per node)
Traversal	Only forward traversal	Forward and backward traversal
Deletion	After a given node	Before and after a given node
Insertion	After a given node	Before and after a given node
Implementation	Simpler	More complex
Performance	Generally faster due to less overhead	Slightly slower due to extra pointer maintenance

Use cases	Suitable for simpler operations, and/or when memory need to be conserved	Suitable for operations requiring bidirectional traversal and flexible insertions/deletions
-----------	--	---

When initializing a doubly linked list, one significant difference compared to a singly linked list lies in the creation of the node class, which includes an additional pointer for the previous node.



Node Class in Doubly-Linked List:

In addition to initializing a 'next' variable to store the reference to the next node, a '**prev**' variable also needs to be initialized to store the reference to the previous node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

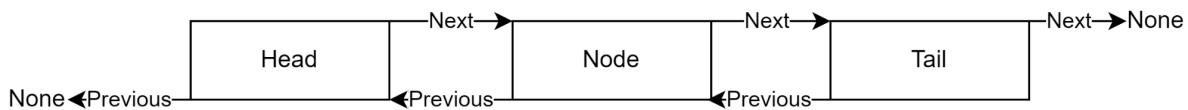
The **prev** reference of the first node (head) and the next reference of the last node (tail) typically point to **None**, indicating the beginning and end of the list, respectively.

DoublyLinkedList class:

For the DoublyLinkedList class, an additional variable '**tail**' is needed.

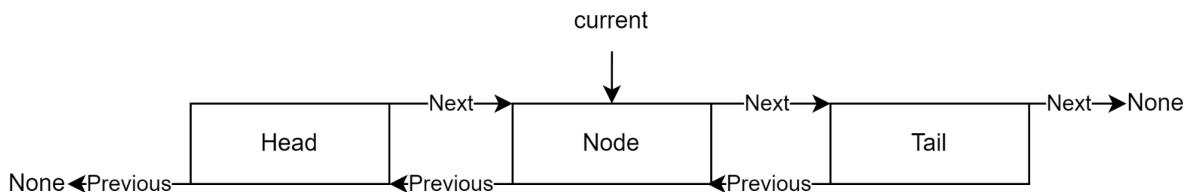
```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None # Reference to the last node
        self.size = 0
```

This variable references the last element of the list, in the same way the head element references the first element. This provides efficient access to both ends of the list.



The utility of the doubly linked list comes mostly from its ability to implement custom methods using its bidirectional links.

This is achieved by introducing a '**current**' node pointer within the DoublyLinkedList class, which tracks the node being accessed by the user.



Methods are created to access the data of this node and iterate through the list by updating the reference of current to the next or previous node as needed.

This is defined in the **DoublyLinkedList** class.

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.current = None
```

Exercises 1 - 10 (MCQ)

Ex 1

What are the benefits of using a doubly linked list? Select all correct answers.

Better traversal

The size of the list can be bigger

Easier to write more efficient methods

- The nodes can store double the data

Explanation:

Better traversal: A doubly linked list allows bidirectional traversal, enabling navigation in both forward and backward directions.

Easier to write more efficient methods: With two pointers (`next` and `prev`), many operations such as insertion and deletion at arbitrary positions are more efficient.

Ex 2

A doubly linked list's nodes are linked to?

- Only the previous node in the list
- Only the next node in the list
- Both the node before and after them in the list
- Both the node after and the head node

Explanation:

Both the node before and after them in the list: Each node in a doubly linked list contains references to both its predecessor and successor, enabling bidirectional navigation.

Ex 3

A doubly linked list can be traversed _____.

Select the correct option to finish the sentence.

- Only back to front
- Only front to back
- Both back to front and front to back
- Not at all

Explanation:

Both back to front and front to back: Due to `prev` and `next` pointers, traversal is possible in either direction.

Ex 4

What extra variable is stored in a doubly linked list class above to keep track of the end of the list?

tail

size

isEmpty

depth

Explanation: **tail:** The `tail` reference keeps track of the last node in the list for efficient operations at the end.

Ex 5

What extra variable stored in the Node class allows backwards traversal?

size

weight

tail

prev

Ex 6

What extra variables could be stored in the Linked List class for a doubly linked list that can improve traversal? Select all correct options.

tail

size

current

depth

Explanation:

tail: Efficiently tracks the last node for backward traversal.

current: Tracks the current node being accessed, simplifying dynamic traversal.

Ex 7

The current node reference is used to traverse the list by _____.

Select the correct option to complete the sentence.

- Setting it to the last node accessed
 - Setting to the node at the middle of the list
 - It can only be updated by adding a new node
 - Setting it to the next or previous node in relation to the node that is referenced by current

Explanation: Setting it to the next or previous node in relation to the node that is referenced by current: Traversal is achieved by updating the `current` reference based on `next` or `prev`.

Ex 8

The doubly linked list is more efficient than a singly linked list _____.
Select the correct option to complete this sentence.

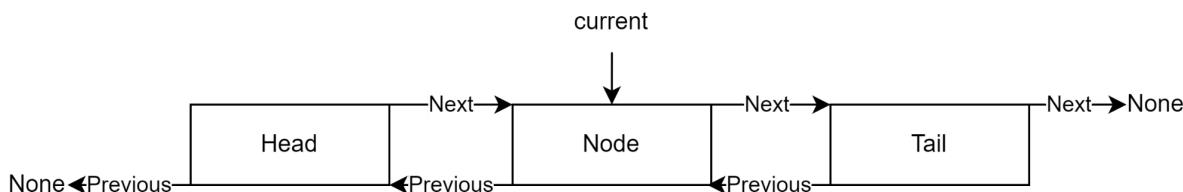
- In all scenarios
 - In some scenarios
 - In no scenarios
 - When the size is under 300

Explanation:

Since a node in a doubly linked list has 2 pointers (one that points to the 'next' and another that points to the 'previous'), applications such as reversal of the list, forward and backward traversal, and insertions and deletions at both ends can be more efficient in comparison to the singly linked list. However, in cases where there are **memory restrictions**, a singly linked list is more efficient since there is no need for another pointer to the previous node.

Ex 9

In the below diagram, the current node is _____.
Select the correct option to complete this sentence.



A reference to the node linking to the Head and Tail

A reference to the Head node

A reference to the Tail node

The data stored in the node at index 1

Explanation: **A reference to the node linking to the Head and Tail:** The `current` node is a dynamic reference that can point to any node, typically the one being accessed.

Ex 10

Most of the doubly linked list's inefficiency compared to a singly linked list is due to _____. Select the correct option to finish the sentence.

The longer class name of DoublyLinkedList

The tail node reference

Its implementation of the current node

Increased memory usage

Explanation:

Increased memory usage: The additional `prev` pointer in each node and maintenance of both `head` and `tail` references increase memory requirements.

Exercises 11 - 20 (Fill in the Blank)

Ex 11

Complete the code below to create the node for a doubly linked list.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        # Your code here
```

Expected answer:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

```
self.prev = None
```

Ex 12

Complete the code below to create the *DoublyLinkedList* class.

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        # Your code here  
        self.size = 0  
        # Your code here
```

Expected answer:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.current = None
```

Ex 13

Complete the code below to print the data of the node before *current* in the list.

```
print([Fill in answer here])
```

Expected answer:

```
print(current.prev.data)
```

Explanation: The *prev* pointer in a doubly linked list allows access to the node before the current one. By using *current.prev.data*, you directly retrieve the data stored in the previous node.

Ex 14

Complete the code below to print the data of the last node in the list *list*.

```
print([Fill in answer here])
```

Expected answer:

```
print(list.tail.data)
```

Explanation: The *tail* pointer in a doubly linked list tracks the last node, making it easy to

access its data using `list.tail.data`

Ex 15

Complete the code below to print the sum of 5 and the data of the node before `current` in the list.

```
print([Fill in answer here])
```

Expected answer:

```
print(current.prev.data + 5)
```

Explanation: Using the `prev` pointer, you access the data of the node before the current one (`current.prev.data`) and add 5 to it before printing the result.

Ex 16

Complete the code below to print the data in the current node of the list `members`.

```
print([Fill in answer here])
```

Expected answer:

```
print(members.current.data)
```

Explanation: The `current` pointer refers to the node currently in focus. Using `members.current.data`, you access and print its data.

Ex 17

Complete the code below to set the data of the current node in the list `letters` to "H".

```
letters[Fill in answer here]
```

Expected answer:

```
letters.current.data = "H"
```

Explanation: By assigning a value to `current.data` (e.g., `letters.current.data = "H"`), you modify the data stored in the node currently referenced by `current`.

Ex 18

Complete the code below to set the current node to be the head node.

```
list[Fill in answer here] = list[Fill in answer here]
```

Expected answer:

```
list.current = list.head
```

Explanation: To reset the traversal to the beginning of the list, set current to the head pointer: list.current = list.head

Ex 19

Complete the code below to set the current node to be the tail node.

```
list[Fill in answer here] = list[Fill in answer here]
```

Expected answer:

```
list.current = list.tail
```

Ex 20

Complete the code below to insert a new node with the data "NewMember" after the current node in the list.

```
# Create a new node with the data "NewMember"
new_node = Node("NewMember")

# Adjust the pointers to insert the new node after the current node
new_node.next = [Fill in answer here]
[Fill in answer here] = current
current.next = [Fill in answer here]
if new_node.next:
    [Fill in answer here] = new_node
```

Expected answer:

```
# Create a new node with the data "NewMember"
new_node = Node("NewMember")

# Adjust the pointers to insert the new node after the current node
# Link new node's next to current's next node
new_node.next = current.next
# Set new node's previous link to current node
new_node.prev = current
# Connect current node to new node
current.next = new_node
# Update next node's previous link if it exists
if new_node.next:
    new_node.next.prev = new_node
```

Ex 21

Complete the code to reverse a doubly Linked List

Given code

```
def reverse(self):
    prev = None
    current = self.head
    while current is not None:
        next = [Fill in answer here]
        current.next = prev
        current.prev = next
        prev = [Fill in answer here]
        current = next
    self.head = prev
```

Expected input

```
def reverse(self):
    # Initialize prev as None (will be the new tail of the list)
    prev = None
    # Start with the head of the list
    current = self.head
    # Traverse through the list
    while current is not None:
        # Store the next node before modifying links
        next = current.next
        # Swap the next and previous pointers
        current.next = prev
        current.prev = next
        # Move prev and current forward for the next iteration
        prev = current
        current = next
    # Update the head to be the last node (which is now the first)
    self.head = prev
```

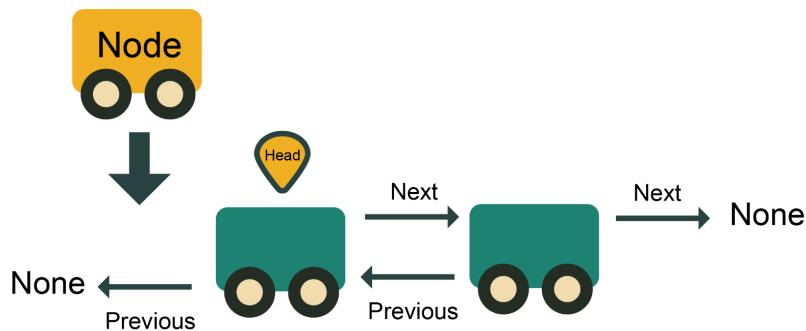
4.2 Standard Methods

As there are several new pointers such as 'current', 'tail', and 'prev' in the Node and List classes, the standard methods require modification to ensure that the doubly linked list will function as required.

The **prepend()**, **append()**, and **insert_at()** methods ensure that the 'tail' reference is updated whenever a new node is added to the end of the linked list.

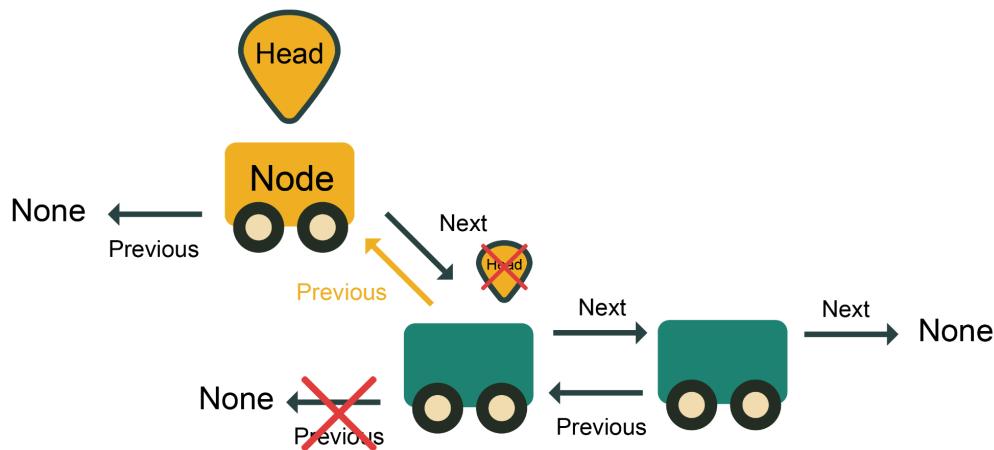
The 'prev' pointer is updated for both the node that is added and also the node that comes after the newly added node. The 'next' pointers for both the newly added node and the node that comes before must be updated similarly (same as with singly linked lists).

prepend() Method



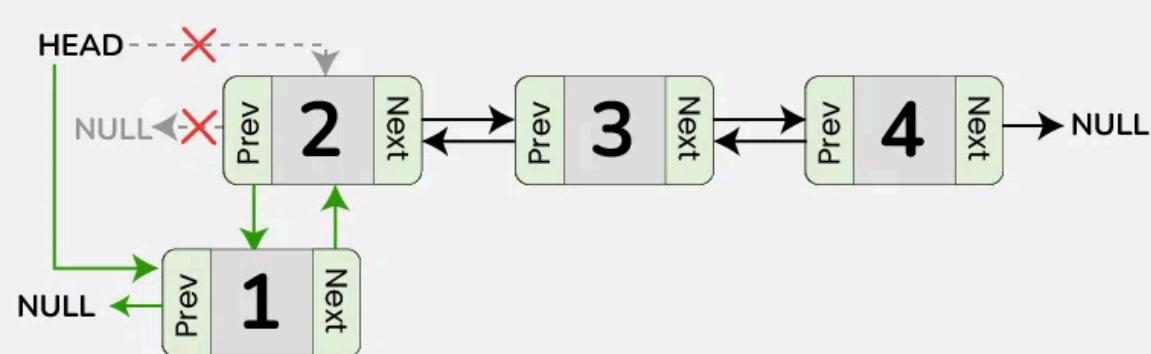
The **prepend** method adds a new node at the **beginning** of the list.

- If the list is empty, the new node becomes both the head and the tail because it is the only node in the list.
- Otherwise, the method:
 - updates the head
 - ensures the new head's **next** pointer points to the old head
 - sets the old head's **prev** to this new node



- In either case, the size is incremented.

Another example of prepend() method:



```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:

```

```

# If empty, the new node becomes both the head and the tail
    self.head = new_node
    self.tail = new_node
else:
    # Otherwise, link the new node to the current head
    new_node.next = self.head
    # Set the previous reference of the current head to the
    # new node
    self.head.prev = new_node
    # Update the head to the new node
    self.head = new_node
# Increment the size of the list
self.size += 1

def print_list(self):
    current = self.head
    while current:
        print(current.data, end=" <-> ")
        current = current.next
    print("None")

```

Example usage

```

dll = DoublyLinkedList()

print("Original Doubly Linked List:")
dll.print_list()

dll.prepend("Alice")
dll.prepend("Ken")
dll.prepend("Vanessa")

print("Doubly Linked List after prepending 'Alice', Ken, Vanessa':")
dll.print_list()

```

Output:

Original Doubly Linked List:

None

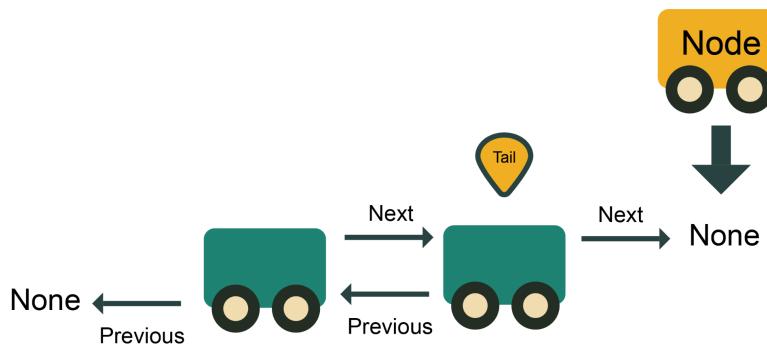
Doubly Linked List after prepending 'Alice':

Vanessa <-> Ken <-> Alice <-> None

append() Method

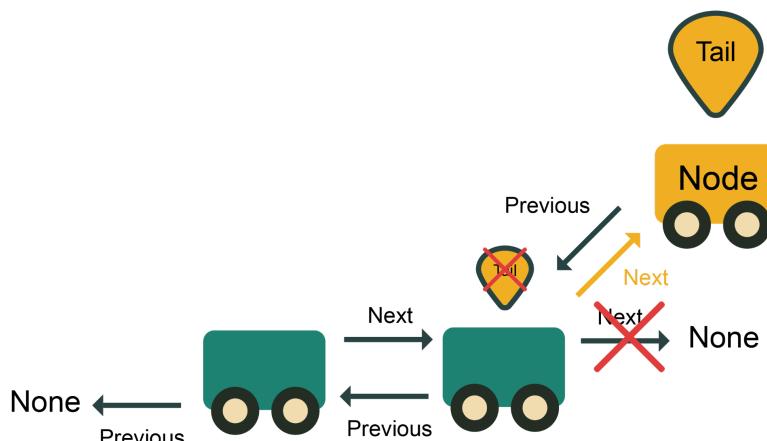
In a singly linked list, each node only references the next node, with no direct reference to the tail. To find the last node, you must start from the head and traverse the entire list using a while loop, making appending nodes less efficient.

In contrast, a doubly linked list has nodes that reference both the next and previous nodes, allowing **direct access** to both the **beginning** and **end of the list without needing to traverse it entirely**.



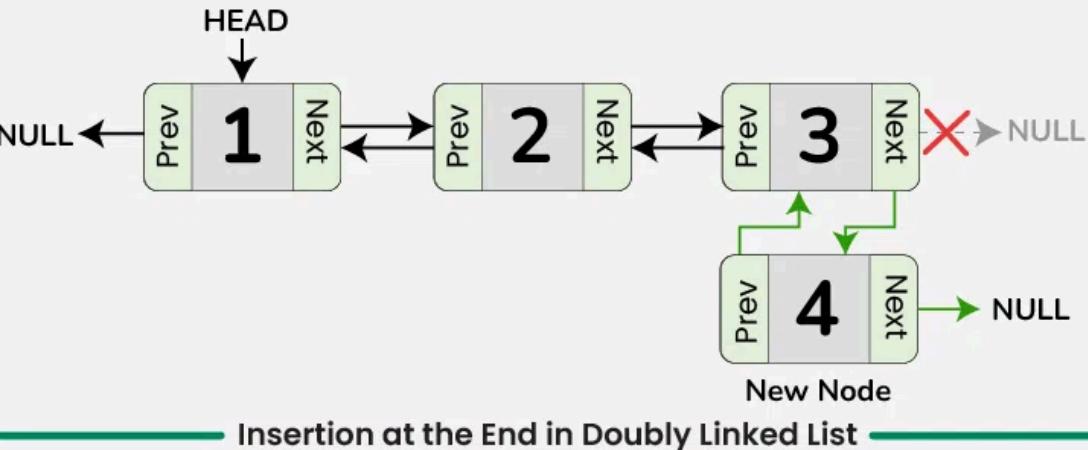
The append method adds a new node to the end of the list.

- If the list is **empty**, the new node becomes both head and tail.
- Otherwise, the method:
 - updates the **last node's next** pointer to point to the new node
 - sets the new node's **prev** pointer to the last node
 - updates the **tail** to be the new node.



- In either case, the size is incremented.

Another example of append() method:



```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            # If the list is empty, the new node becomes both the
            # head and the tail
            self.head = new_node
            self.tail = new_node
        else: # Otherwise, link the new node to the current tail
            # Set the previous pointer of the new node to the current
            # tail
            new_node.prev = self.tail
            # Set the next pointer of the current tail to the new
            # node
            self.tail.next = new_node
            # Update the tail to the new node
            self.tail = new_node
```

```

# Increment the size of the list
self.size += 1

def print_list(self):
    current = self.head
    while current:
        print(current.data, end=" <-> ")
        current = current.next
    print("None")

```

Example usage

```

dll = DoublyLinkedList()
dll.append("John")
dll.append("Ben")
dll.append("Matthew")

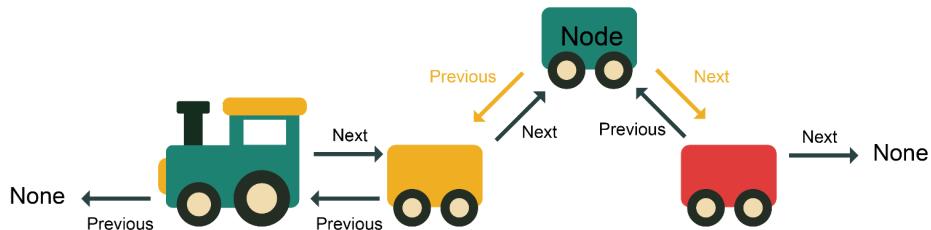
print("Doubly Linked List:")
dll.print_list()

```

Output:

Doubly Linked List:
John <-> Ben <-> Matthew <-> None

insert_at() Method



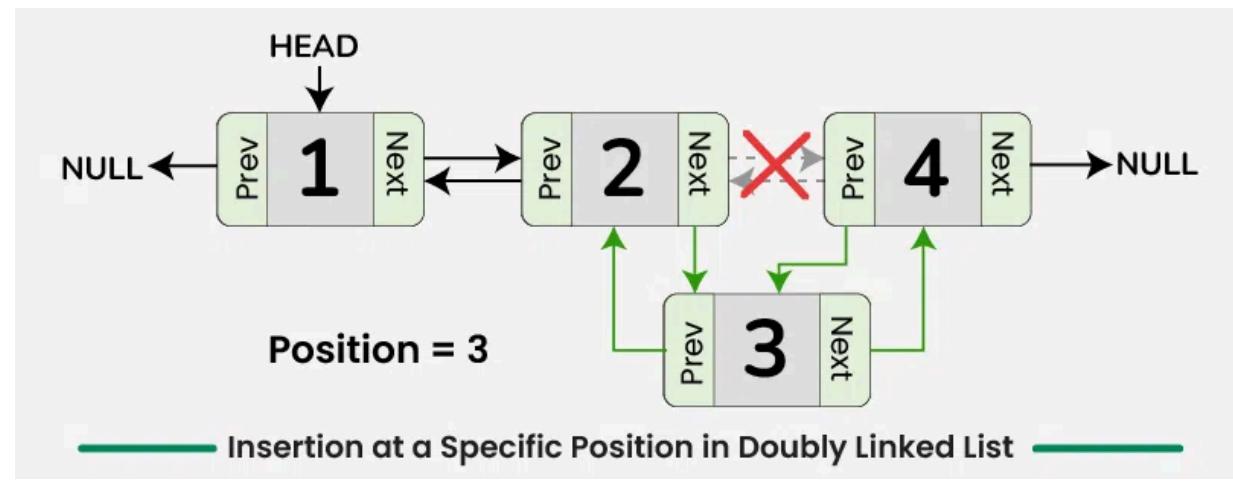
This method inserts a new node with the specified data at the given position in the list. It handles three scenarios:

- **Inserting at the Beginning:** If the position is 0, it is handled by the prepend method.
- **Inserting at the End:** If the position is equal to the size, it is handled by the append method.

- **Inserting in the Middle:** Adjusts the **prev** and **next pointers** of the neighboring nodes to insert the new node at the specified position.

In all cases, the size of the list is incremented after insertion. We also handle the cases where the position is invalid, which is if it is less than 0 or greater than the size.

Another example:



```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.size += 1

    def append(self, data):

```

```

new_node = Node(data)
if self.head is None:
    self.head = self.tail = new_node
else:
    new_node.prev = self.tail
    self.tail.next = new_node
    self.tail = new_node
self.size += 1

def insert_at(self, data, position):
    if position < 0 or position > self.size:
        print("Position out of bounds")
        return

    new_node = Node(data)

    # Handle inserting at the beginning
    if position == 0:
        self.prepend(data)
        return

    # Handle inserting at the end
    if position == self.size:
        self.append(data)
        return

    current = self.head
    count = 0

    # Traverse to the desired position
    while current and count < position:
        current = current.next
        count += 1

    # Handle inserting in the middle
    new_node.prev = current.prev
    new_node.next = current
    if current.prev:
        current.prev.next = new_node
    current.prev = new_node
    self.size += 1

def print_list(self):
    current = self.head

```

```

while current:
    print(current.data, end=" <-> ")
    current = current.next
print("None")

```

Example usage

```

dll = DoublyLinkedList()
dll.append("John")
dll.append("Ben")
dll.append("Matthew")

print("Original Doubly Linked List:")
dll.print_list()

dll.insert_at("Alice", 2)

print("Doubly Linked List after inserting 'Alice' at position 2:")
dll.print_list()

```

Output:

```

Original Doubly Linked List:
John <-> Ben <-> Matthew <-> None
Doubly Linked List after inserting 'Alice' at position 2:
John <-> Ben <-> Alice <-> Matthew <-> None

```

get() Method

The get() method that we will use works exactly the same as it does for singly linked lists. We first check if the index is valid, and then move the current node through the list until we get to the index and return the data. You could optimize it by checking whether the index is in the left or right half of the list and then deciding whether to traverse from head to tail or tail to head.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

```

```

def get(self, index):
    if index < 0 or index >= self.size:
        return -1

    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1

```

index() Method

The index() method that we will use also works the same as it does for singly linked lists, by moving the current node through the list until we get to the data and return the index.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

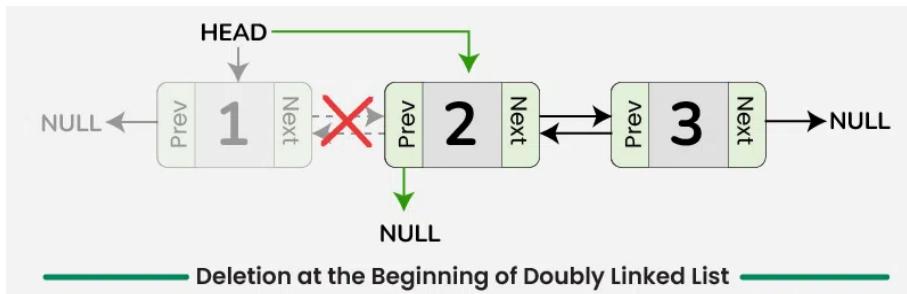
    def index(self, data):
        cur_node = self.head
        count = 0
        while cur_node: # Run to the end of list (None)
            if cur_node.data == data:# If data matches
                return count
            cur_node = cur_node.next # Next node
            count += 1
        return -1 # Not found index

```

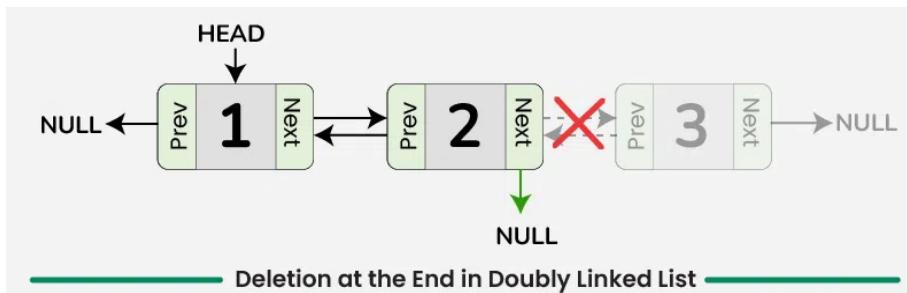
delete() Method

The delete() method has four scenarios where a node can be deleted based on its value:

- **Case 1:** Deleting the only node in the list. If the head, tail, and current node are all equal, then we set the head and tail to None.
- **Case 2:** Deleting the head node. If the node being deleted is the head, then we move the head to the next and remove the previous pointer of the current head.



- **Case 3:** Deleting the tail node. If the node being deleted is the tail, then we move the tail back and remove the next pointer of the current tail.



- **Case 4:** Deleting a middle node. Once we traverse to the node to delete, we move the next pointer of the previous node and previous pointer of the next node to each other.



In all four cases we decrement the size of the Doubly Linked List.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

```

        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def delete(self, data):
        cur_node = self.head
        while cur_node:
            if cur_node.data == data:
# Case 1: The list contains only one node (current node is both
head and tail) -> update head and tail to None
                if cur_node == self.head and cur_node == self.tail:
                    self.head = None
                    self.tail = None
                    self.size -= 1
                    return
# Case 2: The node to delete is the head of the list
                if cur_node == self.head:
                    self.head = cur_node.next
                    self.head.prev = None
                    self.size -= 1
                    return
# Case 3: The node to delete is the tail of the list
                if cur_node == self.tail:
                    self.tail = cur_node.prev
                    self.tail.next = None
                    self.size -= 1
                    return
# Case 4: The node to delete is in the middle of the list
                cur_node.prev.next = cur_node.next
                cur_node.next.prev = cur_node.prev
                self.size -= 1
                return
            cur_node = cur_node.next

```

delete_at() Method

The `delete_at()` method has four scenarios where a node can be deleted based on its index:

- Case 1: Deleting the only node in the list. If the head and tail are equal, and the index to delete is 0, then we set the head and tail to None.
- Case 2: Deleting the head node. If the node being deleted is at index 0, then we move the head to the next and remove the previous pointer of the current head.
- Case 3: Deleting the tail node. If the node being deleted is the tail, then we move the tail back and remove the next pointer of the current tail.
- Case 4: Deleting a middle node. Once we traverse to the node to delete, we move the next pointer of the previous node and previous pointer of the next node to each other.

In all four cases we decrement the size of the Doubly Linked List.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def delete_at(self, index):
        # Position is not valid
        if index < 0 or index >= self.size:
            print("Position out of bounds")
            return

        # Deleting the head node
        if index == 0:
            # Only one node in the list
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                self.head = self.head.next
                self.head.prev = None
            self.size -= 1
            return

        cur_node = self.head
        count = 0

        while cur_node:

```

```

if count == index:
    # Deleting the tail node
    if cur_node == self.tail:
        self.tail = cur_node.prev
        self.tail.next = None
        self.size -= 1
        return

    # Deleting a middle node
    cur_node.prev.next = cur_node.next
    cur_node.next.prev = cur_node.prev
    self.size -= 1
    return

cur_node = cur_node.next
count += 1

```

Summary:

Operation	Time Complexity	Space Complexity	Explanation
prepend()	O(1)	O(1)	Prepending requires updating head and possibly tail, both done in constant time.
append()	O(1)	O(1)	Appending uses the tail pointer to directly add at the end in constant time.
insert_at()	O(n)	O(1)	Insertion requires traversing the list to the specified position, which takes linear time.
get()	O(n)	O(1)	Fetching a value requires traversing the list to the index, which takes linear time.
index()	O(n)	O(1)	Finding the index of a value requires traversing the list, which takes linear time.
delete()	O(n)	O(1)	Deleting requires finding the node first, which involves traversal, making it linear.
delete_at()	O(n)	O(1)	Deleting at a specific index involves traversing the list to the index, which takes linear time.

The doubly linked list is a versatile data structure that excels in scenarios requiring frequent updates at both ends of the list or scenarios involving bidirectional traversal. It is particularly advantageous for applications where quick insertions and deletions are prioritized over random access. However, its higher memory usage and linear traversal for many operations make it less suitable for large-scale datasets where memory efficiency or faster lookups are

crucial.

While doubly linked lists strike a balance between flexibility and performance, their effectiveness depends on the specific use case, often trading off simplicity and speed for adaptability and enhanced functionality.

Exercises 1 - 7 (Fill in the Blank)

Exercise 1

Since we have added a few new variables for our Node and DoublyLinkedList class. The method should be modified as well, to fit the new structure. Fill in the blanks to complete the append() method.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            _____ # set the tail to the new node  
        else:  
            _____ # set the new node's prev to the tail  
            _____ # set the tail's next to the new node  
            self.tail = new_node  
        self.size += 1
```

Expected Answer:

```
def append(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node
```

```

        self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1

```

Exercise 2

Fill in the blanks to complete the prepend() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            _____ # if not, make the head the new node
            _____ # and the tail the new node
        else:
            new_node.next = self.head
            _____ # make the head point to new node
            self.head = new_node
        self.size += 1

```

Hint: The "prev" variable of the old head node can not be None anymore.

Expected Answer:

```

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node

```

```

else:
    new_node.next = self.head
    self.head.prev = new_node
    self.head = new_node
    self.size += 1

```

Exercise 3

Fill in the blanks to complete the `insert_at()` method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def insert_at(self, data, position):
        if position < 0 or position > self.size:
            print("Position out of bounds")
            return
        new_node = Node(data)
        if position == 0:
            _____ # prepend the new node's data
            return
        if position == self.size:
            _____ # append the new node's data
            return
        current = self.head
        count = 0
        while current and count < position:
            _____ # move current to next node
            count += 1
            _____ # adjust the new node's prev pointer
            _____ # adjust the new node's next pointer
        if current.prev:
            current.prev.next = new_node
        current.prev = new_node
        self.size += 1

```

Expected Answer:

```
def insert_at(self, data, position):
    if position < 0 or position > self.size:
        print("Position out of bounds")
        return
    new_node = Node(data)
    if position == 0:
        self.prepend(data)
        return
    if position == self.size:
        self.append(data)
        return
    current = self.head
    count = 0
    while current and count < position:
        current = current.next
        count += 1
    new_node.prev = current.prev
    new_node.next = current
    if current.prev:
        current.prev.next = new_node
    current.prev = new_node
    self.size += 1
```

Exercise 4

Create the get() method.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def get(self, index):
        if index < 0 or index >= self.size:
            return -1
```

```

cur_node = self.head
count = 0
while cur_node:
    if count == index:
        _____ # return the node's data
        _____ # move the current node to next node
        _____ # increment count
    return -1

```

Expected Answer:

```

def get(self, index):
    if index < 0 or index >= self.size:
        return -1

    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1

```

Exercise 5

Create the index() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def index(self, data):
        cur_node = self.head
        count = 0

```

```

while cur_node:
    if cur_node.data == data:
        _____ # return the count
        _____ # move the current node to the next
        _____ # increment the count
return -1

```

Expected Answer:

```

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
return -1

```

Exercise 6

Fill in the blanks to create the delete() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def delete(self, data):
        cur_node = self.head
        while cur_node:
            if cur_node.data == data:
                if cur_node == self.head and cur_node == self.tail:
                    _____ # set the head to None
                    _____ # set the tail to None
                self.size -= 1
            return

```

```

        if cur_node == self.head:
            _____ # set the head to the next
node
            self.head.prev = None
            self.size -= 1
            return
        if cur_node == self.tail:
            _____ # set the tail to the prev
node
            self.tail.next = None
            self.size -= 1
            return
            _____ # adjust previous node's pointer
            _____ # adjust next node's pointer
            self.size -= 1
            return
            _____ # move the current node

```

Expected Answer:

```

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head and cur_node == self.tail:
                self.head = None
                self.tail = None
                self.size -= 1
                return
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = None
                self.size -= 1
                return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next

```

Exercise 7

Fill in the blanks to create the `delete_at()` method.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def delete_at(self, index):
        if index < 0 or index >= self.size:
            print("Position out of bounds")
            return
        if index == 0:
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                _____ # update the head
                self.head.prev = None
                self.size -= 1
            return
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                if cur_node == self.tail:
                    _____ # update the tail
                    self.tail.next = None
                    self.size -= 1
                return
                _____ # adjust the prev node's pointer
                _____ # adjust the next node's pointer
                self.size -= 1
            count += 1
            cur_node = cur_node.next
        _____ # move the current node
        _____ # increment the count
```

Expected Answer:

```
def delete_at(self, index):
    if index < 0 or index >= self.size:
        print("Position out of bounds")
        return
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
            self.size -= 1
        return
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
            return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next
        count += 1
```

Current traversal of the list

The **set_current()** method is used to reset the current variable to reference the start of the list:

```
def set_current(self):
    self.cur_node = self.head
```

The **set_current_to()** sets the current node to a node at the given index.

```
def set_current_to(self, index):
    self.cur_node = self.head
    for i in range(index):
        self.cur_node = self.cur_node.next
```

The **next()** method is used to traverse the list by setting the current variable to the node next to the node it is currently set to:

```
def next(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.next
```

The **prev()** method is used to traverse the list by setting the current variable to the node previous to the node it is currently set to.

```
def prev(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.prev
```

The **get_current()** method implements the retrieval of the data stored in the current node through a simple return statement.

```
def get_current(self):
    return self.cur_node.data
```

Examples:

1. Use `set_current` method to reset the current node

```
# Method to reset the current node to the start of the list
def reset_and_print_current(dll):
    dll.set_current()
    print(dll.get_current())

# Example usage
dll = DoublyLinkedList()
dll.append("John")
dll.append("Ben")
dll.append("Matthew")
reset_and_print_current(dll)  # Expected output: John
```

2. Retrieving Data from the Current Node

```
# Method to retrieve and print the data from the current node
def print_current_data(dll):
    print(dll.get_current())

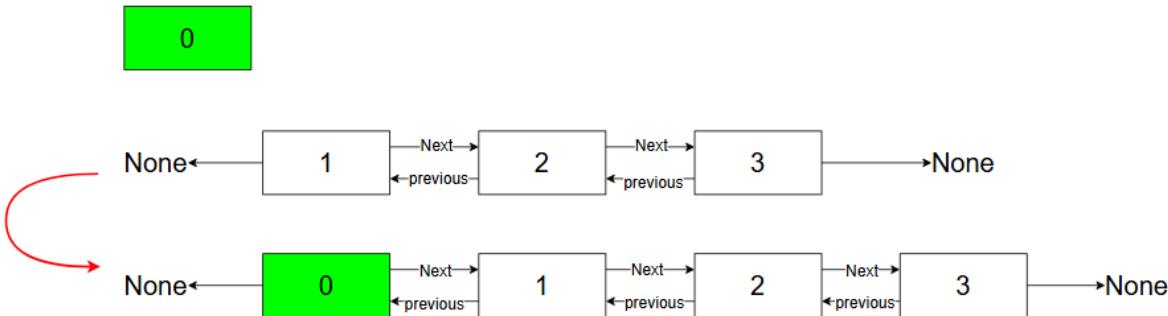
# Example usage
dll = DoublyLinkedList()
dll.append("John")
dll.append("Ben")
dll.append("Matthew")
dll.set_current_to(1)  # Set current to the second node
```

```
print_current_data(dll) # Expected output: Ben
```

Exercises 8 - 12 (Images)

Ex 8

What Linked List method is used in the below image?



prepend(0)

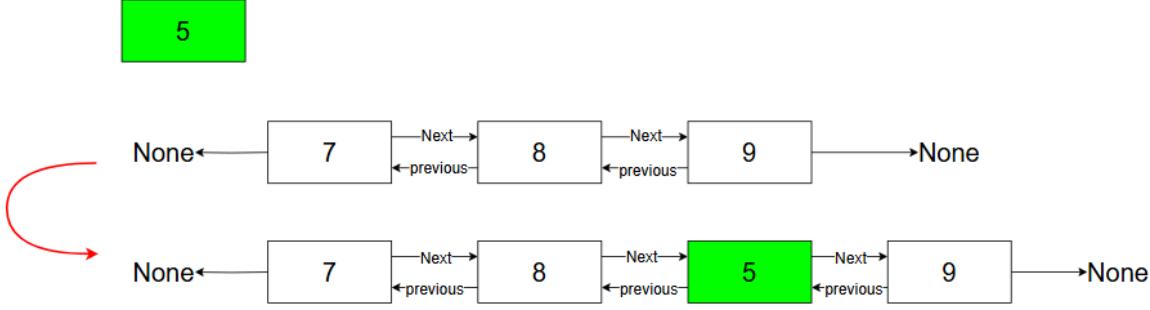
append(0)

insert_at(2, 0)

insert_at(0, 0)

Ex 9

What Linked List method is used in the below image?



prepend(5)

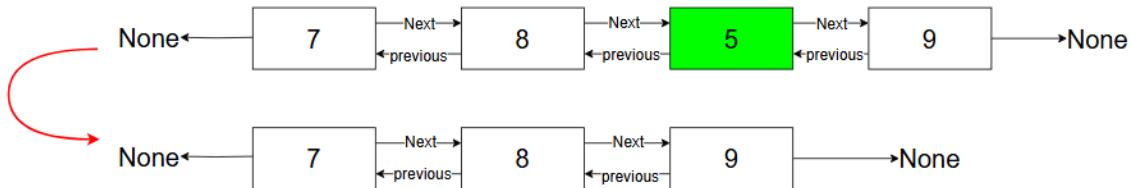
append(5)

insert_at(2, 5)

insert_at(1, 5)

Ex 10

What Linked List method is used in the below image?



prepend(5)

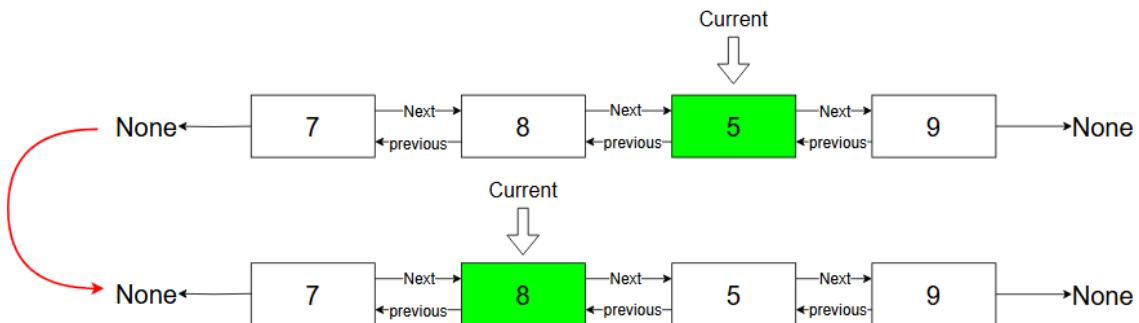
append(5)

delete(5)

delete_at(1)

Ex 11

What Linked List method is used in the below image?



prepend(8)

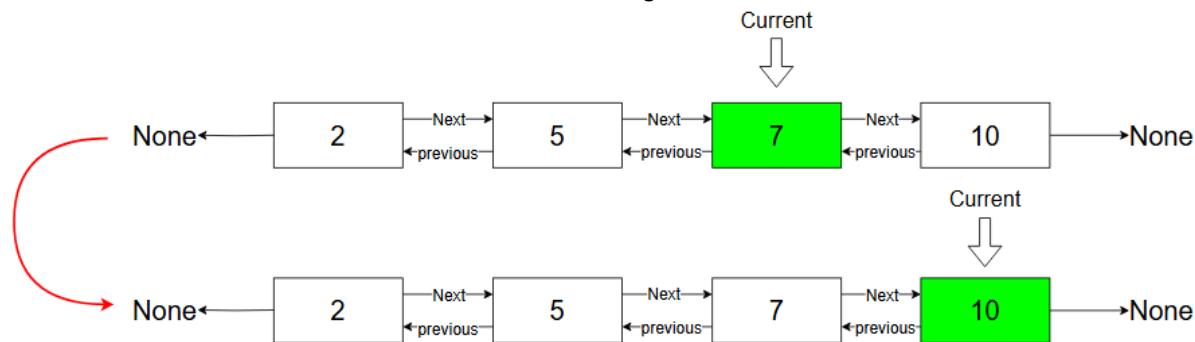
get_current()

prev()

next()

Ex 12

What Linked List method is used in the below image?



- set_current_to(7)
- set_current()
- prev()
- next()

Exercises 13 - 42 (Coding)

Story:

The local library has contacted us about the creation of a digital system for keeping track of the books they have and creating a system for searching for books by title and author. They have suggested that a doubly linked list would be useful as it models a shelf where a book is positioned on the shelf either at the start, end, or in between two books.

Ex 13

Create a class called Book that stores a book's title, author and pages. This will represent the books in the digital system. This object will be the data represented in each Node.

Given code:

```
class Book:  
    # Your code here
```

Expected input:

```
class Book:  
    def __init__(self, title, author, pages):  
        self.title = title  
        self.author = author  
        self.pages = pages
```

Ex 14

Create a basic Node class for a doubly linked list that will store the names of computer users as strings.

Given code:

```
class Node:  
    def __init__(self, data):  
        # Your code here
```

Expected input:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None
```

Ex 15

Create a linked list class for a doubly linked list and define its `__init__` constructor.

Hint: remember to keep track of the head, tail, and the size of the linked list.

```
class DoublyLinkedList:  
    def __init__(self):  
        # Your code here
```

Expected input:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0
```

Ex 16

Write the `append()` method for the doubly linked list.

The 'DoublyLinkedList' class has the following attributes:

- 'head': Points to the first node in the list.

- `tail`: Points to the last node in the list.
- `size`: Represents the number of nodes in the list.

The `Node` class has the following attributes:

- `data`: Stores the value of the node.
- `next`: Points to the next node in the list.
- `prev`: Points to the previous node in the list.

Steps:

- Create a new node (`new_node`) with the provided data.
- Check if the list is empty. If it is, set both the head and the tail to the new node.
- If the list is not empty, update the references to add the new node to the end.
- Update the size of the list.

Given Code:

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        # Your code here
```

Expected input:

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1
```

Ex 17

Write the prepend() method for the doubly linked list.

The `DoublyLinkedList` class has the following attributes:

- `head`: Points to the first node in the list.
- `tail`: Points to the last node in the list.
- `size`: Represents the number of nodes in the list.

The `Node` class has the following attributes:

- `data`: Stores the value of the node.
- `next`: Points to the next node in the list.
- `prev`: Points to the previous node in the list.

Steps:

- Create a new node (`new_node`) with the provided data.
- Check if the list is empty. If it is, set both the head and the tail to the new node.
- If the list is not empty, update the references to add the new node to the start.
- Update the size of the list.

Given code:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
        else:  
            new_node.prev = self.tail  
            self.tail.next = new_node  
            self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        # Your code here
```

Expected input:

```
def prepend(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.tail = new_node  
    else:  
        new_node.next = self.head  
        self.head.prev = new_node  
        self.head = new_node  
    self.size += 1
```

Ex 18

Write the `insert_at()` method for the doubly linked list. (You can reuse the previously defined methods.)

The `'DoublyLinkedList'` class has the following attributes:

- `'head'`: Points to the first node in the list.
- `'tail'`: Points to the last node in the list.

- `size`: Represents the number of nodes in the list.

The `Node` class has the following attributes:

- `data`: Stores the value of the node.
- `next`: Points to the next node in the list.
- `prev`: Points to the previous node in the list.

Steps:

- If the position is invalid, print "Position out of bounds" and return.
- Create a new node (`new_node`) with the provided data.
- If the position is 0, call the `prepend()` method to add the data at the beginning of the list.
- If the position is the size of the list, call the `append()` method to add the data at the end of the list.
- Traverse the list to find the node at the specified index.
- Update the references of the new node, the previous node, and the next node to insert the new node at the specified index.
- Update the size of the list.

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.size += 1

    def insert_at(self, data, position):
        # Your code here
```

Expected input:

```

def insert_at(self, data, position):
    if position < 0 or position > self.size:
        print("Position out of bounds")
        return
    new_node = Node(data)
    if position == 0:
        self.prepend(data)
        return
    if position == self.size:
        self.append(data)
        return
    current = self.head
    count = 0
    while current and count < position:
        current = current.next
        count += 1
    new_node.prev = current.prev
    new_node.next = current
    if current.prev:
        current.prev.next = new_node
    current.prev = new_node
    self.size += 1

```

Ex 19

Write the get() method for the doubly linked list.

Steps:

- If the index is invalid, return -1.
- Initialize a `cur_node` pointer to the head of the list.
- Use a `count` variable to keep track of the current position in the list.
- Traverse the list until you reach the node at the specified index.
- Return the data stored in the node at the specified index.

Given Code:

```

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
            return
        new_node.prev = self.tail
        self.tail.next = new_node

```

```

        self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head is not None:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        else:
            self.head = new_node
            self.tail = new_node
        self.size += 1

    def insert_at(self, data, position):
        if position < 0 or position > self.size:
            print("Position out of bounds")
            return
        new_node = Node(data)
        if position == 0:
            self.prepend(data)
            return
        if position == self.size:
            self.append(data)
            return
        current = self.head
        count = 0
        while current and count < position:
            current = current.next
            count += 1
        new_node.prev = current.prev
        new_node.next = current
        if current.prev:
            current.prev.next = new_node
        current.prev = new_node
        self.size += 1

    def get(self, index):
        # Your code here

```

Expected input:

```

def get(self, index):
    if index < 0 or index >= self.size:
        return -1

    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        count += 1

```

```

    cur_node = cur_node.next
    count += 1
return -1

```

Ex 20

Write the index() method for the doubly linked list.

Steps:

- Initialize a `cur_node` pointer to the head of the list.
- Use a `count` variable to keep track of the current position in the list.
- Traverse the list until you find a node with the specified data.
- Return the index of the first occurrence of the data. If not found, return -1.

Given Code:

```

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
            self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head is not None:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        else:
            self.head = new_node
            self.tail = new_node
        self.size += 1

    def insert_at(self, index, data):
        if index == 0:
            self.prepend(data)
            return
        new_node = Node(data)
        cur_node = self.head

```

```

count = 0
while cur_node:
    if count == index:
        new_node.next = cur_node
        new_node.prev = cur_node.prev
        cur_node.prev.next = new_node
        cur_node.prev = new_node
        self.size += 1
    return
    cur_node = cur_node.next
    count += 1

def get(self, index):
    if index < 0 or index >= self.size:
        return -1

    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return -1

def index(self, data):
    # Your code here

```

Expected input:

```

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

```

Ex 21

Write the delete() method for the doubly linked list.

Steps:

1. Start with a `cur_node` pointer initialized to the head of the list.
2. Traverse the list until you find a node with the specified data.
3. Handle cases where the node to be deleted is the head, tail, or somewhere in between.
4. Adjust the `prev` and `next` references of adjacent nodes accordingly.
5. Update the size of the list.

Given Code:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            self.size += 1  
            return  
        new_node.prev = self.tail  
        self.tail.next = new_node  
        self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head is not None:  
            new_node.next = self.head  
            self.head.prev = new_node  
            self.head = new_node  
        else:  
            self.head = new_node  
            self.tail = new_node  
        self.size += 1  
  
    def insert_at(self, index, data):  
        if index == 0:  
            self.prepend(data)  
            return  
        new_node = Node(data)  
        cur_node = self.head  
        count = 0  
        while cur_node:  
            if count == index:  
                new_node.next = cur_node  
                new_node.prev = cur_node.prev  
                cur_node.prev.next = new_node  
                cur_node.prev = new_node  
                self.size += 1  
                return  
            cur_node = cur_node.next  
            count += 1  
  
    def get(self, index):  
        cur_node = self.head
```

```

count = 0
while cur_node:
    if count == index:
        return cur_node.data
    cur_node = cur_node.next
    count += 1

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def delete(self, data):
    # Your code here

```

Expected input:

```

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head and cur_node == self.tail:
                self.head = None
                self.tail = None
                self.size -= 1
                return
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = None
                self.size -= 1
                return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next

```

Ex 22

Write the `delete_at()` method for the doubly linked list.

Steps:

1. Check if the index is valid and print "Position out of bounds" if not.
2. Check if the index is 0. If so, handle the case for deleting the head node.
3. Traverse the list until you reach the node at the specified index.
4. Handle different cases for deleting a node, considering if it's the tail or somewhere in between.
5. Adjust the `'prev'` and `'next'` references of adjacent nodes accordingly.
6. Update the size of the list.

Given code:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            self.size += 1  
            return  
        new_node.prev = self.tail  
        self.tail.next = new_node  
        self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head is not None:  
            new_node.next = self.head  
            self.head.prev = new_node  
            self.head = new_node  
        else:  
            self.head = new_node  
            self.tail = new_node  
        self.size += 1  
  
    def insert_at(self, index, data):  
        if index == 0:  
            self.prepend(data)  
            return  
        new_node = Node(data)  
        cur_node = self.head  
        count = 0  
        while cur_node:  
            if count == index:  
                new_node.next = cur_node
```

```

        new_node.prev = cur_node.prev
        cur_node.prev.next = new_node
        cur_node.prev = new_node
        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head and cur_node == self.tail:
                self.head = None
                self.tail = None
                self.size -= 1
                return
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = None
                self.size -= 1
                return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next

```

```
def delete_at(self, index):
    # Your code here
```

Expected input:

```
def delete_at(self, index):
    if index < 0 or index >= self.size:
        print("Position out of bounds")
        return
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
        self.size -= 1
        return
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next
        count += 1
```

Ex 23

In the requirements brief the local library has supplied you with, it is an expected functionality for the attributes of the Book class to be printed in a consumer-friendly format. Write the `__str__` method for the Book class that returns a string in the format: "<title> by <author>, <pages> pages." .

Given code:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
```

```
def __str__(self):  
    # Your code here
```

Expected input:

```
def __str__(self):  
    return self.title + " by " + self.author + ", " +  
    str(self.pages) + " pages."
```

Ex 24

The local library has provided you with a small sample of data that can be found in their current inventory so that you can test the code you have made so far.

Create a list sci_fi to represent the sci-fi shelf and add the following books.

- "The Martian", "Andy Weir", 384
- "The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 224
- "The Time Machine", "H.G. Wells", 96
- "The War of the Worlds", "H.G. Wells", 128
- "A Princess of Mars", "Edgar Rice Burroughs", 256

Expected input:

```
sci_fi = DoublyLinkedList()  
sci_fi.append(Book("The Martian", "Andy Weir", 384))  
sci_fi.append(Book("The Hitchhiker's Guide to the Galaxy", "Douglas  
Adams", 224))  
sci_fi.append(Book("The Time Machine", "H.G. Wells", 96))  
sci_fi.append(Book("The War of the Worlds", "H.G. Wells", 128))  
sci_fi.append(Book("A Princess of Mars", "Edgar Rice Burroughs",  
256))
```

Ex 25

By extension, the library expects that there is a way to print out the entire shelf of book data from one function call.

Write a method called `print_list()` for the `DoublyLinkedList` class that prints the data of each node in the list.

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data)
```

```

        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
            return
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head is not None:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        else:
            self.head = new_node
            self.tail = new_node
        self.size += 1

    def insert_at(self, index, data):
        if index == 0:
            self.prepend(data)
            return
        new_node = Node(data)
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                new_node.next = cur_node
                new_node.prev = cur_node.prev
                cur_node.prev.next = new_node
                cur_node.prev = new_node
                self.size += 1
                return
            cur_node = cur_node.next
            count += 1

    def get(self, index):
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                return cur_node.data
            cur_node = cur_node.next
            count += 1

    def index(self, data):
        cur_node = self.head
        count = 0

```

```

while cur_node:
    if cur_node.data == data:
        return count
    cur_node = cur_node.next
    count += 1
return -1

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head and cur_node == self.tail:
                self.head = None
                self.tail = None
            elif cur_node == self.head:
                self.head = cur_node.next
                cur_node.next.prev = None

            elif cur_node == self.tail:
                self.tail = cur_node.prev
                cur_node.prev.next = None

            else:
                cur_node.prev.next = cur_node.next
                cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next

def delete_at(self, index):
    if index == 0:
        self.head = self.head.next
        self.head.prev = None
        self.size -= 1
        return
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.tail:
                self.tail = cur_node.prev
                cur_node.prev.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next
        count += 1

```

```
def print_list(self):  
    # Your code here
```

Expected input:

```
def print_list(self):  
    cur_node = self.head  
    while cur_node:  
        print(cur_node.data)  
        cur_node = cur_node.next
```

Ex 26

Write a single line of code to print all of the books on the sci-fi shelf.

Expected input:

```
sci_fi.print_list()
```

Expected output:

The Martian by Andy Weir, 384 pages.

The Hitchhiker's Guide to the Galaxy by Douglas Adams, 224 pages.

The Time Machine by H.G. Wells, 96 pages.

The War of the Worlds by H.G. Wells, 128 pages.

A Princess of Mars by Edgar Rice Burroughs, 256 pages.

Explanation:

In exercises 25 and 26, encapsulation is shown by bundling the printing logic within the `print_list` method inside the linked list class. Abstraction is seen in how users interact with the linked list through a simple method call without needing to know the internal workings of the list traversal and printing.

Encapsulation and Abstraction principles focus on separating complexities and hiding implementation details:

Encapsulation:

What it means: It's like placing related things in a box. In programming, it involves bundling data and methods that work on the data within a single unit (like a class).

Why it's useful: It helps organize code by grouping related functionalities together, making it easier to manage and understand.

Abstraction:

What it means: It's like using a TV remote without knowing how it works internally. It involves hiding complex implementation details and providing a simpler interface.

Why it's useful: It allows users of a class or object to interact with it using simple and understandable methods or operations, without needing to understand the underlying complexities.

We are now going to simulate the process of looking through the books and determining if the books on either side are by the same author. This will first be done with the standard methods and then be done using current for traversal.

Ex 27

Write a program that displays details of a book from a predefined linked list `sci_fi` based on the user's input.

The user should be prompted to enter the index number of the book they wish to view. Once the index is provided, your program should retrieve the book from the list, save it to a variable named `book`, and then print its details.

Sample Output:

What is the index of the book you want to see? (user input: 1)
The Hitchhiker's Guide to the Galaxy by Douglas Adams, 224 pages

Expected input:

```
index = int(input("What is the index of the book you want to see?"))
book = sci_fi.get(index)
print(book)
```

Ex 28

We need to check if the book (after the current book) is written by the same author if the index is at the start of the list, if it is then print("The next book is by the same author") else print ("The next book is by a different author"). Write an if statement that handles a user input of 0.

Given Input:

```
index = int(input("What is the index of the book you want to see?"))
book = sci_fi.get(index)
print(book)
if <insert condition here>:
    # Your code here
```

Expected input:

```
if index == 0 and sci_fi.size > 1:
    next_book = sci_fi.get(index + 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
```

```
print("The next book is by a different author")
```

Ex 29

We need to check if either the book before or after is written by the same author if the index is at the end of the list, if it is print("The previous book is by the same author") else print("The previous book is by a different author"). Write an elif statement that handles a user input of sci_hi.size - 1.

Note: put this in an elif statement.

Given Code:

Either what they did previously OR

```
index = int(input("What is the index of the book you want to see?"))
book = sci_hi.get(index)
print(book)
if index == 0 and sci_hi.size > 1:
    next_book = sci_hi.get(index + 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
        print("The next book is by a different author")
elif <insert condition here> :
    # Your code here
```

Expected input:

```
elif index == sci_hi.size - 1:
    prev_book = sci_hi.get(index - 1)
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
```

Ex 30

We need to check if either of the books before or after it is written by the same author. If it is then print "The next/previous book is by the same author" else print "The next/previous book is by a different author". Write an else statement that handles all other user inputs within the list.

Given code:

```
index = int(input("What is the index of the book you want to see?"))
book = sci_hi.get(index)
print(book)
if index == 0 and sci_hi.size > 1:
    next_book = sci_hi.get(index + 1)
    if book.author == next_book.author:
```

```

        print("The next book is by the same author")
else:
    print("The next book is by a different author")
elif index == sci_fi.size - 1:
    prev_book = sci_fi.get(index - 1)
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
else:
    # Your code here

```

Expected input:

```

else:
    next_book = sci_fi.get(index + 1)
    prev_book = sci_fi.get(index - 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
        print("The next book is by a different author")
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")

```

Ex 31

Surround the entire code in a while True loop to allow repeated searches.

At the start of each loop:

- Print "Enter the index of the book you want to see"
- Print "Enter -1 to exit"

Capture the user input for the index. If -1 is entered, exit the loop.

Then validate the index:

If the index is out of range, print "That is not a valid index" and restart the loop using continue.

If the index is valid, retrieve the book details using `sci_fi.get(index)` and print it. Then, compare the author of this book with its neighboring books:

- If it's the first book: Compare with the next book.
- If it's the last book: Compare with the previous book.
- If it's in the middle: Compare with both the previous and next books.

Based on these comparisons, print:

- "The next book is by the same author" or "The next book is by a different author"
- "The previous book is by the same author" or "The previous book is by a different author"

Given code:

```
# While loop here
# Print index request here
# print -1 message here
index = int(input())
# Check for exit input
# Check for out of bounds input
book = sci_fi.get(index)
print(book)
if index == 0 and sci_fi.size > 1:
    next_book = sci_fi.get(index + 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
        print("The next book is by a different author")
elif index == sci_fi.size - 1:
    prev_book = sci_fi.get(index - 1)
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
else:
    next_book = sci_fi.get(index + 1)
    prev_book = sci_fi.get(index - 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
        print("The next book is by a different author")
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
```

Expected input:

```
while True:
    print("Enter the index of the book you want to see")
    print("Enter -1 to exit")
    index = int(input())
    if index == -1:
        break
    if index < 0 or index >= sci_fi.size:
        print("That is not a valid index")
        continue
    book = sci_fi.get(index)
    print(book)
    if index == 0 and sci_fi.size > 1:
        next_book = sci_fi.get(index + 1)
```

```

if book.author == next_book.author:
    print("The next book is by the same author")
else:
    print("The next book is by a different author")
elif index == sci_fi.size - 1:
    prev_book = sci_fi.get(index - 1)
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
else:
    next_book = sci_fi.get(index + 1)
    prev_book = sci_fi.get(index - 1)
    if book.author == next_book.author:
        print("The next book is by the same author")
    else:
        print("The next book is by a different author")
    if book.author == prev_book.author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")

```

Now we are going to rewrite the above method using the current methods to gain an understanding of the methods.

Ex 32

Write the `set_current()` and `set_current_to()` methods for the linked list.

The `set_current()` method should set the `cur_node` instance variable to the head node.
 The `set_current_to(index)` should set the `current_node` to the head variable, and then traverse the list to the node at the specified index.

Return None if the relevant nodes do not exist.

Given Code:

```

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None # This has been added as an instance variable to track the
                            # current node

```

```

def set_current(self):
    # Your code here

```

```
def set_current_to(self, index):
    # Your code here
```

Expected input:

```
def set_current(self):
    self.cur_node = self.head

def set_current_to(self, index):
    self.cur_node = self.head
    for i in range(index):
        if self.cur_node is None:
            return self.cur_node
        self.cur_node = self.cur_node.next
```

Ex 33

Write the next() and prev() methods for the linked list.

They should set the cur_node to the next or previous node based on the method name.

Hint: Remember to deal with the case where the current node is None.

Given Code:

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None # This has been added as an instance variable to track the current
node
```

```
def set_current(self):
    self.cur_node = self.head

def set_current_to(self, index):
    self.cur_node = self.head
    for i in range(index):
        if self.cur_node is None:
            return self.cur_node
        self.cur_node = self.cur_node.next

def next(self):
    # Your code here

def prev(self):
    # Your code here
```

Expected input:

```
def next(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.next

def prev(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.prev
```

Ex 34

Write the get_current() method for the linked list.

It should return the data held by the current node.

Given Code:

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None # This has been added as an instance variable to track the current
node

    def set_current(self):
        self.cur_node = self.head

    def set_current_to(self, index):
        self.cur_node = self.head
        for i in range(index):
            if self.cur_node is None:
                return self.cur_node
            self.cur_node = self.cur_node.next

    def next(self):
        if self.cur_node == None:
            return
        self.cur_node = self.cur_node.next

    def prev(self):
        if self.cur_node == None:
            return
        self.cur_node = self.cur_node.prev

    def get_current(self):
```

Your code here

Expected input:

```
def get_current(self):  
    return self.cur_node.data
```

Ex 35

To begin reconstructing the program, write the initial "while True:" loop for handling the index input section, if -1 then break. And instead of setting a book variable, set the current and then print it.

Expected input:

```
while True:  
    print("Enter the index of the book you want to see")  
    print("Enter -1 to exit")  
    index = int(input())  
    if index == -1:  
        break  
    sci_fi.set_current_to(index)  
    print(sci_fi.get_current())
```

Ex 36

Now continue this by rewriting the case of the index being 0 without the book variable.

Given code:

```
while True:  
    print("Enter the index of the book you want to see")  
    print("Enter -1 to exit")  
    index = int(input())  
    if index == -1:  
        break  
    sci_fi.set_current_to(index)  
    print(sci_fi.get_current())  
# Write If statement here
```

Expected input:

```
if index == 0 and sci_fi.size > 1:  
    sci_fi.next()  
    if sci_fi.get_current().author == sci_fi.get(index).author:  
        print("The next book is by the same author")  
    else:  
        print("The next book is by a different author")
```

Ex 37

Now continue this by rewriting the case of the index being `sci_hi.size - 1` without the book variable.

Given Code:

```
while True:  
    print("Enter the index of the book you want to see")  
    print("Enter -1 to exit")  
    index = int(input())  
    if index == -1:  
        break  
    sci_hi.set_current_to(index)  
    print(sci_hi.get_current())  
    if index == 0 and sci_hi.size > 1:  
        sci_hi.next()  
        if sci_hi.get_current().author == sci_hi.get(index + 1).author:  
            print("The next book is by the same author")  
        else:  
            print("The next book is by a different author")  
    elif <insert condition here>:  
        # Your code here
```

Expected input:

```
elif index == sci_hi.size - 1:  
    sci_hi.prev()  
    if sci_hi.get_current().author == sci_hi.get(index).author:  
        print("The previous book is by the same author")  
    else:  
        print("The previous book is by a different author")
```

Ex 38

Now continue this by rewriting the case of any other index without the book variable.

Given code:

```
while True:  
    print("Enter the index of the book you want to see")  
    print("Enter -1 to exit")  
    index = int(input())  
    if index == -1:  
        break  
    sci_hi.set_current_to(index)  
    print(sci_hi.get_current())  
    if index == 0 and sci_hi.size > 1:  
        sci_hi.next()  
        if sci_hi.get_current().author == sci_hi.get(index + 1).author:  
            print("The next book is by the same author")
```

```

else:
    print("The next book is by a different author")
elif index == sci_fi.size - 1:
    sci_fi.prev()
    if sci_fi.get_current().author == sci_fi.get(index - 1).author:
        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")
else:
    # Your code here

```

Expected input:

```

else:
sci_fi.next()
if sci_fi.get_current().author == sci_fi.get(index).author:
    print("The next book is by the same author")
else:
    print("The next book is by a different author")
sci_fi.prev()
sci_fi.prev()
if sci_fi.get_current().author == sci_fi.get(index).author:
    print("The previous book is by the same author")
else:
    print("The previous book is by a different author")
sci_fi.next()

```

Ex 39

The final task we are going to do is print the list in reverse order starting at the tail and ending at the head using the current and prev method.

Expected input:

```

sci_fi.set_current_to(sci_fi.size - 1)
while sci_fi.get_current() != sci_fi.get(0):
    print(sci_fi.get_current())
    sci_fi.prev()
print(sci_fi.get_current())

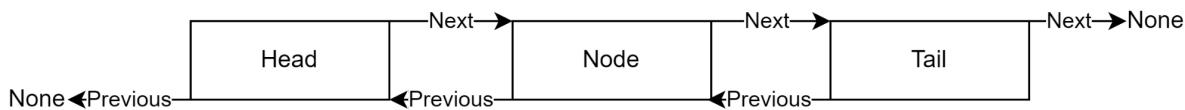
```

Recap (Lesson 4)

Doubly linked lists

Doubly linked lists differ from typical singly linked lists, by being able to operate in a bidirectional manner.

This is done through the usage of a previous pointer (called prev) in their nodes that point to the previous node in the linked list or None if it is the head.



This additional pointer does require some more care when doing methods that work with middle nodes. Doubly linked lists also use a tail pointer, which helps them improve the time complexity of various operations.

Initializing nodes and linked lists

Both operations take O(1) time. However nodes in a doubly linked list will require an extra pointer to point toward the previous node, and the doublyLinkedList will use a tail pointer.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0
```

Prepend

The method is still O(1), but takes a bit more care due to the additional pointer.

```
def prepend(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.tail = new_node  
    else:  
        new_node.next = self.head  
        self.head.prev = new_node  
        self.head = new_node  
    self.size += 1
```

Append

Adding to the end of a list now takes O(1) time due to the fact that the doubly linked lists keep track of the tail node.

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1
```

Insert at

Inserting a node into the middle of a list still requires O(n) time, however you need to be more careful of how you deal with pointers.

```
def insert_at(self, data, position):
    if position < 0 or position > self.size:
        print("Position out of bounds")
        return
    new_node = Node(data)

    if position == 0:
        self.prepend(data)
        return

    if position == self.size:
        self.append(data)
        return

    current = self.head
    count = 0
    while current and count < position:
        current = current.next
        count += 1

    new_node.prev = current.prev
    new_node.next = current
    if current.prev:
        current.prev.next = new_node
    current.prev = new_node
```

```
    self.size += 1
```

Delete

This method looks through the list and deletes a node based on if it matches the data it is looking for.

```
def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head and cur_node == self.tail:
                self.head = None
                self.tail = None
                self.size -= 1
                return

            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = None
                self.size -= 1
                return

            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = None
                self.size -= 1
                return

            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return

        cur_node = cur_node.next
```

There are also methods to delete based on index number.

Current traversal

You can also add another pointer to the linked list and work with a traversal technique that allows the user control over moving back and forth between nodes.

```
def set_current_to(self, index):
    self.cur_node = self.head
    for i in range(index):
        self.cur_node = self.cur_node.next
```

```
def next(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.next
```

```
def prev(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.prev
```

These methods and others allow more control over select parts of the linked list and potentially enable the improvement of performance in some cases.

Conclusion:

Doubly linked lists offer enhanced functionality and flexibility compared to singly linked lists. By allowing bidirectional traversal and efficient insertion and deletion operations, they can be more suitable for certain applications.

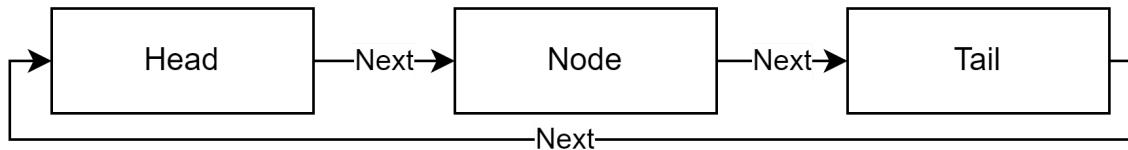
5. Circular Linked List

5.1 Structure

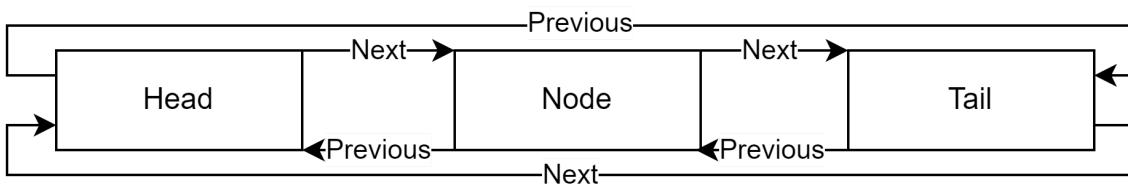
Circular linked lists are a special type of linked lists.

A circular linked list is a type of linked list in which the last node of the list points back to the first node, forming a closed loop. This circular structure distinguishes it from traditional singly or doubly linked lists.

They are very similar to single or doubly linked lists, but have the head and tail nodes connected to each other.



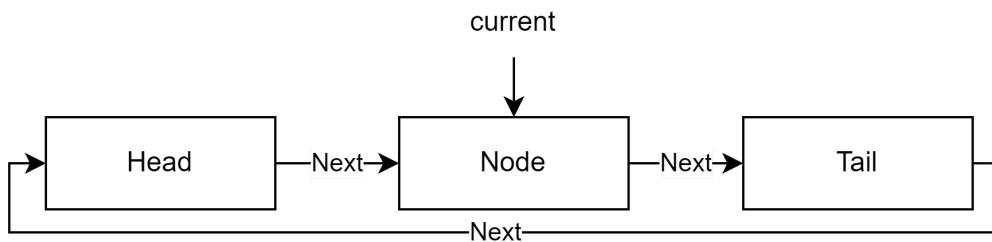
This linking creates a data structure where every link references another **node**, rather than **None**. The above circularly linked list is **singly** linked.



The circular structure of the list efficiently solves problems requiring continuous traversal, while double linking enables bidirectional traversal (as shown in the above example).

The utility of the circular linked list comes mostly from its ability to implement custom methods through **continuous** linking, which enables traversal without considering index positions.

This is achieved by using the node pointer '**current**' in the linked list class, which tracks the node being accessed by the user.



Methods allow us to access the information in each node and navigate through the list. These methods help us update the 'current' pointer to move to the next node, making the circular linked list even more useful.

In a circular linked list, a node cannot be linked to "None". Unlike a linear linked list where the last node points to "None" to indicate the end of the list, in a circular linked list, the last node is connected back to the first node, forming a loop.

Advantages of Circular Linked Lists:

- **Efficient traversal:** In a circular linked list, you can start traversing from any node and still visit all the nodes in the list. There is no need to check for the end of the list explicitly.
- **Efficient insertion and deletion at the beginning or end:** Since the last node points back to the head, inserting or deleting nodes at the beginning or end of the list can be done without the need to traverse the entire list.
- **Useful for implementing circular buffers or queues:** Circular linked lists are commonly used to implement circular buffers or queues, where the last element wraps around to the first element.

However, circular linked lists also have some disadvantages:

- **Increased complexity:** The circular structure of the list requires additional logic to handle the wrapping around from the last node to the head.
- **Risk of infinite loops:** If not handled properly, traversing a circular linked list can lead to infinite loops if the termination condition is not correctly defined.

Exercises 1 - 10 (MCQ)

Ex 1

What structure does a circular linked list have?

Circular chain

Branching tree

Interconnected loop

Several loops

Explanation: A circular linked list is where each node is connected to two other nodes. An interconnected loop would imply nodes can be connected in any way, creating cycles. The connection of the tail and head creates this circular chain.

Ex 2

A node can be linked to None in a circular linked list.

True

False

Explanation: In a circular linked list, no node is linked to None. Instead, the last node links back to the head.

Ex 3

What is the purpose of the *current* variable?

To locate the most important node in the list

To select the node to be deleted

To reference the node a user is currently accessing

To save a new node to the list

Ex 4

What is the main advantage of using a circular linked list?

The list can store more data

The list can be traversed in a loop

The list uses less storage

The list can be traversed

Ex 5

Only a singly linked list can be circular.

True

False

Ex 6

Only a doubly linked list can be circular.

True

False

Ex 7

Which of the following statements accurately describes the traversal capabilities of circular linked lists compared to regular linked lists?

Circular linked lists offer more efficient traversal because you can start from any node and visit all the nodes in the list without explicitly checking for the end of the list.

Circular linked lists have the same traversal capabilities as regular linked lists, as both require explicit checks for the end of the list to avoid infinite loops.

Circular linked lists have less traversal utility compared to regular linked lists because the circular structure makes it more complex to traverse the list.

Circular linked lists and regular linked lists have identical traversal performance, and there is no significant difference in their traversal capabilities.

Explanation: In a circular linked list, the last node points back to the first node, allowing traversal from any starting point to visit all nodes without checking for a null reference as in a regular (singly or doubly) linked list. This property enables continuous traversal until explicitly stopped, unlike regular linked lists that require a check for NULL to avoid going out of bounds.

Ex 8

A circular linked list is significantly more efficient in all scenarios.

True

False

Ex 9

A circular linked list operates with higher efficiency in scenarios such as circular queues.

True

False

Explanation: Circular linked lists are well-suited for implementing **circular queues**, as their structure allows efficient handling of scenarios where the last element of the queue is followed by the first element.

Ex 10

A circular doubly linked list does not allow traversal in both directions.

True

False

5.2 Circular Singly Linked List

The singly circular linked list is the most basic implementation of a circular linked list with its only additional requirements being that the tail node is connected to the head node, creating a continuous loop in the data structure.

The current node can be tracked and updated easily using standard traversal or modification methods.

The Node class is **identical** to the standard singly linked list.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

The circular linked list class is identical to the **doubly** linked list with the tail and current variables used to implement the circular structure.

```
class CircularLinkedList:
```

```

def __init__(self):
    self.head = None
    self.tail = None
    self.size = 0
    self.current = None

```

5.2.1 prepend() Method:

The prepend method adds a new node at the **beginning** of the circular linked list.

- If the list is empty
 - the new node becomes both the head and the tail.
 - its next pointer points to itself, forming a circular structure.
- If the list is **not** empty
 - the new node's next pointer is set to the current head.
 - the tail's next pointer is updated to point to the new node.
 - the head is updated to the new node.

Update the size of the list accordingly, when the insertion is processed.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data)

        # If the list is empty
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node # Point to itself, making the
list circular
        else:
            # Insert the new node at the beginning
            new_node.next = self.head

```

```

        self.tail.next = new_node # Update the last node to
point to the new node
        self.head = new_node # Update the head to be the new
node
        self.size += 1

def print_list(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head
    while True:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.head:
            break
    print("(head)")

```

Example usage:

```

cll = CircularLinkedList()
cll.prepend("John")
cll.prepend("Ben")
cll.prepend("Matthew")

print("Circular Singly Linked List after prepending:")
cll.print_list()

```

Output:

Circular Singly Linked List after prepending:
 Matthew -> Ben -> John -> (head)

5.2.2 append() Method:

The append method adds a new node to the end of the circular linked list.

- If the list is empty
 - the new node becomes both the head and the tail.
 - its next pointer points to itself, forming a circular structure.
- If the list is **not** empty
 - the new node's next pointer is set to the current head.
 - the tail's next pointer is updated to point to the new node.
 - the tail is updated to the new node.

Update the size of the list accordingly, when the insertion is processed.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            # If the list is empty
            self.head = self.tail = new_node
            new_node.next = new_node # Point to itself, making the
list circular
        else:
            # Insert the new node at the end
            new_node.next = self.head # The new node points to the
head
            self.tail.next = new_node # The old tail points to the
new node
            self.tail = new_node # Update the tail to be the new
node
        self.size += 1

    def print_list(self):
        if self.head is None:
            print("The list is empty.")
            return
        current = self.head
        while True:
            print(current.data, end=" -> ")
            current = current.next
            if current == self.head:
                break
        print("(head)")
```

Example usage:

```
cll = CircularLinkedList()
```

```

cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Circular Singly Linked List after appending:")
cll.print_list()

```

Output:

Circular Singly Linked List after appending:
 John -> Ben -> Matthew -> (head)

5.2.3 insert_at() Method:

The `insert_at` method adds a new node at a specific position in the circular linked list.

- If the position is **0**
 - it calls the **prepend** method to insert the node at the beginning of the list.
- If the position is equal to the **size of the list**:
 - it calls the **append** method to insert the node at the end of the list.
- If the position is somewhere in the **middle**
 - it **traverses** the list to find the node just before the desired position.
 - updates the **next pointers** to insert the new node.

Update the size of the list accordingly, when the insertion is processed.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            # If the list is empty
            self.head = self.tail = new_node
            new_node.next = new_node # Point to itself, making the

```

```

list circular
    else:
        # Insert the new node at the end
        new_node.next = self.head # The new node points to the
head
        self.tail.next = new_node # The old tail points to the
new node
        self.tail = new_node # Update the tail to be the new
node
        self.size += 1

def insert_at(self, data, position):
    if index< 0 or index> self.size:
        print("Position out of bounds")
        return

    if position == 0:
        self.prepend(data)
        return

    if position == self.size:
        self.append(data)
        return

    new_node = Node(data)
    current = self.head
    count = 0

    while count < index- 1:
        current = current.next
        count += 1

    new_node.next = current.next
    current.next = new_node
    self.size += 1

def print_list(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head
    while True:
        print(current.data, end=" -> ")
        current = current.next

```

```

        if current == self.head:
            break
    print("(head) ")

# Example usage
cll = CircularLinkedList()
cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Original Circular Singly Linked List:")
cll.print_list()

cll.insert_at("Alice", 2)

print("Circular Singly Linked List after inserting 'Alice' at position 2:")
cll.print_list()

```

Output:

Original Circular Singly Linked List:
 John -> Ben -> Matthew -> (head)

Circular Singly Linked List after inserting 'Alice' at position 2:
 John -> Ben -> Alice -> Matthew -> (head)

5.2.4 delete() Method:

Deletes the first occurrence of a node with a specified value.

- If the list is empty:
 - Nothing happens.
- If the node to be deleted is the head:
 - Update the head to the next node.
 - If the list becomes empty after deletion, set both the head and tail to **None**.
- If the node to be deleted is not the head:
 - Traverse the list to find the node to delete.
 - Update the previous node's **next** pointer to skip the node to be deleted.

Update the size of the list accordingly, only when a deletion is processed.

```
class Node:
```

```

def __init__(self, data):
    self.data = data
    self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    def insert_at(self, data, position):
        new_node = Node(data)
        if index == 0:
            self.prepend(data)
        elif index == self.size:
            self.append(data)
        else:
            current = self.head
            count = 0
            while count < index - 1:
                current = current.next

```

```

        count += 1
        new_node.next = current.next
        current.next = new_node
        self.size += 1

def print_list(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head
    while True:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.head:
            break
    print("(head)")

def delete(self, data):
    # If the list is empty, do nothing
    if self.head is None:
        return
    # Case 1: Node to delete is the head
    if self.head.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            self.tail.next = self.head
        self.size -= 1
        return
    # Case 2: Node to delete is not the head
    current = self.head
    while current.next != self.head:
        if current.next.data == data:
            if current.next == self.tail:
                self.tail = current
            current.next = current.next.next
            self.size -= 1
            return
        current = current.next

# Example usage
cll = CircularLinkedList()

```

```

cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Original Circular Singly Linked List:")
cll.print_list()

cll.insert_at("Alice", 2)

print("Circular Singly Linked List after inserting 'Alice' at position 2:")
cll.print_list()

cll.delete("Ben")

print("Circular Singly Linked List after deleting 'Ben':")
cll.print_list()

```

Output:

Original Circular Singly Linked List:
 John -> Ben -> Matthew -> (head)

Circular Singly Linked List after inserting 'Alice' at position 2:
 John -> Alice -> Ben -> Matthew -> (head)

Circular Singly Linked List after deleting 'Ben':
 John -> Alice -> Matthew -> (head)

5.2.5 delete_at() Method:

Deletes the node at the given index.

- If the list is empty:
 - Do nothing.
- If the index is 0:
 - If there is only one node:
 - Make the head and tail None.
 - Set the current to 0.
 - Otherwise:
 - Move the head to the next node and adjust the tail's next pointer to point to the new head.
 - Set the current to 0.
- Otherwise, traverse the list until the index is reached. If the node is found:
 - Update the previous node's next pointer. If the node deleted was the tail, move the tail to the previous node and connect the new tail's pointer to the head.
- If not found, do nothing.

Update the size of the list accordingly, only when a deletion is processed.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.head = new_node  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.tail = new_node  
            self.size += 1  
  
    def insert_at(self, data, index):  
        new_node = Node(data)  
        if index == 0:  
            self.prepend(data)  
        elif index == self.size:  
            self.append(data)  
        else:  
            current = self.head  
            count = 0
```

```

        while count < index- 1:
            current = current.next
            count += 1
        new_node.next = current.next
        current.next = new_node
        self.size += 1

    def print_list(self):
        if self.head is None:
            print("The list is empty.")
            return
        current = self.head
        while True:
            print(current.data, end=" -> ")
            current = current.next
            if current == self.head:
                break
        print("(head)")

    def delete(self, data):
        if self.head is None:
            return
        if self.head.data == data:
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                self.head = self.head.next
                self.tail.next = self.head
            self.size -= 1
            return
        current = self.head
        while current.next != self.head:
            if current.next.data == data:
                if current.next == self.tail:
                    self.tail = current
                current.next = current.next.next
                self.size -= 1
                return
            current = current.next

    def delete_at(self, index):
        if self.head is None:
            return

```

```

        if index < 0 or index >= self.size:
            return "Invalid index"

        cur_node = self.head
        if index == 0:
            # Deleting the head node
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                self.head = cur_node.next
                self.tail.next = self.head
                cur_node = None
                self.size -= 1
            return
        # Traverse the list to find the node at the given index
        count = 0
        prev = None
        while count < index:
            prev = cur_node
            cur_node = cur_node.next
            count += 1
            if cur_node is self.head: # prevent infinite loop
                return
        prev.next = cur_node.next
        if cur_node == self.tail:
            self.tail = prev
        cur_node = None
        self.size -= 1
    
```

```

# Example usage
cll = CircularLinkedList()
cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Original Circular Singly Linked List:")
cll.print_list()

cll.insert_at("Alice", 2)

print("Circular Singly Linked List after inserting 'Alice' at position 2:")
cll.print_list()

cll.delete_at(2)
    
```

```
print("Circular Singly Linked List after deleting 'Ben' at position 2:")
cll.print_list()
```

Output:

Original Circular Singly Linked List:
John -> Ben -> Matthew -> (head)

Circular Singly Linked List after inserting 'Alice' at position 2:
John -> Alice -> Ben -> Matthew -> (head)

Circular Singly Linked List after deleting 'Ben' at position 2:")
John -> Alice -> Matthew -> (head)

5.2.6 get() Method:

Retrieves the data at a specific position in the circular linked list.

- If the position is invalid:
 - Return None.
- Otherwise:
 - Traverse the list to the desired position and return the node's data.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node
```

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
        new_node.next = new_node
    else:
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1

def insert_at(self, data, position):
    new_node = Node(data)
    if position == 0:
        self.prepend(data)
    elif position == self.size + 1:
        self.append(data)
    else:
        current = self.head
        count = 1
        while count < position - 1:
            current = current.next
            count += 1
        new_node.next = current.next
        current.next = new_node
    self.size += 1

def print_list(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head
    while True:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.head:
            break
    print("(head)")

def delete(self, data):
    if self.head is None:
        return
    if self.head.data == data:
        if self.head == self.tail:

```

```

        self.head = None
        self.tail = None
    else:
        self.head = self.head.next
        self.tail.next = self.head
    self.size -= 1
    return
current = self.head
while current.next != self.head:
    if current.next.data == data:
        if current.next == self.tail:
            self.tail = current
            current.next = current.next.next
            self.size -= 1
            return
        current = current.next

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head: # prevent infinite loop
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
    cur_node = None
    self.size -= 1

```

```

def get(self, position):
    if position < 1 or position > self.size: # Check for invalid
index
        return None
    current = self.head
    count = 1
    while count < position:
        current = current.next
        count += 1
    return current.data

```

Example usage

```

cll = CircularLinkedList()
cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Original Circular Singly Linked List:")
cll.print_list()

cll.insert_at("Alice", 2)

print("Circular Singly Linked List after inserting 'Alice' at position 2:")
cll.print_list()

print("Data at position 3:", cll.get(3))

```

cll.delete("Ben")

```

print("Circular Singly Linked List after deleting 'Ben':")
cll.print_list()

```

Output:

Original Circular Singly Linked List:
 John -> Ben -> Matthew -> (head)

Circular Singly Linked List after inserting 'Alice' at position 2:
 John -> Alice -> Ben -> Matthew -> (head)

Data at position 3: Ben

Circular Singly Linked List after deleting 'Ben':
 John -> Alice -> Matthew -> (head)

5.2.7 index() Method:

Retrieves the data with a specific value in the circular linked list.

- Traverse the list until you get a node with data the same as the value.
 - If it is found, return the index.
 - Otherwise, return -1.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.head = new_node  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.tail = new_node  
        self.size += 1  
  
    def insert_at(self, data, index):  
        new_node = Node(data)  
        if index == 0:
```

```

        self.prepend(data)
    elif index == self.size:
        self.append(data)
    else:
        current = self.head
        count = 0
        while count < index - 1:
            current = current.next
            count += 1
        new_node.next = current.next
        current.next = new_node
        self.size += 1

def print_list(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head
    while True:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.head:
            break
    print("(head)")

def delete(self, data):
    if self.head is None:
        return
    if self.head.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            self.tail.next = self.head
        self.size -= 1
        return
    current = self.head
    while current.next != self.head:
        if current.next.data == data:
            if current.next == self.tail:
                self.tail = current
            current.next = current.next.next
            self.size -= 1

```

```

        return
        current = current.next

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
            cur_node = None
            self.size -= 1
    return
count = 0
prev = None
while count < index:
    prev = cur_node
    cur_node = cur_node.next
    count += 1
    if cur_node is self.head: # prevent infinite loop
        return
prev.next = cur_node.next
if cur_node == self.tail:
    self.tail = prev
cur_node = None
self.size -= 1

def get(self, index):
    if index < 0 or index >= self.size:
        return None
    current = self.head
    count = 0
    while count < index:
        current = current.next
        count += 1
    return current.data

def index(self, data):
    # Find the index of the first node with the given data
    cur_node = self.head

```

```

count = 0
while cur_node:
    if cur_node.data == data:
        return count
    cur_node = cur_node.next
    count += 1
return -1

# Example usage
cll = CircularLinkedList()
cll.append("John")
cll.append("Ben")
cll.append("Matthew")

print("Original Circular Singly Linked List:")
cll.print_list()

cll.insert_at("Alice", 2)

print("Circular Singly Linked List after inserting 'Alice' at position 2:")
cll.print_list()

print("Index of node Ben:", cll.index("Ben"))

cll.delete_at(2)

print("Circular Singly Linked List after deleting 'Ben' at position 2:")
cll.print_list()

Output:
Original Circular Singly Linked List:
John -> Ben -> Matthew -> (head)

Circular Singly Linked List after inserting 'Alice' at position 2:
John -> Alice -> Ben -> Matthew -> (head)

Index of node Ben: 1

Circular Singly Linked List after deleting 'Ben' at position 2:
John -> Alice -> Matthew -> (head)

```

Summary:

Method	Time Complexity	Space Complexity	Explanation
prepend()	O(1)	O(1)	Adding a node at the start only updates head and tail pointers in constant time.
append()	O(1)	O(1)	Adding a node at the end only updates tail and next pointers in constant time.
insert_at()	O(n)	O(1)	Inserting at a specific position requires traversing the list, which takes linear time.
delete()	O(n)	O(1)	Finding and deleting a node involves traversal to locate the node, which is linear.
delete_at()	O(n)	O(1)	Deleting a node at a specific index requires traversing the list to the index, which is linear.
get()	O(n)	O(1)	Fetching data at a position requires traversal to that position, taking linear time.
index()	O(n)	O(1)	Finding the index of a node requires traversing the list to locate the data, taking linear time.

Exercises 1 - 7 (Fill in the Blank)

Exercise 1

Fill in the blanks to complete the prepend() method.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
```

```

        self.size = 0
        self.current = None

def prepend(self, data):
    new_node = Node(data)
    if self.head == None:
        _____
        _____
        new_node.next = new_node
        self.size += 1
    else:
        _____
        _____
        _____
        self.size += 1

```

Hint:

Consider what should happen when the list is empty (no head or tail). If the list isn't empty, make sure the new node points to the current head, and the current tail connects back to this new head to maintain the circular structure.

Expected Answer:

```

def prepend(self, data):
    new_node = Node(data)
    if self.head == None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    else:
        new_node.next = self.head
        self.tail.next = new_node
        self.head = new_node
        self.size += 1

```

Exercise 2

All of the methods to add a node need to be updated to ensure that the circular linking is created and maintained as elements are added to the linked list. Fill in the blanks to complete the append() method.

```

class Node:
    def __init__(self, data):

```

```

        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            _____
            _____
            _____
            self.size += 1
        else:
            new_node.next = self.head
            _____
            _____
            self.size += 1

```

Expected Answer:

```

def append(self, data):
    new_node = Node(data)
    if self.head == None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    else:
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

```

Exercise 3

Fill in the blanks to complete the insert_at() method.

```

class Node:
    def __init__(self, data):
        self.data = data

```

```

        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def insert_at(self, index, data):
        if index == 0:
            _____
            return
        if index == self.size - 1:
            _____
            return
        new_node = Node(data)
        _____
        count = 0
        while cur_node:
            if count == index - 1:
                temp = cur_node.next
                new_node.next = temp
                _____
                self.size += 1
                return
            _____
            count += 1

```

Hint:

Think about different scenarios: if inserting at the beginning, use the prepend method; if at the end, use append. For inserting in the middle, find the node at the specified position and adjust the pointers to fit the new node.

Expected Answer:

```

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)

```

```

cur_node = self.head
count = 0
while cur_node:
    if count == index - 1:
        temp = cur_node.next
        new_node.next = temp
        cur_node.next = new_node
        self.size += 1
    return
cur_node = cur_node.next
count += 1

```

Exercise 4

The delete() method is updated to ensure that while attempting to find the node saving the data the circle is not traversed forever. Fill in the blanks to complete the delete method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def delete(self, data): # delete a node with a specific value
        cur_node = self.head
        if cur_node and cur_node.data == data: # If the node to be
        deleted is the head node
            self.head = cur_node.next
            _____
            cur_node = None
            self.size -= 1
            return
        prev = None
        while cur_node.data != data: # Traverse the linked list to
        find the node with the specified value
            _____
            cur_node = cur_node.next
            if cur_node == self.head: # prevent infinite loop

```

```

        return
    if cur_node == self.tail: # If the node to be deleted is
the tail node
        self.tail = prev
    _____
    self.size -= 1
    return
    _____
cur_node = None
self.size -= 1

```

Expected Answer:

```

def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node and cur_node.data == data:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head: # prevent infinite loop
            return
    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

```

Exercise 5

The `delete_at()` method is updated to ensure that if an index larger than the size of the list is inputted the traversal will not return to the start and continue to delete a node that should not be. Fill in the blanks to complete the `delete_at()` method.

```
class Node:
```

```

def __init__(self, data):
    self.data = data
    self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def delete_at(self, index):
        if self.head is None:
            return
        cur_node = self.head
        if index == 0:
            if self.head == self.tail:
                _____ # Update head
                self.tail = None
            else:
                _____ # Update head
                self.tail.next = self.head
            cur_node = None
            self.size -= 1
            return
        count = 0
        prev = None
        while count < index:
            prev = cur_node
            _____ # Move current
            count += 1
            if cur_node == self.head: # prevent infinite loop
                return
            _____ # Update the previous node's next pointer
        if cur_node == self.tail:
            _____ # Update tail
            cur_node = None
            self.size -= 1

```

Expected Answer:

```

def delete_at(self, index):
    if self.head is None:
        return

```

```

cur_node = self.head
if index == 0:
    if self.head == self.tail:
        self.head = None
        self.tail = None
    else:
        self.head = cur_node.next
        self.tail.next = self.head
    cur_node = None
    self.size -= 1
return
count = 0
prev = None
while count < index:
    prev = cur_node
    cur_node = cur_node.next
    count += 1
    if cur_node is self.head: # prevent infinite loop
        return
prev.next = cur_node.next
if cur_node == self.tail:
    self.tail = prev
cur_node = None
self.size -= 1

```

Exercise 6

Fill in the blanks to complete the get() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def get(self, position):
        if position < 1 or position > self.size:
            return None

```

```
current = self.head
count = 1
while count < position:  
    _____  
    _____  
    _____
```

Expected Answer:

```
def get(self, position):
    if position < 1 or position > self.size:
        return None
    current = self.head
    count = 1
    while count < position:
        current = current.next
        count += 1
    return current.data
```

Exercise 7

Fill in the blanks to complete the index() method.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def index(self, data):
        cur_node = self.head
        count = 0
        while cur_node:  
            _____  
            return count  
            _____  
            _____  
        return -1
```

Expected Answer:

```
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
```

Current traversal of the list

The set_current() method is used to reset the current variable to reference the start of the list.

```
def set_current(self):
    self.current = self.head
```

The set_current_to() sets the current node to a node at the given index.

```
def set_current_to(self, index):
    self.current = self.head
    for i in range(index):
        self.current = self.current.next
```

The next() method is used to traverse the list by setting the current variable to the node next to the node it is currently set to.

```
def next(self):
    if self.current == None:
        return
    self.current = self.current.next
```

The get_current() method implements the retrieval of the data stored in the current node through a simple return statement.

```
def get_current(self):
    return self.current.data
```

Exercises 8 - 12 (Images)

Ex 8

The below image is a valid representation of a circular singly linked list.

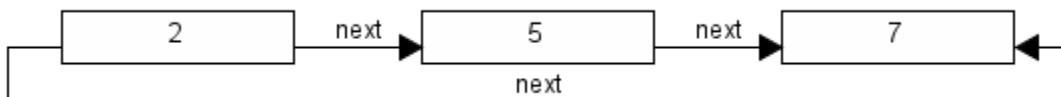


True

False

Ex 9

The below image is a valid representation of a circular singly linked list.



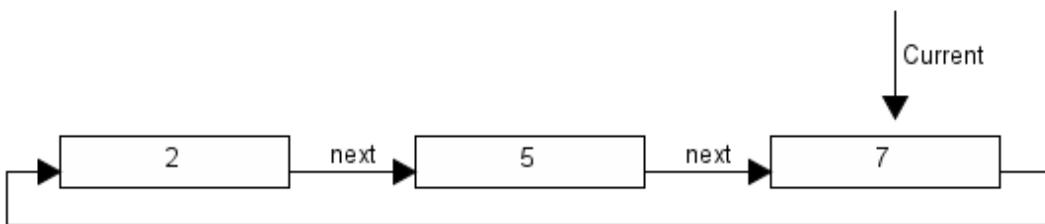
True

False

Explanation: The loop arrow is going the wrong direction.

Ex 10

Given the below image, what would be a valid way to move the 'current' variable to the node with data '2'?



next()

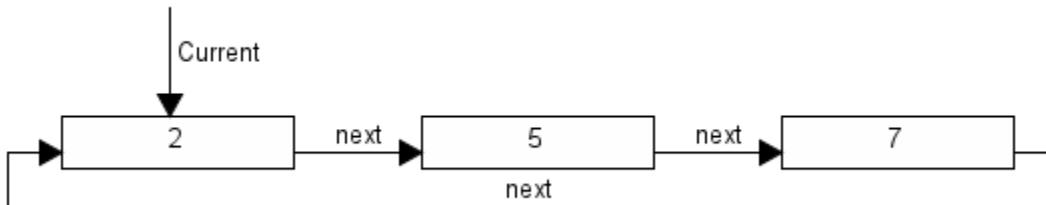
set_current()

set_current_to(0)

previous()

Ex 11

Given the below image, what would be a valid way to move the 'current' variable to the node with data '7'?



previous()

set_current()

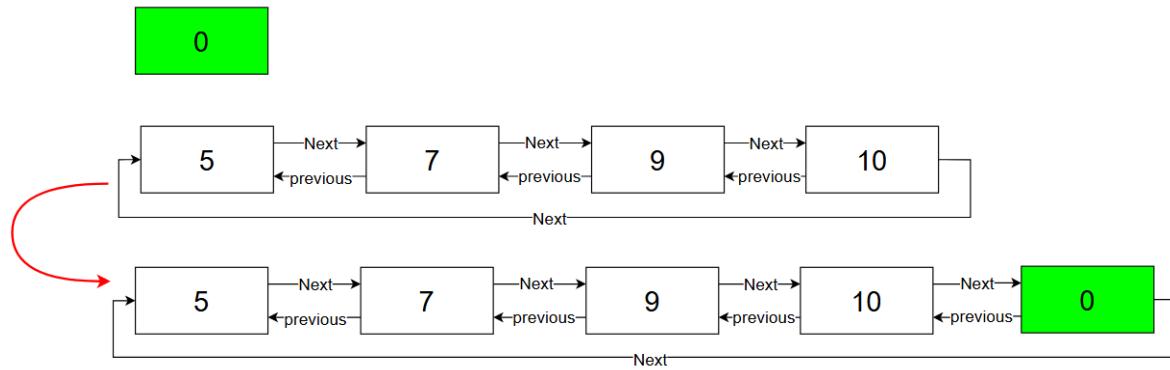
set_current_to(1)

next().next()

Explanation: next().next() is the only valid answer. The previous() method might work in a circular doubly linked list, but in this example we don't have prev pointers to use as it is a circular singly linked list.

Ex 12

Given the below image, which linked list method does this illustrate?



append(0)

set_current()

- | |
|-------------------------------------|
| <input type="checkbox"/> next() |
| <input type="checkbox"/> prepend(0) |

Exercises 13 - 44 (Coding)

Story:

Annabel is a software development intern at CoolGameCompany. She is tasked with making a small part of a multiplayer game the company is designing. This game is meant to be based off of the card game '21' and Annabel knows the best way to begin is by making a circular linked list. This is a big task for an intern and doing well means that she might get a promotion! She isn't very used to Python syntax, so she has asked you for assistance!

Ex 13

Create the Node class for a Singly Circular Linked List.

Given Code:

```
class Node:
    def __init__(self, data):
        # Your code here
```

Expected input:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Ex 14

Create a Singly Circular Linked List class and `__init__()` method which keeps track of head, tail and size of a list and current node called `cur_node`.

Given Code:

```
class CircularLinkedList:
    # Your code here
```

Expected input:

```
class CircularLinkedList:
    def __init__(self):
        self.head = None
```

```
self.tail = None
self.size = 0
self.cur_node = None
```

Ex 15

Create the append() method for the circular Linked list class. Remember to update that size.

Hint: Consider dealing with the case where there are no nodes, before dealing with adding the node to a list of nodes.

Given Code:

```
class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        # Your code here
```

Expected input:

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1
```

Ex 16

Create the prepend() method for the circular linked list class.

Hint: Consider dealing with the case where there are no nodes, before dealing with adding the node to a list of nodes.

Given Code:

```
class CircularLinkedList:
    def __init__(self):
```

```

self.head = None
self.tail = None
self.size = 0
self.cur_node = None

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1

def prepend(self, data):
    # Your code here

```

Expected input:

```

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

```

Ex 17

Create the insert_at() method for the circular linked list class.

Hint: consider checking if the index is for the first or last item, and call the append/prepend methods you have already created for those cases.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

```

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    # Your code here

```

Expected input:

```

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next

```

```
count += 1
```

Ex 18

Create the delete() method for the circular linked list class.

There are a couple of things you should consider:

- What if the head and tail are the same node?
- What does the tail need to do if the deleted node is the head?
- What does the head need to do if the deleted node is the tail?

Hint: First check if the list is empty. Next check if the head node is the node to delete, then loop through the list to find the node to delete and the prev node. Lastly, check if the node to delete is tail or not before dealing with the delete operation.

Given Code:

```
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.cur_node = None  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node  
        self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node  
        self.head = new_node  
        self.size += 1
```

```

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data):
    # Your code here

```

Expected input:

```

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head:
            return

    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head

```

```

cur_node = None
self.size -= 1
return

prev.next = cur_node.next
cur_node = None
self.size -= 1

```

Ex 19

Create the delete_at() method for the circular linked list class.

The shape of this method is very similar to how you did the delete method. Check the hints if you want the key steps highlighted again.

Hint: First check if the list is empty. Next check if the head node is the node to delete, then loop through the list to find the node to delete and the prev node. Lastly, check if the node to delete is tail or not before dealing with the delete operation.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head

```

```

self.tail.next = new_node
self.head = new_node
self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head:
            return

    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None

```

```

    self.size -= 1
    return

    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index):
    # Your code here

```

Expected input:

```

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
    cur_node = None
    self.size -= 1

```

Ex 20

Create the get() method for the circular linked list class.

Hint: Remember to check for invalid indexes.

Given Code:

```

class CircularLinkedList:
    def __init__(self):

```

```

self.head = None
self.tail = None
self.size = 0
self.cur_node = None

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

```

```

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head:
            return

    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None

```

```

while count < index:
    prev = cur_node
    cur_node = cur_node.next
    count += 1
    if cur_node is self.head:
        return
    prev.next = cur_node.next
if cur_node == self.tail:
    self.tail = prev
cur_node = None
self.size -= 1

def get(self, index):
    # Your code here

```

Expected input:

```

def get(self, index):
    if index < 0 or index >= self.size:
        return None
    current = self.head
    count = 0
    while count < index:
        current = current.next
        count += 1
    return current.data

```

Ex 21

Create the index() method for the circular linked list class.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node

```

```

    self.tail = new_node
    self.size += 1

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
            cur_node = None
            self.size -= 1
        return

```

```

prev = None
while cur_node.data != data:
    prev = cur_node
    cur_node = cur_node.next
    if cur_node == self.head:
        return

if cur_node == self.tail:
    self.tail = prev
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

prev.next = cur_node.next
cur_node = None
self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
            cur_node = None
            self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head: # prevent infinite loop
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
    cur_node = None
    self.size -= 1

def get(self, index):
    if index < 0 or index >= self.size:
        return None

```

```

current = self.head
count = 0
while count < index:
    current = current.next
    count += 1
return current.data

def index(self, data):
    # Your code here

```

Expected input:

```

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

```

Ex 22

Create the set_current() method for the circular linked list class. The method should set the current head to be the cur_node.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

```

```

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev = None
    while cur_node.data != data:

```

```

prev = cur_node
cur_node = cur_node.next
if cur_node == self.head:
    return

if cur_node == self.tail:
    self.tail = prev
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

prev.next = cur_node.next
cur_node = None
self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
            cur_node = None
            self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
        cur_node = None
        self.size -= 1

def get(self, index):
    if index < 0 or index >= self.size:
        return None
    current = self.head
    count = 0
    while count < index:

```

```

        current = current.next
        count += 1
    return current.data

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def set_current(self):
    # Your code here

```

Expected input:

```
def set_current(self):
    self.cur_node = self.head
```

Ex 23

Create the next() method for the circular linked list class. The method should set the cur_node to the next node in the list.

Hint: Make sure the cur_node is not None.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

```

```

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
    return

prev = None

```

```

while cur_node.data != data:
    prev = cur_node
    cur_node = cur_node.next
    if cur_node == self.head:
        return

if cur_node == self.tail:
    self.tail = prev
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

prev.next = cur_node.next
cur_node = None
self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
        cur_node = None
        self.size -= 1

def get(self, index):
    if index < 0 or index >= self.size:
        return None
    current = self.head
    count = 0

```

```

while count < index:
    current = current.next
    count += 1
return current.data

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def set_current(self):
    self.cur_node = self.head

def next(self):
    # Your code here

```

Expected input:

```

def next(self):
    if self.cur_node is None:
        return
    self.cur_node = self.cur_node.next

```

Ex 24

Create the get_current() method for the circular linked list class. The method returns the data of the current node.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
        return

```

```

new_node.next = self.head
self.tail.next = new_node
self.tail = new_node
self.size += 1

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
            cur_node = None
    else:
        cur_node = cur_node.next

```

```

    self.size -= 1
    return

prev = None
while cur_node.data != data:
    prev = cur_node
    cur_node = cur_node.next
    if cur_node == self.head:
        return

    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return

    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev
    cur_node = None
    self.size -= 1

def get(self, index):

```

```

if index < 0 or index >= self.size:
    return None
current = self.head
count = 0
while count < index:
    current = current.next
    count += 1
return current.data

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def set_current(self):
    self.cur_node = self.head

def next(self):
    if self.cur_node is None:
        return
    self.cur_node = self.cur_node.next

def get_current(self):
    # Your code here

```

Expected input:

```

def get_current(self):
    if self.cur_node is None:
        return None
    return self.cur_node.data

```

Ex 25

Story:

Now that we have created the data structure, we need to work on the objects the list will store. Because this game is based on a card game, the brief given to Annabel states that a Card class would be best suited for this solution.

Help Annabel create a class Card with the method `__init__` that takes suit and value and initializes them .

Create the method `__repr__` that returns the string in form value of suit.

The string should be like:

“6 of Diamonds”

Where “6” is a value and “Diamonds” is a suit, both of them have the type string.

Given code:

```
class Card:  
    def __init__(self, suit, value):  
        # Your code here  
  
    def __repr__():  
        # Your code here
```

Expected input:

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__():  
        return self.value + " of " + self.suit
```

Ex 26

Story:

Now that we have a Card class, a Deck class should be made next that stores Card objects.

Create a class Deck with the method `__init__` that creates a circular linked list called cards and calls a method called build without any parameter (we will implement this `build()` method later).

Given Code:

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__():  
        return self.value + " of " + self.suit  
  
class Deck:  
    # Your code here
```

Expected input:

```
class Deck:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.build()
```

Ex 27

Story:

Annabel needs to implement a standard deck of playing cards using the code we have made so far.

Create a method called *build()* for creating each of the cards with values "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" for the suits "Hearts", "Diamonds", "Clubs", "Spades" and adding them to the cards list.

Hint: We used 2 for loops (nested).

Given Code:

```
class Card:
    def __init__(self, suit, value):
        self.suit = suit
        self.value = value
    def __repr__(self):
        return self.value + " of " + self.suit

class Deck:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.build()

    def build(self):
        # Your code here
```

Expected input:

```
def build(self):
    for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:
        for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:
            self.cards.append(Card(suit, value))
```

Ex 28

Story:

Before every game starts, the deck of cards is shuffled to make sure each game is fair. Annabel has been tasked with implementing this.

Help her by creating a method called *shuffle()* that for as many cards as there are in the deck takes a random card and inserts it at the index of the for loop counter removing the original version of the card.

Note: To select a random car use the random.randint() function by adding "import random" to the top of the code document and using the function as random.randint("lowest random value", "highest random value").

Given Code:

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__(self):  
        return self.value + " of " + self.suit  
  
class Deck:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.build()  
  
    def build(self):  
        for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:  
            for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:  
                self.cards.append(Card(suit, value))  
  
    def shuffle(self):  
        # Your code here
```

Expected input:

```
def shuffle(self):  
    for i in range(0, self.cards.size):  
        index = random.randint(0, self.cards.size - 1)  
        card = self.cards.get(index)  
        self.cards.insert_at(i, card)  
        self.cards.delete_at(index + 1)
```

Ex 29**Story:**

Drawing cards is an integral part of any card game. Annabel knows that we can simulate this by returning and removing the card at the top of the list.

Create a method *draw(self)* that takes no parameters and returns the card at the start of the deck (index 0) and removes it from the deck.

Hint: We used the *get* and *delete_at* methods.

Given Code:

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__(self):
```

```

return self.value + " of " + self.suit

class Deck:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.build()

    def build(self):
        for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:
            for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:
                self.cards.append(Card(suit, value))

    def shuffle(self):
        for i in range(0, self.cards.size):
            index = random.randint(0, self.cards.size - 1)
            card = self.cards.get(index)
            self.cards.insert_at(i, card)
            self.cards.delete_at(index + 1)

    def draw(self):
        # Your code here

```

Expected input:

```

def draw(self):
    card = self.cards.get(0)
    self.cards.delete_at(0)
    return card

```

Ex 30

Story:

Annabel has realised that she has forgotten to add a function to print all items in a given list. This method is important to functionalities we are yet to implement!

Go back to the circular linked list and create a method `print_list()` that prints all of the nodes in the list once on separate lines.

Given Code:

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node

```

```

        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
        return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None

```

```

else:
    self.head = cur_node.next
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

prev = None
while cur_node.data != data:
    prev = cur_node
    cur_node = cur_node.next
    if cur_node == self.head:
        return

if cur_node == self.tail:
    self.tail = prev
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

prev.next = cur_node.next
cur_node = None
self.size -= 1

def delete_at(self, index):
    if self.head is None:
        return
    cur_node = self.head
    if index == 0:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count < index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
    prev.next = cur_node.next
    if cur_node == self.tail:
        self.tail = prev

```

```

cur_node = None
self.size -= 1

def get(self, index):
    if index < 0 or index >= self.size:
        return None
    current = self.head
    count = 0
    while count < index:
        current = current.next
        count += 1
    return current.data

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def set_current(self):
    self.cur_node = self.head

def next(self):
    if self.cur_node is None:
        return
    self.cur_node = self.cur_node.next

def get_current(self):
    if self.cur_node is None:
        return None
    return self.cur_node.data

def print_list(self):
    # Your code here

```

Expected input:

```

def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
    if cur_node == self.head:
        break

```

Ex 31

Create a method **show()** that takes no parameters and prints every card in the deck.

The output should be like:

```
A of Hearts  
2 of Hearts  
3 of Hearts  
4 of Hearts  
. . .
```

You may use the `print_list` method in the `CircularLinkedList` class.

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__(self):  
        return self.value + " of " + self.suit  
  
class Deck:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.build()  
  
    def build(self):  
        for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:  
            for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:  
                self.cards.append(Card(suit, value))  
  
    def shuffle(self):  
        for i in range(0, self.cards.size):  
            index = random.randint(0, self.cards.size - 1)  
            card = self.cards.get(index)  
            self.cards.insert_at(i, card)  
            self.cards.delete_at(index + 1)  
  
    def draw(self):  
        card = self.cards.get(0)  
        self.cards.delete_at(0)  
        return card  
  
# Your code here
```

Expected input:

```
def show(self):  
    self.cards.print_list()
```

Ex 32

Story:

Annabel wants to make sure that the code you have made together works as expected. She has given you a list of instructions to complete in the main code that will prove that all of the functions we have made runs correctly.

1. Create an instance of the Deck class.
2. Display the deck size before shuffling.
3. Then, display the initial state of the deck after shuffling it. Next, display the deck size after shuffling.
4. Finally, display the shuffled deck.

The output should be like:

Deck size before shuffling: 22

Deck size after shuffling: 22

```
class Card:  
    def __init__(self, suit, value):  
        self.suit = suit  
        self.value = value  
    def __repr__(self):  
        return self.value + " of " + self.suit  
  
class Deck:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.build()  
  
    def build(self):  
        for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:  
            for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:  
                self.cards.append(Card(suit, value))  
  
    def shuffle(self):  
        for i in range(0, self.cards.size):  
            index = random.randint(0, self.cards.size - 1)  
            card = self.cards.get(index)  
            self.cards.insert_at(i, card)  
            self.cards.delete_at(index + 1)  
  
    def draw(self):  
        card = self.cards.get(0)  
        self.cards.delete_at(0)  
        return card  
  
    def show(self):  
        self.cards.print_list()
```

Your code here

Expected input:

```
deck = Deck()  
print("Deck size before shuffling:", deck.cards.size)  
deck.show()  
deck.shuffle()  
print("Deck size after shuffling:", deck.cards.size)  
deck.show()
```

Ex 33

Story:

Each player will have their own cards as well! The brief also says we need to have a data store for each player and the cards they have in their hand in a given round.

Create a class Hand that will act as the hand of a player, create an `__init__` method (takes no parameter) that creates a linked list called cards and the variables stand and bust both initialized as false.

Given Code:

class Hand:

```
def __init__(self):  
    # Your code here
```

Expected input:

class Hand:

```
def __init__(self):  
    self.cards = CircularLinkedList()  
    self.stand = False  
    self.bust = False
```

Ex 34

Create the methods add() and show() to the Hand class with add taking a card input and adding it to the players stored cards and show printing all of their stored cards.

Hint: You may need to use the `print_list()` method for one of these functions.

Given Code:

class Hand:

```
def __init__(self):  
    self.cards = CircularLinkedList()  
    self.stand = False
```

```

self.bust = False

def add(self, card):
    # Your code here

def show(self):
    # Your code here

```

Expected input:

```

def add(self, card):
    self.cards.append(card)

def show(self):
    self.cards.print_list()

```

Ex 35

Create the score() method for the hand class that will be used to add up the total score of the cards and return it. When adding the values are:

"A" = 11

"J", "Q", "K" = 10

All numbers are their own value as an integer.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        # Your code here

```

Expected input:

```

def score(self):
    score = 0
    for i in range(0, self.cards.size()):
        card = self.cards.get(i)
        if card is None:
            continue
        if card.value == "A":

```

```

        score += 11
    elif card.value in ["J", "Q", "K"]:
        score += 10
    else:
        score += int(card.value)
return score

```

Ex 36

Now that the building blocks are complete it's time to construct the game, first create a prompt that asks for the number of players "*Enter the number of players (1-4):*" and with the response create a list called players that stores each of their hands. Before a hand is added to the list make sure it is dealt two cards.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        score = 0
        for i in range(0, self.cards.size()):
            card = self.cards.get(i)
            if card is None:
                continue
            if card.value == "A":
                score += 11
            elif card.value in ["J", "Q", "K"]:
                score += 10
            else:
                score += int(card.value)
        return score

```

Your code here

Expected input:

```

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()

```

```

player.add(deck.draw())
player.add(deck.draw())
players.append(player)

```

Ex 37

Now create a separate hand called dealer and add two cards to it. In the game '21' all of the players are playing against the dealer, the dealer plays after all of the players and will be controlled by the computer.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        score = 0
        for i in range(0, self.cards.size()):
            card = self.cards.get(i)
            if card is None:
                continue
            if card.value == "A":
                score += 11
            elif card.value in ["J", "Q", "K"]:
                score += 10
            else:
                score += int(card.value)
        return score

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)
# Your code here

```

Expected input:

```
dealer = Hand()
```

```
dealer.add(deck.draw())
dealer.add(deck.draw())
```

Ex 38

Now it is time to handle the gameplay. However to demonstrate the use of the current node use only the set_current() method to access players. To begin, set the current node to the beginning of the list, create a variable 'i' that will be used in print statements to denote the number of the player and a boolean game_over and set it to False.

Given Code:

```
class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        score = 0
        for i in range(0, self.cards.size()):
            card = self.cards.get(i)
            if card is None:
                continue
            if card.value == "A":
                score += 11
            elif card.value in ["J", "Q", "K"]:
                score += 10
            else:
                score += int(card.value)
        return score

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())
# Your code here
```

Expected input:

```
players.set_current()  
i = 1  
game_over = False
```

Ex 39

Now create a loop that will continue while the game is not over. Within the loop, set game_over to true so that once all players have played, the loop exits. Then in the loop, create another loop that iterates until the player either stands or busts. In that loop, set game_over to false to prevent the outer loop ending.

Given Code:

```
class Hand:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.stand = False  
        self.bust = False  
  
    def add(self, card):  
        self.cards.append(card)  
  
    def show(self):  
        self.cards.print_list()  
  
    def score(self):  
        score = 0  
        for i in range(0, self.cards.size()):  
            card = self.cards.get(i)  
            if card is None:  
                continue  
            if card.value == "A":  
                score += 11  
            elif card.value in ["J", "Q", "K"]:  
                score += 10  
            else:  
                score += int(card.value)  
        return score  
  
num_players = int(input("Enter the number of players (1-4): "))  
players = CircularLinkedList()  
for i in range(0, num_players):  
    player = Hand()  
    player.add(deck.draw())  
    player.add(deck.draw())  
    players.append(player)  
dealer = Hand()  
dealer.add(deck.draw())
```

```
dealer.add(deck.draw())
players.set_current()
i = 1
game_over = False
```

Your code here

Expected input:

```
while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
```

Ex 40

Now that we have two loops, one to iterate over all players, one to iterate for a single player until they haven't busted or chosen to stand. As long as the player hasn't chosen to stand or bust, we'll keep showing their score as "Player [player_number] score: [player_score]" along with the cards in their hand. The player then needs to decide whether to hit or stand by typing 'h' for hit or 's' for stand.

Given Code:

```
class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        score = 0
        for i in range(0, self.cards.size()):
            card = self.cards.get(i)
            if card is None:
                continue
            if card.value == "A":
                score += 11
            elif card.value in ["J", "Q", "K"]:
                score += 10
            else:
                score += int(card.value)
        return score
```

```

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())
players.set_current()
i = 1
game_over = False

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        # Your code here

```

Expected input:

```

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        print(f"Player {i} score: {player.score()}")
        player.show()
        choice = input("Hit or stand? (h/s): ")

```

Ex 41

Story:

Annabel now needs to work on criteria that causes the game to end, one of these being when a player draws a card and the value of the hand exceeds 21.

Now let's handle the user input during a player's turn. If they entered 'h', then add a card to the player's hand and check if their score is now more than 21. If it is, then they have busted and need to be shown "Player {i} busted!". And if the player chose 's', then it means they chose to stand, so set stand to true so that the loop can be exited.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False

```

```

self.bust = False

def add(self, card):
    self.cards.append(card)

def show(self):
    self.cards.print_list()

def score(self):
    score = 0
    for i in range(0, self.cards.size()):
        card = self.cards.get(i)
        if card is None:
            continue
        if card.value == "A":
            score += 11
        elif card.value in ["J", "Q", "K"]:
            score += 10
        else:
            score += int(card.value)
    return score

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())
players.set_current()
i = 1
game_over = False

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        print(f"Player {i} score: {player.score()}")
        player.show()
        choice = input("Hit or stand? (h/s): ")
        # Your code here

```

Expected input:

```

while not game_over:
    game_over = True

```

```

player = players.get_current()
while not player.stand and not player.bust:
    game_over = False
    print(f"Player {i} score: {player.score()}")
    player.show()
    choice = input("Hit or stand? (h/s): ")
    if choice == "h":
        player.add(deck.draw())
        if player.score() > 21:
            player.bust = True
            print(f"Player {i} busted!")
    elif choice == "s":
        player.stand = True

```

Ex 42

Finally, we need to move to the next player after the inner loop ends. We do this using the `next()` method and incrementing `i`.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):
        self.cards.print_list()

    def score(self):
        score = 0
        for i in range(0, self.cards.size()):
            card = self.cards.get(i)
            if card is None:
                continue
            if card.value == "A":
                score += 11
            elif card.value in ["J", "Q", "K"]:
                score += 10
            else:
                score += int(card.value)
        return score

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):

```

```

player = Hand()
player.add(deck.draw())
player.add(deck.draw())
players.append(player)
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())
players.set_current()
i = 1
game_over = False

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        print(f"Player {i} score: {player.score()}")
        player.show()
        choice = input("Hit or stand? (h/s): ")
        if choice == "h":
            player.add(deck.draw())
            if player.score() > 21:
                player.bust = True
                print(f"Player {i} busted!")
        elif choice == "s":
            player.stand = True
# Your code here

```

Expected input:

```

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        print(f"Player {i} score: {player.score()}")
        player.show()
        choice = input("Hit or stand? (h/s): ")
        if choice == "h":
            player.add(deck.draw())
            if player.score() > 21:
                player.bust = True
                print(f"Player {i} busted!")
        elif choice == "s":
            player.stand = True
    players.next()
    i += 1

```

Ex 43

Now that all of the players have taken their turns it is time for the dealer to take their turn, the dealer is required to continue drawing cards while their score is less than 16. Once it is not less than 16, we check if it is greater than 21. If it is, then set the dealer's 'bust' to true and print "Dealer busted!"

Given Code:

```
class Hand:  
    def __init__(self):  
        self.cards = CircularLinkedList()  
        self.stand = False  
        self.bust = False  
  
    def add(self, card):  
        self.cards.append(card)  
  
    def show(self):  
        self.cards.print_list()  
  
    def score(self):  
        score = 0  
        for i in range(0, self.cards.size()):  
            card = self.cards.get(i)  
            if card is None:  
                continue  
            if card.value == "A":  
                score += 11  
            elif card.value in ["J", "Q", "K"]:  
                score += 10  
            else:  
                score += int(card.value)  
        return score  
  
num_players = int(input("Enter the number of players (1-4): "))  
players = CircularLinkedList()  
for i in range(0, num_players):  
    player = Hand()  
    player.add(deck.draw())  
    player.add(deck.draw())  
    players.append(player)  
dealer = Hand()  
dealer.add(deck.draw())  
dealer.add(deck.draw())  
players.set_current()  
i = 1  
game_over = False  
  
while not game_over:  
    game_over = True  
    player = players.get_current()
```

```

while not player.stand and not player.bust:
    game_over = False
    print(f"Player {i} score: {player.score()}")
    player.show()
    choice = input("Hit or stand? (h/s): ")
    if choice == "h":
        player.add(deck.draw())
        if player.score() > 21:
            player.bust = True
            print(f"Player {i} busted!")
    elif choice == "s":
        player.stand = True
    players.next()
    i += 1

```

Your code here

Expected input:

```

while dealer.score() < 16:
    dealer.add(deck.draw())
if dealer.score() > 21:
    dealer.bust = True
    print("Dealer busted!")

```

Ex 44

Now that all members of the game have played using the current node to traverse the list as was done in the gameplay loop, determine if the player has won, lost or tied with the dealer. Printing "Player", i, "loses!", "Player", i, "ties!" or "Player", i, "wins!", where i is the number of the player.

Use the following conditions:

- If the player busted, then they lose.
- Else, if the dealer busts or the player has a higher score than the dealer, they win.
- If the player's score is less than the dealer's score, then they lose.
- Otherwise, neither player has busted and the scores are equal, so it is a tie.

Given Code:

```

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False

    def add(self, card):
        self.cards.append(card)

    def show(self):

```

```

    self.cards.print_list()

def score(self):
    score = 0
    for i in range(0, self.cards.size):
        card = self.cards.get(i)
        if card is None:
            continue
        if card.value == "A":
            score += 11
        elif card.value in ["J", "Q", "K"]:
            score += 10
        else:
            score += int(card.value)
    return score

num_players = int(input("Enter the number of players (1-4): "))
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())
players.set_current()
i = 1
game_over = False

while not game_over:
    game_over = True
    player = players.get_current()
    while not player.stand and not player.bust:
        game_over = False
        print(f"Player {i} score: {player.score()}")
        player.show()
        choice = input("Hit or stand? (h/s): ")
        if choice == "h":
            player.add(deck.draw())
            if player.score() > 21:
                player.bust = True
                print(f"Player {i} busted!")
        elif choice == "s":
            player.stand = True
    players.next()
    i += 1

while dealer.score() < 16:
    dealer.add(deck.draw())

```

```
if dealer.score() > 21:  
    dealer.bust = True  
    print("Dealer busted!")
```

Your code here

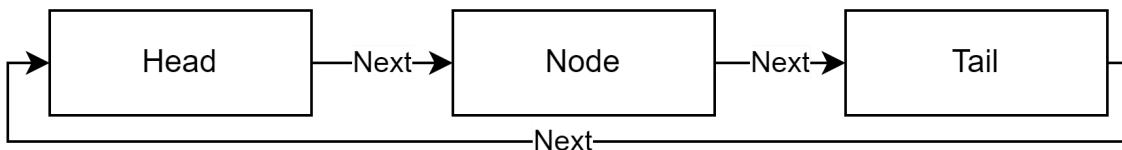
Expected input:

```
players.set_current()  
for i in range(players.size):  
    player = players.get_current()  
    if player.bust:  
        print(f"Player {i + 1} loses!")  
    elif dealer.bust or player.score() > dealer.score():  
        print(f"Player {i + 1} wins!")  
    elif player.score() < dealer.score():  
        print(f"Player {i + 1} loses!")  
    else:  
        print(f"Player {i + 1} ties!")  
players.next()
```

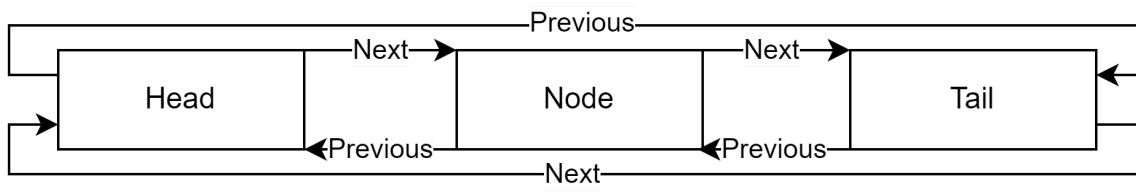
Recap (Lesson 5)

Circular Linked Lists

Circular linked lists are a modification of singly or doubly linked lists, where the tail node points back to the head of the list instead of `None`



In the case of a doubly linked list the Head will also point to the tail instead of None.



This allows the user to choose to work with the list in a circular manner, where they can continually choose to go again and again through the list without terminating the method or function.

Typically however, many methods are still designed to end after one complete look through all the nodes.

Caution is necessary when implementing circular linked lists. Their circular nature can lead to infinite loops if not handled properly. For example, where we usually check for `None` to terminate a traversal, we must instead track whether we've returned to the starting node. Where previously we would check for the next field being none, this can no longer be performed, as circular linked lists have every node connected to each adjacent node.

Implementing nodes and Circular linked lists

Like stated previously, you can choose to use a unidirectional or bidirectional travel method by adding appropriate pointers to your Node class.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

```

The example above uses single directional travel (singly linked list).

The Circular Linked List will commonly keep track of a current node to enable 'current traversal'.

Prepend

The prepend method now needs to ensure that the tail node's `next` field points to this new node.

```

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.head = new_node
    self.size += 1

```

Append

This is the same for the append node, which now needs to keep track of the tail and head as well.

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
        self.size += 1
    return
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node
    self.size += 1

```

Insert and Delete

In circular linked lists, `insert_at()` works similarly to singly and doubly linked lists. The insertion logic adjusts the pointers of the surrounding nodes to accommodate the new node while maintaining the circular structure

```

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head

```

```

count = 0
while cur_node:
    if count == index - 1:
        temp = cur_node.next
        new_node.next = temp
        cur_node.next = new_node
        self.size += 1
    return
    cur_node = cur_node.next
    count += 1

def delete(self, data):
    if self.head is None:
        return
    cur_node = self.head
    if cur_node.data == data:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = cur_node.next
            self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head:
            return

    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

```

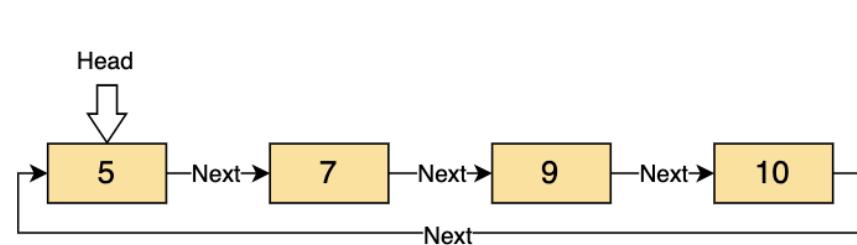
Traversal methods

As stated earlier, circular linked lists also have the set_current, next, (optionally prev) traversal methods. There is no change from what is expected.

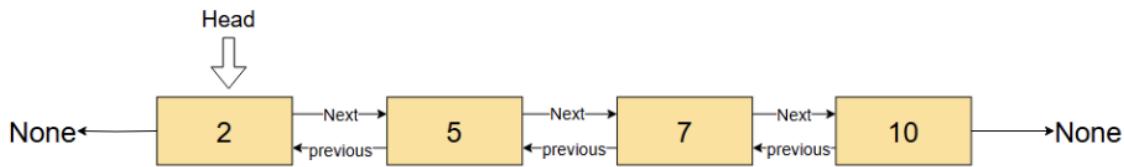
6. Linked List Applications

Here is a summary of the different types of linked lists:

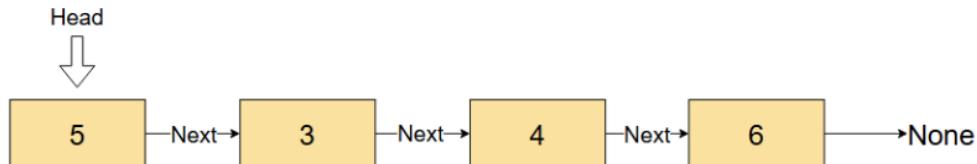
Circular Linked List:



Doubly Linked List:



Singly Linked List:



Now that we have reviewed each of the standard linked list configurations it is time to look at some examples of how linked lists can be used to create useful and efficient programs with ease.

The previous sections have focused on setting up the linked list, the creation of basic methods and then some examples of their use.

In practice, programmers will not create an entire linked list from scratch each time they require one and will likely use a prebuilt linked list to save time and only modify or extend them with custom methods tailored to the program's specific requirements.

We will now apply our linked lists. This will provide more experience using linked lists to their full potential.

6.1 Music Player

Story:

Imagine you are a university student completing their data structures and algorithms course. You have been given an assignment after learning about linked lists by your lecturer! This assignment involves creating a simulated music player using a circular linked list, allowing us to imitate the functionality of playing music tracks seamlessly.

A circular linked list can make sense because we want to utilize the circular nature to allow us to always be able to click on the next music piece and we will never reach the end because the music never ends!

The code below showcases the Node and CircularLinkedList classes we will use.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.current = None  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head == None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node  
        self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head == None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node
```

```

        self.head = new_node
        self.size += 1

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next
            if cur_node == self.head:
                break

    def get(self, index):
        if index < 0 or index >= self.size:
            return
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                return cur_node.data
            cur_node = cur_node.next
            count += 1
            if cur_node == self.head:
                return None

    def index(self, data):
        cur_node = self.head
        count = 0
        while count < self.size:
            if cur_node.data == data:
                return count
            cur_node = cur_node.next
            count += 1
        return -1

    def insert_at(self, index, data):
        if index < 0 or index > self.size:
            return
        if index == 0:
            self.prepend(data)
            return
        if index == self.size:
            self.append(data)
            return
        new_node = Node(data)
        cur_node = self.head
        count = 0
        while True:
            if count == index - 1:
                temp = cur_node.next
                new_node.next = temp
                cur_node.next = new_node
                break
            cur_node = cur_node.next
            count += 1

```

```

        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node.data == data:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
        if cur_node == self.head: # prevent infinite loop
            return
    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index): # delete a node at a specific index
    if index < 0 or index >= self.size:
        return
    cur_node = self.head
    if index == 0:
        if self.size == 1:
            self.head = None
            self.tail = None
    else:
        self.head = cur_node.next
        self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

    count = 0
    prev = None
    while count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1

```

```

        prev.next = cur_node.next
        if cur_node is self.tail:
            self.tail = prev
        cur_node = None
        self.size -= 1

    def set_current(self):
        self.current = self.head

    def set_current_to(self, index):
        self.current = self.head
        for i in range(index):
            self.current = self.current.next

    def next(self):
        if self.current == None:
            return False
        self.current = self.current.next
        return True

    def get_current(self):
        if self.current:
            return self.current.data

        return None

```

Exercises 1 - 2 (Song Class)

Exercise 1

To begin coding the music player, we are going to create a song class. This song class is going to be the data that we shall store in our nodes. Create a constructor for the `Song` class. Each instance variable should match the corresponding parameter name: `name`, `artist`, `genre`, and `favorite`. Initialize these variables in the constructor.

Given code

```

class Song:
    def __init__(self, name, artist, genre, favorite=False):
        # Your code here

    # Testing
    test_song = Song("test_name", "test_artist", "test_genre", True)
    print(test_song.name)
    print(test_song.artist)
    print(test_song.genre)
    print(test_song.favorite)

```

Expected input

```

class Song:
    def __init__(self, name, artist, genre, favorite=False):
        self.name = name
        self.artist = artist
        self.genre = genre
        self.favorite = favorite

# Testing
test_song = Song("test_name", "test_artist", "test_genre", True)
print(test_song.name)
print(test_song.artist)
print(test_song.genre)
print(test_song.favorite)

```

Expected Output

test_name
 test_artist
 test_genre
 True

Exercise 2

To help with printing in the future. Let's fill out a `__repr__` method. If the song has the following attributes:

- Name: “Fried Circuits”
- Artist: “The Electric Dreamers”
- Genre: “Synthwave”
- Favorite: False

The text printed out should match:

“(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)”

Given code

```

class Song:
    def __init__(self, name, artist, genre, favorite=False):
        self.name = name
        self.artist = artist
        self.genre = genre
        self.favorite = favorite

    def __repr__(self):
        # Your code here

# Testing
test_song = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
print(test_song)

```

Expected input

```

class Song:
    def __init__(self, name, artist, genre, favorite=False):
        self.name = name
        self.artist = artist
        self.genre = genre
        self.favorite = favorite

    def __repr__(self):
        return f"{{self.name}} by {{self.artist}}, Genre: {{self.genre}}, Favorite: {{'Yes' if self.favorite else 'No'}}"

test_song = Song("Fried Circuits", "The Electric Dreamers",
                  "Synthwave")
print(test_song)

```

Expected Output

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

Exercises 3 - 8 (Music Player Class)

Exercise 3

Now, let's create a music player class. To start with, create the constructor. This should initialize a CircularLinkedList to the instance variable play_list.

```

class MusicPlayer:
    def __init__(self):
        # Your code here

    # Testing
player = MusicPlayer()
print(isinstance(player.play_list, CircularLinkedList))

```

Expected input

```

class MusicPlayer:
    def __init__(self):
        self.play_list = CircularLinkedList()

    # Testing
player = MusicPlayer()
print(isinstance(player.play_list, CircularLinkedList))

```

Expected Output

True

Exercise 4

Now, let's create a way to add and print all our songs from the Music Player. The add_song method should call the play_list's append method. While the print_songs method should call the play_list's print_list method.

```
class MusicPlayer:  
    def __init__(self):  
        self.play_list = CircularLinkedList()  
  
    def add_song(self, song):  
        # Your code here  
  
    def print_songs(self):  
        # Your code here  
  
# Testing  
song1 = Song("Fried Circuits", "The Electric Dreamers",  
"Synthwave")  
player = MusicPlayer()  
player.add_song(song1)  
player.print_songs()
```

Expected input

```
class MusicPlayer:  
    def __init__(self):  
        self.play_list = CircularLinkedList()  
  
    def add_song(self, song):  
        self.play_list.append(song)  
  
    def print_songs(self):  
        self.play_list.print_list()  
  
# Testing  
song1 = Song("Fried Circuits", "The Electric Dreamers",  
"Synthwave")  
player = MusicPlayer()  
player.add_song(song1)  
player.print_songs()
```

Expected Output

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

Exercise 5

For extra marks on the assignment, let's add some methods to retrieve songs that we exactly want. The get_song_index should use the play_list's index method, while the get_song method should use the play_list's get method.

```
class MusicPlayer:
```

```

def __init__(self):
    self.play_list = CircularLinkedList()

def add_song(self, song):
    self.play_list.append(song)

def print_songs(self):
    self.play_list.print_list()

def get_song_index(self, song):
    # Your code here

def get_song(self, index):
    # Your code here

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
player = MusicPlayer()
player.add_song(song1)
print(player.get_song_index(song1))
print(player.get_song(0))

```

Expected input

```

class MusicPlayer:
    def __init__(self):
        self.play_list = CircularLinkedList()

    def add_song(self, song):
        self.play_list.append(song)

    def print_songs(self):
        self.play_list.print_list()

    def get_song_index(self, song):
        return self.play_list.index(song)

    def get_song(self, index):
        return self.play_list.get(index)

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
player = MusicPlayer()
player.add_song(song1)
print(player.get_song_index(song1))
print(player.get_song(0))

```

Expected Output

0

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

Exercise 6

Story:

Many music playing apps used on mass today have a favourite's system for a user's favourite song(s), so we are going to implement that as well! We can accomplish this by creating the toggle_favorite method. This method will take in a song, and will first call the get_song_index with that song. If the index is not -1, then that means that the song exists, and we can then get the song with the get_song method and set the favorite variable to be opposite to what it is currently (If it is True, set it to False, if it is False, set it to True). Lastly, return a string based on:

- If successful return the string: "{song.name} was set to favorite"
- If unsuccessful return the string: "{song.name} was not found"

```
class MusicPlayer:  
    def __init__(self):  
        self.play_list = CircularLinkedList()  
  
    def add_song(self, song):  
        self.play_list.append(song)  
  
    def print_songs(self):  
        self.play_list.print_list()  
  
    def get_song_index(self, song):  
        return self.play_list.index(song)  
  
    def get_song(self, index):  
        return self.play_list.get(index)  
  
    def toggle_favorite(self, song):  
        # Your code here  
  
# Testing  
song1 = Song("Fried Circuits", "The Electric Dreamers",  
"Synthwave")  
player = MusicPlayer()  
player.add_song(song1)  
print(player.get_song(0))  
print(player.set_favorite(song1))  
print(player.get_song(0))
```

Expected input

```
class MusicPlayer:  
    def __init__(self):  
        self.play_list = CircularLinkedList()
```

```

def add_song(self, song):
    self.play_list.append(song)

def print_songs(self):
    self.play_list.print_list()

def get_song_index(self, song):
    return self.play_list.index(song)

def get_song(self, index):
    return self.play_list.get(index)

def toggle_favorite(self, song):
    song_index = self.get_song_index(song)

    if song_index != -1:
        song = self.get_song(song_index)
        song.favorite = not song.favorite
        if (song.favorite):
            return f"{song.name} was set to favorite"
        else:
            return f"{song.name} is no longer a favorite"

    return f"{song.name} not found"

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
player = MusicPlayer()
player.add_song(song1)
print(player.get_song(0))
print(player.toggle_favorite(song1))
print(player.get_song(0))

```

Expected Output

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

Fried Circuits was set to favorite

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: Yes)

Exercise 7

To have a proper Music Player, we are going to implement some controls. First off, create the next_song method. This method needs to do two things. It will first call the next method from the play_list. But sometimes that will reach the end, and return None. If this happens, we need to set up the first song by clicking set_current. If you look, our next method in this instance returns True/False in this instance which allows for this. This can also be done in other ways. Additionally, create a get_current_song method. You can return None if there isn't a current song.

```
class MusicPlayer:
```

```

def __init__(self):
    self.play_list = CircularLinkedList()

def add_song(self, song):
    self.play_list.append(song)

def print_songs(self):
    self.play_list.print_list()

def get_song_index(self, song):
    return self.play_list.index(song)

def get_song(self, index):
    return self.play_list.get(index)

def toggle_favorite(self, song):
    song_index = self.get_song_index(song)

    if song_index != -1:
        song = self.get_song(song_index)
        song.favorite = not song.favorite
        if (song.favorite):
            return f"{song.name} was set to favorite"
        else:
            return f"{song.name} is no longer a favorite"

    return f"{song.name} not found"

def next_song(self):
    # Your code here

def get_current_song(self):
    # Your code here

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
player = MusicPlayer()
player.add_song(song1)
print(player.get_song(0))
print(player.toggle_favorite(song1))
print(player.get_song(0))
print(player.toggle_favorite(song1))
print(player.get_song(0))
print(player.toggle_favorite(song2))

```

Expected input

```

class MusicPlayer:
    def __init__(self):

```

```

    self.play_list = CircularLinkedList()

    def add_song(self, song):
        self.play_list.append(song)

    def print_songs(self):
        self.play_list.print_list()

    def get_song_index(self, song):
        return self.play_list.index(song)

    def get_song(self, index):
        return self.play_list.get(index)

    def toggle_favorite(self, song):
        song_index = self.get_song_index(song)

        if song_index != -1:
            song = self.get_song(song_index)
            song.favorite = not song.favorite
            if (song.favorite):
                return f"{song.name} was set to favorite"
            else:
                return f"{song.name} is no longer a favorite"

        return f"{song.name} not found"

    def next_song(self):
        success = self.play_list.next()
        if not success:
            self.play_list.set_current()

    def get_current_song(self):
        return self.play_list.get_current()

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)

print(player.get_current_song())
player.next_song()
print(player.get_current_song())
player.next_song()
print(player.get_current_song())
player.next_song()
print(player.get_current_song())

```

Expected Output

None

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

(Ctrl Alt Love by The Keyboard Warriors, Genre: Indie Pop, Favorite: No)

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, Favorite: No)

Exercise 8

We also have a way to get the next favorite song. Fill out the `next_favorite_song` method so that the next song will either be the next available favorite song or the current song if there are no favorites in the `play_list`. We recommend setting up a counter so that you don't loop through the circular list for infinity. We also recommend taking advantage of the `CircularLinkedList`'s `size` property.

Hint: We set up a while loop that loops until there is a false `current_song` and our song count is equal to or greater than the size of the `play_list` and when we are on a favorite song. Essentially the loop ends when any of these are true.

```
class MusicPlayer:  
    def __init__(self):  
        self.play_list = CircularLinkedList()  
  
    def add_song(self, song):  
        self.play_list.append(song)  
  
    def print_songs(self):  
        self.play_list.print_list()  
  
    def get_song_index(self, song):  
        return self.play_list.index(song)  
  
    def get_song(self, index):  
        return self.play_list.get(index)  
  
    def toggle_favorite(self, song):  
        song_index = self.get_song_index(song)  
  
        if song_index != -1:  
            song = self.get_song(song_index)  
            song.favorite = not song.favorite  
            if (song.favorite):  
                return f"{song.name} was set to favorite"  
            else:  
                return f"{song.name} is no longer a favorite"  
  
        return f"{song.name} not found"  
  
    def next_song(self):  
        success = self.play_list.next()  
        if not success:
```

```

        self.play_list.set_current()

    def get_current_song(self):
        return self.play_list.get_current()

    def next_favorite_song(self):
        # Your Code here

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()

player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

print(player.set_favorite(song3))
player.next_song()
print(player.get_current_song())
player.next_favorite_song()
print(player.get_current_song())

```

Expected input

```

class MusicPlayer:
    def __init__(self):
        self.play_list = CircularLinkedList()

    def add_song(self, song):
        self.play_list.append(song)

    def print_songs(self):
        self.play_list.print_list()

    def get_song_index(self, song):
        return self.play_list.index(song)

    def get_song(self, index):
        return self.play_list.get(index)

    def toggle_favorite(self, song):
        song_index = self.get_song_index(song)

        if song_index != -1:

```

```

        song = self.get_song(song_index)
        song.favorite = not song.favorite
        if (song.favorite):
            return f"{song.name} was set to favorite"
        else:
            return f"{song.name} is no longer a favorite"

    return f"{song.name} not found"

def next_song(self):
    success = self.play_list.next()
    if not success:
        self.play_list.set_current()

def get_current_song(self):
    return self.play_list.get_current()

def next_favorite_song(self):
    current_song = self.get_current_song()
    song_count = self.play_list.size
    counter = 0
    while current_song and counter < song_count and not
current_song.favorite:
        self.play_list.next()
        current_song = self.get_current_song()
        counter += 1

# Testing
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()

player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

print(player.set_favorite(song3))
player.next_song()
print(player.get_current_song())
player.next_favorite_song()
print(player.get_current_song())

```

Expected Output

404 Blues was set to favorite

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, favorite: No)

(404 Blues by The Error Codes, Genre: Blues Rock, favorite: Yes)

To recap, here is our full music player class:

```
class MusicPlayer:
    def __init__(self):
        self.play_list = CircularLinkedList()

    def add_song(self, song):
        self.play_list.append(song)

    def print_songs(self):
        self.play_list.print_list()

    def get_song_index(self, song):
        return self.play_list.index(song)

    def get_song(self, index):
        return self.play_list.get(index)

    def set_favorite(self, song):
        song_index = self.get_song_index(song)

        if song_index != -1:
            song = self.get_song(song_index)
            song.favorite = True
            return f"{song.name} was set to favorite"

        return f"{song.name} not found"

    def toggle_favorite(self, song):
        song_index = self.get_song_index(song)

        if song_index != -1:
            song = self.get_song(song_index)
            song.favorite = not song.favorite
            if (song.favorite):
                return f"{song.name} was set to favorite"
            else:
                return f"{song.name} is no longer a favorite"

        return f"{song.name} not found"

    def next_song(self):
        success = self.play_list.next()
        if not success:
            self.play_list.set_current()
```

```

def get_current_song(self):
    return self.play_list.get_current()

def next_favorite_song(self):
    current_song = self.get_current_song()
    song_count = self.play_list.size
    counter = 0
    while current_song and counter < song_count and not
current_song.favorite:
        self.play_list.next()
        current_song = self.get_current_song()
        counter += 1

```

Note: The code above does not explicitly define the next pointer for the node Song but still works normally. This behavior works because Python allows you to **dynamically** add attributes to objects. When the `next` pointer is assigned during the linking process in the circular linked list, it becomes an attribute of the specific `Song` instance, enabling the linked list functionality.

Now, let's move onto making our program that uses this class!

Exercises 9 - 24 (Creating the program)

Exercise 9

Let's start the implementation of the code and finish the assignment. The `clear()` method should first request for an input. This input should have the text: "Press any button to continue". Then, two new lines should print after the input. We will use this to clean the console slightly after every command.

Given Code:

```

def clear():
    # Your code here

# Testing
clear()

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

# Testing
clear()

```

Expected Output

Press any button to continue

Exercise 10

Next we are going to make a function called `print_choice()` that will print our options (this will represent the 'main menu' of this program). Ensure there is a new line following the options.

The lines one by one are:

1. "Here are your choices:"
2. a - print all songs
3. p - play current song
4. n - next song
5. nf - next favorite song
6. q - quit

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    # Your code here

# Testing
print_choice()
```

Expected input

```
def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print()

# Testing
print_choice()
```

Expected Output

Here are your choices:
a - print all songs
p - play current song
n - next song
nf - next favorite song
q - quit

Exercise 11

Let's get to the fun part! We are going to use 4 songs. Use the song class to make the following and then print them:

1. Name: "Fried Circuits", "The Electric Dreamers", "Synthwave".
2. Name: "Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop".
3. Name: "404 Blues", "The Error Codes", "Blues Rock".
4. Name: "Debugging My Heart", "The Stack Tracers", "Country".

Call each of these song1, song2, song3, song4 in order of them being listed.

Hint: Thanks to our `__repr__` method. We can print these just with the `print()` statement. Like: `print(song_variable_name)`.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")  
  
# Your code here  
  
# Testing
print(song1)
print(song2)
print(song3)
print(song4)
```

Expected input

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
```

```

print("")

song1 = Song("Fried Circuits", "The Electric Dreamers", "Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

# Testing
print(song1)
print(song2)
print(song3)
print(song4)

```

Expected Output

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, favorite: No)
 (Ctrl Alt Love by The Keyboard Warriors, Genre: Indie Pop, favorite: No)
 (404 Blues by The Error Codes, Genre: Blues Rock, favorite: No)
 (Debugging My Heart by The Stack Tracers, Genre: Country, favorite: No)

Exercise 12

Next, let's create a `MusicPlayer` object called `player`. Then add all the songs to this `MusicPlayer` using the appropriate method `add_song()`.

Given Code:

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

# Your code here

# Testing
player.print_songs()

```

Expected input

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

# Testing
player.print_songs()
```

Expected Output

(Fried Circuits by The Electric Dreamers, Genre: Synthwave, favorite: No)
(Ctrl Alt Love by The Keyboard Warriors, Genre: Indie Pop, favorite: No)
(404 Blues by The Error Codes, Genre: Blues Rock, favorite: No)
(Debugging My Heart by The Stack Tracers, Genre: Country, favorite: No)

Exercise 14

We are now going to create a while loop that displays all of our options listed in our print_choice function.

Let's first set up an infinite while loop.

The first thing in this loop you should do is call the print_choice function. Next make an input with the text: "Select Your Option: ". Save the user input to a variable called user_choice.

Then in the loop set a condition where if the lower case of the user_choice is 'q', you break out of the loop.

Hint: You can get the lowercase of a string with the `.lower()` method.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")  
  
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")  
  
player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)  
  
# Your code here
```

Expected input

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")  
  
song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
```

```

song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

```

Exercise 15

Excellent! Now that we have an infinite loop, we can exit the program. The next thing to do is to make sure we can print the songs! Add another condition to the loop, that when the user enters an input that resolves to the lowercase letter 'a', first print all the songs in the current player, and then call the clear function.

Given Code:

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

```

```

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    # Your code here

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

```

Exercise 16

Next cause of action, let's write the conditions for our n option. Follow the same course of action as we have done so far. In the n case we will call the next_song method in the player. Then we will get the returned value and set it to a variable. We will then make a print statement with the text: "{current song name} is ready to play!" and then call clear().

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

# Your code here
```

Expected input

```
def clear():
    input("Press any button to continue")
    print()
    print()
```

```

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

```

Exercise 17

Ok now that we have dealt with our ‘n’ action. Let’s do our ‘nf’ action. Again, set it up like the other conditionals, and then call the music player’s next_favorite_song method. Then get the current song. Now, this could be None, so we have to first check if there is a song, and if so, we will print: “{current song name} is ready to play!”. Otherwise, we will print “There was no song selected to play.” Call the clear() function after either option.

Given Code:

```

def clear():
    input("Press any button to continue")
    print()

```

```

print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

# Your code here

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")

```

```

print("a - print all songs")
print("p - play current song")
print("n - next song")
print("nf - next favorite song")
print("q - quit")
print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

```

Exercise 18

We are now going to do our p command. This command will do a couple of things. First off though, we need to get our current song. If there is a song, you will print: "Now playing: {song.name}". Otherwise you need to print: "No song currently selected. Select 'n' to load a

song". Lastly, regardless of which of the options printed, you need to call the clear() method.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
```

```

        print(f"{current_song.name} is ready to play!")
    else:
        print("There was no song selected to play.")
    clear()

```

Your code here

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")

```

```

    clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

```

Exercise 19

Nice we are looking pretty good, but we have this favorite functionality and no way to set it currently. Let's fix that. First, add a print statement to the print_choice which says: "t - toggle favorite status". Place this print statement after the nf statement.

Given Code:

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    # Your code here
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()

```

```

player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")

```

```

print("n - next song")
print("nf - next favorite song")
print("t - toggle favorite status")
print("q - quit")
print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")

```

```
clear()
```

Exercise 20

Now let's add an option to our loop for our 't' user choice. Do this in the style of the others. If a song is selected, toggle the favorite boolean of the current song. Then if it is now a favorite print: "{current song name} is now {added to} favorites." If it is not a favorite, it should write to the console: "{current song} name is now removed from favorites." If there was no song currently selected, instead write "No song is currently selected." Regardless of what is printed, use the clear function at the end.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()
```

```

if user_choice.lower() == "n":
    player.next_song()
    current_song = player.get_current_song()
    print(f"{current_song.name} is ready to play!")
    clear()

if user_choice.lower() == "nf":
    player.next_favorite_song()
    current_song = player.get_current_song()
    if current_song:
        print(f"{current_song.name} is ready to play!")
    else:
        print("There was no song selected to play.")
    clear()

if user_choice.lower() == "p":
    song = player.get_current_song()
    if song:
        print("Now playing:", song.name)
    else:
        print("No song currently selected. Select 'n' to load a
song.")
    clear()

# Your code here

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("q - quit")
    print("")

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)

```

```

player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:
            player.toggle_favorite(current_song)
            print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
        else:
            print("No song is currently selected.")

        clear()

```

Exercise 21

You are nearly ready to submit this assignment! Now we just want to be able to add new songs to our music player. First off, create a new function named `create_song()`. This function is going to first ask the user for 3 inputs. Each input will be given the following statements:

- “Enter a name for your song: ”
- “Who is the artist? ”
- “What is the genre? ”

After that, create a new `Song` object from these three inputs. The first input will be the first argument, the second input will be the second, and the last input will be the last argument.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("q - quit")
    print("")

# Your code here

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
```

```

player.print_songs()
clear()

if user_choice.lower() == "n":
    player.next_song()
    current_song = player.get_current_song()
    print(f"{current_song.name} is ready to play!")
    clear()

if user_choice.lower() == "nf":
    player.next_favorite_song()
    current_song = player.get_current_song()
    if current_song:
        print(f"{current_song.name} is ready to play!")
    else:
        print("There was no song selected to play.")
    clear()

if user_choice.lower() == "p":
    song = player.get_current_song()
    if song:
        print("Now playing:", song.name)
    else:
        print("No song currently selected. Select 'n' to load a
song.")
    clear()

if user_choice.lower() == "t":
    current_song = player.get_current_song()
    if current_song:
        player.toggle_favorite(current_song)
        print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
    else:
        print("No song is currently selected.")

    clear()

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")

```

```

print("t - toggle favorite status")
print("q - quit")
print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()

```

```

if song:
    print("Now playing:", song.name)
else:
    print("No song currently selected. Select 'n' to load a
song.")
    clear()

if user_choice.lower() == "t":
    current_song = player.get_current_song()
    if current_song:
        player.toggle_favorite(current_song)
        print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
    else:
        print("No song is currently selected.")

clear()

```

Exercise 22

Let's allow for the user to be able to cancel their request if it isn't to their liking. Add a while loop to `create_song()` that loops forever. In it, ask the user with an input the following: "Add this song to the playlist? {print the new song object} (y/n)". Then add two conditions if the user enters something that turns into lowercase y, return the new song, otherwise return None if the response is n. If it isn't either of those print "That is not a valid response" and call the clear function.

Given Code:

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    # Your code here

```

```

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:

```

```

        player.toggle_favorite(current_song)
        print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
    else:
        print("No song is currently selected.")

    clear()

```

Expected input

```

def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    while True:
        response = input(f"Add this song to the playlist? {new_song}
(y/n)")

        if response.lower() == 'y':
            return new_song

        if response.lower() == 'n':
            return None

        print("That is not a valid response")
        clear()

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()

```

```

player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:
            player.toggle_favorite(current_song)
            print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
        else:
            print("No song is currently selected.")

        clear()

```

Exercise 23

Now that we have finished our create song function, let's get ready to add it to our loop by adding a print statement to our choices that says: "add - add a new song".

Hint: Add a new line to the print_choice function to display 'add - add a new song' as an option.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    # Your code here
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    while True:
        response = input(f"Add this song to the playlist?
{new_song} (y/n)")

        if response.lower() == 'y':
            return new_song

        if response.lower() == 'n':
            return None

        print("That is not a valid response")
        clear()

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
```

```

player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:
            player.toggle_favorite(current_song)
            print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
        else:
            print("No song is currently selected.")

        clear()

```

Expected input

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("add - add a new song")
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    while True:
        response = input(f"Add this song to the playlist?
{new_song} (y/n)")

        if response.lower() == 'y':
            return new_song

        if response.lower() == 'n':
            return None

        print("That is not a valid response")
        clear()

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
```

```

user_choice = input("Select Your Option: ")

if user_choice.lower() == "q":
    break

if user_choice.lower() == "a":
    player.print_songs()
    clear()

if user_choice.lower() == "n":
    player.next_song()
    current_song = player.get_current_song()
    print(f"{current_song.name} is ready to play!")
    clear()

if user_choice.lower() == "nf":
    player.next_favorite_song()
    current_song = player.get_current_song()
    if current_song:
        print(f"{current_song.name} is ready to play!")
    else:
        print("There was no song selected to play.")
    clear()

if user_choice.lower() == "p":
    song = player.get_current_song()
    if song:
        print("Now playing:", song.name)
    else:
        print("No song currently selected. Select 'n' to load a
song.")
    clear()

if user_choice.lower() == "t":
    current_song = player.get_current_song()
    if current_song:
        player.toggle_favorite(current_song)
        print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
    else:
        print("No song is currently selected.")

    clear()

```

Exercise 24

We can now finish our project by adding the last condition to the loop when the user types anything that evaluates to “add” in lower case. First call the `create_song` function, and save it to a variable, then if a song is returned, add it to the `MusicPlayer` and print: “{song name} has been added to the music player.”. Otherwise print “No song has been added to the music

player." At the end of the condition call the clear function.

Given Code:

```
def clear():
    input("Press any button to continue")
    print()
    print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("add - add a new song")
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    while True:
        response = input(f"Add this song to the playlist?
{new_song} (y/n)")

        if response.lower() == 'y':
            return new_song

        if response.lower() == 'n':
            return None

        print("That is not a valid response")
        clear()

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)
```

```

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":
        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:
            player.toggle_favorite(current_song)
            print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
        else:
            print("No song is currently selected.")

        clear()

# Your code here

```

Expected input

```
def clear():
```

```

input("Press any button to continue")
print()
print()

def print_choice():
    print("Here are your choices: ")
    print("a - print all songs")
    print("p - play current song")
    print("n - next song")
    print("nf - next favorite song")
    print("t - toggle favorite status")
    print("add - add a new song")
    print("q - quit")
    print("")

def create_song():
    name = input("Enter a name for your song: ")
    artist = input("Who is the artist? ")
    genre = input("What is the genre? ")

    new_song = Song(name, artist, genre)
    while True:
        response = input(f"Add this song to the playlist?\n{new_song} (y/n)")

        if response.lower() == 'y':
            return new_song

        if response.lower() == 'n':
            return None

        print("That is not a valid response")
        clear()

song1 = Song("Fried Circuits", "The Electric Dreamers",
"Synthwave")
song2 = Song("Ctrl Alt Love", "The Keyboard Warriors", "Indie Pop")
song3 = Song("404 Blues", "The Error Codes", "Blues Rock")
song4 = Song("Debugging My Heart", "The Stack Tracers", "Country")

player = MusicPlayer()
player.add_song(song1)
player.add_song(song2)
player.add_song(song3)
player.add_song(song4)

while True:
    print_choice()
    user_choice = input("Select Your Option: ")

    if user_choice.lower() == "q":

```

```

        break

    if user_choice.lower() == "a":
        player.print_songs()
        clear()

    if user_choice.lower() == "n":
        player.next_song()
        current_song = player.get_current_song()
        print(f"{current_song.name} is ready to play!")
        clear()

    if user_choice.lower() == "nf":
        player.next_favorite_song()
        current_song = player.get_current_song()
        if current_song:
            print(f"{current_song.name} is ready to play!")
        else:
            print("There was no song selected to play.")
        clear()

    if user_choice.lower() == "p":
        song = player.get_current_song()
        if song:
            print("Now playing:", song.name)
        else:
            print("No song currently selected. Select 'n' to load a
song.")
        clear()

    if user_choice.lower() == "t":
        current_song = player.get_current_song()
        if current_song:
            player.toggle_favorite(current_song)
            print(f"{current_song.name} is now {'added to ' if
current_song.favorite else 'removed from '}favorites.")
        else:
            print("No song is currently selected.")

        clear()

    if user_choice.lower() == "add":
        song = create_song()
        if song:
            player.add_song(song)
            print(f"{song.name} has been added to the music player.")
        else:
            print("No song has been added to the music player.")

        clear()

```

And that is our music player! You can add more to it if you wish, we haven't added the ability to delete songs. Feel free to take the current project and expand it if you want to!

Be careful though, if you delete songs you need to make sure that every action can deal with there being no songs in the music player.

Aside from that, you may notice that our final project doesn't use much of the content we have learned in this chapter.

That is the reality of using data structures. As you get closer to the applications that users interact with, data structures will only matter when performance comes into play.

Before that, you will often not even know what type of data structure is being used. Think for yourself for a second, if you were just given the MusicPlayer, would you know that there is a linked list behind it?

Great data structures should be almost obscured by the time we make real applications.

They are still important to learn though, especially as projects get larger, eventually you need to make your own data structures to be able to optimize the performance of your projects.

6.2 Mark Maker

The next program that we are going to construct is a tool used by school staff to store the test results of students and provide information about class averages and the details of a student's records.

This program aims to digitise testing records, simplifying storage and providing students with easy access to their information.

The code below details a singly linked list and will be used throughout the following exercises:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self): # initialize the linked list  
        self.head = None  
        self.size = 0  
  
    def append(self, data): # add to the end of the list  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            self.size += 1
```

```

        return
    last_node = self.head
    while last_node.next:
        last_node = last_node.next
    last_node.next = new_node
    self.size += 1

def prepend(self, data): # add to the beginning of the list
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
    self.size += 1

def get(self, index): # get the data at a specific index
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return None

def index(self, data): # get the index of a specific value
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def insert_at(self, index, data): # insert a new node at a
specific index
    if index == 0:
        self.prepend(data)
        return
    new_node = Node(data)
    count = 0
    cur_node = self.head
    while cur_node:
        if count == index - 1:
            new_node.next = cur_node.next

```

```

        cur_node.next = new_node
        self.size += 1
        return
    cur_node = cur_node.next
    count += 1

def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node and cur_node.data == data:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node and cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while cur_node and count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def get_size(self):
    return self.size

```

Exercises 1 - 2 (Coding - Grade Class)

Exercise 1

To begin, create the class **Grade** as a node that will store an inputted **name** and **score**. This will be used to represent the individual test results for students.

Given code

Your code here

Expected input

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score
```

Exercise 2

Now create get methods for the **Grade** class to get the name and score. This will improve the use of classes and data structures, allowing our methods to be better structured and have better readability.

Given code

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
    # Your code here  
  
    # Your code here
```

Expected input

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
    def get_name(self):  
        return self.name  
  
    def get_score(self):  
        return self.score
```

Exercises 3 - 7 (Coding - Student Class)

Exercise 3

Create a class **Student** with input **name** that will store the name of a student and a linked list called **grades**.

Given code

```
# Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()
```

Exercise 4

We now need a method to add grades to a student's profile. Create a method **add_grade()** for the **Student** class that will add a grade object to the end of the grades list.

Given code

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        # Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)
```

Exercise 5

To get an idea of the overall quality of a student's work we need to create a method **get_average()** that will determine the average of all of the grades stored for a student and return them.

Note: Calculate the average by adding all of the grades in the list together and then dividing the sum by the size of the list.

Given code

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()
```

```
def add_grade(self, grade):
    self.grades.append(grade)
```

Your code here

Expected input

```
class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size()
```

Exercise 6

To provide a visual representation of all of the grades, create a method **print_grades()** for **Student** class that prints all of the grades in the form: \t Name of subject Score.

Given code

```
class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size()

# Your code here
```

Expected input

```
class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)
```

```

def get_average(self):
    total = 0
    for i in range(self.grades.size()):
        total += self.grades.get(i).score
    return total / self.grades.size()

def print_grades(self):
    for i in range(self.grades.size()):
        print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

```

Test case and sample output:

```

student = Student("Alice")
student.add_grade(Grade("Math", 85))
student.add_grade(Grade("Science", 90))
print(student.get_average())
student.print_grades()

```

Expected Output:

```

87.5
    Math 85
    Science 90

```

Exercise 7

Now create the method **get_grades()** to return the grades linked list in the **Student** class and a **get_name()** to return the name of the student.

Given code

```

class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size()

    def print_grades(self):
        for i in range(self.grades.size()):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

```

```
# Your code here
```

```
# Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.grades.size()):  
            total += self.grades.get(i).score  
        return total / self.grades.size()  
  
    def print_grades(self):  
        for i in range(self.grades.size()):  
            print("\t" + self.grades.get(i).name,  
self.grades.get(i).score)  
  
    def get_grades(self):  
        return self.grades  
  
    def get_name(self):  
        return self.name
```

Test Case:

```
student = Student("Alice")  
student.add_grade(Grade("Math", 85))  
student.add_grade(Grade("Science", 90))  
print("Student Name:", student.get_name())  
  
print("All Grades:")  
grades_list = student.get_grades()  
for i in range(grades_list.get_size()):  
    grade = grades_list.get(i)  
    print("\t", grade.get_name(), grade.get_score())
```

Expected Output:

```
Student Name: Alice  
All Grades:  
    Math 85  
    Science 90
```

Exercises 8 - 14 (Coding - YearGroup Class)

Exercise 8

We now have a student class that stores grades for each student but we need a way of organizing a group of students. Create a class **YearGroup** that takes **year** as an input and stores year and students, where students are represented as a linked list.

Given code

```
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()
```

Exercise 9

Create a method called **add_student()** to add a student to the end of the list.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    # Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)
```

Exercise 10

While we can check how an individual student is performing, we need to be able to check this against their peers. Write a **get_average()** method for the YearGroup class.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)
```

```
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size
```

Exercise 11

We need a quick way to print out the details of all of the students in the year group. Create a `print_students()` method which prints the student's name and their grades.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size
```

```
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)
```

```

def get_average(self):
    total = 0
    for i in range(self.students.size()):
        total += self.students.get(i).get_average()
    return total / self.students.size

def print_students(self):
    for i in range(self.students.size()):
        print(self.students.get(i).name)
        self.students.get(i).print_grades()

```

Test Cases:

```

student1 = Student("Alice")
student1.add_grade(Grade("Math", 85))
student1.add_grade(Grade("Science", 90))

student2 = Student("Bob")
student2.add_grade(Grade("Math", 78))
student2.add_grade(Grade("Science", 88))

student3 = Student("Charlie")
student3.add_grade(Grade("Math", 92))
student3.add_grade(Grade("Science", 81))
student3.add_grade(Grade("History", 87))

# Create a year group and add the students
year_group_2024 = YearGroup(2024)
year_group_2024.add_student(student1)
year_group_2024.add_student(student2)
year_group_2024.add_student(student3)

print("Year Group Average:", year_group_2024.get_average())
print("\nStudents and their grades:")
year_group_2024.print_students()

```

Expected Output:

Year Group Average: 85.7222222222223

Students and their grades:

```

Alice
    Math 85
    Science 90
Bob
    Math 78
    Science 88
Charlie
    Math 92
    Science 81
    History 87

```

Exercise 12

Create a method `get_student()` that will return a student with a given `name` or None if the student does not exist.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size()):  
            total += self.students.get(i).get_average()  
        return total / self.students.size()  
  
    def print_students(self):  
        for i in range(self.students.size()):  
            print(self.students.get(i).name)  
            self.students.get(i).print_grades()  
  
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size()):  
            total += self.students.get(i).get_average()  
        return total / self.students.size()  
  
    def print_students(self):  
        for i in range(self.students.size()):  
            print(self.students.get(i).name)  
            self.students.get(i).print_grades()  
  
    def get_student(self, name):  
        for i in range(self.students.size()):  
            if self.students.get(i).name == name:  
                return self.students.get(i)
```

```
return None
```

Exercise 13

The `get_student()` method is useful for methods retrieving a certain student's details; for more complex methods we need to return the full list object. Create the method `get_students()` to implement this.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size  
  
    def print_students(self):  
        for i in range(self.students.size):  
            print(self.students.get(i).name)  
            self.students.get(i).print_grades()  
  
    def get_student(self, name):  
        for i in range(self.students.size):  
            if self.students.get(i).name == name:  
                return self.students.get(i)  
        return None  
  
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size
```

```

def print_students(self):
    for i in range(self.students.size()):
        print(self.students.get(i).name)
        self.students.get(i).print_grades()

def get_student(self, name):
    for i in range(self.students.size()):
        if self.students.get(i).name == name:
            return self.students.get(i)
    return None

def get_students(self):
    return self.students

```

Exercise 14

Create the method **get_size()** for the linked list to return the size of the list.

Given code

```

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    def get_students(self):
        return self.students

```

Your code here

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size  
  
    def print_students(self):  
        for i in range(self.students.size):  
            print(self.students.get(i).name)  
            self.students.get(i).print_grades()  
  
    def get_student(self, name):  
        for i in range(self.students.size):  
            if self.students.get(i).name == name:  
                return self.students.get(i)  
        return None  
  
    def get_students(self):  
        return self.students  
  
    def get_size(self):  
        return self.students.size
```

Test Cases:

```
student1 = Student("Alice")  
student1.add_grade(Grade("Math", 85))  
student1.add_grade(Grade("Science", 90))  
  
student2 = Student("Bob")  
student2.add_grade(Grade("Math", 78))  
student2.add_grade(Grade("Science", 88))  
  
student3 = Student("Charlie")  
student3.add_grade(Grade("Math", 92))  
student3.add_grade(Grade("Science", 81))  
student3.add_grade(Grade("History", 87))  
  
# Create a year group and add the students  
year_group_2024 = YearGroup(2024)  
year_group_2024.add_student(student1)  
year_group_2024.add_student(student2)
```

```

year_group_2024.add_student(student3)

print("\nRetrieving a specific student by name:")
search_name = "Charlie"
found_student = year_group_2024.get_student(search_name)
if found_student:
    print(f"Found student: {found_student.get_name()}")
    found_student.print_grades()
else:
    print(f"Student {search_name} not found in year group.")

```

Expected Output:

```

Retrieving a specific student by name:
Found student: Charlie
    Math 92
    Science 81
    History 87

```

Exercises 15 - 25 (Coding - LinkedList Class)

The following exercises will utilize all three classes you just created (Grade, Student, and YearGroup). Also, the Node and LinkedList classes are still being used but try to use the methods we made in the other classes before using the LinkedList methods if possible. The three classes you created are below:

```

class Grade:
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_name(self):
        return self.name

    def get_score(self):
        return self.score

class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0

```

```

        for i in range(self.grades.size):
            total += self.grades.get(i).score
        return total / self.grades.size

    def print_grades(self):
        for i in range(self.grades.size):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

    def get_grades(self):
        return self.grades

    def get_name(self):
        return self.name

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    def get_students(self):
        return self.students

    def get_size(self):
        return self.students.size

```

Exercises 26 - 29 (MCQ - YearGroup Time Complexity)

Exercise 26

What is the time complexity of the `add_student()` method in `YearGroup`?

Hint: n is the number of nodes in `self.students`.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    # Other methods
```

- $O(1)$
- $\Theta(n)$
- $O(n^2)$
- $O(2^n)$

Explanation:

`add_student()` calls the `append()` method of the linked list `self.students`, which has a time complexity of $O(n)$, meaning that `add_student()` is also $O(n)$.

Exercise 27

What is the time complexity of the `get_average()` method in `YearGroup`?

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size):  
            total += self.students.get(i).get_average()  
        return total / self.students.size  
  
    # Other methods
```

- $O(1)$
- $O(n)$

- $\Theta(n^2)$
 $O(2^n)$

Explanation:

Using the $f(n)$ notation introduced in section 2.1 of Time Complexity.

- Initializing the variable `total` happens in constant time. $f(n) = 1$
- `get_average()` contains a for loop that iterates n times, where n is the size of `students`. $f(n) = 1 + n$
- In each iteration of the loop, the `get()` method of the linked list `students` is called, which has time complexity $O(n)$. $f(n) = 1 + n * n$
- Getting the retrieved student's average and adding it to `total` takes constant time. $f(n) = 1 + n * (n + 1)$
 - Note: `Student.get_average()` is treated as $O(1)$ here, because its execution time doesn't depend on the size of `students`. If n was instead the size of `Student.grades`, `Student.get_average()` would be $O(n)$.
- When the loop finishes, `total` is divided by `students.size`, and the result is returned. This is a constant-time operation. $f(n) = 1 + n * (n + 1) + 1$
- This expression simplifies to $f(n) = n^2 + n + 2$. The dominating term in this expression is n^2 , so `get_student()` is $O(n^2)$.

In the exercises to follow, we will simplify the expression as we go, following the logic of Time Complexity section 3 (Analyzing Algorithms).

Exercise 28

What is the time complexity of the `print_students()` method in `YearGroup`?

Given code

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    # Other methods
```

- O(1)
 O(n)
 $\Theta(n^2)$
 O(2^n)

Explanation:

Using the $f(n)$ notation introduced in section 2.1 of Time Complexity.

- `print_students()` contains a for loop that iterates n times, where n is the size of `students`. $f(n) = n$
- In each iteration of the loop, the `get()` method of the linked list `students`, which has time complexity $O(n)$, is called twice. $f(n) = n * 2n$
 - Note: Here we are omitting the terms corresponding to constant time operations (`print` and `print_grades`), because we know they won't affect the dominating term.
- This simplifies to $f(n) = 2n^2$, giving us $O(n^2)$.

Exercise 29

What is the time complexity of the `get_student()` method in `YearGroup`?

Given code

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    # Other methods
```

- O(1)

- $O(n)$
 $\Theta(n^2)$
 $O(2^n)$

Explanation:

Using the $f(n)$ notation introduced in section 2.1 of Time Complexity.

- `get_student()` contains a for loop that iterates n times, where n is the size of students. $f(n) = n$
- In each iteration of the loop, the `get()` method of the linked list `students`, which has time complexity $O(n)$, is called twice. $f(n) = n^2$
 - `get()` is called again when the student is found, but we can tell that this won't affect the dominating term.
- This tells us that `get_student()` has a time complexity of $O(n^2)$.

Exercises 30 - 35 (Coding - List-Backed YearGroup Class)

In the previous set of exercises we found that many of the common methods of our `YearGroup` class had time complexities of $O(n^2)$. This means that their execution time will increase relatively quickly as the number of students increases - for every doubling in student count, these methods will take roughly four times longer to execute.

This was because the methods all accessed all of the students in the `YearGroup` by index. We noted earlier that accessing a node of a linked list by its index was an $O(n)$ operation, and that this was one of the areas in which linked lists were weaker than an ordinary list.

By accessing every student in this way, the $O(n)$ operation is repeated n times every time these methods are called, meaning the methods have time complexities of $O(n^2)$.

Python lists can retrieve an element at a particular index in constant time, $O(1)$. This means that for applications where items are frequently accessed by index they will perform better than linked lists.

Exercise 30

We will now spend a few exercises creating a new class, `ListYearGroup`, that has the same functionality as `YearGroup`, but uses a Python `list` instead of a `LinkedList`. Write the `__init__()` method for this class below. Like the `__init__()` method on `YearGroup`, it should take a year as an argument, and initialize variables for the year and the data structure holding the students, which in this case is a `list`.

Given code

```
class ListYearGroup:  
    # Your code here
```

Expected input

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []
```

Exercise 31

Next, implement the `add_student()` method. Just like the equivalent method in `YearGroup`, it should take a student as an argument, and add them to the end of the list.

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    # Your code here
```

Expected input

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)
```

Exercise 32

Next, implement the `get_average()` method. You can assume that the same `Student` and `Grade` classes from before are being used.

Hint: While the implementation of `add_student()` was identical to the one in `YearGroup`, this one won't be.

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []
```

```
def add_student(self, student):
    self.students.append(student)

# Your code here
```

Expected input

```
class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)
```

Explanation

Recall that the `list` class is an iterable. This means that we can loop over its contents directly from our for loop, rather than by using a `range` and accessing them by index as in the case of the `LinkedList` class.

Exercise 33

Using the same approach as was used in `get_average()`, implement `print_students()`, which prints out each student's name and grades.

Given code

```
class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
```

```

        total += student.get_average()
    return total / len(self.students)

# Your code here

```

Expected input

```

class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)

    def print_students(self):
        for student in self.students:
            print(student.name)
            student.print_grades()

```

Test Cases:

```

student1 = Student("Alice")
student1.add_grade(Grade("Math", 85))
student1.add_grade(Grade("Science", 90))

student2 = Student("Bob")
student2.add_grade(Grade("Math", 78))
student2.add_grade(Grade("Science", 88))

student3 = Student("Charlie")
student3.add_grade(Grade("Math", 92))
student3.add_grade(Grade("Science", 81))
student3.add_grade(Grade("History", 87))

list_year_group_2024 = ListYearGroup(2024)

# Add students
list_year_group_2024.add_student(student1)
list_year_group_2024.add_student(student2)
list_year_group_2024.add_student(student3)

# After adding, we can do a simple check by printing all students

```

```

print("Students added to ListYearGroup 2024:")
list_year_group_2024.print_students()

# Calculate and print the overall average for the year group
overall_average = list_year_group_2024.get_average()
print("\nOverall Year Group Average:", overall_average)

```

Expected Output:

Students added to ListYearGroup 2024:

Alice

Math 85
Science 90

Bob

Math 78
Science 88

Charlie

Math 92
Science 81
History 87

Overall Year Group Average: 85.7222222222223

Exercise 34

Implement `get_student()`, which takes a name as an argument and returns the first student with a matching name, or `None` if no matching student is found.

Given code

```

class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)

    def print_students(self):
        for student in self.students:
            print(student.name)

```

```
student.print_grades()  
  
# Your code here
```

Expected input

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for student in self.students:  
            total += student.get_average()  
        return total / len(self.students)  
  
    def print_students(self):  
        for student in self.students:  
            print(student.name)  
            student.print_grades()  
  
    def get_student(self, name):  
        for student in self.students:  
            if student.name == name:  
                return student  
        return None
```

Exercise 35

Complete the `ListYearGroup` class by implementing `get_students()`, which should return the full list of students, and `get_size()`, which should return the size of the list.

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)
```

```

def get_average(self):
    total = 0
    for student in self.students:
        total += student.get_average()
    return total / len(self.students)

def print_students(self):
    for student in self.students:
        print(student.name)
        student.print_grades()

def get_student(self, name):
    for student in self.students:
        if student.name == name:
            return student
    return None

def get_students(self):
    # Your code here

def get_size(self):
    # Your code here

```

Expected input

```

class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)

    def print_students(self):
        for student in self.students:
            print(student.name)
            student.print_grades()

    def get_student(self, name):
        for student in self.students:
            if student.name == name:

```

```

        return student
    return None

def get_students(self):
    return self.students

def get_size(self):
    return len(self.students)

```

Exercises 36 - 39 (MCQ - YearGroup/ListYearGroup Comparison)

We will use the completed `ListYearGroup` class you wrote in the previous exercises and examine its time complexity in terms of the number of students in `students`.

```

class ListYearGroup:
    def __init__(self, year):
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)

    def print_students(self):
        for student in self.students:
            print(student.name)
            student.print_grades()

    def get_student(self, name):
        for student in self.students:
            if student.name == name:
                return student
        return None

    def get_students(self):
        return self.students

    def get_size(self):
        return len(self.students)

```

Exercise 36

We will now examine the time complexity of the methods of the `ListYearGroup` class. What is the time complexity of the `add_student()` method?

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    # Other methods
```

- $\Theta(1)$
- $O(n)$
- $O(n^2)$
- $O(2^n)$

Explanation:

Appending to a `list` is an $O(1)$ operation. We simply add the element to the end of the list without needing to shift other values.

Exercise 37

What is the time complexity of the `get_average()` method of `ListYearGroup`?

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for student in self.students:  
            total += student.get_average()  
        return total / len(self.students)  
  
    # Other methods
```

- $O(1)$
- $\Theta(n)$
- $O(n^2)$
- $O(2^n)$

Explanation:

`get_average()` contains a `for` loop that iterates over all of `students`. Aside from this, every other operation in the method is constant time, so `get_average()` is $O(n)$.

Exercise 38

What is the time complexity of the `print_students()` method of `ListYearGroup`?

Given code

```
class ListYearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = []  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for student in self.students:  
            total += student.get_average()  
        return total / len(self.students)  
  
    def print_students(self):  
        for student in self.students:  
            print(student.name)  
            student.print_grades()  
  
    # Other methods
```

- $O(1)$
- $\Theta(n)$
- $O(n^2)$
- $O(2^n)$

Explanation:

`print_students()` contains a `for` loop that iterates over all of `students`. The operations performed within the loop are all constant time operations, so `print_students()` has time complexity $O(n)$.

Exercise 39

What is the time complexity of the `get_student()` method of `ListYearGroup`?

Given code

```
class ListYearGroup:  
    def __init__(self, year):
```

```

        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for student in self.students:
            total += student.get_average()
        return total / len(self.students)

    def print_students(self):
        for student in self.students:
            print(student.name)
            student.print_grades()

    def get_student(self, name):
        for student in self.students:
            if student.name == name:
                return student
        return None

    # Other methods

```

- O(1)
- O(n)
- O(n^2)
- O(2^n)

Explanation:

Like `print_students()`, `get_student()` contains only a single for loop that iterates over the students in `students`, and which contains only constant time operations. As such, in the worst case scenario it must iterate n times, and so it has $O(n)$ time complexity.

We can now compare the time complexities of the methods of `YearGroup` and `ListYearGroup`. Again, n here refers to the number of students in the year group.

Method	<code>YearGroup</code>	<code>ListYearGroup</code>
<code>add_student()</code>	$O(n)$	$O(1)$
<code>get_average()</code>	$O(n^2)$	$O(n)$

<code>print_students()</code>	$O(n^2)$	$O(n)$
<code>get_student()</code>	$O(n^2)$	$O(n)$
<code>get_students()</code>	$O(1)$	$O(1)$
<code>get_size()</code>	$O(1)$	$O(1)$
Looking at the table above we can see that by selecting a <code>list</code> as the backing data structure for our year group class we dramatically reduced the time complexity of many of its important methods. <code>ListYearGroup</code> will perform much better than <code>YearGroup</code> when being used to manage large year groups.		
Being able to identify when or when not to use a particular data structure is an important skill. In this case, the need to frequently access elements by their index made the <code>LinkedList</code> class a poor choice.		
That being said, a <code>list</code> is not the only other option we could have considered, or even the best option. If we had changed our <code>LinkedList</code> class to be an iterable, we would have been able to loop over its contents directly in exactly the same way as we looped over the list's contents, reducing the time complexity of the slowest methods to $O(n)$. Adding a <code>tail</code> variable to the class would have let us add elements to the rear of the linked list in $O(1)$ time, just like a Python list. With these changes we could have implemented each method using the linked list while achieving the same time complexity as we did with a Python list, with the added benefit of being able to remove elements in $O(1)$ time.		
Alternatively, we could have used a dictionary. Using the students' names as keys and the <code>Student</code> objects themselves as values, <code>get_student()</code> could be made $O(1)$, while the other methods could still be implemented in $O(n)$ time using the <code>values</code> iterator of the dictionary.		

7. Debugging Exercises (10 Exercises)

Ex 1.1

Kharim is creating a singly linked list to store his to-do list in. However, he made a mistake in its initialization. Identify the error present in the following code:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        head = None  
        size = 0
```

- The initialization for the ~~LinkedList~~ class should initialize ~~self.head~~ and ~~self.size~~ instead of just head and size
- head should initialize a single dummy node instead of None
- The size of the new Linked List should be 2
- The code is error-free

Ex 1.2

Now that you have located the errors in the previous question. Correct those errors in the following code

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        head = None  
        size = 0
```

Expected Input

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0
```

Ex 2.1

Sumia is developing a singly linked list to hold items in a scrolling sidebar for a website. She needs to be able to add new elements to the end of the list, but her code contains a mistake. Identify the error(s) present in her code below:

```
def append(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.size += 1  
        return  
    last_node = self.head  
    while last_node.next:  
        last_node = last_node.next  
    last_node = new_node  
    self.size += 1
```

- If the list is empty, size should be set to 0 instead of incrementing by 1
- ~~The new node overwrites the final node instead of being appended to it~~
- The last node should be set to the final node in the list, not the head
- The code is error-free

Ex 2.2

Find and correct the errors present in the code from the previous question:

```
def append(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.size += 1  
        return  
    last_node = self.head  
    while last_node.next:  
        last_node = last_node.next  
    last_node = new_node  
    self.size += 1
```

Expected Input

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.size += 1
        return
    last_node = self.head
    while last_node.next:
        last_node = last_node.next
    last_node.next = new_node
    self.size += 1
```

Ex 3.1

Eris is developing a singly linked list to store different recipes for a cookbook. However, her code to access the contents of the nodes contains an error. Identify the error(s) present in the code below:

```
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        self.size += 1
    return -1
```

- The current node is never moved along, and thus will cause an infinite loop
- The variable count is never incremented, and thus the desired node will never be found
- Accessing a node should not modify the linked list size.
- There is no return statement to return the contents of the node when it is found

Ex 3.2

Find and correct the errors present in the code from the previous question:

```
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
```

```

        return cur_node.data
    cur_node = cur_node.next
    self.size += 1
return -1

```

Expected Input

```

def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
return -1

```

Ex 4.1

Chat logs in a social media service are stored in a singly linked list format. Users can delete messages they have sent by telling the service the contents of the message that they want deleted. Recently, an error was reported where messages were not being deleted properly. Identify the error(s) present in the code to delete a user's message:

```

def delete(self, value):
    current = self.head
    while current.next is None:
        if current.next.message == value:
            current.next = current.next.next
            self.size -= 1
            return
        current = current.next

```

- The function never moves through the linked list, and will become stuck in an infinite loop
- The code never deletes the message, only removing it from the chain, resulting in memory overload
- The size of the linked list is not decreased
- The ~~while loop will only try to find the message if the linked list is empty~~

Ex 4.2

Find and correct the errors present in the code from the previous question:

```

def delete(self, value):
    current = self.head
    while current.next is None:
        if current.next.message == value:
            current.next = current.next.next
            self.size -= 1
            return
    current = current.next

```

Expected Input

```

def delete(self, value):
    current = self.head
    while current.next is not None:
        if current.next.message == value:
            current.next = current.next.next
            self.size -= 1
            return
    current = current.next

```

Ex 5.1

A streaming service uses a doubly linked list to hold the episodes of the shows they offer. When a new season comes out, they need to add the episodes onto the end of the list. However, their programmer made a mistake when writing the code to do so. Identify the error(s) present in the code:

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail = new_node

```

- The line to connect the old tail node with the new with the new node is missing
- The tail node should still be None if the list is one node long
- In the else statement, the first line should be new_node.next = self.tail
- The code is error-free

Ex 5.2

Find and correct the errors present in the code from the previous question:

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail = new_node

```

Expected Input

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

```

Ex 6.1

A music-streaming service uses a doubly linked list to store the music currently in the user's queue. However, their code to remove a particular song based on its position in the list contains errors. Identify the error(s) present in the code below:

```

def delete_at(self, index):
    if index == 0:
        self.head = self.head.next
        self.head.prev = None
        self.size -= 1
        return
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.tail:
                self.tail = cur_node.prev
                cur_node.prev.next = None
                self.size -= 1
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next

```

```
count += 1
```

- The code does not have special code to handle the case that we want to delete the head node
- ~~The code will delete two nodes instead of one if the tail node is the node we want to delete~~
- The code decrements size each time the loop occurs, instead of only when the desired node is found
- Count is incremented outside of the while loop instead of inside of it

Ex 6.2

Find and correct the errors present in the code from the previous question:

```
def delete_at(self, index):  
    if index == 0:  
        self.head = self.head.next  
        self.head.prev = None  
        self.size -= 1  
        return  
    cur_node = self.head  
    count = 0  
    while cur_node:  
        if count == index:  
            if cur_node == self.tail:  
                self.tail = cur_node.prev  
                cur_node.prev.next = None  
                self.size -= 1  
                cur_node.prev.next = cur_node.next  
                cur_node.next.prev = cur_node.prev  
                self.size -= 1  
                return  
            cur_node = cur_node.next  
        count += 1
```

Expected Input

```
def delete_at(self, index):  
    if index == 0:  
        self.head = self.head.next  
        self.head.prev = None  
        self.size -= 1  
        return  
    cur_node = self.head  
    count = 0  
    while cur_node:  
        if count == index:  
            if cur_node == self.tail:
```

```

        self.tail = cur_node.prev
        cur_node.prev.next = None
        self.size -= 1
        return
    cur_node.prev.next = cur_node.next
    cur_node.next.prev = cur_node.prev
    self.size -= 1
    return
cur_node = cur_node.next
count += 1

```

Ex 7.1

An organization develops a slideshow software that uses a doubly linked list to display slides in presentation mode. Their software allows slides to be accessed by title instead of index when presenting. However, the code to find the index of the desired slide has an error. Identify the error in the code below:

```

def index(self, title):
    cur_node = self
    count = 0
    while cur_node:
        if cur_node.title == title:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

```

- ~~cur_node set to the entire data structure instead of only the head node~~
- The code compares the index of the current node with the desired title, instead of two titles
- The code returns the title of the slide instead of the index
- The code is error-free

Ex 7.2

Find and correct the errors present in the code from the previous question:

```

def index(self, title):
    cur_node = self
    count = 0
    while cur_node:
        if cur_node.title == title:
            return count
        cur_node = cur_node.next

```

```
    count += 1
return -1
```

Expected Input

```
def index(self, title):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.title == title:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
```

Ex 8.1

A restaurant's website has a scrolling bar of special deals they offer, which is implemented as a circular linked list. Sometimes the restaurant creates new special offers, and adds them to the start of the linked list so that customers see it first when they open the website. However, a bug was recently discovered in the code to do so. Identify the error(s) present in their code:

```
class Node:
    def __init__(self, name, image):
        self.name = name
        self.image = image
        self.next = None

class CircularOffers:
    #code for __init__ function

    def prepend(self, name, image):
        new_node = Node(name, image)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node
        self.size += 1
```

- The size of the linked list is only updated when there is more than one node in the list
- The size of the linked list is updated twice
- The code to make the node loop back on itself when there is only one node is missing

- The rest of the linked list is not attached properly when there is more than one node, resulting in the rest of the data being lost

Ex 8.2

Find and correct the errors present in the code from the previous question:

```
class Node:
    def __init__(self, name, image):
        self.name = name
        self.image = image
        self.next = None

class CircularOffers:
    #code for __init__ function

    def prepend(self, name, image):
        new_node = Node(name, image)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node
            self.size += 1
```

Expected Input

```
class Node:
    def __init__(self, name, image):
        self.name = name
        self.image = image
        self.next = None

class CircularOffers:
    #code for __init__ function

    def prepend(self, name, image):
        new_node = Node(name, image)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
```

```

        new_node.next = self.head
        self.tail.next = new_node
        self.head = new_node
self.size += 1

```

Ex 9.1

Instructions for a manufacturing machine are stored in a circular linked list that the machine endlessly iterates through and executes. However, the code contains an error that is preventing the conveyor belt from moving after the first time. Identify the cause of the error in the code below:

```

def get_instruction(self, num_instructions):
    if self.head is None:
        return -1    #If no instructions, return
    cur_node = self.head
    count = 0
    while True:    #loop until turned off
        #Executing instructions
        execute_instructions(cur_node)
        cur_node = cur_node.next
        count += 1
        if count == num_instructions: #We have finished creating
            a single product and need to move the conveyor belt to make the
            next
            print("Product finished. Moving onto the next...")
            move_conveyor_belt()

```

- The while loop does not have a condition to move the conveyor belt
- The check to see if the list is empty will always happen, preventing any instructions from executing
- ~~The count is never reset, and therefore the check to see if we should move the conveyor belt will never be satisfied again~~
- The code does not reset the current node to the head node after moving the conveyor belt

Ex 9.2

Find and correct the errors present in the code from the previous question:

```

def get_instruction(self, num_instructions):
    if self.head is None:
        return -1    #If no instructions, return
    cur_node = self.head
    count = 0
    while True:    #loop until turned off
        #Executing instructions
        execute_instructions(cur_node)
        cur_node = cur_node.next
        count += 1
        if count == num_instructions: #We have finished creating
            a single product and need to move the conveyor belt to make the
            next
            print("Product finished. Moving onto the next...")
            move_conveyor_belt()

```

Expected Input

```

def get_instruction(self, num_instructions):
    if self.head is None:
        return -1    #If no instructions, return
    cur_node = self.head
    count = 0
    while True:    #loop until turned off
        #Executing instructions
        execute_instructions(cur_node)
        cur_node = cur_node.next
        count += 1
        if count == num_instructions: #We have finished creating
            a single product and need to move the conveyor belt to make the
            next
            print("Product finished. Moving onto the next...")
            move_conveyor_belt()
count = 0

```

Ex 10.1

A circular linked list is used by a company to create a virtual line for their help desk. When a person comes to the desk, they are removed from the start of the linked list. However, the code to do so has an error. Identify the error present in the code below:

```
def delete_at_head(self):
```

```

if self.head is None:
    return
cur_node = self.head
if self.head == self.tail:
    self.head = None
else:
    self.head = cur_node.next
    self.tail.next = self.head
cur_node = None
self.size -= 1
return

```

- The code will turn the second-last element in the linked list into the new head when there is more than one element, rather than the second element
- Setting cur_node = None causes the entire linked list to be lost
- The code will always return before the size of the linked list can be decreased
- When there is only one person in the queue, they are not properly removed, as the tail pointer still references them

Ex 10.2

Find and correct the errors present in the code from the previous question:

```

def delete_at_head(self):
    if self.head is None:
        return
    cur_node = self.head
    if self.head == self.tail:
        self.head = None
    else:
        self.head = cur_node.next
        self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return

```

Expected Input

```

def delete_at_head(self):
    if self.head is None:
        return
    cur_node = self.head
    if self.head == self.tail:
        self.head = None
        self.tail = None
    else:
        self.head = cur_node.next

```

```
    self.tail.next = self.head
    cur_node = None
    self.size -= 1
    return
```

8. Overall Questions (22 Exercises)

We are going to start the overall questions by getting you warmed up on the methods that we have already worked on throughout all of our general Linked List lessons. Take a careful look at what class you are working in, and create the methods with all instance variables working as expected.

Story:

You are a budding graduate software engineer and are working for a software development company on a large-scale coded solution. Your manager hasn't finished interviewing the client on the specific requirements of this solution, but has instructed you that building data structures may be beneficial in the meantime. Start by adding methods that are expected to be used in most use cases.

The boss has asked you to prepare numerous types of linked lists as many different data structures should be used, she has also allowed access to the templates she has on hand for you to modify.

Exercise 1

Create the `prepend` method for the `CircularLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched.

Hint: Do not forget about the tail node. You need to make sure that the tail always points toward the head.

Given Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

```

def prepend(self, data):
    # Your code here

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list")
    while count < self.size:
        print("( Node", count, ":", current.data, ") -> ",
end=""")
        count += 1
        current = current.next
        if (current == self.head):
            break

    print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.prepend(n)

LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def prepend(self, data):
        new_node = Node(data)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head

```

```

        self.tail.next = new_node
        self.head = new_node

    self.size += 1

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next
        if (current == self.head):
            break

    print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.prepend(n)

LL.print_list()

```

Expected Output:

Starting to print list:
(Node 0 : 5) -> (Node 1 : 4) -> (Node 2 : 3) -> (Node 3 : 2) -> (Node 4 : 1) -> None

Exercise 2

Create the `append` method for the `DoublyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched.

Hint: You are working with a doubly linked list. Make sure that you link a node to both the next and previous nodes.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:

```

```

def __init__(self):
    self.head = None
    self.tail = None
    self.cur_node = None
    self.size = 0

def append(self, data):
    # Your code here

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:

```

```

        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()

```

Expected Output:

Starting to print list
(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Exercise 3

Create the `insert_at` method for the `SinglyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. We have provided the appropriate error handling of the code, you have to do the rest.

Hint: Because we have ensured that the index is valid, you should only need to travel from the head node to your desired index - 1, and then put the new node after where you have stopped your iteration.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

```

```

        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def insert_at(self, position, data):
        new_node = Node(data)
        if position < 0 or position > self.size:
            raise IndexError("Invalid position")
# Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = SinglyLinkedList()

for n in lst:
    LL.append(n)

LL.insert_at(3, 3)

LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:

```

```

def __init__(self):
    self.head = None
    self.size = 0

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node
    self.size += 1

def insert_at(self, position, data):
    new_node = Node(data)
    if position < 0 or position > self.size:
        raise IndexError("Invalid position")
    if position == 0:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        for _ in range(position - 1):
            current = current.next
        new_node.next = current.next
        current.next = new_node
    self.size += 1

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = SinglyLinkedList()

for n in lst:
    LL.append(n)

LL.insert_at(3, 3)

```

```
LL.print_list()
```

Expected Output:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 3) -> (Node 4 : 4) -> (Node 5 : 5) -> None

Exercise 4

Create the `delete` method for the `DoublyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. Do not return any nodes or values.

Hint: When you have looped through and found the data node you want to delete, there are four scenarios you need to consider carefully:

- When the node to delete is the head and tail.
- When the node is just the head.
- When the node is just the tail.
- When the node is anything else.

Given Code:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = self.tail = new_node  
        else:  
            new_node.prev = self.tail  
            self.tail.next = new_node  
            self.tail = new_node  
  
    def delete(self, data):  
        # Your code here  
  
    def print_list(self):  
        current = self.head  
        count = 0  
  
        print("Starting to print list")
```

```

        while current:
            print("( Node", count, ":", current.data, ") -> ",
end=""")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

print("Initial list:")
LL.print_list()

LL.delete(3)

print("List after deleting:")
LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def delete(self, data):
        cur_node = self.head
        while cur_node:
            if cur_node.data == data:
                if cur_node == self.head:

```

```

        self.head = cur_node.next
        if self.head:
            self.head.prev = None
        if cur_node == self.tail:
            self.tail = None
        return
    if cur_node == self.tail:
        self.tail = cur_node.prev
        cur_node.prev.next = None
        return
    cur_node.prev.next = cur_node.next
    cur_node.next.prev = cur_node.prev
    return
    cur_node = cur_node.next

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

print("Initial list:")
LL.print_list()

LL.delete(3)

print("List after deleting:")
LL.print_list()

```

Expected Output:

Initial list:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

List after deleting:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 4) -> (Node 3 : 5) -> None

Exercise 5

Create the `index` method for the `SinglyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. Return `None` when you do not find any index that matches your data. You may return the first occurring index if there are multiple.

Hint: Create a counter and iterate through the nodes until you find the right one.

Given Code:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def prepend(self, value):  
        new_node = Node(value)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def index(self, value):  
        # Your code  
  
    def print_list(self):  
        current = self.head  
        count = 0  
  
        print("Starting to print list")  
        while current:  
            print("( Node", count, ":", current.value, ") -> ",  
end=""")  
            count += 1  
            current = current.next  
  
        print("None")  
  
lst = [1, 2, 3, 4, 5]  
  
LL = SinglyLinkedList()  
  
for n in lst:  
    LL.prepend(n)  
  
print(LL.index(2))
```

```
print(LL.index(4))
```

Expected input:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def prepend(self, value):  
        new_node = Node(value)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def index(self, value):  
  
        cur_node = self.head  
        count = 0  
        while cur_node:  
            if cur_node.value == value:  
                return count  
            cur_node = cur_node.next  
            count += 1  
        return None  
  
    def print_list(self):  
        current = self.head  
        count = 0  
  
        print("Starting to print list")  
        while current:  
            print("( Node", count, ":", current.value, ") -> ",  
end=""")  
            count += 1  
            current = current.next  
  
        print("None")  
  
lst = [1, 2, 3, 4, 5]  
  
LL = SinglyLinkedList()  
  
for n in lst:
```

```
LL.prepend(n)

print(LL.index(2))
print(LL.index(4))
```

Expected Output:

```
3
1
```

Exercise 6

Create the `get` method for the `CircularLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. Return `None` if the index is out of bounds.

Hint: This is similar to the `index` method. Create a counter and stop the iteration when the count and index are equal.

Given Code:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def get(self, index):
        # Your code here

    def print_list(self):
        current = self.head
        count = 0
```

```

        print("Starting to print list:")
        while count < self.size:
            print("( Node", count, ":", current.value, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

print(LL.get(2))
print(LL.get(4))
print(LL.get(6))

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def get(self, index):
        if index < 0 or index >= self.size:
            return None

        current = self.head

```

```

count = 0

while count != index:
    current = current.next
    count += 1

return current.value

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list:")
    while count < self.size:
        print("( Node", count, ":", current.value, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

print(LL.get(2))
print(LL.get(4))
print(LL.get(6))

```

Expected Output:

3
5
None

Exercise 7

Create the `set_current_to` and `get_current_value` method for the `DoublyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. For `set_current_to`, if the index is greater than the limits, raise an `IndexError` with the text: “That index is out of bounds”. For `get_current_value`, if no current node has been set, return `None`.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data

```

```

        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def set_current_to(self, index):
        if index < 0:
            raise IndexError("Index should not be less than 0")

        # Your code here

    def get_current_value(self):
        # Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end=""")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.set_current_to(2)

print(LL.get_current_value())

```

Expected input:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.cur_node = None  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = self.tail = new_node  
        else:  
            new_node.prev = self.tail  
            self.tail.next = new_node  
            self.tail = new_node  
  
    def set_current_to(self, index):  
        if index < 0:  
            raise IndexError("Index should not be less than 0")  
  
        self.cur_node = self.head  
        for i in range(index):  
            if self.cur_node is None:  
                raise IndexError("That index is out of bounds")  
            self.cur_node = self.cur_node.next  
  
    def get_current_value(self):  
        if self.cur_node == None:  
            return None  
  
        return self.cur_node.data  
  
    def print_list(self):  
        current = self.head  
        count = 0  
  
        print("Starting to print list")  
        while current:  
            print("( Node", count, ":", current.data, ") -> ",  
end="")  
            count += 1  
            current = current.next  
  
        print("None")
```

```

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.set_current_to(2)

print(LL.get_current_value())

```

Expected Output:

3

Exercise 8

Create the `set_current` method for the `CircularLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. If there is no index provided, the index is `None` and in that case set the current node to the head node. Otherwise set it to the specified index. If the index is invalid (negative or greater than size) raise an `IndexError` with the text "Index should be between 0 and {size}".

Hint: Don't forget the case where there are no nodes in the list.

Given Code:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

```

```

def set_current(self, index=None):
    # Your code here

def get_current_value(self):
    if self.cur is None:
        return None

    return self.cur.value

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list:")
    while count < self.size:
        print("( Node", count, ":", current.value, ") ->",
end=""")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

LL.set_current()
print(LL.get_current_value())
LL.set_current(3)
print(LL.get_current_value())

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def append(self, value):

```

```

new_node = Node(value)
if self.size == 0:
    self.head = new_node
    self.tail = new_node
    new_node.next = new_node
else:
    new_node.next = self.head
    self.tail.next = new_node
    self.tail = new_node

self.size += 1

def set_current(self, index=None):
    if self.size == 0:
        return None

    if index is None:
        self.cur = self.head
        return

    if index < 0 or index >= self.size:
        raise IndexError("Index should be between 0 and",
self.size - 1)

    count = 0
    current = self.head

    while count < index:
        current = current.next
        count += 1

    self.cur = current

def get_current_value(self):
    if self.cur is None:
        return None

    return self.cur.value

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list:")
    while count < self.size:
        print("( Node", count, ":", current.value, ") ->",
end="")
        count += 1
        current = current.next

    print("None")

```

```

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

LL.set_current()
print(LL.get_current_value())
LL.set_current(3)
print(LL.get_current_value())

```

Expected Output:

```

1
4

```

Exercise 9

Create the `next()` and `prev()` method for the DoublyLinkedList provided in the given code. Ensure all variables are satisfied so that the expected output is matched. If the current node is `None`, just return.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:

```

```

        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

    def set_current_to(self, index):
        if index < 0:
            raise IndexError("Index should not be less than 0")

        self.cur_node = self.head
        for i in range(index):
            self.cur_node = self.cur_node.next

    def next(self):
        # Your code here

    def prev(self):
        # Your code here

    def get_current_value(self):
        if self.cur_node == None:
            return

        return self.cur_node.data

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.set_current_to(2)
print(LL.get_current_value())
LL.prev()
LL.prev()
print(LL.get_current_value())
LL.next()
LL.next()
LL.next()
LL.next()
print(LL.get_current_value())

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:

```

```

def __init__(self):
    self.head = None
    self.tail = None
    self.cur_node = None

def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

def set_current_to(self, index):
    if index < 0:
        raise IndexError("Index should not be less than 0")

    self.cur_node = self.head
    for i in range(index):
        self.cur_node = self.cur_node.next

def next(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.next

def prev(self):
    if self.cur_node == None:
        return
    self.cur_node = self.cur_node.prev

def get_current_value(self):
    if self.cur_node == None:
        return

    return self.cur_node.data

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

```

```
for n in lst:  
    LL.append(n)  
  
LL.set_current_to(2)  
print(LL.get_current_value())  
LL.prev()  
LL.prev()  
print(LL.get_current_value())  
LL.next()  
LL.next()  
LL.next()  
LL.next()  
print(LL.get_current_value())
```

Expected Output:

3
1
5

Now that you are refreshed on the methods that we have taught throughout the lessons, let's nail down the big goal of creating your own data structures. They allow you to add whatever methods you want. Let's create some simple methods that you may want to use now.

Do note, the type of data structure is still changing!

Story:

Your boss has finished interviewing the client and has let you know what methods need to be created next. She has given you until next meeting to complete the following methods:

1. reverse()
2. to_list()
3. clear()
4. count()
5. is_same_list()

Exercise 10

Create the `reverse()` method for the `SinglyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. What we would suggest you do is to initialize a variable to hold previous nodes to `None`, and another variable to hold the current node and start it from the head. Then iterate through a loop and reverse the two variable's directions

Hint: After the loop, make sure you set the head node to be equal to the variable you set to

hold your previous nodes.

Given Code:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def prepend(self, value):  
        new_node = Node(value)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def reverse(self):  
        # Your code here  
  
    def print_list(self):  
        current = self.head  
        count = 0  
  
        print("Starting to print list")  
        while current:  
            print("( Node", count, ":", current.value, ") -> ",  
end=""")  
            count += 1  
            current = current.next  
  
        print("None")  
  
lst = [1, 2, 3, 4, 5]  
  
LL = SinglyLinkedList()  
  
for n in lst:  
    LL.prepend(n)  
  
LL.print_list()  
LL.reverse()  
LL.print_list()
```

Expected input:

```
class Node:  
    def __init__(self, value):  
        self.value = value
```

```

        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def prepend(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def reverse(self):
        prev = None
        current = self.head
        while current is not None:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.value, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = SinglyLinkedList()

for n in lst:
    LL.prepend(n)

LL.print_list()
LL.reverse()
LL.print_list()

```

Expected Output:

Starting to print list

(Node 0 : 5) -> (Node 1 : 4) -> (Node 2 : 3) -> (Node 3 : 2) -> (Node 4 : 1) -> None

Starting to print list

```
( Node 0 : 1 ) -> ( Node 1 : 2 ) -> ( Node 2 : 3 ) -> ( Node 3 : 4 ) -> ( Node 4 : 5 ) -> None
```

Exercise 11

Create the `to_list()` method for the `CircularLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This method should create a list, and add every node value in the linked list to the new list. Return the list at the end of the function.

Given Code:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
  
    def append(self, value):  
        new_node = Node(value)  
        if self.size == 0:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.tail = new_node  
  
        self.size += 1  
  
    def to_list(self):  
        # Your code here  
  
    def print_list(self):  
  
        current = self.head  
        count = 0  
  
        print("Starting to print list:")  
        while count < self.size:  
            print("( Node", count, ":", current.value, " ) -> ",  
end=""")  
            count += 1  
            current = current.next  
  
        print("None")
```

```

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL_to_list = LL.to_list()

print(LL_to_list)

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def to_list(self):
        result = []
        current = self.head
        count = 0

        while count < self.size:
            result.append(current.value)
            current = current.next
            count += 1

        return result

```

```

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list:")
    while count < self.size:
        print("( Node", count, ":", current.value, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL_to_list = LL.to_list()

print(LL_to_list)

```

Expected Output:

Starting to print list:

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None
[1, 2, 3, 4, 5]

Exercise 12

Create the `clear()` method for the `DoublyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This method should remove all nodes from the linked list. It is possible for this to happen in O(1) time.

Hint: The O(1) solution can be done by setting all instance variables to None.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

```

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

def clear(self):
    # Your code here

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.clear()
LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

```

```

def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

def clear(self):
    self.head = None
    self.tail = None

def print_list(self):

    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.clear()
LL.print_list()

```

Expected Output:

Starting to print list:

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Starting to print list:

None

Exercise 13

Create the `count()` method for the `SinglyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This should take in a value, and return a count equal to the number of nodes that equal that value.

Given Code:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def prepend(self, value):  
        new_node = Node(value)  
        new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def append(self, value):  
        new_node = Node(value)  
        if self.head is None:  
            self.head = new_node  
        else:  
            current = self.head  
            while current.next is not None:  
                current = current.next  
            current.next = new_node  
        self.size += 1  
  
    def count(self, value):  
        # Your code here  
  
    def print_list(self):  
  
        current = self.head  
        count = 0  
  
        print("Starting to print list:")  
        while current:  
            print("( Node", count, ":", current.value, ") -> ",  
end=""")  
            count += 1  
            current = current.next  
  
        print("None")  
  
lst = [1, 2, 3, 4, 5]  
LL = SinglyLinkedList()  
for n in lst:
```

```

LL.prepend(3)

print(LL.count(3))
LL.append(3)
LL.append(2)
LL.append(3)
print(LL.count(3))

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def prepend(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def count(self, value):
        current = self.head
        count = 0
        while current is not None:
            if current.value == value:
                count += 1
            current = current.next
        return count

    def print_list(self):
        current = self.head
        count = 0

```

```

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.value, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL = SinglyLinkedList()

for n in lst:
    LL.prepend(n)

print(LL.count(3))
LL.append(3)
LL.append(2)
LL.append(3)
print(LL.count(3))

```

Expected Output:

1
3

Exercise 14

Create the `is_same_list()` method for the `CircularLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This method first checks to see if the other object is a `CircularLinkedList` (we have already done this for you), and then goes through every node in both lists and identifies whether every node has a matching position and value. As an example:

1 -> 2 -> 3 -> 4
1 -> 3 -> 2 -> 4

Are not the same list because 2 and 3 are in the wrong order, even though they are both in both lists.

Hint: It might be easier to first check if the size of both lists are the same, before going through and checking the values.

Given Code:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

```

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def is_same_list(self, other):
        if not isinstance(other, CircularLinkedList):
            return False

# Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.value, ") -> ",
end=""")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

LL1 = CircularLinkedList()
LL2 = CircularLinkedList()

for n in lst:
    LL1.append(n)
    LL2.append(n)

print("Checking that LL1 is same as LL1")
print(LL1.is_same_list(LL1))
print("Checking that LL1 is same as LL2")

```

```

print(LL1.is_same_list(LL2))
print("Adding 2 to LL1")
LL1.append(2)
print("Checking LL1 is same as LL2")
print(LL1.is_same_list(LL2))
print("Adding 2 to LL2")
LL2.append(2)
print("Checking LL1 is same as LL2")
print(LL1.is_same_list(LL2))

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def append(self, value):
        new_node = Node(value)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def is_same_list(self, other):
        if not isinstance(other, CircularLinkedList):
            return False

        if self.size != other.size:
            return False

        current_self = self.head
        current_other = other.head

        for _ in range(self.size):
            if current_self.value != current_other.value:
                return False
            current_self = current_self.next

```

```

        current_other = current_other.next

    return True

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.value, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL1 = CircularLinkedList()
LL2 = CircularLinkedList()

for n in lst:
    LL1.append(n)
    LL2.append(n)

print("Checking that LL1 is same as LL1")
print(LL1.is_same_list(LL1))
print("Checking that LL1 is same as LL2")
print(LL1.is_same_list(LL2))
print("Adding 2 to LL1")
LL1.append(2)
print("Checking LL1 is same as LL2")
print(LL1.is_same_list(LL2))
print("Adding 2 to LL2")
LL2.append(2)
print("Checking LL1 is same as LL2")
print(LL1.is_same_list(LL2))

```

Expected Output:

Checking that LL1 is same as LL1

True

Checking that LL1 is same as LL2

True

Adding 2 to LL1

Checking LL1 is same as LL2

False

Adding 2 to LL2

Checking LL1 is same as LL2

True

Exercise 15

Your manager is thrilled with how fast you completed those methods. She has supplied you with a set of instructions to test the functionality of the code you have created so far. The list can be found below. Perform the following operations and print the results of those operations.

1. Add all the items in lst to the CircularLinkedList LL.
2. Print the text: "When the linked list was first created:".
3. Print the current Linked list.
4. Append 6 to the linked list.
5. Save the linked list using the to_list() method to a variable.
6. Prepend 0, Prepend 2, Append 3.
7. Save the linked list using the to_list() method to a different variable.
8. Print the text: "Last modified Linked List:". And then print the linked list.
9. Print the text: "When the list was first pulled:". And then print the first list you saved with to_list.
10. Print the text: "When the list was last pulled:". Then print the second list.

See if you matched the expected result. Consider what actually happened, why are the two lists different? When might you want to use this?

Given Code:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.cur = None  
  
    def prepend(self, value):  
        new_node = Node(value)  
        if self.size == 0:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
        else:  
            new_node.next = self.head  
            self.tail.next = new_node  
            self.head = new_node  
  
        self.size += 1  
  
    def append(self, value):  
        new_node = Node(value)
```

```

        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def to_list(self):
        result = []
        current = self.head
        count = 0

        while count < self.size:
            result.append(current.value)
            current = current.next
            count += 1

        return result

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") ->",
end="")
            count += 1
            current = current.next
            if (current == self.head):
                break

        print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

# Your code here

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

```

```

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.cur = None

    def prepend(self, data):
        new_node = Node(data)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node

        self.size += 1

    def append(self, data):
        new_node = Node(data)
        if self.size == 0:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1

    def to_list(self):
        result = []
        current = self.head
        count = 0

        while count < self.size:
            result.append(current.data)
            current = current.next
            count += 1

        return result

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")

```

```

        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next
            if (current == self.head):
                break

        print("None")

lst = [1, 2, 3, 4, 5]

LL = CircularLinkedList()

for n in lst:
    LL.append(n)

print("When the linked list was first created:")
LL.print_list()
LL.append(6)
list1 = LL.to_list()
LL.prepend(0)
LL.prepend(2)
LL.append(3)
list2 = LL.to_list()

print("Last modified Linked List:")
LL.print_list()
print("When the list was first pulled:")
print(list1)
print("When the list was last pulled:")
print(list2)

```

Expected Output:

When the linked list was first created:

Starting to print list:

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Last modified Linked List:

Starting to print list:

(Node 0 : 2) -> (Node 1 : 0) -> (Node 2 : 1) -> (Node 3 : 2) -> (Node 4 : 3) -> (Node 5 : 4) -> (Node 6 : 5) -> (Node 7 : 6) -> (Node 8 : 3) -> None

When the list was first pulled:

[1, 2, 3, 4, 5, 6]

When the list was last pulled:

[2, 0, 1, 2, 3, 4, 5, 6, 3]

Exercise 16

Story:

There was a miscommunication in the meeting that meant that you didn't also create a

method reverse() for the DoublyLinkedList class alongside the SinglyLinkedList method that you completed earlier. We need to fix this quickly before the boss finds out!!!!

Create a reverse method for the DoublyLinkedList provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This is similar to the SinglyLinkedList version, but you need to be careful of the prev instance variable for nodes. As a tip, for this one, use a temp variable that stores the current node's previous variable. Declare this outside of your loop, as you need to deal with the final condition similar to what you did with the SinglyLinkedList.

Hint: You need to set the head at the end equal to the temp node's previous value. But be careful that you check that temp is not None first it will raise an error.

Given Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def reverse(self):
        # Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")
```

```

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()
LL2 = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.reverse()
LL.print_list()
print("reversing list 2")
LL2.reverse()
LL2.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def reverse(self):
        current = self.head
        temp = None
        while current:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

        if temp:
            self.head = temp.prev
            self.tail = temp

```

```

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]

LL = DoublyLinkedList()
LL2 = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.reverse()
LL.print_list()
print("reversing list 2")
LL2.reverse()
LL2.print_list()

```

Expected Output:

Starting to print list:

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Starting to print list

(Node 0 : 5) -> (Node 1 : 4) -> (Node 2 : 3) -> (Node 3 : 2) -> (Node 4 : 1) -> None

reversing list 2

Starting to print list:

None

Story:

After a code review with the client your manager has extracted more use cases. Because she is so pleased with the progress you have made and the initiative you have taken, she has asked you to continue with your work by implementing the following methods:

1. find_middle()
2. pop()

Exercise 17

Create the `find_middle` method for the `SinglyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. It should return the value of the node if there is one otherwise return `None`.

This can be a touch hard to come up with by yourself, so here is some intuition:

If you have two pointers. One pointer travels the linked list one node at a time. The other pointer travels the linked list two nodes at a time. When the second pointer is at the end, where should the first pointer be?

Hint: The answer is that the first pointer is going to be in the middle. So set up a loop that does the above steps starting from the head node for both pointers, and have them both traverse the list until the faster one has reached the end.

Given Code:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def prepend(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def find_middle(self):
        # Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]
LL = SinglyLinkedList()
```

```

for n in lst:
    LL.prepend(n)

print(LL.find_middle())
LL.prepend(6)
print(LL.find_middle())
LL.prepend(7)
print(LL.find_middle())

```

Expected input:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def prepend(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def find_middle(self):
        slow = self.head
        fast = self.head

        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next

        return slow.value if slow else None

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end=""")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]

```

```

LL = SinglyLinkedList()

for n in lst:
    LL.prepend(n)

print(LL.find_middle())
LL.prepend(6)
print(LL.find_middle())
LL.prepend(7)
print(LL.find_middle())

```

Expected Output:

3
3
4

Exercise 18

Create the `pop` method for the `DoublyLinkedList` provided in the given code. Ensure all variables are satisfied so that the expected output is matched. This should remove the last node from the list and return the data it holds to you.

Hint: This is simply the application of `delete_last` for doubly linked lists. We recommend checking if the tail is set before doing anything more.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def pop(self):
        # Your code here

```

```

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list:")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [1, 2, 3, 4, 5]
LL = DoublyLinkedList()
LL2 = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.pop()
LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.cur_node = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def pop(self):
        if self.tail is None:
            return None

```

```

        data = self.tail.data
        if self.head == self.tail:
            self.head = self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None
        return data

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list:")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [1, 2, 3, 4, 5]
LL = DoublyLinkedList()
LL2 = DoublyLinkedList()

for n in lst:
    LL.append(n)

LL.print_list()
LL.pop()
LL.print_list()

```

Expected Output:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> None

Explanation:

Data structure naming isn't universal, often you are going to have to look around and see what functions do and how they are named. OR make your own so you can choose the name. Here someone chose to have a naming convention that is similar to python lists. Probably so that the user didn't have to check if they were working with a normal list or a linked list first.

To end these overall exercises, let's really emphasize that the data structures we have taught are just templates. Now that you know how to make some, make them yours to fit your use case!

Story:

Another colleague was working alongside you on these tasks and was assigned to implement a sorted data structure. He completed some of this, but he caught the flu and needs time to recover. You have volunteered to take on his work while he sits at home and watches 90s sitcoms.

Below is a list of methods he has yet to implement:

1. insert()
2. remove_smallest()
3. to_list()

Exercise 19

We are going to construct a `SortedDoublyLinkedList` class. We have done the `Nodes` and some of the methods, but you need to create the `insert()` method. The `insert` method will self-sort the list, so whenever an item is added, it will be added such that every node in the list is sorted in ascending order. To help you, first check if the list is empty and the first node to be added. Then check if the node is smaller than the head's node data, or if it is larger than the tail node's data. If none of those, then it is somewhere in the middle, so you should traverse the list and find the correct spot to place it.

Given Code:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class SortedDoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
  
    def insert(self, data):  
        # Your code here  
  
    def print_list(self):  
  
        current = self.head  
        count = 0  
  
        print("Starting to print list")  
        while current:  
            print("( Node", count, ":", current.data, ") -> ",  
end="")  
            count += 1  
            current = current.next  
  
        print("None")  
  
lst = [3, 4, 1, 5, 2]
```

```

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
            return

        if data < self.head.data:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
            return

        if data >= self.tail.data:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
            return

        current = self.head
        while current.next is not None and current.next.data < data:
            current = current.next
        if current.next is not None:
            current.next.prev = new_node

        new_node.next = current.next
        current.next = new_node

    def print_list(self):
        current = self.head
        count = 0

```

```

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()

```

Expected Output:

Starting to print list
(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Exercise 20

Now we are going to have you add another method called `remove_smallest` that removes the node with the smallest data from the list. Remember the linked list is sorted in ascending order. Return the data of the node that was deleted, otherwise `None` if there are no nodes.

Hint: As the linked list is sorted in ascending order, it will be the first node that should be removed.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
        return

```

```

        if data < self.head.data:
            new_node.next = self.head
            self.head = new_node
            return

        if data >= self.tail.data:
            self.tail.next = new_node
            self.tail = new_node
            return

        current = self.head
        while current.next is not None and current.next.data <
data:
            current = current.next

        new_node.next = current.next
        current.next = new_node

    def remove_smallest(self):
        # Your code here

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()
print(LL.remove_smallest())
LL.print_list()

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

```

```

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
            return

        if data < self.head.data:
            new_node.next = self.head
            self.head = new_node
            return

        if data >= self.tail.data:
            self.tail.next = new_node
            self.tail = new_node
            return

        current = self.head
        while current.next is not None and current.next.data <
data:
            current = current.next

        new_node.next = current.next
        current.next = new_node

    def remove_smallest(self):
        if self.head is None:
            return None

        smallest_value = self.head.data

        self.head = self.head.next

        if self.head is None:
            self.tail = None

        return smallest_value

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",

```

```

        end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()
print(LL.remove_smallest())
LL.print_list()

```

Expected Output:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None
1

Starting to print list

(Node 0 : 2) -> (Node 1 : 3) -> (Node 2 : 4) -> (Node 3 : 5) -> None

Exercise 21

Lastly, create a `to_list()` method and add it to the class. It simply creates a list from our sorted linked list.

Given Code:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
            return

        if data < self.head.data:
            new_node.next = self.head
            self.head = new_node

```

```

        return

    if data >= self.tail.data:
        self.tail.next = new_node
        self.tail = new_node
        return

    current = self.head
    while current.next is not None and current.next.data <
data:
        current = current.next

    new_node.next = current.next
    current.next = new_node

def remove_smallest(self):
    if self.head is None:
        return None

    smallest_value = self.head.data

    self.head = self.head.next

    if self.head is None:
        self.tail = None

    return smallest_value

def to_list(self):
    # Your code here

def print_list(self):
    current = self.head
    count = 0

    print("Starting to print list")
    while current:
        print("( Node", count, ":", current.data, ") -> ",
end="")
        count += 1
        current = current.next

    print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

```

```

LL.print_list()
print(LL.remove_smallest())
LL.print_list()

print("Turning the Linked List back into a list")
print(LL.to_list())

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
            return

        if data < self.head.data:
            new_node.next = self.head
            self.head = new_node
            return

        if data >= self.tail.data:
            self.tail.next = new_node
            self.tail = new_node
            return

        current = self.head
        while current.next is not None and current.next.data <
data:
            current = current.next

            new_node.next = current.next
            current.next = new_node

    def remove_smallest(self):
        if self.head is None:
            return None

        smallest_value = self.head.data

        self.head = self.head.next

```

```

        if self.head is None:
            self.tail = None

        return smallest_value

    def to_list(self):
        result = []
        current = self.head
        while current:
            result.append(current.data)
            current = current.next
        return result

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()
print(LL.remove_smallest())
LL.print_list()

print("Turning the Linked List back into a list")
print(LL.to_list())

```

Expected Output:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None
1

Starting to print list

(Node 0 : 2) -> (Node 1 : 3) -> (Node 2 : 4) -> (Node 3 : 5) -> None

Turning the Linked List back into a list

[2, 3, 4, 5]

Exercise 22

Story:

After submitting your implementation for review, your manager supplied you with a set of instructions to carry out to ensure that the code you have created works correctly.

1. Create a SortedDoublyLinkedList.
2. Add every element in lst to the linked list.
3. Print the list and visually confirm it is sorted.
4. Insert 0 and 7 into the list.
5. Now print the list and make sure it is still sorted.
6. Return and print the final list with the to_list method.

It should all be working as expected. We can now use our data structure knowledge of linked lists to make linked lists that do whatever we want!

Given Code:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class SortedDoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
  
    def insert(self, data):  
        new_node = Node(data)  
  
        if self.head is None:  
            self.head = self.tail = new_node  
            return  
  
        if data < self.head.data:  
            new_node.next = self.head  
            self.head = new_node  
            return  
  
        if data >= self.tail.data:  
            self.tail.next = new_node  
            self.tail = new_node  
            return  
  
        current = self.head  
        while current.next is not None and current.next.data <  
data:  
            current = current.next  
  
        new_node.next = current.next
```

```

        current.next = new_node

    def remove_smallest(self):
        if self.head is None:
            return None

        smallest_value = self.head.data

        self.head = self.head.next

        if self.head is None:
            self.tail = None

        return smallest_value

    def to_list(self):
        result = []
        current = self.head
        while current:
            result.append(current.data)
            current = current.next
        return result

    def print_list(self):

        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") ->",
end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

# Your code here

```

Expected input:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SortedDoublyLinkedList:
    def __init__(self):
        self.head = None

```

```

        self.tail = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = self.tail = new_node
            return

        if data < self.head.data:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
            return

        if data >= self.tail.data:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
            return

        current = self.head
        while current.next is not None and current.next.data <
data:
            current = current.next

        if current.next is not None:
            current.next.prev = new_node

        new_node.next = current.next
        current.next = new_node

    def remove_smallest(self):
        if self.head is None:
            return None

        smallest_value = self.head.data

        self.head = self.head.next

        if self.head is None:
            self.tail = None

        return smallest_value

    def to_list(self):
        result = []
        current = self.head
        while current:
            result.append(current.data)
            current = current.next

```

```

        return result

    def print_list(self):
        current = self.head
        count = 0

        print("Starting to print list")
        while current:
            print("( Node", count, ":", current.data, ") -> ",
end="")
            count += 1
            current = current.next

        print("None")

lst = [3, 4, 1, 5, 2]

LL = SortedDoublyLinkedList()

for n in lst:
    LL.insert(n)

LL.print_list()

LL.insert(0)
LL.insert(7)

LL.print_list()

result_list = LL.to_list()
print(result_list)

```

Expected Output:

Starting to print list

(Node 0 : 1) -> (Node 1 : 2) -> (Node 2 : 3) -> (Node 3 : 4) -> (Node 4 : 5) -> None

Starting to print list

(Node 0 : 0) -> (Node 1 : 1) -> (Node 2 : 2) -> (Node 3 : 3) -> (Node 4 : 4) -> (Node 5 :

5) -> (Node 6 : 7) -> None

[0, 1, 2, 3, 4, 5, 7]

Story:

Congratulations on going above and beyond at your internship! Your manager has recognised you for your hard work by offering you a full-time position AND a labelled space in the work fridge.

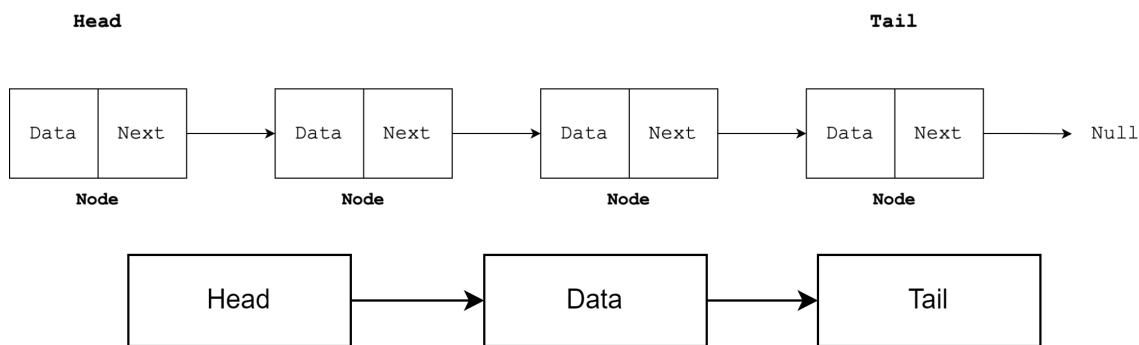
Removed sections

1.2.1 Linear Data Structures

So, now we know the basics, let's go into an implementation in Python that we can use, **LinkedLists!**

LinkedList

The linked list is the simplest of linear lists, with its structure being a simple chain of data connected together.



The data object ("Node") at the start of the list is known as the "**head**", and the object at the end is the "**tail**".

The head and tail store data and behave like any other **node** (the object where the data is stored), which is composed of two items - the data and a reference to the next node. The last node has a reference to **null**.

If the list is empty then the head is a **null** reference. There can be any number of nodes in between the head and tail.

Let's quickly recap the Python list. A general 1D list in Python is a dynamic array, this means that the list has:

- Dynamic size so you don't need to specify the size when declaring it.
- Dynamic type so you don't need to specify the data type of the elements in the list

For example, to declare a list in Python, you can use square brackets [] and separate the elements with commas:

```
people = [None] * 5
```

To access an element of the list, for example to assign a value to it, or to change its contents, you can refer to the element by its index number:

```
people[0] = "Homer"  
people[1] = "Marge"  
people[2] = "Kevin"
```

```
people[3] = "Lisa"  
people[4] = "Maggie"
```

0	1	2	3	4
Homer	Marge	Kevin	Lisa	Maggie

LinkedLists behave the same as lists so both can grow and shrink dynamically but LinkedLists behave differently when **deleting** elements.

In lists, if you remove an element, you may need to shift all the remaining elements to fill the gap. In LinkedLists, you just need to update the references in the neighboring nodes, making deletions more efficient.

Thus making this operation relatively fast as it doesn't require shifting elements.

Example

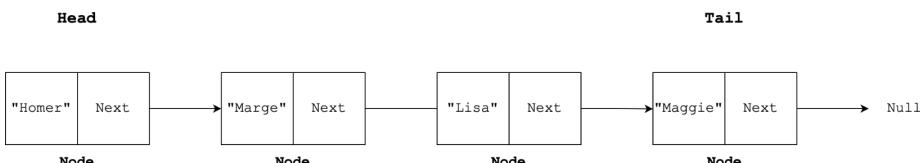
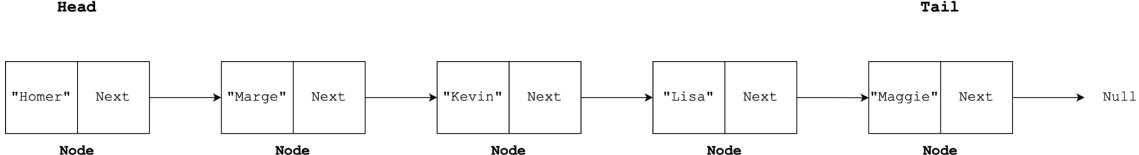
In a list, if we delete "Kevin" which is in index 2, it will involve shifting the elements after "Kevin" to fill the gap.

0	1	2	3	4
Homer	Marge	Kevin	Lisa	Maggie


Need to delete

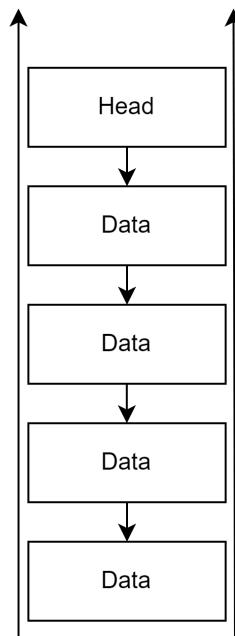
0	1	2	3	4
Homer	Marge	Lisa	Maggie	None

In a LinkedList, deleting "Kevin" will involve reassigning "Maggie's" next pointer:



Stack

A stack is another type of linear list, similar to a linked list, but with a key difference in how data is accessed and modified.



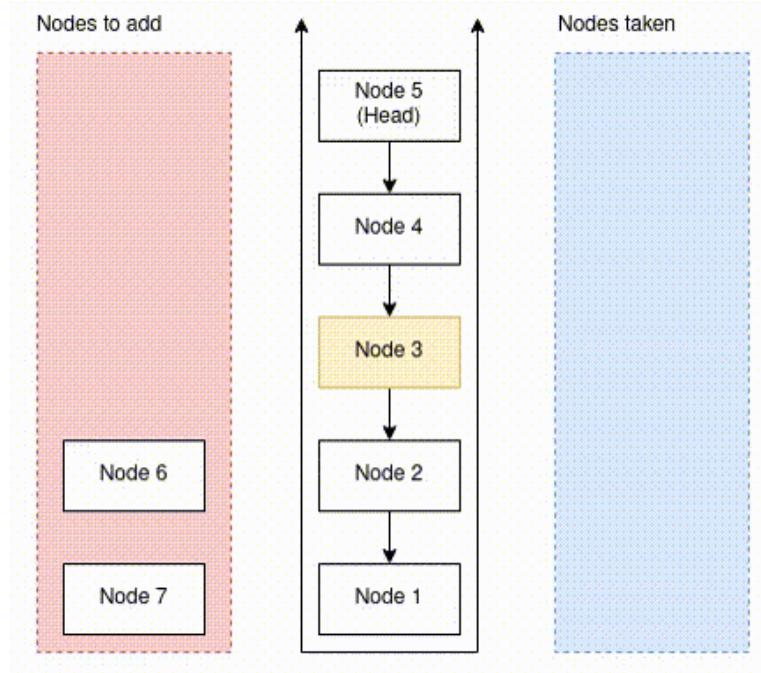
Imagine you are stacking books, you place one book on top of another, and when you need a book, you take the one from the top. The last book you put on the stack is the first one you take off.

This can be described as **last-in-first-out (LIFO)** and is exactly how a stack functions!

More specifically we can say that data can be added to or removed from the top of the stack,

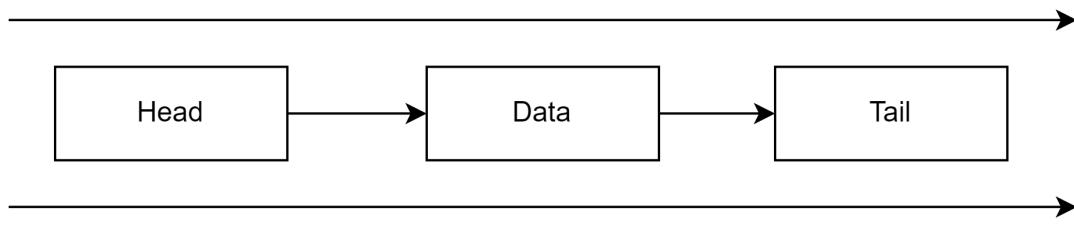
but not from any other part.

The LIFO operation is depicted in the GIF below. If Node 3 is the node that needs to be removed, notice that Node 4 and Node 5 must be removed first. Additionally, adding extra nodes (Node 6 and Node 7) is similar to placing extra books on top of a stack of books.



Queue

A queue is similar to a stack, but it follows the opposite principle for data access.



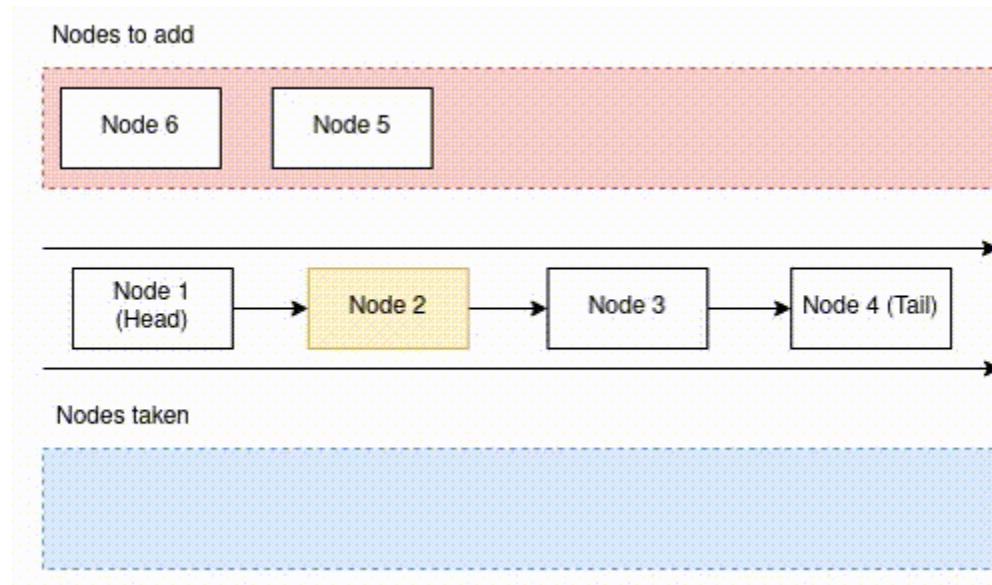
Queue data access can be described as **first-in-first-out (FIFO)**, where the first element added will be the first one removed. Adding to a queue will append the new item to the end where it will wait its turn.

Think of a queue as waiting in line at a grocery store. The first person to get in line is the first person to be served and leave.

Similarly, more people can only join the queue at the end of the line and must wait until everyone in front of them has been served.

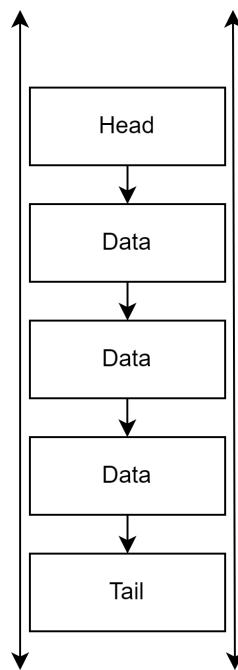
Queues are often used for managing processes in the order they need to be completed, ensuring that tasks are executed in the sequence they were added. This is particularly useful for managing tasks with specific priority or dependency requirements.

The FIFO operation is depicted in the GIF below. If Node 2 is the node that needs to be removed, notice that Node 4 and Node 3 must be removed first.



Deque

A deque, or double-ended queue, is a queue that can have data added and removed from both its front and back. It combines the access functionality of both stacks and queues, greatly increasing its flexibility.



Think of a deque as a ticket purchasing line. Normally, people join the queue at the end and are served from the front, similar to a queue. However, sometimes a person who has already purchased a ticket may come back to the front of the line to ask a follow-up question.

Because they have already purchased a ticket, they have the privilege to come to the front and be served again. Similarly, a person can leave the line from the rear if they decide not to wait anymore.

Deques are often used in applications where elements need to be added and removed from both ends efficiently. They are useful in scenarios like task scheduling, sliding window algorithms, and buffering data streams.

The deque is useful as it can be used as both a stack and queue, depending on the specific needs of an application.

Comparison Between Stack, Queue and Deque

Feature	Stack	Queue	Deque
Access Ends	Top (one end)	Front and Back	Both ends
Data Organization	Linear	Linear	Linear
Use cases	LIFO scenarios	FIFO scenarios	Versatile

1.2.1 Linear Data Structures

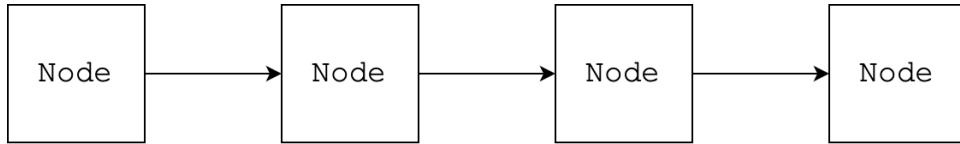
These data structures are made up of a line of data objects (or “**nodes**”) that are able to maintain an order, and where every object has at least one but no more than two neighbors.

Think about a line of people trying to buy something at a shop - every person has one person in front of them and one person behind them (unless they are at the front or end of the line), and the order of the line does not randomly change.

For some types of linear lists, there are different ways to implement them. A common approach is using a “linked” data structure where each element is connected to the one next to it, creating a chain-like structure.

In linear lists, there is always a clear beginning and an end. This makes it simpler to go through each element in the list one by one, a process we call traversal or iteration.

Below is a generalized diagram which shows this “linked” structure



It's like reading a book from the first page to the last, or walking through a line of people, starting with the first person and ending with the last.

As mentioned before linear data structures include Arrays, Linked Lists, Queues and Stacks.

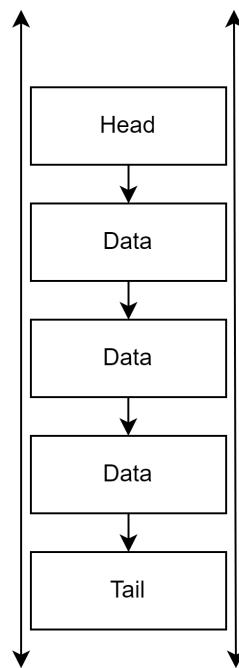
Each is structured linearly but what differs in element accessibility and because of this each have their own pros and cons in various use cases.

We will dive deeper into specific implementations in subsequent lessons.

1.2.1 Exercises

Exercise 1

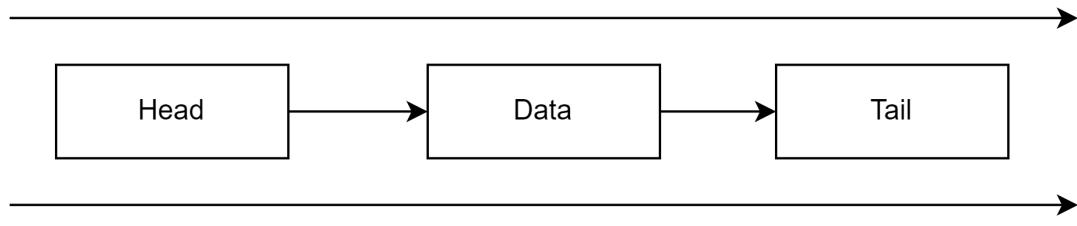
What data structure is represented by the image?



- Queue
- Deque
- Linked List
- Stack

Exercise 2

What data structure is represented by the image



- Queue
- Tree
- Stack
- Linked List

Exercise 3

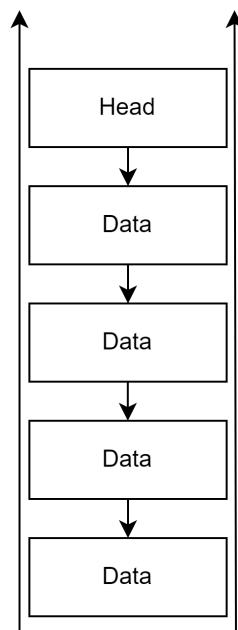
What data structure is represented by the image?



- Queue
- Tree
- Linked List
- Graph

Exercise 4

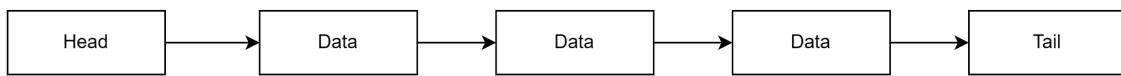
What data structure is represented by the image?



- Queue
- Stack
- Linked List
- Tree

Exercise 5

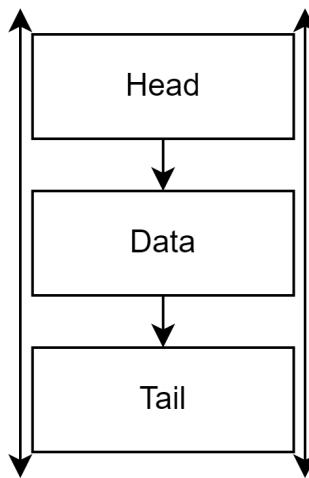
What data structure is represented by the image?



- Linked List
- Tree
- Stack
- Graph

Exercise 6

What data structure is represented by the image?



- Deque
- Queue
- Linked List
- Stack

Exercise 13

What is the organizational structure of a stack? Select all correct answers

- First in first out
- First in last out
- Last in first out
- Last in last out

Exercise 14

What is the organizational structure of a queue? Select all correct answers

- First in first out
- First in last out
- Last in first out
- Last in last out

Exercise 15

What is the organizational structure of a deque? Select all correct answers

- First in first out
- First in last out
- Last in first out
- Last in last out

Exercise 16

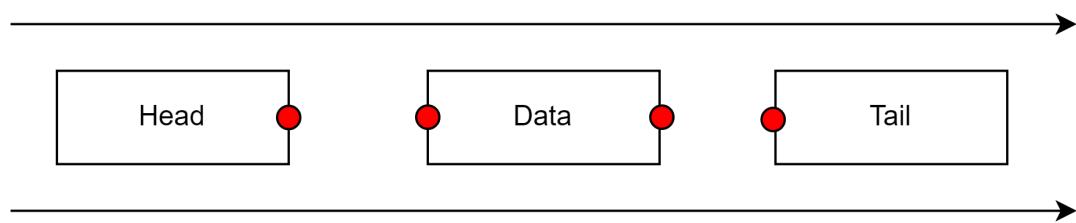
What are the first and last objects in a linked list called? Select all correct answers

- Head
- Start
- Tail
- End

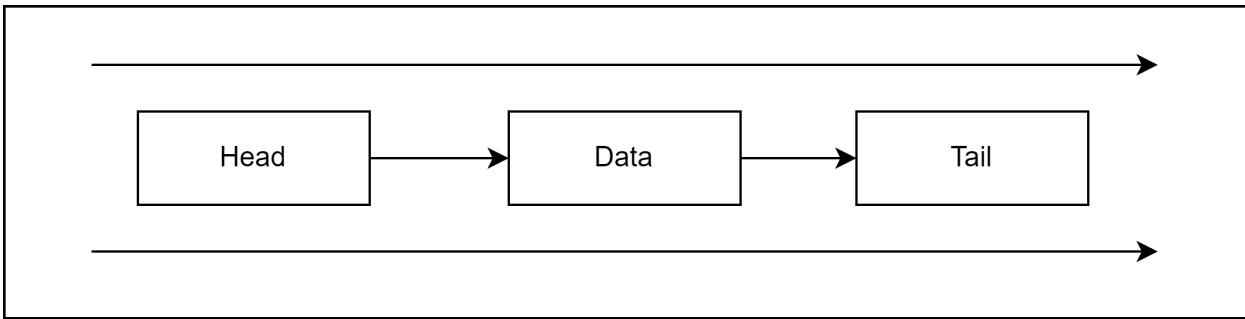
Exercises 7 - 12 (Draw Links)

Exercise 7

Link the data with the arrows to create a queue.



Expected Input



Exercise 8

Link the data with the arrows to create a linked list.

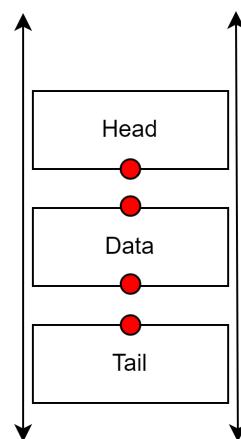


Expected Input

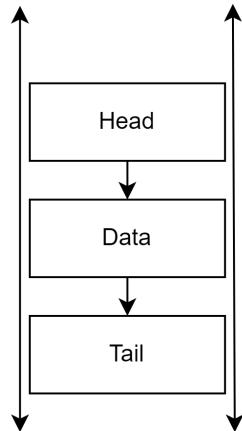


Exercise 9

Link the data with the arrows to create a deque.

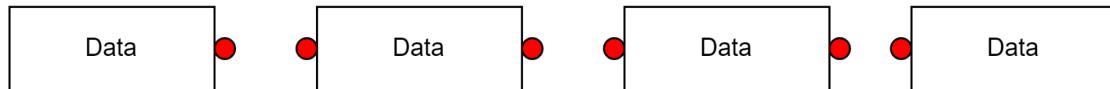


Expected Input

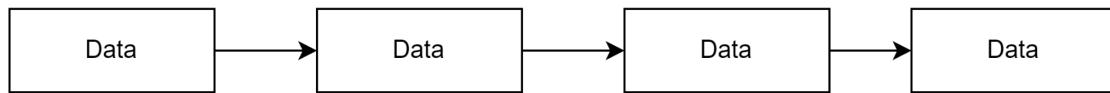


Exercise 10

Link the data with the arrows to create a linked list.

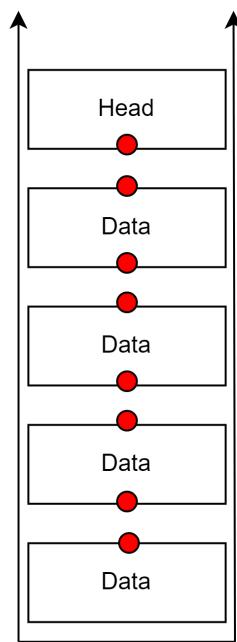


Expected Input

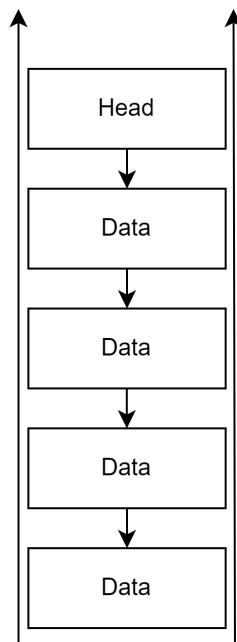


Exercise 11

Link the data with the arrows to create a stack.

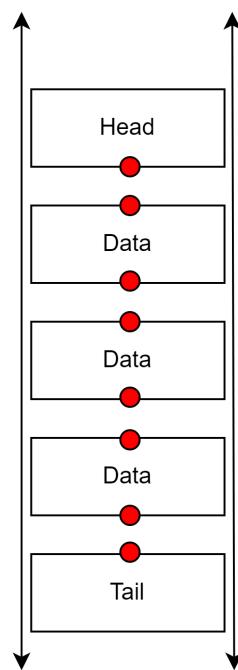


Expected Input

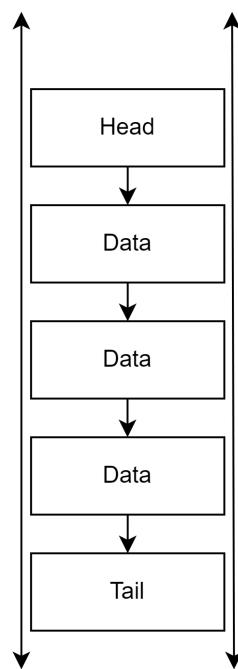


Exercise 12

Link the data with the arrows to create a deque.



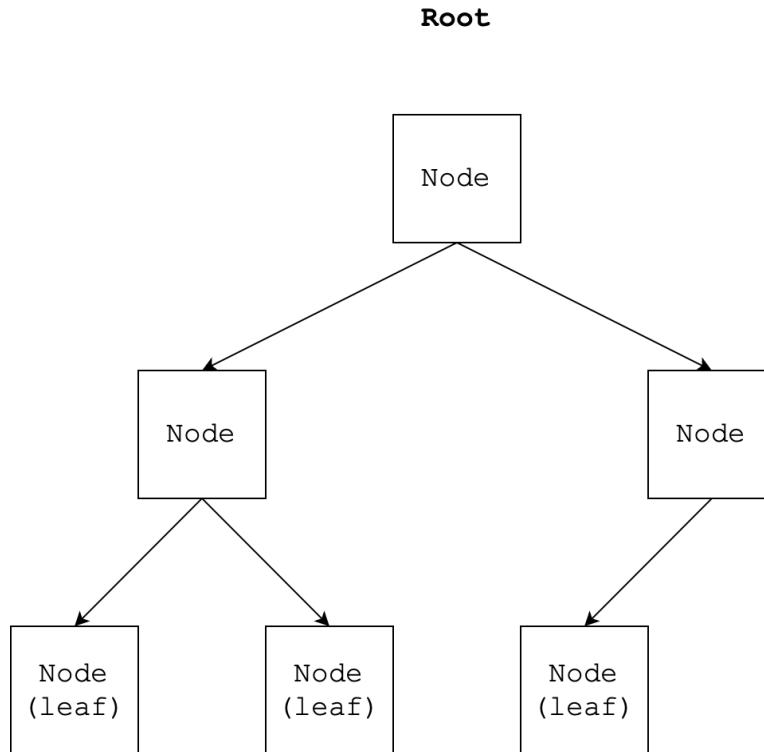
Expected Input



1.2.2 Non-Linear Data Structures

Trees

Trees are a non-linear structure made up of a single root data object, that in a branching structure, links to objects that in turn branch to other nodes and so on, creating a structure that consists of layers of nodes linking to nodes on the layer below them.



The tree structure is useful in searching algorithms as data can be organized in a hierarchical structure that allows a searching algorithm to compare the branches of any individual object before moving down the tree in a search, eliminating options and reducing the search time. Similar to the **head** of Linear list, there is only one node at the top that is **Root**. However, there are more than one **tail** in Tree data structure, all of them are **Leaf**.

Examples:

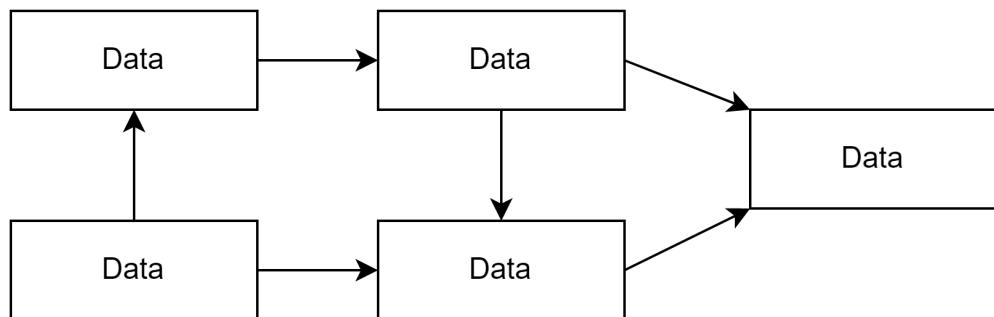
Imagine a family tree:

- The root is the oldest ancestor (great-grandparent).
- Each child of the root is a grandparent.
- Each grandparent has children who are parents.
- Each parent has children who are siblings to each other.
- The youngest generation are the leaves of the tree.

In this family tree, you can trace the path from any leaf back to the root to see the lineage. Traversing the tree might involve visiting each generation level by level or tracing individual family lines.

Graphs:

Graphs are a non-linear data structure made up of **vertices** and **edges**. The edges connect any two nodes in the graph, and the nodes are also known as vertices. Graphs are by far the most complex data structure, being made up of data linked to any other data in the graph.

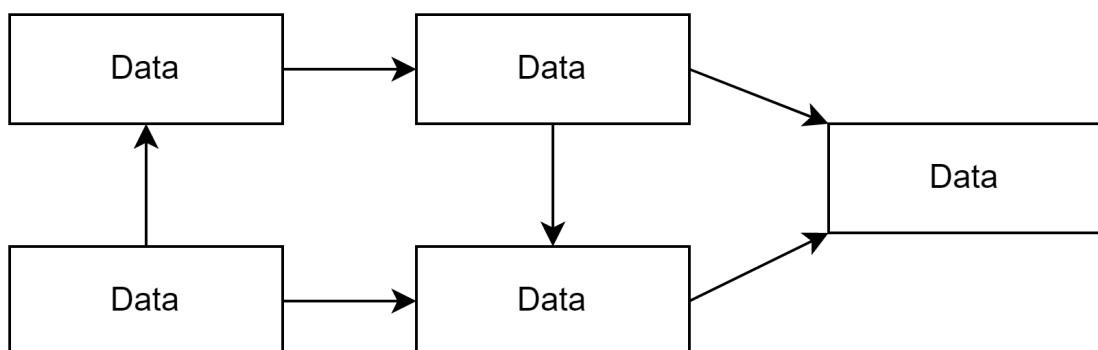


This allows their structure to vary to a large degree. The extremely versatile structure of graphs allows them to model complex real-world data such as routes between locations, social networks and the World Wide Web. Graphs are used to solve a number of unique problems; however, their complex structure requires complex algorithms to perform many simple tasks, such as searching, route optimization in GPS navigation systems.

Exercises

Exercise 1

What data structure is represented in the image?

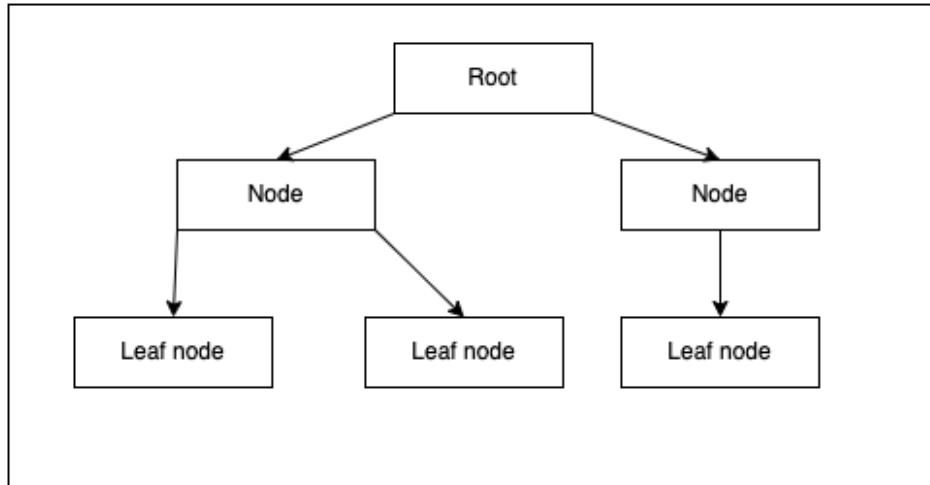


- Queue
- Tree

- Linked List
- Graph

Exercise 2

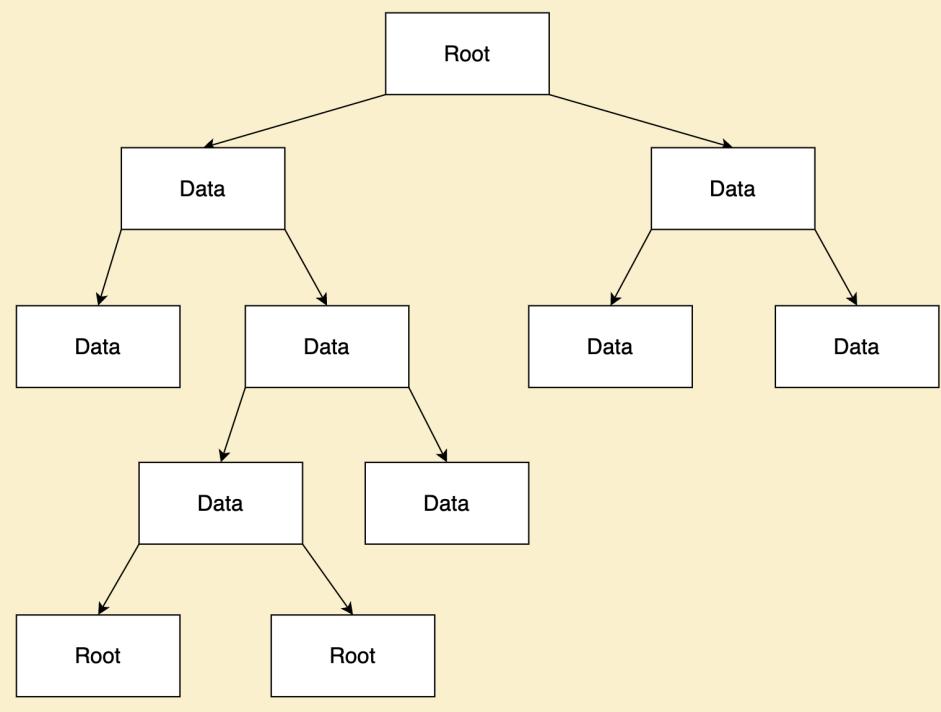
What data structure is represented in the image?



- Graph
- Tree
- Linked List
- Queue

Exercise 3

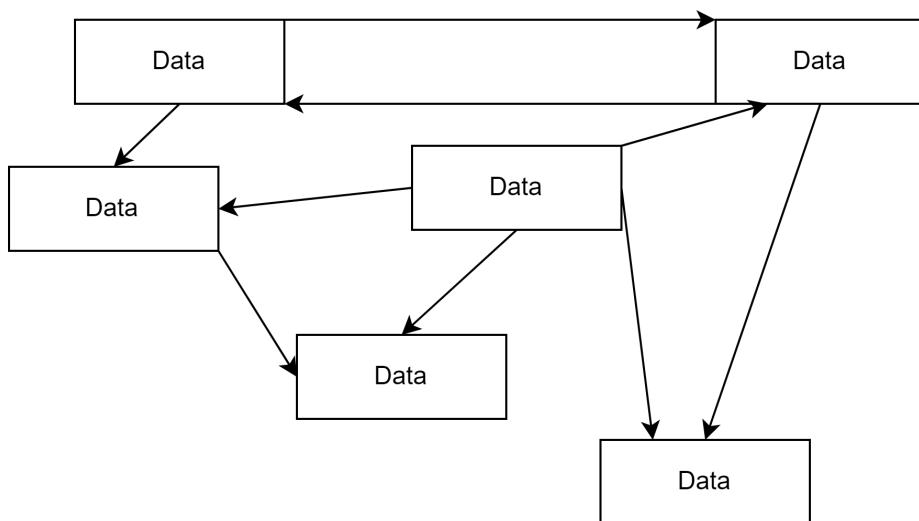
What data structure is represented in the image?



- Queue
- Tree
- Linked List
- Graph

Exercise 4

What data structure is represented by the image?



- Queue
- Tree
- Linked List
- Graph

Exercise 9

What is the data object at the top layer of a tree called?

- Root
- Head
- Top
- Main

Exercise 12

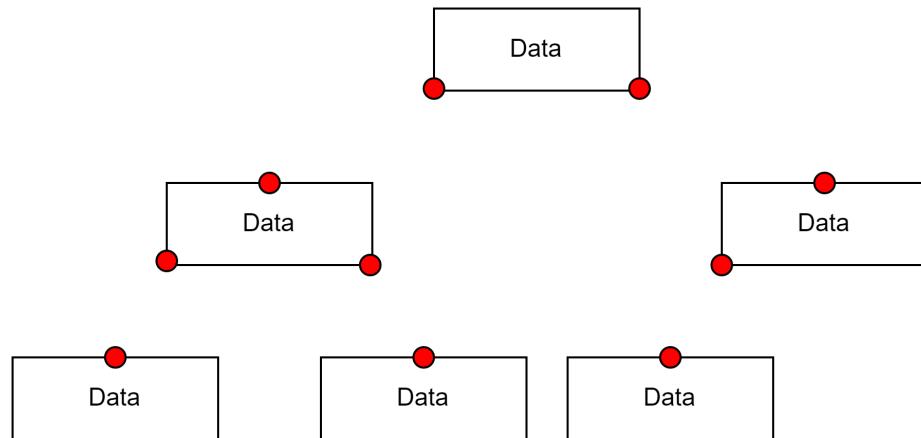
What are the benefits of using the advanced data structures we have seen (e.g. queue, stack, tree) as compared to Python's built-in data structures? Select all correct answers.

- Advanced data structures can provide more efficient algorithms for specific problems, improving time and space complexity.
- Advanced data structures are easier to implement and use than Python's built-in data structures.
- Advanced data structures eliminate the need for error handling and data validation, unlike Python's built-in data structures.
- Python's built-in data structures offer more flexibility and functionality than advanced data structures like queue, stack, and tree.

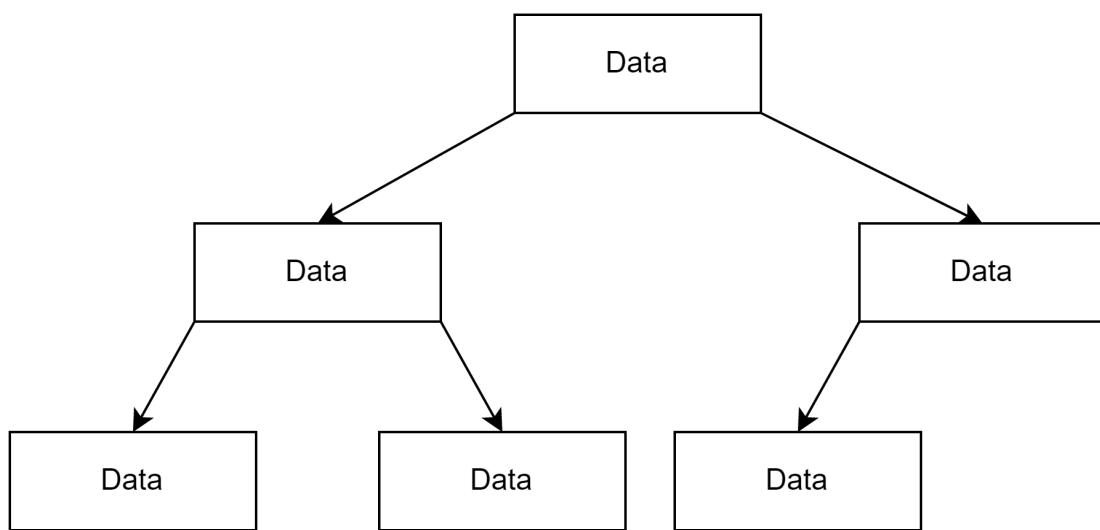
Exercises 5 - 8 (Draw Links)

Exercise 5

Link the data with the arrows to create a tree.

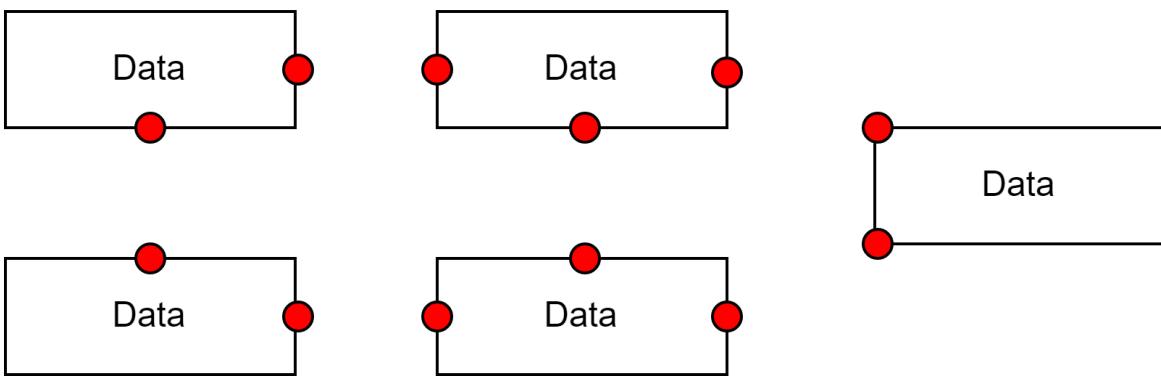


Expected Input



Exercise 6

Link the data with the arrows to create a graph. Note: the direction of the arrows does not matter.

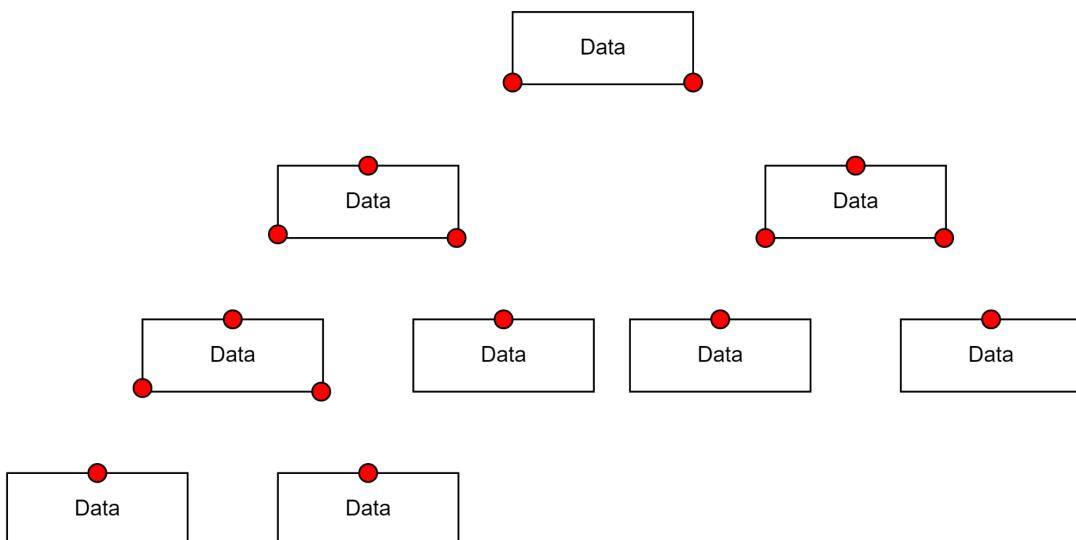


Expected Input

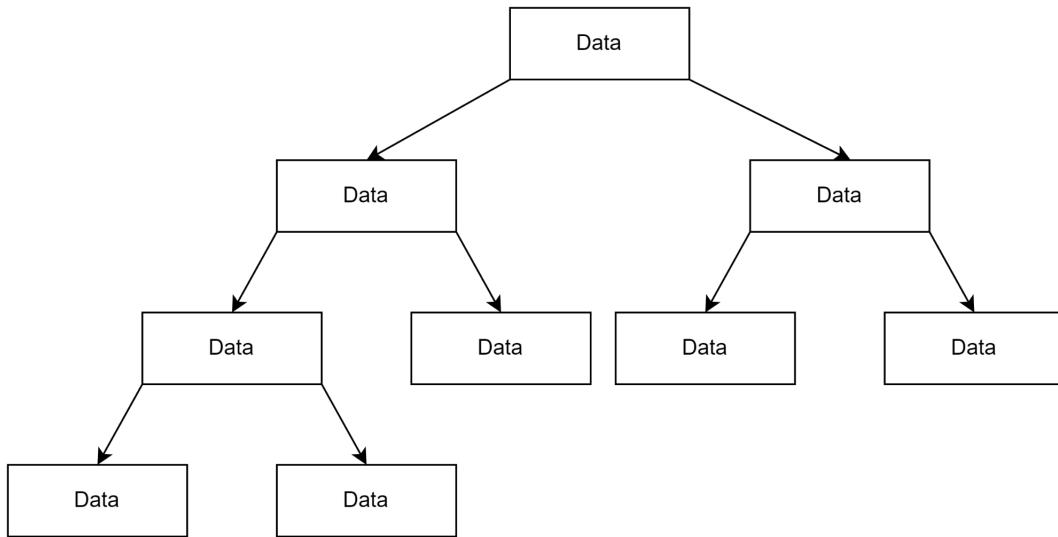
Any combination of connections is correct.

Exercise 7

Link the data with the arrows to create a tree.

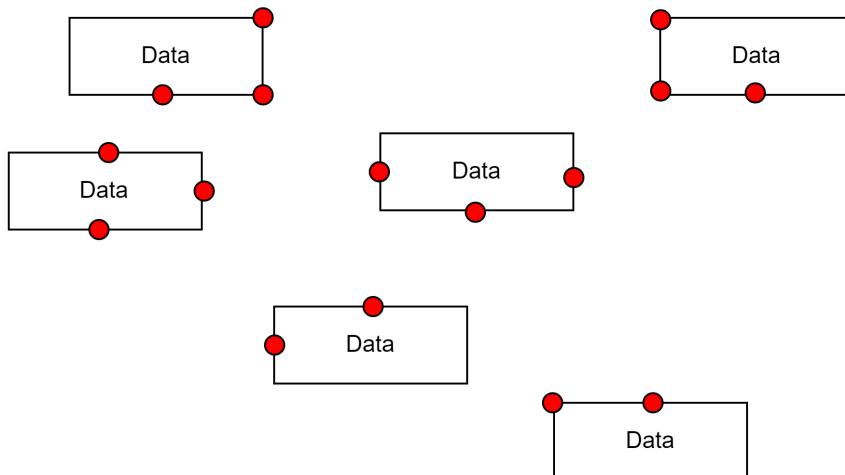


Expected Input



Exercise 8

Link the data with the arrows to create a graph. Note: the direction of the arrows does not matter.



Expected Input

Any combination of connections is correct.

1.2.2 Non-Linear Data Structures

Non-linear data structures differ from linear lists because of the non-linear arrangement of their nodes.

While linear lists are made up of objects linked together with at most two adjacent objects in a

chain, non-linear structures contain linking structures that can have any number of links (> 1) to any other object in the structure.

This means that these structures have no limit on the patterns of linking that can be created, but it also causes their structure to become more complex and challenging to manage, as performing traversals of the data structure becomes difficult when it is non-linear.

Implementations we will explore later include trees and graphs, both of which follow a non-linear structure but differ in their specific implementations.

3.2 Inserting data in a singularly linked list

Ex 20

Write a function to prepend a node at the beginning of a linked list

Expected input

```
def insert_at_beginning(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node
    self.size += 1
```

5.3 Circular Doubly Linked List

A circular doubly linked list is a data structure that combines the features of a **doubly linked list** and a **circular linked list**. It consists of a sequence of nodes, where each node contains three components: **data**, a **reference to the next node**, and a **reference to the previous node**. The last node's next reference points back to the head node, forming a circular loop, and the head node's **prev** reference points to the last node, completing the circular structure.

The circular doubly linked list is almost identical to the singly linked list in terms of the changes needed for a circular implementation.

This data structure maintains the benefits of circular singly linked lists in terms of traversal, but also allows bidirectional traversal.

The methods used to add nodes to a circular linked list need to carefully manage the linking between the head and tail nodes of the list.

This is because they are inserted into and we need to ensure that indexes outside of the range are not accessed incorrectly.

The same management also needs to take into account the bidirectional links when

performing these actions.

Circular doubly linked lists support various basic operations, including:

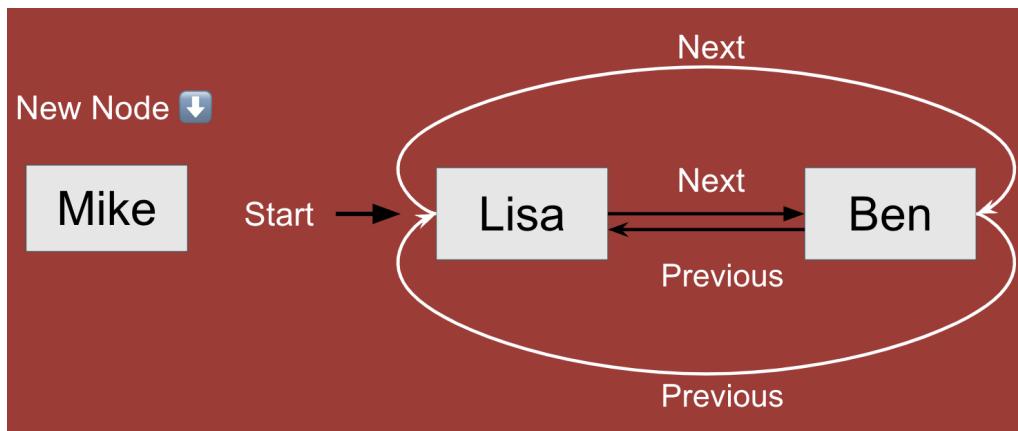
- **Insertion:** Adding a new node at the beginning, end, or at a specific position in the list.
- **Deletion:** Removing a node from the beginning, end, or from a specific position in the list.
- **Traversal:** Accessing each node in the list sequentially, either in a forward or backward direction.
- **Searching:** Finding a specific node based on its data value.

First, let's go through the insertion operations.

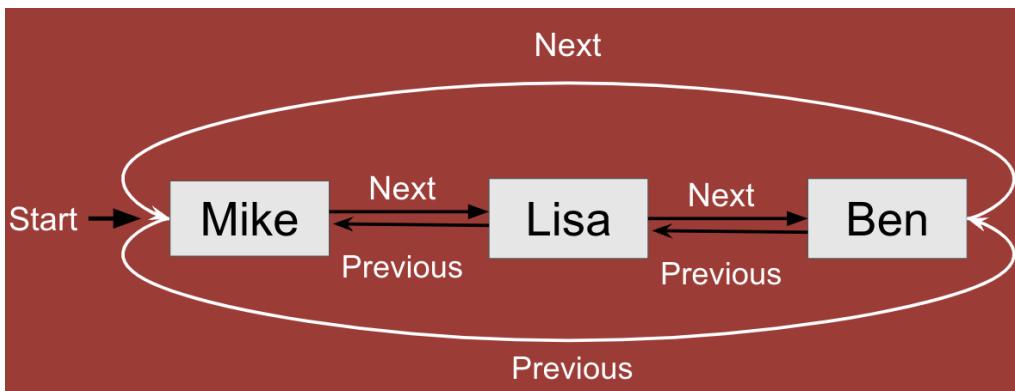
Inserting at the beginning - prepend

Inserting a node at the beginning of a circular doubly linked list involves adding a new node as the new head of the list. As we've learned similar method before, let's see a quick example:

There is a circular doubly linked list with 2 nodes, one node with data "Lisa" and one node with data "Ben". Now we want to insert a new node with data "Mike" at the beginning of the linked list.



The next of "Mike" should point to the first node(data with "Lisa") of the list and the previous pointer of "Mike" should point to the last node(data with "Ben") of the list, the next of last node(data with "Ben") should point to this new node(data with "Mike"). The previous pointer of the first node(data with "Lisa") also needs to point to the new node. Finally, shift the 'Start' pointer to the new node.



Let's see the basic logic before implementation:

1. **Create a new node with the given data.**
2. **Make the new node's next reference point to the current head.**
 - a. The new node will become the new head, so its next reference should point to the current head node.
3. **Make the new node's prev reference point to the last node (head's prev).**
 - a. Since the list is circular, the prev reference of the new head should point to the last node in the list.
 - b. The last node can be accessed through the prev reference of the current head.
4. **Update the last node's next reference to point to the new node.**
 - a. The last node's next reference should now point to the new node, establishing the circular connection.
5. **Update the head's prev reference to point to the new node.**
 - a. The current head's prev reference should now point to the new node, completing the circular connection.
6. **Make the new node the new head.**
 - a. Update the head pointer to point to the new node, effectively making it the new head of the list.
7. **Increment the size of the list.**
 - a. Increase the size variable by 1 to reflect the addition of the new node.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None

```

```

self.size = 0

def prepend(self, data):
    new_node = Node(data) # Create a new node with the given data

    if self.head is None: # If the list is empty
        self.head = new_node # Make the new node the head
        new_node.next = new_node # Make the new node's next reference point to itself
        new_node.prev = new_node # Make the new node's prev reference point to itself
    else:
        new_node.next = self.head # Make the new node's next reference point to the current
head
        new_node.prev = self.head.prev # Make the new node's prev reference point to the
last node
        self.head.prev.next = new_node # Update the last node's next reference to point to
the new node
        self.head.prev = new_node # Update the head's prev reference to point to the new
node
        self.head = new_node # Make the new node the new head

    self.size += 1 # Increment the size of the list

def print_list(self):
    if self.head is None: # If the list is empty
        print("Empty List")
        return

    current = self.head # Start from the head node
    print(current.data, end=" ") # Print the head node's data
    current = current.next # Move to the next node

    while current != self.head: # Traverse the list until we reach the head again
        print(current.data, end=" ") # Print the current node's data
        current = current.next # Move to the next node

    print() # Print a newline after the list is printed

# Create a circular doubly linked list
cdll = CircularDoublyLinkedList()

# Insert nodes at the beginning
cdll.prepend(3)
cdll.prepend(2)

```

```

cdll.prepend(1)

# Print the list
print("Circular Doubly Linked List:")
cdll.print_list()

```

Output:

Circular Doubly Linked List:

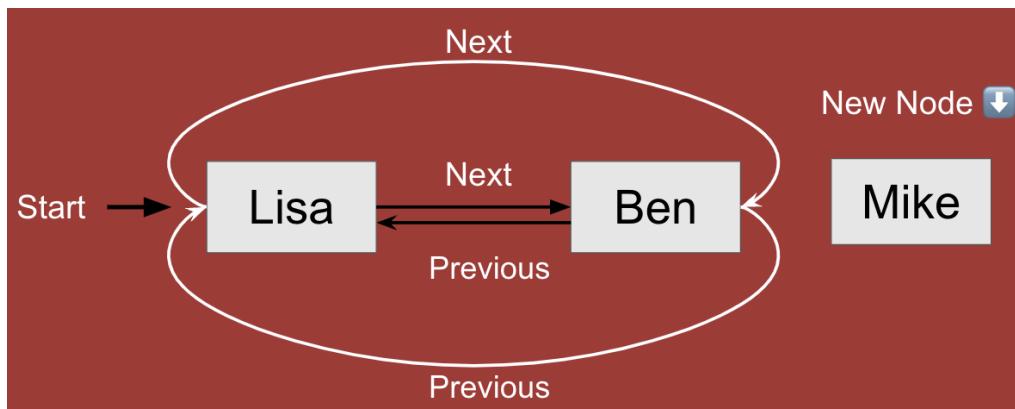
1 2 3

Let's move on to the next operation: inserting a node at the end of a circular doubly linked list!

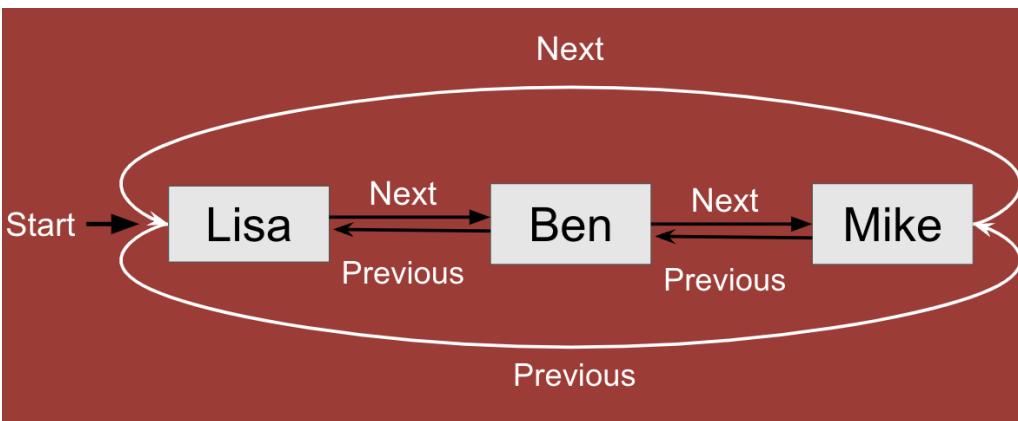
Inserting at the End (Append):

Inserting a node at the end of a circular doubly linked list involves adding a new node as the last node in the list.

Consider a circular doubly linked list with 2 nodes, one node with data "Lisa" and one node with data "Ben". The "Lisa" node is the head of the list, and the "Ben" node is the last node. Now, we want to insert a new node with data "Mike" at the end of the linked list.



The next pointer of the new node "Mike" should point to the head node (data with "Lisa") of the list, establishing the circular connection. The previous pointer of the new node "Mike" should point to the current last node (data with "Ben") of the list. Update the next pointer of the current last node "Ben" to point to the new node "Mike", effectively making "Mike" the new last node. Update the previous pointer of the head node "Lisa" to point to the new node "Mike", completing the circular connection.



Let's see the basic logic before implementation:

1. **Create a new node with the given data.**
2. **Make the new node's next reference point to the head.**
 - a. Since the list is circular, the last node's next reference should point to the head node.
3. **Make the new node's prev reference point to the current last node.**
 - a. The current last node can be accessed through the prev reference of the head.
4. **Update the last node's next reference to point to the new node.**
 - a. The current last node's next reference should now point to the new node.
5. **Update the head's prev reference to point to the new node.**
 - a. The head's prev reference should now point to the new node, establishing the new node as the last node.
6. **Increment the size of the list.**
 - a. Increase the size variable by 1 to reflect the addition of the new node.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data) # Create a new node with the given data

        if self.head is None: # If the list is empty

```

```

        self.head = new_node # Make the new node the head
        new_node.next = new_node # Make the new node's next reference point to itself
        new_node.prev = new_node # Make the new node's prev reference point to itself
    else:
        new_node.next = self.head # Make the new node's next reference point to the head
        new_node.prev = self.head.prev # Make the new node's prev reference point to the
        current last node
        self.head.prev.next = new_node # Update the current last node's next reference to
        point to the new node
        self.head.prev = new_node # Update the head's prev reference to point to the new
        node

    self.size += 1 # Increment the size of the list

def print_list(self):
    if self.head is None: # If the list is empty
        print("Empty List")
        return

    current = self.head # Start from the head node
    print(current.data, end=" ") # Print the head node's data
    current = current.next # Move to the next node

    while current != self.head: # Traverse the list until we reach the head again
        print(current.data, end=" ") # Print the current node's data
        current = current.next # Move to the next node

    print() # Print a newline after the list is printed

# Create a circular doubly linked list
cdll = CircularDoublyLinkedList()

# Append nodes at the end
cdll.append(1)
cdll.append(2)
cdll.append(3)

# Print the list
print("Circular Doubly Linked List:")
cdll.print_list()

```

Output:

Circular Doubly Linked List:

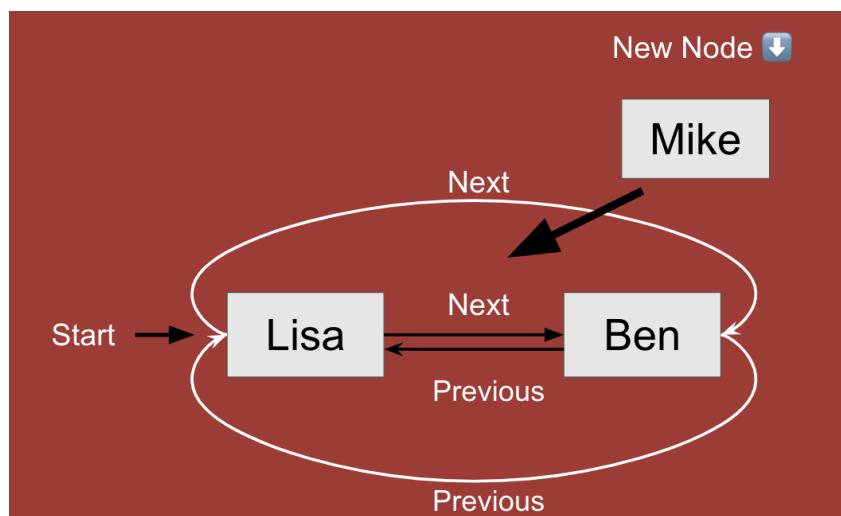
1 2 3

Let's move on to the next operation: inserting a node at a specific position in a circular doubly linked list.

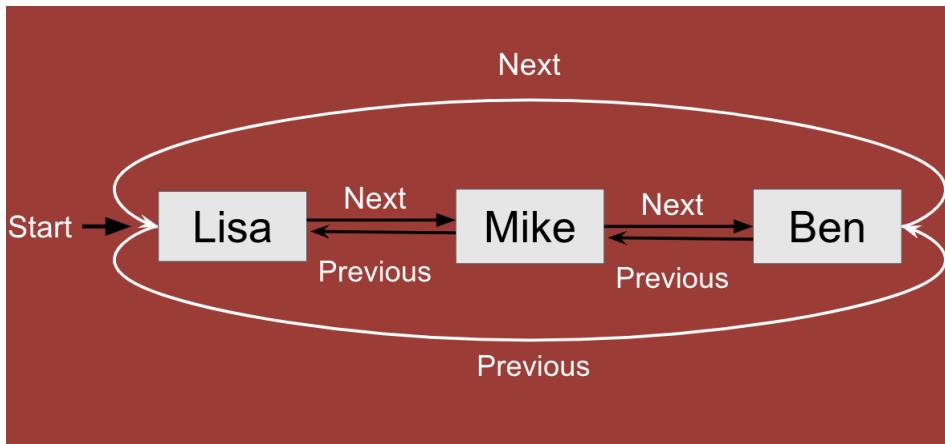
Inserting at a Specific Position:

Inserting a node at a specific position in a circular doubly linked list involves adding a new node at the desired position and updating the references of the surrounding nodes accordingly.

Consider a circular doubly linked list with 2 nodes, one node with data "Lisa" and one node with data "Ben". The "Lisa" node is the head of the list, and the "Ben" node is the last node. Now, we want to insert a new node with data "Mike" between "Lisa" and "Ben".



We should traverse the list to the node just before the desired position, which is the "Lisa" node in this case. Then make the new node's next reference point to the node after the current position, which is the "Ben" node; make the new node's prev reference point to the node before the current position, which is the "Lisa" node; Update the next reference of the node before the current position ("Lisa") to point to the new node ("Mike") and update the prev reference of the node after the current position ("Ben") to point to the new node ("Mike").



Let's see the basic logic before implementation:

1. If the position is 0, use the prepend method to insert the node at the beginning.
2. If the position is equal to the size of the list, use the append method to insert the node at the end.
3. Otherwise, traverse the list to the node just before the desired position.
 - a. Initialize a variable (e.g., current) to store the current node during traversal.
 - b. Set current to the head of the list.
 - c. Traverse the list until the desired position is reached or the end of the list is encountered.
4. Create a new node with the given data.
5. Make the new node's next reference point to the node after the current position.
 - a. The node after the current position can be accessed through the next reference of the current node.
6. Make the new node's prev reference point to the node before the current position (i.e., current).
7. Update the next reference of the node before the current position (current) to point to the new node.
8. Update the prev reference of the node after the current position to point to the new node.
9. Increment the size of the list.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class CircularDoublyLinkedList:
    def __init__():

```

```

self.head = None
self.size = 0

def insert_at(self, position, data):
    if position < 0 or position > self.size: # Check if the position is valid
        raise IndexError("Invalid position")

    if position == 0: # If the position is 0, use the prepend method
        self.prepend(data)
    elif position == self.size: # If the position is equal to the size, use the append method
        self.append(data)
    else:
        new_node = Node(data) # Create a new node with the given data
        current = self.head # Start from the head node

        for _ in range(position - 1): # Traverse to the node just before the desired position
            current = current.next

        new_node.next = current.next # Make the new node's next reference point to the
        node after the current position
        new_node.prev = current # Make the new node's prev reference point to the node
        before the current position
        current.next.prev = new_node # Update the prev reference of the node after the
        current position to point to the new node
        current.next = new_node # Update the next reference of the node before the current
        position to point to the new node

        self.size += 1 # Increment the size of the list

def print_list(self):
    if self.head is None: # If the list is empty
        print("Empty List")
        return

    current = self.head # Start from the head node
    print(current.data, end=" ") # Print the head node's data
    current = current.next # Move to the next node

    while current != self.head: # Traverse the list until we reach the head again
        print(current.data, end=" ") # Print the current node's data
        current = current.next # Move to the next node

    print() # Print a newline after the list is printed

```

```

# Create a circular doubly linked list
cdll = CircularDoublyLinkedList()

# Insert nodes at specific positions
cdll.insert_at(0, 1) # Insert 1 at position 0
cdll.insert_at(1, 2) # Insert 2 at position 1
cdll.insert_at(1, 3) # Insert 3 at position 1
cdll.insert_at(3, 4) # Insert 4 at position 3

# Print the list
print("Circular Doubly Linked List:")
cdll.print_list()

```

Output:

Circular Doubly Linked List:

1 3 2 4

Congratulations! Now let's move on to the deletion operations in a circular doubly linked list. We'll cover three scenarios: deleting the head node, deleting the last node, and deleting a node at a specific position.

Deleting the Head Node (`remove_first`):

To delete the head node from a circular doubly linked list, we need to update the references of the surrounding nodes and handle the case where the list becomes empty.

Let's see the basic logic before implementation:

1. Check if the list is empty. If it is, there's nothing to delete, so we return.
2. Check if there is only one node in the list (i.e., the head's next reference points to itself). In this case, we simply set the head to None, effectively emptying the list.
3. If there are multiple nodes in the list:
 - a. Update the prev reference of the new head (the node after the current head) to point to the last node (the prev of the current head).
 - b. Update the next reference of the last node (the prev of the current head) to point to the new head (the node after the current head).
 - c. Make the next node (after the current head) the new head of the list.
4. Decrement the size of the list by 1 to reflect the removal of the head node.

```
def remove_first(self):
```

```

if self.head is None: # If the list is empty
    return

if self.head.next == self.head: # If there is only one node in the list
    self.head = None
else:
    self.head.next.prev = self.head.prev # Update the prev reference of the new head
    self.head.prev.next = self.head.next # Update the next reference of the last node
    self.head = self.head.next # Make the next node the new head

self.size -= 1 # Decrement the size of the list

```

Deleting the Last Node (`remove_last`):

To delete the last node from a circular doubly linked list, we need to update the references of the surrounding nodes and handle the case where the list becomes empty.

Let's see the basic logic before implementation:

1. Check if the list is empty. If it is, there's nothing to delete, so we return.
2. Check if there is only one node in the list (i.e., the head's next reference points to itself). In this case, we simply set the head to None, effectively emptying the list.
3. If there are multiple nodes in the list:
 - a. Get the last node by accessing the prev reference of the head.
 - b. Update the next reference of the new last node (the prev of the current last node) to point to the head.
 - c. Update the prev reference of the head to point to the new last node (the prev of the current last node).
4. Decrement the size of the list by 1 to reflect the removal of the last node.

```

def remove_last(self):
    if self.head is None: # If the list is empty
        return

    if self.head.next == self.head: # If there is only one node in the list
        self.head = None
    else:
        last_node = self.head.prev # Get the last node
        last_node.prev.next = self.head # Update the next reference of the new last node
        self.head.prev = last_node.prev # Update the prev reference of the head

    self.size -= 1 # Decrement the size of the list

```

Deleting a Node at a Specific Position (`remove_at`):

To delete a node at a specific position in a circular doubly linked list, we need to traverse the list to the desired position and update the references of the surrounding nodes accordingly.

Let's see the basic logic before implementation:

1. Check if the given position is valid. If it is less than 0 or greater than or equal to the size of the list, raise an `IndexError` with an appropriate message.
2. If the position is 0, use the `remove_first` method to delete the head node.
3. If the position is the last index (`size - 1`), use the `remove_last` method to delete the last node.
4. If the position is within the valid range (excluding the head and last node):
 - a. Start from the head node and traverse the list until the desired position is reached.
 - b. Update the next reference of the previous node (the `prev` of the current node) to point to the next node (the `next` of the current node).
 - c. Update the `prev` reference of the next node (the `next` of the current node) to point to the previous node (the `prev` of the current node).
5. Decrement the size of the list by 1 to reflect the removal of the node.

```
def remove_at(self, position):
    if position < 0 or position >= self.size: # Check if the position is valid
        raise IndexError("Invalid position")

    if position == 0: # If the position is 0, use the remove_first method
        self.remove_first()
    elif position == self.size - 1: # If the position is the last index, use the remove_last method
        self.remove_last()
    else:
            current = self.head # Start from the head node

            for _ in range(position): # Traverse to the node at the desired position
                    current = current.next

                current.prev.next = current.next # Update the next reference of the previous node
                current.next.prev = current.prev # Update the prev reference of the next node

        self.size -= 1 # Decrement the size of the list
```

Ex 1-7 (Guide to modify the linked list method)

Exercise 1:

Fill in the blanks to complete the append() method.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.prev = None  
        self.next = None  
  
class CircularDoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
        new_node = Node(data) # Create a new node with the given data  
  
        if self.head is None: # If the list is empty  
            _____ # Make the new node the head  
            _____ # Make the new node's next reference point to itself  
            _____ # Make the new node's prev reference point to itself  
        else:  
            _____ # Make the new node's next reference point to the head  
            _____ # Make the new node's prev reference point to the current last node  
            _____ # Update the current last node's next reference to point to the new node  
            _____ # Update the head's prev reference to point to the new node  
  
        _____ # Increment the size of the list
```

Expected Answer:

```
def append(self, data):  
    new_node = Node(data) # Create a new node with the given data  
  
    if self.head is None: # If the list is empty  
        self.head = new_node # Make the new node the head  
        new_node.next = new_node # Make the new node's next reference point to itself  
        new_node.prev = new_node # Make the new node's prev reference point to itself  
    else:  
        new_node.next = self.head # Make the new node's next reference point to the head  
        new_node.prev = self.head.prev # Make the new node's prev reference point to the  
        current last node  
        self.head.prev.next = new_node # Update the current last node's next reference to  
        point to the new node
```

```

        self.head.prev = new_node # Update the head's prev reference to point to the new
node

        self.size += 1 # Increment the size of the list

```

Exercise 2:

Fill in the blanks to complete the prepend() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def prepend(self, data):
        new_node = Node(data) # Create a new node with the given data

        _____ # If the list is empty
        _____ # Make the new node the head
        _____ # Make the new node's next reference point to itself
        _____ # Make the new node's prev reference point to itself
else:
        _____ # Make the new node's next reference point to the current head
        _____ # Make the new node's prev reference point to the last node
        _____ # Update the last node's next reference to point to the new node
        _____ # Update the head's prev reference to point to the new node
        _____ # Make the new node the new head

        _____ # Increment the size of the list

```

Expected Answer:

```

def prepend(self, data):
    new_node = Node(data) # Create a new node with the given data

    if self.head is None: # If the list is empty

```

```

        self.head = new_node # Make the new node the head
        new_node.next = new_node # Make the new node's next reference point to itself
        new_node.prev = new_node # Make the new node's prev reference point to itself
    else:
        new_node.next = self.head # Make the new node's next reference point to the current
head
        new_node.prev = self.head.prev # Make the new node's prev reference point to the
last node
        self.head.prev.next = new_node # Update the last node's next reference to point to
the new node
        self.head.prev = new_node # Update the head's prev reference to point to the new
node
        self.head = new_node # Make the new node the new head

    self.size += 1 # Increment the size of the list

```

Exercise 3:

Fill in the blanks to complete the insert_at() method.

```

def insert_at(self, position, data):
    if position < 0 or position > self.size: # Check if the position is valid
        raise IndexError("Invalid position")

    if position == 0: # If the position is 0, use the prepend method
        _____
    elif position == self.size: # If the position is equal to the size, use the append method
        _____
    else:
        _____ # Create a new node with the given data
        _____ # Start from the head node
        _____ # Traverse to the node just before the desired position
        _____
        _____ # Make the new node's next reference point to the node after the
current position
        _____ # Make the new node's prev reference point to the node before the
current position
        _____ # Update the prev reference of the node after the current position to
point to the new node
        _____ # Update the next reference of the node before the current position to

```

point to the new node

_____ # Increment the size of the list

Expected Answer:

```
def insert_at(self, position, data):
    if position < 0 or position > self.size: # Check if the position is valid
        raise IndexError("Invalid position")

    if position == 0: # If the position is 0, use the prepend method
        self.prepend(data)
    elif position == self.size: # If the position is equal to the size, use the append method
        self.append(data)
    else:
        new_node = Node(data) # Create a new node with the given data
        current = self.head # Start from the head node

        for _ in range(position - 1): # Traverse to the node just before the desired position
            current = current.next

        new_node.next = current.next # Make the new node's next reference point to the
        # node after the current position
        new_node.prev = current # Make the new node's prev reference point to the node
        # before the current position
        current.next.prev = new_node # Update the prev reference of the node after the
        # current position to point to the new node
        current.next = new_node # Update the next reference of the node before the current
        # position to point to the new node

    self.size += 1 # Increment the size of the list
```

Exercise 4:

The deletion methods are almost identical to the single circular linked list only providing extra support for the reassignment of links in the other direction. Fill in the blanks to complete the delete() method.

```
class Node:
```

```

def __init__(self, data):
    self.data = data
    self.next = None
    self.prev = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def delete(self, data):
        cur_node = self.head
        while cur_node:
            if cur_node.data == data:
                if cur_node == self.head:
                    _____
                    _____
                    _____
                    self.size -= 1
                    return
                if cur_node == self.tail:
                    _____
                    _____
                    _____
                    self.size -= 1
                    return
                _____
                self.size -= 1
                return

            if cur_node == self.head:
                break

```

Expected Answer:

```

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = self.tail

```

```

    self.tail.next = self.head
    self.size -= 1
    return
if cur_node == self.tail:
    self.tail = cur_node.prev
    self.tail.next = self.head
    self.head.prev = self.tail
    self.size -= 1
    return
cur_node.prev.next = cur_node.next
cur_node.next.prev = cur_node.prev
self.size -= 1
return
cur_node = cur_node.next
if cur_node == self.head:
    break

```

Exercise 5:

Fill in the blanks to complete the delete_at() method.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def delete_at(self, index):
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                if cur_node == self.head:
                    _____
                    _____
                    _____

```

```

self.size -= 1
return
if cur_node == self.tail:
    _____
    _____
    _____
self.size -= 1
return
_____
_____
_____
self.size -= 1
return
_____
count += 1
if cur_node == self.head:
    break

```

Expected Answer:

```

def delete_at(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = self.tail
                self.tail.next = self.head
                self.size -= 1
            return
        if cur_node == self.tail:
            self.tail = cur_node.prev
            self.tail.next = self.head
            self.head.prev = self.tail
            self.size -= 1
        cur_node.prev.next = cur_node.next
        cur_node.next.prev = cur_node.prev
        self.size -= 1
    return
    cur_node = cur_node.next
    count += 1
    if cur_node == self.head:
        break

```

Exercise 6:

Create the get() method.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.current = None  
  
    def get(self, index):  
        pass
```

Expected Answer:

```
def get(self, index):  
    cur_node = self.head  
    count = 0  
    while cur_node:  
        if count == index:  
            return cur_node.data  
        cur_node = cur_node.next  
        count += 1  
    if cur_node == self.head:  
        break
```

Exercise 7:

Create the index() method.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.prev = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None
```

```
self.tail = None  
self.size = 0  
self.current = None  
  
def index(self, data):  
    pass
```

Expected Answer:

```
def index(self, data):  
    cur_node = self.head  
    count = 0  
    while cur_node:  
        if cur_node.data == data:  
            return count  
        cur_node = cur_node.next  
        count += 1  
    if cur_node == self.head:  
        break  
    return -1
```

Current traversal of the list

The set_current() method is used to reset the current variable to reference the start of the list.

```
def set_current(self):  
    self.current = self.head
```

The set_current_to() sets the current node to a node at the given index.

```
def set_current_to(self, index):  
    self.current = self.head  
    for i in range(index):  
        self.current = self.current.next
```

The next() method is used to traverse the list by setting the current variable to the node next to the node it is currently set to.

```
def next(self):  
    if self.current == None:  
        return  
    self.current = self.current.next
```

The prev() method is used to traverse the list by setting the current variable to the node previous to the node it is currently set to.

```
def prev(self):  
    if self.current == None:  
        return  
    self.current = self.current.prev
```

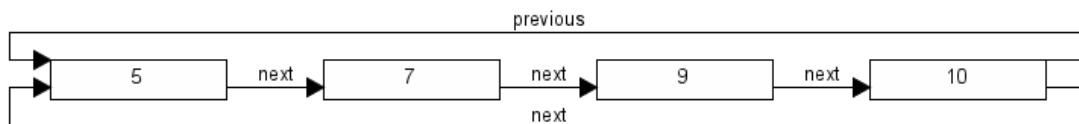
The get_current() method implements the retrieval of the data stored in the current node through a simple return statement.

```
def get_current(self):  
    return self.cur_node.data
```

Ex 8 (Visual exercises)

Ex 8

The below image is a valid representation of a circular doubly linked list. Is this statement true or false?

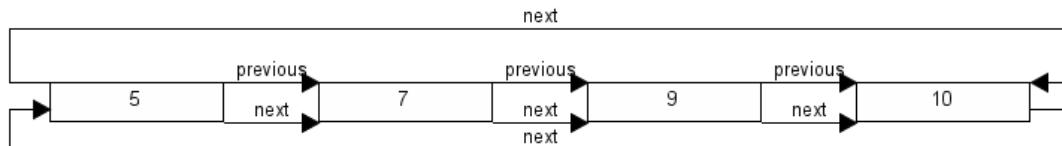


True

False

Ex 9

The below image is a valid representation of a circular singly linked list. Is this statement true or false?

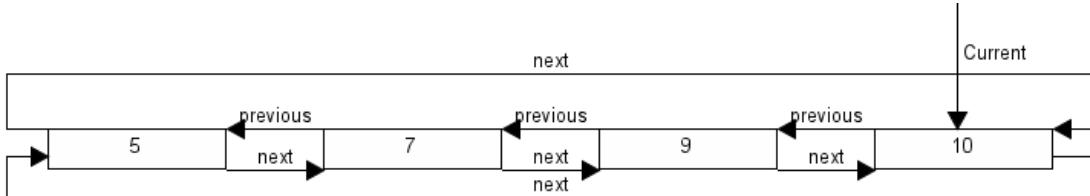


True

False

Ex 10

Given the below image, what would be a valid way to have the 'current' variable equal to 5?
Select all correct options.



`next()`

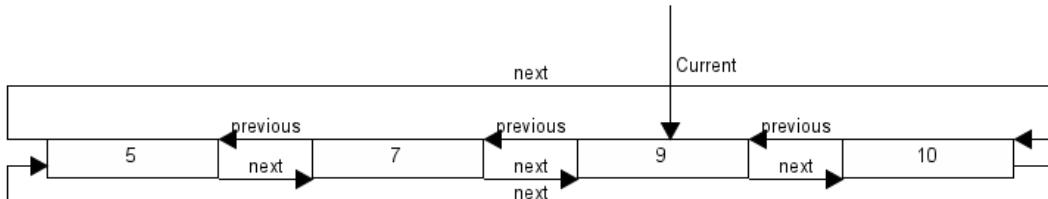
`previous()`

`previous()`

`set_current_to(1)`

Ex 11

Given the below image, what would be a valid way to have the 'current' variable equal to 7?



`previous()`

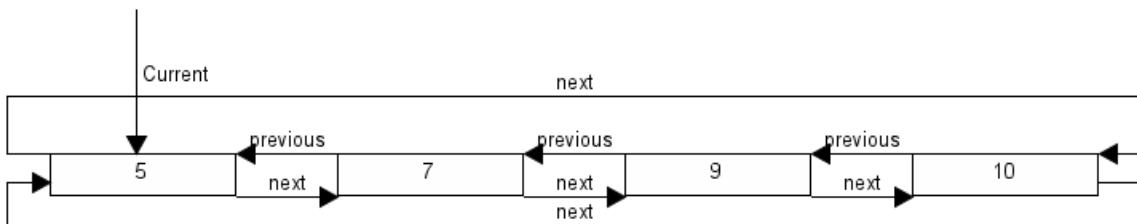
`set_current()`

`next()`

`set_current_to(3)`

Ex 12

Given the below image, what would be a valid way to have the 'current' variable equal to 10?



- next()
- set_current()
- previous()
- set_current_to(2)

Ex 13-42 (Train Station loop management example)

The Australian government has committed to the creation of a massive high speed rail project that will create a vast train loop around Australia connection to many major cities

Ex 13

Create the Node class for the doubly linked list.

Expected input:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Ex 14

Create the CircularDoublyLinkedList class for the doubly linked list.

Expected input:

```
class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None
```

Ex 15

Create the append() method for the CircularDoublyLinkedList.

Expected input:

```
def append(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        self.size += 1
        new_node.next = self.head
        new_node.prev = self.tail
        return
    new_node.prev = self.tail
    self.tail.next = new_node
    self.tail = new_node
    self.tail.next = self.head
    self.head.prev = self.tail
    self.size += 1
```

Ex 16

Create the prepend() method for the CircularDoublyLinkedList.

Expected input:

```
def prepend(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        self.size += 1
        new_node.next = self.head
        new_node.prev = self.tail
        return
    new_node.next = self.head
    self.head.prev = new_node
    self.head = new_node
    self.head.prev = self.tail
    self.tail.next = self.head
    self.size += 1
```

Ex 18

Create the get() method for the CircularDoublyLinkedList.

Expected input:

```
def get(self, index):
```

```

cur_node = self.head
count = 0
while cur_node:
    if count == index:
        return cur_node.data
    cur_node = cur_node.next
    count += 1
    if cur_node == self.head:
        break

```

Ex 19

Create the index() method for the CircularDoublyLinkedList.

Expected input:

```

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
        if cur_node == self.head:
            break
    return -1

```

Ex 20

Create the insert_at() method for the CircularDoublyLinkedList.

Expected input:

```

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            new_node.prev = cur_node
            cur_node.next = new_node
            break
        cur_node = cur_node.next
        count += 1

```

```

temp.prev = new_node
self.size += 1
return
cur_node = cur_node.next
count += 1
if cur_node == self.head:
    break

```

Ex 21

Create the delete() method for the CircularDoublyLinkedList.

Expected input:

```

def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = self.tail
                self.tail.next = self.head
                self.size -= 1
            return
        if cur_node == self.tail:
            self.tail = cur_node.prev
            self.tail.next = self.head
            self.head.prev = self.tail
            self.size -= 1
        cur_node.prev.next = cur_node.next
        cur_node.next.prev = cur_node.prev
        self.size -= 1
        return
    cur_node = cur_node.next
    if cur_node == self.head:
        break

```

Ex 22

Create the delete_at() method for the CircularDoublyLinkedList.

Expected input:

```

def delete_at(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.head:

```

```

    self.head = cur_node.next
    self.head.prev = self.tail
    self.tail.next = self.head
    self.size -= 1
    return
if cur_node == self.tail:
    self.tail = cur_node.prev
    self.tail.next = self.head
    self.head.prev = self.tail
    self.size -= 1
    return
cur_node.prev.next = cur_node.next
cur_node.next.prev = cur_node.prev
self.size -= 1
return
cur_node = cur_node.next
count += 1
if cur_node == self.head:
    break

```

Ex 23

Create the set_current() method for the CircularDoublyLinkedList.

Expected input:

```

def set_current(self, index):
    self.current = self.get(index)

```

Ex 24

Create the get_current() method for the CircularDoublyLinkedList.

Expected input:

```

def get_current(self):
    return self.current

```

Ex 25

Create the next() and prev() methods for the CircularDoublyLinkedList.

Expected input:

```

def next(self):
    self.current = self.current.next
def prev(self):
    self.current = self.current.prev

```

Ex 26

Create the set_current_to() method for the CircularDoublyLinkedList.

Expected input:

```
def set_current_to(self, index):
    self.current = self.head
    for i in range(index):
        self.current = self.current.next
```

Ex 27

Create the print_list() method for the CircularDoublyLinkedList.

Expected input:

```
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
    if cur_node == self.head:
        break
```

Ex 28

Now create a Station class that takes and stores an inputted name and a Circular doubly linked list called passengers.

Assume that the Circular doubly linked list is created as previous exercises.

Expected input:

```
class Station:
    def __init__(self, name):
        self.name = name
        self.passengers = CircularDoublyLinkedList()
```

Ex 29

Create an add_passenger() method to add a new passenger to the station. This method takes the passenger name as an input.

Expected input:

```
def add_passenger(self, passenger):
    self.passengers.append(passenger)
```

Ex 30

Create a remove_passenger() method that removes the first passenger with the inputted

name from the station.

Expected input:

```
def remove_passenger(self, passenger):  
    self.passengers.delete(passenger)
```

Ex 31

Create a get_passenger() method to return and delete the first passenger that has the same name as the inputted name from the list.

Expected input:

```
def get_passenger(self, passenger):  
    self.passengers.delete(passenger)
```

Ex 32

Now create the federal_loop as a Circular doubly linked list.

Expected input:

```
federal_loop = CircularDoublyLinkedList()
```

Ex 33

Now add the stations in the order, Newcastle, Sydney, Canberra, Melbourne, Adelaide, Perth, Darwin, Cairns, Brisbane Add 20 passengers to each station, and name each passenger after the station they are associated with. For example, all passengers at the 'Newcastle' station should be named 'Newcastle', and so on for each subsequent station.

Expected input:

```
federal_loop.append(Station("Newcastle"))  
for i in range(20):  
    federal_loop.get(0).add_passenger("Newcastle")  
federal_loop.append(Station("Sydney"))  
for i in range(20):  
    federal_loop.get(1).add_passenger("Sydney")  
federal_loop.append(Station("Canberra"))  
for i in range(20):  
    federal_loop.get(2).add_passenger("Canberra")  
federal_loop.append(Station("Melbourne"))  
for i in range(20):  
    federal_loop.get(3).add_passenger("Melbourne")  
federal_loop.append(Station("Adelaide"))  
for i in range(20):  
    federal_loop.get(4).add_passenger("Adelaide")  
federal_loop.append(Station("Perth"))
```

```

for i in range(20):
    federal_loop.get(5).add_passenger("Perth")
federal_loop.append(Station("Darwin"))
for i in range(20):
    federal_loop.get(6).add_passenger("Darwin")
federal_loop.append(Station("Cairns"))
for i in range(20):
    federal_loop.get(7).add_passenger("Cairns")
federal_loop.append(Station("Brisbane"))
for i in range(20):
    federal_loop.get(8).add_passenger("Brisbane")

```

Ex 34

Create the class Train saving a name, a linked list of passengers and a current_station set to None. The name should be passed as an input.

Expected input:

```

class Train:
    def __init__(self, name):
        self.name = name
        self.passengers = CircularDoublyLinkedList()
        self.current_station = None

```

Ex 35

Create the method set_station() for the Train class.

Expected input:

```

def set_station(self, station):
    self.current_station = station

```

Ex 36

Create the train1 variable initializing the train called "Train1" and set it as the first station in the loop.

Expected input:

```

train1 = Train("Train 1")
train1.current_station = federal_loop.get(0)

```

Ex 37

Now set the train to take on 5 passengers from the station it is currently at, and move to the next station in the list and unload those 5 passengers. Do this 20 times and use current traversal for greater efficiency.

Expected input:

```
federal_loop.set_current()
for i in range(20):
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    federal_loop.next()
    train1.set_station(federal_loop.get_current())
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
        train1.passengers.delete_at(0)
```

Ex 38

Now do the exact same thing starting at the current station but moving in the opposite direction.

Expected input:

```
for i in range(20):
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    federal_loop.prev()
    train1.set_station(federal_loop.get_current())
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
        train1.passengers.delete_at(0)
```

Ex 39

To see the results of this now print out the passengers for each station in the form:

Station name:

"\t" + name or passenger 1
"\t" + name or passenger 2
"\t" + name or passenger 3
...

Expected input:

```
for i in range(9):
    print(federal_loop.get(i).name + ":")
    for j in range(federal_loop.get(i).passengers.size()):
        print("\t" + federal_loop.get(i).passengers.get(j))
```

Ex 40

Now for 10 stops from the current position, transport people as before using next() but move two stations each time.

Expected input:

```
for i in range(10):
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    federal_loop.next()
    federal_loop.next()
    train1.set_station(federal_loop.get_current())
    # let off 5 passengers at the current station
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
    train1.passengers.delete_at(0)
```

Ex 41

Now complete the same loop in the other direction.

Expected input:

```
for i in range(10):
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    federal_loop.prev()
    federal_loop.prev()
    train1.set_station(federal_loop.get_current())
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
    train1.passengers.delete_at(0)
```

Ex 42

As before, print out the stations and the people at them.

Expected input:

```
for i in range(9):
    print(federal_loop.get(i).name + ":")
    for j in range(federal_loop.get(i).passengers.size):
        print("\t" + federal_loop.get(i).passengers.get(j))
```

6.2 Data Structure Quest

The following exercises will be used to create a text-based game called "Data Structure Quest", the Circular Linked List will be used to manage turn orders for players and enemies.

The game is structured as follow:

Node and CircularLinkedList Classes:

- The Node class represents a node in a linked list, with data and a reference to the next node.
- The CircularLinkedList class is a circular linked list implementation. It includes methods for appending, prepending, printing the list, getting the element at a specific index, finding the index of a specific value, inserting at a specific index, deleting a node with a specific value, deleting a node at a specific index, and setting the current node in the list.

Character Class:

- The character_class class defines a character class with attributes like name, health multiplier, attack multiplier, defense multiplier, and weapon.

Enemy Class:

- The Enemy class represents enemies in the game. It has attributes similar to the Character class

Party Class:

- The Party class represents the player's party, consisting of characters. It has methods for adding characters, getting the party, checking if the party is defeated, and setting the defeated status.

Game Initialization:

- Instances of the Party class and enemies are created for each round.

Player Class Selection:

- Players are prompted to select a character class for each of the four party members.

Game Rounds:

- The game consists of multiple rounds where the party faces different enemies. Each round involves the party attacking enemies and enemies attacking the party in turn.

Game Logic:

- The game progresses through different rounds, and characters level up after defeating enemies. The game ends when the party defeats all enemies or is defeated.

User Input:

- The game involves user input to select character classes, choose enemies to attack, and make decisions during encounters.

The linked list to be used in the program is given below.

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class CircularLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.size = 0  
        self.current = None  
  
    def append(self, data):  
        new_node = Node(data)  
        if self.head == None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node  
        self.tail = new_node  
        self.size += 1  
  
    def prepend(self, data):  
        new_node = Node(data)  
        if self.head == None:  
            self.head = new_node  
            self.tail = new_node  
            new_node.next = new_node  
            self.size += 1  
            return  
        new_node.next = self.head  
        self.tail.next = new_node  
        self.head = new_node  
        self.size += 1  
  
    def print_list(self):  
        cur_node = self.head  
        while cur_node:
```

```

print(cur_node.data)
cur_node = cur_node.next
if cur_node == self.head:
    break

def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return None

def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while True:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next

```

```

count += 1

def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node.data == data:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node == self.head: # prevent infinite loop
        return
    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
    if cur_node is self.head:
        return
    prev.next = cur_node.next

```

```
cur_node = None
self.size -= 1

def set_current(self):
    self.current = self.head

def set_current_to(self, index):
    self.current = self.head
    for i in range(index):
        self.current = self.current.next

def next(self):
    if self.current == None:
        return
    self.current = self.current.next

def prev(self):
    if self.current == None:
        return
    self.current = self.current.prev

def get_current(self):
    return self.current.data
```

Exercises 1 - 30 (Coding)

Ex 1

Create the class character_class inputting the values name, health_multiplier, attack_multiplier, defense_multiplier, and weapon to be stored.

Expected input:

```
class character_class:  
    def __init__(self, name, health_multiplier, attack_multiplier, defense_multiplier, weapon):  
        self.name = name  
        self.health_multiplier = health_multiplier  
        self.attack_multiplier = attack_multiplier  
        self.defense_multiplier = defense_multiplier  
        self.weapon = weapon
```

Ex 2

Create the methods get_name(), get_health(), get_attack() and get_defense() to return the desired values.

Given Code:

```
class character_class:  
    def __init__(self, name, health_multiplier, attack_multiplier, defense_multiplier, weapon):  
        self.name = name  
        self.health_multiplier = health_multiplier  
        self.attack_multiplier = attack_multiplier  
        self.defense_multiplier = defense_multiplier  
        self.weapon = weapon
```

Your code here

Expected input:

```
def get_name(self):  
    return self.name  
  
def get_health(self):  
    return self.health_multiplier  
  
def get_attack(self):  
    return self.attack_multiplier  
  
def get_defense(self):  
    return self.defense_multiplier
```

Ex 3

Create the following character classes:

"Warrior", 2.0, 2.0, 1.0, "Sword"
"Cleric", 1.0, 3.0, 1.0, "Holy Icon"
"Mage", 1.0, 2.0, 2.0, "Magic Staff"
"Ranger", 1.0, 2.0, 2.0, "Bow"

Given Code:

```
class character_class:  
    def __init__(self, name, health_multiplier, attack_multiplier, defense_multiplier, weapon):  
        self.name = name  
        self.health_multiplier = health_multiplier  
        self.attack_multiplier = attack_multiplier  
        self.defense_multiplier = defense_multiplier  
        self.weapon = weapon  
  
    def get_name(self):  
        return self.name  
  
    def get_health(self):  
        return self.health_multiplier  
  
    def get_attack(self):  
        return self.attack_multiplier  
  
    def get_defense(self):  
        return self.defense_multiplier
```

Your code here

Expected input:

```
warrior = character_class("Warrior", 2.0, 2.0, 1.0, "Sword")  
cleric = character_class("Cleric", 1.0, 3.0, 1.0, "Holy Icon")  
mage = character_class("Mage", 1.0, 2.0, 2.0, "Magic Staff")  
ranger = character_class("Ranger", 1.0, 2.0, 2.0, "Bow")
```

Ex 4

Create the class Character with input name and character_class setting the variable:

1. max_health = 100 * the health multiplier
2. attack = 10 * attack multipliers
3. defense = 10 * defense multipliers
4. health = max_health,
5. weapon = the class weapon
6. alive = True.

Expected input:

class Character:

```
def __init__(self, name, character_class):
    self.name = name
    self.character_class = character_class
    self.max_health = 100 * character_class.get_health()
    self.health = self.max_health
    self.attack = 10 * character_class.get_attack()
    self.defense = 10 * character_class.get_defense()
    self.weapon = character_class.weapon
    self.alive = True
```

Ex 5

Create the get_name(), get_class(), get_health(), get_attack(), get_defense(), get_weapon(), get_alive() methods to return the values.

Given Code:

class Character:

```
def __init__(self, name, character_class):
    self.name = name
    self.character_class = character_class
    self.max_health = 100 * character_class.get_health()
    self.health = self.max_health
    self.attack = 10 * character_class.get_attack()
    self.defense = 10 * character_class.get_defense()
    self.weapon = character_class.weapon
    self.alive = True
```

Your code here

Expected input:

```
def get_name(self):
    return self.name

def get_class(self):
    return self.character_class

def get_health(self):
    return self.health

def get_attack(self):
    return self.attack

def get_defense(self):
    return self.defense

def get_weapon(self):
    return self.weapon
```

```
def get_alive(self):  
    return self.alive
```

Ex 6

Create the set_alive() method that sets alive to true and sets health to max_health.

Given Code:

```
class Character:  
    def __init__(self, name, character_class):  
        self.name = name  
        self.character_class = character_class  
        self.max_health = 100 * character_class.get_health()  
        self.health = self.max_health  
        self.attack = 10 * character_class.get_attack()  
        self.defense = 10 * character_class.get_defense()  
        self.weapon = character_class.weapon  
        self.alive = True  
  
    def get_name(self):  
        return self.name  
  
    def get_class(self):  
        return self.character_class  
  
    def get_health(self):  
        return self.health  
  
    def get_attack(self):  
        return self.attack  
  
    def get_defense(self):  
        return self.defense  
  
    def get_weapon(self):  
        return self.weapon  
  
    def get_alive(self):  
        return self.alive  
  
#Your code here
```

Expected input:

```
def set_alive(self, alive):  
    self.alive = alive  
    self.health = self.max_health
```

Ex 7

Write the level_up() method that scales max_health, attack and defense by * 1.5 and update health to equal max_health.

Given Code:

```
class Character:  
    def __init__(self, name, character_class):  
        self.name = name  
        self.character_class = character_class  
        self.max_health = 100 * character_class.get_health()  
        self.health = self.max_health  
        self.attack = 10 * character_class.get_attack()  
        self.defense = 10 * character_class.get_defense()  
        self.weapon = character_class.weapon  
        self.alive = True  
  
    def get_name(self):  
        return self.name  
  
    def get_class(self):  
        return self.character_class  
  
    def get_health(self):  
        return self.health  
  
    def get_attack(self):  
        return self.attack  
  
    def get_defense(self):  
        return self.defense  
  
    def get_weapon(self):  
        return self.weapon  
  
    def get_alive(self):  
        return self.alive  
  
    def set_alive(self, alive):  
        self.alive = alive  
        self.health = self.max_health  
  
# Your code here
```

Expected input:

```
def level_up(self):  
    self.max_health *= 1.5  
    self.health = self.max_health  
    self.attack *= 1.5
```

```
self.defense *= 1.5
```

Ex 8

Write the attack() method that will take a target as the input and will on a 50% random chance either determine damage as the difference between the character's attack and the target's defense or determine it as the attack of the character. Once determined, make sure damage is not negative and remove the damage from the target's health. If it is now below 0, set them as dead and call the announce_death method on them.

```
class Character:  
    def __init__(self, name, character_class):  
        self.name = name  
        self.character_class = character_class  
        self.max_health = 100 * character_class.get_health()  
        self.health = self.max_health  
        self.attack = 10 * character_class.get_attack()  
        self.defense = 10 * character_class.get_defense()  
        self.weapon = character_class.weapon  
        self.alive = True  
  
    def get_name(self):  
        return self.name  
  
    def get_class(self):  
        return self.character_class  
  
    def get_health(self):  
        return self.health  
  
    def get_attack(self):  
        return self.attack  
  
    def get_defense(self):  
        return self.defense  
  
    def get_weapon(self):  
        return self.weapon  
  
    def get_alive(self):  
        return self.alive  
  
    def set_alive(self, alive):  
        self.alive = alive  
        self.health = self.max_health  
  
    def level_up(self):  
        self.max_health *= 1.5  
        self.health = self.max_health  
        self.attack *= 1.5
```

```
self.defense *= 1.5
```

#Your code here

Expected input:

```
def attack(self, target):
    crit = random.randint(0, 1)
    if crit == 1:
        damage = self.get_attack()
    else:
        damage = self.get_attack() - target.get_defense()
    if damage < 0:
        damage = 0
    target.health -= damage
    if target.get_health() <= 0:
        target.alive = False
        target.announce_death()
```

Ex 9

Now create the announce_death() method that prints:

name + " has died!, don't worry, you can revive them if you survive the round!"

For example:

Warrior has died!, don't worry, you can revive them if you survive the round!

Given Code:

class Character:

```
def __init__(self, name, character_class):
    self.name = name
    self.character_class = character_class
    self.max_health = 100 * character_class.get_health()
    self.health = self.max_health
    self.attack = 10 * character_class.get_attack()
    self.defense = 10 * character_class.get_defense()
    self.weapon = character_class.weapon
    self.alive = True
```

```
def get_name(self):
    return self.name
```

```
def get_class(self):
    return self.character_class
```

```
def get_health(self):
    return self.health
```

```
def get_attack(self):
```

```

        return self.attack

    def get_defense(self):
        return self.defense

    def get_weapon(self):
        return self.weapon

    def get_alive(self):
        return self.alive

    def set_alive(self, alive):
        self.alive = alive
        self.health = self.max_health

    def level_up(self):
        self.max_health *= 1.5
        self.health = self.max_health
        self.attack *= 1.5
        self.defense *= 1.5

    def attack(self, target):
        crit = random.randint(0, 1)
        if crit == 1:
            damage = self.get_attack()
        else:
            damage = self.get_attack() - target.get_defense()
        if damage < 0:
            damage = 0
        target.health -= damage
        if target.get_health() <= 0:
            target.alive = False
            target.announce_death()

# Your code here

```

Expected input:

```

def announce_death(self):
    print(self.get_name() + " has died!, don't worry, you can revive them if you survive the
round!")

```

Ex 10

Now create the Enemy class that has name, health, attack, and defense as inputs to be saved, and set the variable alive to true.

Expected input:

```
class Enemy:
```

```
def __init__(self, name, health, attack, defense, weapon):
    self.name = name
    self.health = health
    self.attack = attack
    self.defense = defense
    self.alive = True
```

Ex 11

Write the get_name(), get_health(), get_attack(), get_defense(), and get_alive() methods identical to the characters methods.

Given Code:

```
class Enemy:
```

```
    def __init__(self, name, health, attack, defense, weapon):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense
        self.alive = True
```

Expected input:

```
def get_name(self):
    return self.name

def get_health(self):
    return self.health

def get_attack(self):
    return self.attack

def get_defense(self):
    return self.defense

def get_alive(self):
    return self.alive
```

Ex 12

Write the attack() method identical to the characters method that will take a target as the input and will on a 50% random chance either determine damage as the difference between the character's attack and the target's defense or determine it as the attack of the character. Once determined, make sure damage is not negative and remove the damage from the target's health. If it is now below 0, set them as dead and call the announce_death method on them.

Given Code:

```
class Enemy:
    def __init__(self, name, health, attack, defense, weapon):
        self.name = name
```

```

self.health = health
self.attack = attack
self.defense = defense
self.alive = True

def get_name(self):
    return self.name

def get_health(self):
    return self.health

def get_attack(self):
    return self.attack

def get_defense(self):
    return self.defense

def get_alive(self):
    return self.alive

#Your code here

```

Expected input:

```

def attack(self, target):
    crit = random.randint(0, 1)
    if crit == 1:
        damage = self.get_attack()
    else:
        damage = self.get_attack() - target.get_defense()
    if damage < 0:
        damage = 0
    target.health -= damage
    if target.get_health() <= 0:
        target.alive = False
        target.announce_death()

```

Ex 13

Write the announce_death() method printing:
name + " has died!"

For example:

Warrior has died!

```

class Enemy:
    def __init__(self, name, health, attack, defense, weapon):
        self.name = name
        self.health = health

```

```

self.attack = attack
self.defense = defense
self.alive = True

def get_name(self):
    return self.name

def get_health(self):
    return self.health

def get_attack(self):
    return self.attack

def get_defense(self):
    return self.defense

def get_alive(self):
    return self.alive

def attack(self, target):
    crit = random.randint(0, 1)
    if crit == 1:
        damage = self.get_attack()
    else:
        damage = self.get_attack() - target.get_defense()
    if damage < 0:
        damage = 0
    target.health -= damage
    if target.get_health() <= 0:
        target.alive = False
        target.announce_death()

```

Your code here

Expected input:

```

def announce_death(self):
    print(self.get_name() + " has died!")

```

Ex 14

Create the Party class that will manage the adventurers, the class will have a linked list called party and the variable defeated set to false.

Expected input:

```

class Party:
    def __init__(self):
        self.party = CircularLinkedList()
        self.defeated = False

```

Ex 15

Create methods in the Party class: 'add' to take a character as an input and add that character to the party's linked list, 'get_party' to retrieve the party's character list, and 'get_defeated' to check and return the party's defeat status.

Given Code:

```
class Party:  
    def __init__(self):  
        self.party = CircularLinkedList()  
        self.defeated = False
```

Expected input:

```
def add(self, character):  
    self.party.append(character)  
  
def get_party(self):  
    return self.party  
  
def get_defeated(self):  
    return self.defeated
```

Ex 16

Write the method `check_defeated()` that will check if there are any alive members of the party and update the defeated attribute to True.

Given Code:

```
class Party:  
    def __init__(self):  
        self.party = CircularLinkedList()  
        self.defeated = False  
  
    def add(self, character):  
        self.party.append(character)  
  
    def get_party(self):  
        return self.party  
  
    def get_defeated(self):  
        return self.defeated
```

Expected input:

```
def check_defeated(self):  
    for i in range(self.party.size):  
        if self.party.get_current().get_alive():  
            return  
            self.party.next()
```

```
self.defeated = True
```

Ex 17

Create 6 enemies called goblin1 - goblin6 with the same stats:
"Goblin", 50, 10, 5, "Dagger"

Expected input:

```
goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
```

Ex 18

Now create a linked list called round_1 and add all of the goblins to the list.

Given Code:

```
goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
```

Expected input:

```
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
```

Ex 19

Create the list round_2 and add 4 orks (Enemy) with the same stats:
"Ork", 100, 20, 10, "Club"

Expected input:

```
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
```

```
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
```

Ex 20

Create the list round_3 with a single enemy with stats:
"Dragon", 200, 30, 20, "Fire Breath"

Expected input:

```
round_3 = CircularLinkedList()  
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))
```

Ex 21

Create a party called heroes and loop 4 times printing the following text:

"Player " + str(i+1) + " please select your class:"

"1. Warrior"

"2. Cleric"

"3. Mage"

"4. Ranger"

Once this is printed, store the user's response as the variable class_choice.

Expected input:

```
heroes = Party()  
for i in range(4):  
    print("Player " + str(i+1) + " please select your class:")  
    print("1. Warrior")  
    print("2. Cleric")  
    print("3. Mage")  
    print("4. Ranger")  
    class_choice = int(input())
```

Ex 22

Now using an if statement, ask for the user's name and create their character.

Hint: You should use the character_class objects created in ex[3] to create the character here.

For example, if you choose Warrior as your character, this should be print out:

You have chosen the Warrior class

Please enter your name:

Given Code:

```
heroes = Party()  
for i in range(4):  
    print("Player " + str(i+1) + " please select your class:")  
    print("1. Warrior")  
    print("2. Cleric")
```

```
print("3. Mage")
print("4. Ranger")
class_choice = int(input())
```

Expected input:

```
classType = ""
if class_choice == 1:
    classType = warrior
    print("You have chosen the Warrior class")
elif class_choice == 2:
    classType = cleric
    print("You have chosen the Cleric class")
elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)
```

Ex 23

Print the following text as shown:

Welcome to the game!

You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.

You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.

Round 1: Goblins!

You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.

You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.

Expected input:

```
print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil
dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of
enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
```

```

print("You have encountered a group of goblins! They are weak, but they are in large
numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the
goblins. You can choose which Enemy you want to attack but choose carefully because the
goblins will attack after you.")

```

Ex 24

Create the attacking loop for the players against the goblins showing the player their health and the health of the goblins before taking their selection and making the attack by updating the lists.

Given Code:

```

goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

heroes = Party()
for i in range(4):
    print("Player " + str(i+1) + " please select your class:")
    print("1. Warrior")
    print("2. Cleric")
    print("3. Mage")
    print("4. Ranger")
    class_choice = int(input())

    classType = ""
    if class_choice == 1:
        classType = warrior
        print("You have chosen the Warrior class")
    elif class_choice == 2:
        classType = cleric
        print("You have chosen the Cleric class")

```

```

elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.")

```

Expected input:

```

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()
        # ask the user which goblin they want to attack
        print("Which goblin do you want to attack?")
        goblin_choice = int(input())
        # attack the goblin
        heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
        # check if the goblin is dead
        if round_1.get(goblin_choice-1).get_health() <= 0:
            # remove the goblin from the list
            round_1.delete_at(goblin_choice-1)
        # check if the goblin list is empty
        if round_1.size == 0:

```

```
break  
heroes.get_party().next()
```

Ex 25

Create the attack loop for the goblins where they select a random player to attack, only attacking if that player is alive, but selecting another target if they are not. Once an attack is made, check for a defeat.

Given Code:

```
goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")  
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")  
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")  
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")  
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")  
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")  
round_1 = CircularLinkedList()  
round_1.append(goblin1)  
round_1.append(goblin2)  
round_1.append(goblin3)  
round_1.append(goblin4)  
round_1.append(goblin5)  
round_1.append(goblin6)  
round_2 = CircularLinkedList()  
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))  
round_3 = CircularLinkedList()  
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))  
  
heroes = Party()  
for i in range(4):  
    print("Player " + str(i+1) + " please select your class:")  
    print("1. Warrior")  
    print("2. Cleric")  
    print("3. Mage")  
    print("4. Ranger")  
    class_choice = int(input())  
  
    classType = ""  
    if class_choice == 1:  
        classType = warrior  
        print("You have chosen the Warrior class")  
    elif class_choice == 2:  
        classType = cleric  
        print("You have chosen the Cleric class")  
    elif class_choice == 3:  
        classType = mage  
        print("You have chosen the Mage class")
```

```

elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil
dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of
enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large
numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the
goblins. You can choose which Enemy you want to attack but choose carefully because the
goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()
        # ask the user which goblin they want to attack
        print("Which goblin do you want to attack?")
        goblin_choice = int(input())
        # attack the goblin
        heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
        # check if the goblin is dead
        if round_1.get(goblin_choice-1).get_health() <= 0:
            # remove the goblin from the list
            round_1.delete_at(goblin_choice-1)
        # check if the goblin list is empty
        if round_1.size == 0:
            break
    heroes.get_party().next()

```

#Your code here

Expected input:

```

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))
            have_attacked = True
    if heroes.check_defeated():
        break
    round_1.next()

```

Ex 26

Modify the code to implement the outcome of the battle against the goblins. If the heroes are defeated, print 'You have been defeated! Game over!' Otherwise, print 'You have defeated the goblins! You have leveled up!' and proceed to level up each hero in the party. Ensure that the provided loop correctly iterates through each hero in the party and applies the 'level_up()' method.

Given Code:

```

goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

heroes = Party()
for i in range(4):

```

```

print("Player " + str(i+1) + " please select your class:")
print("1. Warrior")
print("2. Cleric")
print("3. Mage")
print("4. Ranger")
class_choice = int(input())

classType = ""
if class_choice == 1:
    classType = warrior
    print("You have chosen the Warrior class")
elif class_choice == 2:
    classType = cleric
    print("You have chosen the Cleric class")
elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()

```

```

# ask the user which goblin they want to attack
print("Which goblin do you want to attack?")
goblin_choice = int(input())
# attack the goblin
heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
# check if the goblin is dead
if round_1.get(goblin_choice-1).get_health() <= 0:
    # remove the goblin from the list
    round_1.delete_at(goblin_choice-1)
# check if the goblin list is empty
if round_1.size == 0:
    break
heroes.get_party().next()

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))
            have_attacked = True
    if heroes.check_defeated():
        break
round_1.next()

```

Your code here

Expected input:

```

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

```

Ex 27

Create the round card with the text:

Round 2: Orks!

You have encountered a group of orks! They are stronger than the goblins, but they are fewer in number. You must defeat them all to move on to the next round.

Given Code:

```

goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

heroes = Party()
for i in range(4):
    print("Player " + str(i+1) + " please select your class:")
    print("1. Warrior")
    print("2. Cleric")
    print("3. Mage")
    print("4. Ranger")
    class_choice = int(input())

    classType = ""
    if class_choice == 1:
        classType = warrior
        print("You have chosen the Warrior class")
    elif class_choice == 2:
        classType = cleric
        print("You have chosen the Cleric class")
    elif class_choice == 3:
        classType = mage
        print("You have chosen the Mage class")
    elif class_choice == 4:
        classType = ranger
        print("You have chosen the Ranger class")
    print("Please enter your name:")
    name = input()
    player = Character(name, classType)
    heroes.add(player)

print("Welcome to the game!")

```

```

print("You are a party of 4 adventurers, and you have been tasked with defeating the evil
dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of
enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large
numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the
goblins. You can choose which Enemy you want to attack but choose carefully because the
goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()
        # ask the user which goblin they want to attack
        print("Which goblin do you want to attack?")
        goblin_choice = int(input())
        # attack the goblin
        heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
        # check if the goblin is dead
        if round_1.get(goblin_choice-1).get_health() <= 0:
            # remove the goblin from the list
            round_1.delete_at(goblin_choice-1)
        # check if the goblin list is empty
        if round_1.size == 0:
            break
        heroes.get_party().next()

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))

```

```

have_attacked = True
if heroes.check_defeated():
    break
round_1.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

```

Your code here

Expected input:

```

print("Round 2: Orks!")
print("You have encountered a group of orks! They are stronger than the goblins, but they are
fewer in number. You must defeat them all to move on to the next round.")

```

Ex 28

Create the attacking loop for the players against the orks showing the player their health and the health of the goblins before taking their selection and making the attack updating the lists.

Given Code:

```

goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

heroes = Party()
for i in range(4):

```

```

print("Player " + str(i+1) + " please select your class:")
print("1. Warrior")
print("2. Cleric")
print("3. Mage")
print("4. Ranger")
class_choice = int(input())

classType = ""
if class_choice == 1:
    classType = warrior
    print("You have chosen the Warrior class")
elif class_choice == 2:
    classType = cleric
    print("You have chosen the Cleric class")
elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()

```

```

# ask the user which goblin they want to attack
print("Which goblin do you want to attack?")
goblin_choice = int(input())
# attack the goblin
heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
# check if the goblin is dead
if round_1.get(goblin_choice-1).get_health() <= 0:
    # remove the goblin from the list
    round_1.delete_at(goblin_choice-1)
# check if the goblin list is empty
if round_1.size == 0:
    break
heroes.get_party().next()

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))
            have_attacked = True
    if heroes.check_defeated():
        break
    round_1.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

print("Round 2: Orks!")
print("You have encountered a group of orks! They are stronger than the goblins, but they are fewer in number. You must defeat them all to move on to the next round.")

```

Your code here

Expected input:

```

while not heroes.get_defeated():
    if round_2.size == 0:
        break
    heroes.get_party().set_current()

```

```

for i in range(heroes.get_party().size):
    # print the current character's name and health
    print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
    # print out the orks' health, name and index
    round_2.set_current()
    for i in range(round_2.size):
        print("Ork " + str(i+1) + " has " + str(round_2.get_current().get_health()) + " health.")
        round_2.next()
    # ask the user which ork they want to attack
    print("Which ork do you want to attack?")
    ork_choice = int(input())
    # attack the ork
    heroes.get_party().get_current().attack(round_2.get(ork_choice-1))
    # check if the ork is dead
    if round_2.get(ork_choice-1).get_health() <= 0:
        # remove the ork from the list
        round_2.delete_at(ork_choice-1)
    # check if the ork list is empty
    if round_2.size == 0:
        break
    heroes.get_party().next()

```

Ex 29

Create the attack loop for the orks where they select a random player to attack, only attacking if that player is alive, but selecting another target if they are not. Once an attack is made, check for a defeat. In case of a victory, print: “You have defeated the orks! You have leveled up!” and level-up all the characters. In case of a defeat, print: “You have been defeated! Game Over!”

Given Code:

```

goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))

```

```

round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

heroes = Party()
for i in range(4):
    print("Player " + str(i+1) + " please select your class:")
    print("1. Warrior")
    print("2. Cleric")
    print("3. Mage")
    print("4. Ranger")
    class_choice = int(input())

classType = ""
if class_choice == 1:
    classType = warrior
    print("You have chosen the Warrior class")
elif class_choice == 2:
    classType = cleric
    print("You have chosen the Cleric class")
elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")

```

```

# print out the goblins' health, name and index
round_1.set_current()
for i in range(round_1.size):
    print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
    round_1.next()
# ask the user which goblin they want to attack
print("Which goblin do you want to attack?")
goblin_choice = int(input())
# attack the goblin
heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
# check if the goblin is dead
if round_1.get(goblin_choice-1).get_health() <= 0:
    # remove the goblin from the list
    round_1.delete_at(goblin_choice-1)
# check if the goblin list is empty
if round_1.size == 0:
    break
heroes.get_party().next()

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))
            have_attacked = True
    if heroes.check_defeated():
        break
    round_1.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

print("Round 2: Orks!")
print("You have encountered a group of orks! They are stronger than the goblins, but they are fewer in number. You must defeat them all to move on to the next round.")

while not heroes.get_defeated():
    if round_2.size == 0:
        break

```

```

heroes.get_party().set_current()
for i in range(heroes.get_party().size):
    # print the current character's name and health
    print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
    # print out the orks' health, name and index
    round_2.set_current()
    for i in range(round_2.size):
        print("Ork " + str(i+1) + " has " + str(round_2.get_current().get_health()) + " health.")
        round_2.next()
    # ask the user which ork they want to attack
    print("Which ork do you want to attack?")
    ork_choice = int(input())
    # attack the ork
    heroes.get_party().get_current().attack(round_2.get(ork_choice-1))
    # check if the ork is dead
    if round_2.get(ork_choice-1).get_health() <= 0:
        # remove the ork from the list
        round_2.delete_at(ork_choice-1)
    # check if the ork list is empty
    if round_2.size == 0:
        break
    heroes.get_party().next()

```

Your code here

Expected input:

```

while not heroes.get_defeated():
    if round_2.size == 0:
        break
    round_2.set_current()
    for i in range(round_2.size):
        have_attacked = False
        while have_attacked == False:
            # select a random character to attack
            random_character = random.randint(0, heroes.get_party().size-1)
            # check if the character is alive
            if heroes.get_party().get(random_character).get_alive():
                # attack the character
                round_2.get_current().attack(heroes.get_party().get(random_character))
                have_attacked = True
        if heroes.check_defeated():
            break
        round_2.next()
    if heroes.get_defeated():
        print("You have been defeated! Game over!")
    else:
        print("You have defeated the orks! You have leveled up!")
        for i in range(heroes.get_party().size):

```

```
heroes.get_party().get_current().level_up()
```

Ex 30

Now create the final round where the heroes finally defeat the dragon.

Print 'Round 3: Defeating the Dragon!' to set the stage for the final battle. Implement a loop that continues until either the heroes are defeated or the dragon list is empty.

Within the loop:

- a. Display the name and health of the current hero.
- b. Show the dragon's current health.
- c. Ask the user if they want the hero to attack the dragon. Accept 'y' for yes and 'n' for no.
- d. If the user chooses to attack, perform the hero's attack on the dragon.
- e. Remove the dragon if its health drops to or below 0.
- f. Check if the dragon list is empty, and exit the loop if it is.
- g. Move to the next hero in the party.

After heroes complete their attacks, simulate the dragon's attack on a random living hero.

If the heroes are defeated, print 'You have been defeated! Game over!'. Otherwise, print 'You have defeated the dragon! You have won the game!' at the end.

Given Code:

```
goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")
round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)
round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))
round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))
```

```
heroes = Party()
for i in range(4):
    print("Player " + str(i+1) + " please select your class:")
    print("1. Warrior")
    print("2. Cleric")
    print("3. Mage")
    print("4. Ranger")
    class_choice = int(input())
```

```

classType = ""
if class_choice == 1:
    classType = warrior
    print("You have chosen the Warrior class")
elif class_choice == 2:
    classType = cleric
    print("You have chosen the Cleric class")
elif class_choice == 3:
    classType = mage
    print("You have chosen the Mage class")
elif class_choice == 4:
    classType = ranger
    print("You have chosen the Ranger class")
print("Please enter your name:")
name = input()
player = Character(name, classType)
heroes.add(player)

print("Welcome to the game!")
print("You are a party of 4 adventurers, and you have been tasked with defeating the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")
print("You have encountered a group of goblins! They are weak, but they are in large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn attacking the goblins. You can choose which Enemy you want to attack but choose carefully because the goblins will attack after you.")

while not heroes.get_defeated():
    if round_1.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the goblins' health, name and index
        round_1.set_current()
        for i in range(round_1.size):
            print("Goblin " + str(i+1) + " has " + str(round_1.get_current().get_health()) + " health.")
            round_1.next()
        # ask the user which goblin they want to attack
        print("Which goblin do you want to attack?")
        goblin_choice = int(input())
        # attack the goblin
        heroes.get_party().get_current().attack(round_1.get(goblin_choice-1))
        # check if the goblin is dead

```

```

if round_1.get(goblin_choice-1).get_health() <= 0:
    # remove the goblin from the list
    round_1.delete_at(goblin_choice-1)
# check if the goblin list is empty
if round_1.size == 0:
    break
heroes.get_party().next()

if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character
            round_1.get_current().attack(heroes.get_party().get(random_character))
            have_attacked = True
    if heroes.check_defeated():
        break
    round_1.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

print("Round 2: Orks!")
print("You have encountered a group of orks! They are stronger than the goblins, but they are fewer in number. You must defeat them all to move on to the next round.")

while not heroes.get_defeated():
    if round_2.size == 0:
        break
    heroes.get_party().set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the orks' health, name and index
        round_2.set_current()
        for i in range(round_2.size):
            print("Ork " + str(i+1) + " has " + str(round_2.get_current().get_health()) + " health.")
            round_2.next()
        # ask the user which ork they want to attack

```

```

print("Which ork do you want to attack?")
ork_choice = int(input())
# attack the ork
heroes.get_party().get_current().attack(round_2.get(ork_choice-1))
# check if the ork is dead
if round_2.get(ork_choice-1).get_health() <= 0:
    # remove the ork from the list
    round_2.delete_at(ork_choice-1)
# check if the ork list is empty
if round_2.size == 0:
    break
heroes.get_party().next()

while not heroes.get_defeated():
    if round_2.size == 0:
        break
    round_2.set_current()
    for i in range(round_2.size):
        have_attacked = False
        while have_attacked == False:
            # select a random character to attack
            random_character = random.randint(0, heroes.get_party().size-1)
            # check if the character is alive
            if heroes.get_party().get(random_character).get_alive():
                # attack the character
                round_2.get_current().attack(heroes.get_party().get(random_character))
                have_attacked = True
        if heroes.check_defeated():
            break
    round_2.next()
if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the orks! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

```

#Your code here

Expected input:

```

print("Round 3: Defeating the Dragon!")
    print("You have encountered the dragon! It is the most powerful enemy you have ever
faced. You must defeat it to win the game.")
    # round 3
    while not heroes.get_defeated():
        if round_3.size == 0:
            break
        heroes.get_party().set_current()
        for i in range(heroes.get_party().size):

```

```

# print the current character's name and health
print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
# print out the dragon's health
print("The dragon has " + str(round_3.get_current().get_health()) + " health.")
# ask the user if they want to attack the dragon
print("Do you want to attack the dragon? (y/n)")
attack_choice = input()
if attack_choice == "y":
    # attack the dragon
    heroes.get_party().get_current().attack(round_3.get_current())
# check if the dragon is dead
if round_3.get_current().get_health() <= 0:
    # remove the dragon from the list
    round_3.remove(0)
# check if the dragon list is empty
if round_3.size == 0:
    break

    heroes.get_party().next()
# now that all the characters have attacked, the dragon gets to attack
if round_3.size == 0:
    break
have_attacked = False
while have_attacked == False:
    # select a random character to attack
    random_character = random.randint(0, heroes.get_party().size-1)
    # check if the character is alive
    if heroes.get_party().get(random_character).get_alive():
        # attack the character
        round_3.get_current().attack(heroes.get_party().get(random_character))
        have_attacked = True
    if heroes.check_defeated():
        break
    round_3.next()
if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the dragon! You have won the game!")

```

Code Appendix

1: Singly Linked List

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self): # initialize the linked list
        self.head = None
        self.size = 0
    def append(self, data): # add to the end of the list
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.size += 1
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        self.size += 1
    def prepend(self, data): # add to the beginning of the list
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1
    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
```

```

        cur_node = cur_node.next
    def get(self, index): # get the data at a specific index
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
                return cur_node.data
            cur_node = cur_node.next
            count += 1
    def index(self, data): # get the index of a specific value
        cur_node = self.head
        count = 0
        while cur_node:
            if cur_node.data == data:
                return count
            cur_node = cur_node.next
            count += 1
        return -1

    def insert_at(self, index, data): # insert a new node at a specific index
        if index == 0:
            self.prepend(data)
            return
        new_node = Node(data)
        count = 0
        cur_node = self.head
        while cur_node:
            if count == index - 1:
                new_node.next = cur_node.next
                cur_node.next = new_node
                self.size += 1
                return
            cur_node = cur_node.next
            count += 1
    def delete(self, data): # delete a node with a specific value
        cur_node = self.head
        if cur_node and cur_node.data == data:
            self.head = cur_node.next
            cur_node = None

```

```

        self.size -= 1
        return
    prev = None
    while cur_node and cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while cur_node and count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

# List1 = LinkedList()
# item1 = Node("soup")
# List1.head = item1
# List1.size += 1

```

```

# print(list1.head.data)
# print(list1.size)

# item2 = Node("carrots")
# list1.head.next = item2
# print(list1.head.next.data)

# item3 = Node("milk")
# list1.head.next.next = item3
# print(list1.head.next.next.data)

# food_list = LinkedList()
# item1 = Node("grapes")
# food_list.head = item1
# food_list.size += 1
# item2 = Node("milk")
# food_list.head.next = item2
# food_list.size += 1
# item3 = Node("onions")
# food_list.head.next.next = item3
# food_list.size += 1
# item4 = Node("chips")
# food_list.head.next.next.next = item4
# food_list.size += 1

# food_list.head.data = "apples"

# # print out the list
# food_list.print_list()

```

```

user_list = LinkedList()
user_list.append("Phil")
user_list.append("George")
user_list.append("Mary")
user_list.append("Lee")
user_list.append("John")
user_list.append("Kelvin")

```

```

user_list.append("Patrick")
user_list.append("Jane")

# print(user_list.get(3))
# print(user_list.get(4))

user_list.prepend("Lucy")

user_list.insert_at(3, "Sally")

# user_list.print_list()

# determine the number of four letter names in the list
count = 0
while count < user_list.size:
    if len(user_list.get(count)) == 4:
        print(user_list.get(count))
    count += 1

# check if a name is in the list
name = "Jane"
count = 0
while count < user_list.size:
    if user_list.get(count) == name:
        print("True")
    count += 1

# change the data at a specific index

bookshelf = LinkedList()
book1 = Book("The Hobbit", "J.R.R. Tolkien", 295)
new_node = Node(book1)
bookshelf.head = new_node

print("The first book on the shelf is " + bookshelf.head.data.title)
print("The author of the first book on the shelf is " +
bookshelf.head.data.author)

```

```

print("The first book on the shelf is " + str(bookshelf.head.data.pages) + " pages long")

class Member:
    def __init__(self, name, months_left, tier):
        self.name = name
        self.months_left = months_left
        self.tier = tier
    def print_member(self):
        print("Name: " + self.name)
        print("Months Left: " + str(self.months_left))
        print("Tier: " + self.tier)

club_members = LinkedList()

club_members.append(Member("Phil", 3, "Gold"))
club_members.append(Member("George", 2, "Silver"))
club_members.append(Member("Mary", 1, "Bronze"))
club_members.append(Member("Lee", 4, "Gold"))
club_members.append(Member("John", 5, "Gold"))
club_members.append(Member("Kelvin", 6, "Silver"))
club_members.append(Member("Patrick", 7, "Gold"))
club_members.append(Member("Jane", 8, "Silver"))

# delete all members with a tier of silver
count = 0
while count < club_members.size:
    if club_members.get(count).tier == "Silver":
        club_members.delete_at(count)
    count += 1

# use the delete method to delete all members with less than 5 months left
count = 0
while count < club_members.size:
    if club_members.get(count).months_left < 5:
        club_members.delete(club_members.get(count))
    count += 1

```

2: Doubly Linked List

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return self.title + " by " + self.author + ", " + str(self.pages) + " "
pages.

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
            return
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1
    def prepend(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
```

```

        self.tail = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node
    self.size += 1
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            new_node.next = cur_node
            new_node.prev = cur_node.prev

```

```

        cur_node.prev.next = new_node
        cur_node.prev = new_node
        self.size += 1
        return
    cur_node = cur_node.next
    count += 1
def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head:
                self.head = cur_node.next
                cur_node.next.prev = None
                self.size -= 1
                return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                cur_node.prev.next = None
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next
def delete_at(self, index):
    if index == 0:
        self.head = self.head.next
        self.head.prev = None
        self.size -= 1
        return
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.tail:
                self.tail = cur_node.prev
                cur_node.prev.next = None
                self.size -= 1
                return

```

```

        return
    cur_node.prev.next = cur_node.next
    cur_node.next.prev = cur_node.prev
    self.size -= 1
    return
cur_node = cur_node.next
count += 1
def set_current(self):
    self.current = self.head
def set_current_to(self, index):
    self.current = self.head
    for i in range(index):
        self.current = self.current.next
def next(self):
    if self.current == None:
        return
    self.current = self.current.next
def prev(self):
    if self.current == None:
        return
    self.current = self.current.prev
def get_current(self):
    return self.current.data

```

```
list3 = DoublyLinkedList()
```

```
list3.append(1)
```

```
list3.append(2)
```

```
list3.append(3)
```

```
print(list3.tail.data)
```

```
# set the current node to the head
```

```
list3.current = list3.head
```

```
sci_fi = DoublyLinkedList()
```

```
sci_fi.append(Book("The Martian", "Andy Weir", 384))
```

```

sci_fi.append(Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 224))
sci_fi.append(Book("The Time Machine", "H.G. Wells", 96))
sci_fi.append(Book("The War of the Worlds", "H.G. Wells", 128))
sci_fi.append(Book("A Princess of Mars", "Edgar Rice Burroughs", 256))

sci_fi.print_list()

# Demonstration of the Doubly Linked List data structure

# Print the title of each book and whether or not the books next to it is by the same author
while True:
    print("Enter the index of the book you want to see")
    print("Enter -1 to exit")
    index = int(input())
    if index == -1:
        break
    book = sci_fi.get(index)
    print(book)
    if index == 0:
        next_book = sci_fi.get(index + 1)
        if book.author == next_book.author:
            print("The next book is by the same author")
        else:
            print("The next book is by a different author")
    elif index == sci_fi.size - 1:
        prev_book = sci_fi.get(index - 1)
        if book.author == prev_book.author:
            print("The previous book is by the same author")
        else:
            print("The previous book is by a different author")
    else:
        next_book = sci_fi.get(index + 1)
        prev_book = sci_fi.get(index - 1)
        if book.author == next_book.author:
            print("The next book is by the same author")
        else:
            print("The next book is by a different author")
        if book.author == prev_book.author:

```

```

        print("The previous book is by the same author")
    else:
        print("The previous book is by a different author")

# Rewrite the above code to use the current next and prev method and not the
variable book

while True:
    print("Enter the index of the book you want to see")
    print("Enter -1 to exit")
    index = int(input())
    if index == -1:
        break
    sci_fi.set_current_to(index)
    print(sci_fi.get_current())
    if index == 0:
        sci_fi.next()
        if sci_fi.get_current().author == sci_fi.get(index + 1).author:
            print("The next book is by the same author")
        else:
            print("The next book is by a different author")
    elif index == sci_fi.size - 1:
        sci_fi.prev()
        if sci_fi.get_current().author == sci_fi.get(index - 1).author:
            print("The previous book is by the same author")
        else:
            print("The previous book is by a different author")
    else:
        sci_fi.next()
        if sci_fi.get_current().author == sci_fi.get(index + 1).author:
            print("The next book is by the same author")
        else:
            print("The next book is by a different author")
        sci_fi.prev()
        sci_fi.prev()
        if sci_fi.get_current().author == sci_fi.get(index - 1).author:
            print("The previous book is by the same author")
        else:
            print("The previous book is by a different author")
    sci_fi.next()

```

```
# finish by adding a method to the DoublyLinkedList class that will print the
# list in reverse order starting at the tail and ending at the head using the
# current and prev method
sci_fi.set_current_to(sci_fi.size - 1)
while sci_fi.get_current() != sci_fi.get(0):
    print(sci_fi.get_current())
    sci_fi.prev()
print(sci_fi.get_current())
```

3: Circular Linked List

```
import random

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.head = new_node
        self.size += 1
```

```

def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
        if cur_node == self.head:
            break
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while True:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node

```

```

        self.size += 1
        return
    cur_node = cur_node.next
    count += 1
def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node.data == data:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node == self.head: # prevent infinite loop
        return
    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count != index:
        prev = cur_node

```

```

        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
        prev.next = cur_node.next
        cur_node = None
        self.size -= 1
    def set_current(self):
        self.cur_node = self.head
    def next(self):
        if self.cur_node == None:
            return
        self.cur_node = self.cur_node.next
    def get_current(self):
        return self.cur_node.data

# implement 21 using a circular linked list
class Card:
    def __init__(self, suit, value):
        self.suit = suit
        self.value = value
    def __repr__(self):
        return self.value + " of " + self.suit

class Deck:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.build()
    def build(self):
        for suit in ["Hearts", "Diamonds", "Clubs", "Spades"]:
            for value in ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]:
                self.cards.append(Card(suit, value))
    def show(self):
        self.cards.print_list()
    def shuffle(self):
        # shuffle the deck

```

```

        for i in range(0, self.cards.size):
            card = self.cards.get(i)
            rand = random.randint(0, self.cards.size - 1)
            self.cards.delete_at(i)
            self.cards.insert_at(rand, card)

    def draw(self):
        card = self.cards.get(0)
        self.cards.delete_at(0)
        return card

deck = Deck()
print("Deck size before shuffling:", deck.cards.size)
deck.show()
deck.shuffle()
print("Deck size after shuffling:", deck.cards.size)
deck.show()

class Hand:
    def __init__(self):
        self.cards = CircularLinkedList()
        self.stand = False
        self.bust = False
    def add(self, card):
        self.cards.append(card)
    def show(self):
        self.cards.print_list()
    def score(self):
        score = 0
        for i in range(0, self.cards.size):
            card = self.cards.get(i)
            if card.value == "A":
                score += 11
            elif card.value == "J" or card.value == "Q" or card.value == "K":
                score += 10
            else:
                score += int(card.value)
        return score

# determine the number of players from input (2-5)

```

```

num_players = int(input("Enter the number of players (1-4): "))

# create a circular linked list of players hands to keep track of the order
players = CircularLinkedList()
for i in range(0, num_players):
    player = Hand()
    player.add(deck.draw())
    player.add(deck.draw())
    players.append(player)

# print("Players:", players.size)

# create a dealer hand the dealer will play after the players
dealer = Hand()
dealer.add(deck.draw())
dealer.add(deck.draw())

# Run through the players hands

players.set_current()
i = 1
game_over = False
while game_over == False:
    player = players.get_current()
    if player.stand == True or player.bust == True:
        game_over = True
    else:
        while player.stand == False and player.bust == False:
            print("Player", i, "score:", player.score())
            player.show()
            choice = input("Hit or stand? (h/s): ")
            if choice == "h":
                player.add(deck.draw())
                # check if the player busted
                if player.score() > 21:
                    player.bust = True

```

```

        print("Player", i, "busted!")
        player.bust = True
    else:
        player.stand = True
players.next()
i += 1

# Run through the dealer hand

# print the dealer's hand
print("Dealer score:", dealer.score())
dealer.show()

while dealer.score() < 16:
    dealer.add(deck.draw())
    print("Dealer score:", dealer.score())
    dealer.show()
    # check if the dealer busted
    if dealer.score() > 21:
        dealer.bust = True
        print("Dealer busted!")

# determine the winner without using the get method
players.set_current()

for i in range(0, players.size):
    if players.get_current().bust == False and dealer.bust == False:
        if players.get_current().score() > dealer.score():
            print("Player", i + 1, "wins!")
        elif players.get_current().score() < dealer.score():
            print("Player", i + 1, "loses!")
        else:
            print("Player", i + 1, "ties!")
    elif players.get_current().bust == True:
        print("Player", i + 1, "loses!")
    else:

```

```
    print("Player", i + 1, "wins!")
players.next()
```

4: Double Circular Linked List

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
            new_node.next = self.head
            new_node.prev = self.tail
            return
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
        self.tail.next = self.head
        self.head.prev = self.tail
        self.size += 1
    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            self.size += 1
            new_node.next = self.head
            new_node.prev = self.tail
            return
        new_node.next = self.head
        new_node.prev = self.tail
        self.head.prev = new_node
        self.tail.next = new_node
        self.size += 1
    new_node.next = self.head
```

```

    self.head.prev = new_node
    self.head = new_node
    self.head.prev = self.tail
    self.tail.next = self.head
    self.size += 1
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
        if cur_node == self.head:
            break
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
        if cur_node == self.head:
            break
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
        if cur_node == self.head:
            break
    return -1
def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)

```

```

        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            new_node.prev = cur_node
            cur_node.next = new_node
            temp.prev = new_node
            self.size += 1
        return
    cur_node = cur_node.next
    count += 1
    if cur_node == self.head:
        break
def delete(self, data):
    cur_node = self.head
    while cur_node:
        if cur_node.data == data:
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = self.tail
                self.tail.next = self.head
                self.size -= 1
            return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = self.head
                self.head.prev = self.tail
                self.size -= 1
            return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
        return
    cur_node = cur_node.next
    if cur_node == self.head:

```

```

        break
def delete_at(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            if cur_node == self.head:
                self.head = cur_node.next
                self.head.prev = self.tail
                self.tail.next = self.head
                self.size -= 1
                return
            if cur_node == self.tail:
                self.tail = cur_node.prev
                self.tail.next = self.head
                self.head.prev = self.tail
                self.size -= 1
                return
            cur_node.prev.next = cur_node.next
            cur_node.next.prev = cur_node.prev
            self.size -= 1
            return
        cur_node = cur_node.next
        count += 1
        if cur_node == self.head:
            break
def set_current(self):
    self.current = self.head
def set_current_to(self, index):
    self.current = self.head
    for i in range(index):
        self.current = self.current.next
def get_current(self):
    return self.current.data
def next(self):
    self.current = self.current.next
def prev(self):
    self.current = self.current.prev

```

```

# create a program modeling a new high speed rail system in Australia called the
"Federal Loop"
# the Federal Loop will connect the major cities of Newcastle, Sydney, Melbourne,
Brisbane, Adelaide, Perth, and Canberra, Cairns, and Darwin
# - the Federal Loop will be a circular doubly linked list

class Station:
    def __init__(self, name):
        self.name = name
        self.passengers = CircularDoublyLinkedList()
    def add_passenger(self, passenger):
        self.passengers.append(passenger)
    def remove_passenger(self):
        self.passengers.delete_at(0)
    def get_passenger(self):
        return self.passengers.get(0)
    def __str__(self) -> str:
        return self.name

# create the stations in the Federal Loop and add 20 passengers to each station
with the passengers being strings of their starting stations name

federal_loop = CircularDoublyLinkedList()
federal_loop.append(Station("Newcastle"))
for i in range(20):
    federal_loop.get(0).add_passenger("Newcastle")
federal_loop.append(Station("Sydney"))
for i in range(20):
    federal_loop.get(1).add_passenger("Sydney")
federal_loop.append(Station("Canberra"))
for i in range(20):
    federal_loop.get(2).add_passenger("Canberra")
federal_loop.append(Station("Melbourne"))
for i in range(20):
    federal_loop.get(3).add_passenger("Melbourne")
federal_loop.append(Station("Adelaide"))
for i in range(20):
    federal_loop.get(4).add_passenger("Adelaide")
federal_loop.append(Station("Perth"))

```

```

for i in range(20):
    federal_loop.get(5).add_passenger("Perth")
federal_loop.append(Station("Darwin"))
for i in range(20):
    federal_loop.get(6).add_passenger("Darwin")
federal_loop.append(Station("Cairns"))
for i in range(20):
    federal_loop.get(7).add_passenger("Cairns")
federal_loop.append(Station("Brisbane"))
for i in range(20):
    federal_loop.get(8).add_passenger("Brisbane")
# the order of the stations in the Federal Loop is Newcastle, Sydney, Canberra,
Melbourne, Adeleide, Perth, Darwin, Cairns, Brisbane

federal_loop.print_list()

class Train:
    def __init__(self, name):
        self.name = name
        self.passengers = CircularDoublyLinkedList()
        self.current_station = None
    def set_station(self, station):
        self.current_station = station

# now make use of the Federal Loop to create a program that will simulate the
movement of the trains and the passengers on the Federal Loop using the current
next and prev methods to show off doubly linked circular lists
train1 = Train("Train 1")

# set the current station of each train to the first station in the Federal Loop
train1.current_station = federal_loop.get(0)

# now move the trains around the Federal Loop and have the passengers get on and
off the trains
federal_loop.set_current()
print(federal_loop.get_current())

```

```

federal_loop.get_current().passengers.print_list()

for i in range(20):
    # take on 5 passengers from the current station
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    # move the train to the next station
    federal_loop.next()
    train1.set_station(federal_loop.get_current())
    # let off 5 passengers at the current station
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
        train1.passengers.delete_at(0)

# now the train has made several loops in one direction go the other direction
for i in range(20):
    # take on 5 passengers from the current station
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    # move the train to the next station
    federal_loop.prev()
    train1.set_station(federal_loop.get_current())
    # let off 5 passengers at the current station
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
        train1.passengers.delete_at(0)

# now print out which passengers are at each station
for i in range(9):
    print(federal_loop.get(i).name + ":")
    for j in range(federal_loop.get(i).passengers.size()):
        print("\t" + federal_loop.get(i).passengers.get(j))

# send the train on a loop stopping every second station and letting off 5
# passengers at each station
for i in range(10):

```

```

# take on 5 passengers from the current station
for j in range(5):
    train1.passengers.append(federal_loop.get_current().get_passenger())
    federal_loop.get_current().remove_passenger()
# move the train to the next station
federal_loop.next()
federal_loop.next()
train1.set_station(federal_loop.get_current())
# let off 5 passengers at the current station
for j in range(5):
    federal_loop.get_current().add_passenger(train1.passengers.get(0))
    train1.passengers.delete_at(0)

# now the train has made several loops in one direction go the other direction
for i in range(10):
    # take on 5 passengers from the current station
    for j in range(5):
        train1.passengers.append(federal_loop.get_current().get_passenger())
        federal_loop.get_current().remove_passenger()
    # move the train to the next station
    federal_loop.prev()
    federal_loop.prev()
    train1.set_station(federal_loop.get_current())
    # let off 5 passengers at the current station
    for j in range(5):
        federal_loop.get_current().add_passenger(train1.passengers.get(0))
        train1.passengers.delete_at(0)

```

5: Mark Maker

```
import random

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self): # initialize the linked list
        self.head = None
        self.size = 0
    def append(self, data): # add to the end of the list
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.size += 1
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        self.size += 1
    def prepend(self, data): # add to the beginning of the list
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1
    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next
    def get(self, index): # get the data at a specific index
        cur_node = self.head
        count = 0
        while cur_node:
            if count == index:
```

```

        return cur_node.data
    cur_node = cur_node.next
    count += 1
def index(self, data): # get the index of a specific value
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
def insert_at(self, index, data): # insert a new node at a specific index
    if index == 0:
        self.prepend(data)
        return
    new_node = Node(data)
    count = 0
    cur_node = self.head
    while cur_node:
        if count == index - 1:
            new_node.next = cur_node.next
            cur_node.next = new_node
            self.size += 1
            return
        cur_node = cur_node.next
        count += 1
def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node and cur_node.data == data:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node and cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node is None:

```

```

        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while cur_node and count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def get_size(self):
    return self.size

```

```

class Grade:
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def get_name(self):
        return self.name
    def get_score(self):
        return self.score

```

```

class Student:
    def __init__(self, name):

```

```

        self.name = name
        self.grades = LinkedList()
    def add_grade(self, grade):
        self.grades.append(grade)
    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size()
    def print_grades(self):
        for i in range(self.grades.size()):
            print("\t" + self.grades.get(i).name, self.grades.get(i).score)
    def get_grades(self):
        return self.grades
    def get_name(self):
        return self.name

class Year_group:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()
    def add_student(self, student):
        self.students.append(student)
    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size()
    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()
    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None
    def get_students(self):
        return self.students

```

```

# create a year group of 30 students
year_11 = Year_group(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30): # add 5 grades to each student
    year_11.students.get(i).add_grade(Grade("Maths", random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("English", random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science", random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History", random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography", random.randint(0, 100)))

#print out all of the students and their grades
year_11.print_students()

# print the average of the year group
print(year_11.get_average())

# print the average of a specific student
print(year_11.students.get(0).get_average())

# print the average of a specific subject
total = 0
count = 0
for i in range(year_11.get_students().get_size()):
    for j in range(year_11.get_students().get(i).get_grades().get_size()):
        if year_11.get_students().get(i).get_grades().get(j).get_name() ==
"Maths":
            total += year_11.get_students().get(i).get_grades().get(j).get_score()
            count += 1
print(total / count)

# print the average of a specific subject for a specific student

for i in range(year_11.get_student("Student 0").get_grades().get_size()):

```

```

    if year_11.get_student("Student 0").get_grades().get(i).get_name() ==
"Maths":
        print(year_11.get_student("Student 0").get_grades().get(i).get_score())


# delete the student with the name "Student 18"
year_11.get_students().delete_at(18)

# print out the names and grades of the top 5 students
top_5 = LinkedList()
# Loop 5 times
for i in range(5):
    # find the student with the highest average
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    # add the student to the top 5 list
    top_5.append(highest_student)
    # remove the student from the year group

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

# print out the names and grades of the top 5 students
print("Top 5 Students")
for i in range(top_5.get_size()):
    print(top_5.get(i).get_name())
    top_5.get(i).print_grades()

# perform a bubble sort on the students by their average highest to lowest
# sorted = False
# while not sorted:
#     sorted = True
#     for i in range(year_11.get_students().get_size() - 1):
#         if year_11.get_students().get(i).get_average() >
year_11.get_students().get(i + 1).get_average():
#             sorted = False

```

```

#           temp = year_11.get_students().get(i)
#           year_11.get_students().set(i, year_11.get_students().get(i + 1))
#           year_11.get_students().set(i + 1, temp)
# # perform a bubble sort on the students by their average lowest to highest
# sorted = False
# while not sorted:
#     sorted = True
#     for i in range(year_11.get_students().get_size() - 1):
#         if year_11.get_students().get(i).get_average() <
year_11.get_students().get(i + 1).get_average():
#             sorted = False
#             temp = year_11.get_students().get(i)
#             year_11.get_students().set(i, year_11.get_students().get(i + 1))
#             year_11.get_students().set(i + 1, temp)

# determine how many students have a grade greater than 90 in each subject
print("Students with a grade greater than 90")
for i in range(year_11.get_students().get_size()):
    for j in range(year_11.get_students().get(i).get_grades().get_size()):
        if year_11.get_students().get(i).get_grades().get(j).get_score() > 90:
            print(year_11.get_students().get(i).get_name() + " has a grade
greater than 90 in " +
year_11.get_students().get(i).get_grades().get(j).get_name())

print()

# determine how many students in the top 5 have a grade greater than 90 in each
subject
print("Students in the top 5 with a grade greater than 90")
for i in range(top_5.get_size()):
    for j in range(top_5.get(i).get_grades().get_size()):
        if top_5.get(i).get_grades().get(j).get_score() > 90:
            print(top_5.get(i).get_name() + " has a grade greater than 90 in " +
top_5.get(i).get_grades().get(j).get_name())


# determine the average of each subject
print("Average of each subject")

```

```

for i in range(year_11.get_students().get(0).get_grades().get_size()):
    total = 0
    count = 0
    for j in range(year_11.get_students().get_size()):
        total += year_11.get_students().get(j).get_grades().get(i).get_score()
        count += 1
    print(year_11.get_students().get(0).get_grades().get(i).get_name() + ": " +
str(total / count))

# determine the average of each subject for the top 5 students
print("Average of each subject for the top 5 students")
for i in range(top_5.get(0).get_grades().get_size()):
    total = 0
    count = 0
    for j in range(top_5.get_size()):
        total += top_5.get(j).get_grades().get(i).get_score()
        count += 1
    print(top_5.get(0).get_grades().get(i).get_name() + ": " + str(total /
count))

# creat an input system that allows the user to enter a student name and a
subject name and then print out the grade for that student in that subject
student_name = input("Enter a student name: ")
subject_name = input("Enter a subject name: ")
for i in range(year_11.get_students().get_size()):
    if year_11.get_students().get(i).get_name() == student_name:
        for j in range(year_11.get_students().get(i).get_grades().get_size()):
            if year_11.get_students().get(i).get_grades().get(j).get_name() ==
subject_name:

print(year_11.get_students().get(i).get_grades().get(j).get_score())

```

6: Data Structure Quest

```
import random

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
        self.current = None

    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = new_node
            new_node.next = new_node
            self.size += 1
            return
        new_node.next = self.head
        self.tail.next = new_node
        self.head = new_node
        self.size += 1
```

```

def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data)
        cur_node = cur_node.next
        if cur_node == self.head:
            break
def get(self, index):
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
def index(self, data):
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1
def insert_at(self, index, data):
    if index == 0:
        self.prepend(data)
        return
    if index == self.size - 1:
        self.append(data)
        return
    new_node = Node(data)
    cur_node = self.head
    count = 0
    while True:
        if count == index - 1:
            temp = cur_node.next
            new_node.next = temp
            cur_node.next = new_node

```

```

        self.size += 1
        return
    cur_node = cur_node.next
    count += 1
def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node.data == data:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node == self.head: # prevent infinite loop
        return
    if cur_node == self.tail:
        self.tail = prev
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        self.tail.next = self.head
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while count != index:
        prev = cur_node

```

```

        cur_node = cur_node.next
        count += 1
        if cur_node is self.head:
            return
        prev.next = cur_node.next
        cur_node = None
        self.size -= 1
    def set_current(self):
        self.current = self.head
    def set_current_to(self, index):
        self.current = self.head
        for i in range(index):
            self.current = self.current.next
    def next(self):
        if self.current == None:
            return
        self.current = self.current.next
    def prev(self):
        if self.current == None:
            return
        self.current = self.current.prev
    def get_current(self):
        return self.current.data

class character_class:
    def __init__(self, name, health_multiplier, attack_multiplier,
defense_multiplier, weapon):
        self.name = name
        self.health_multiplier = health_multiplier
        self.attack_multiplier = attack_multiplier
        self.defense_multiplier = defense_multiplier
        self.weapon = weapon
    def get_name(self):
        return self.name
    def get_health(self):
        return self.health_multiplier
    def get_attack(self):

```

```

        return self.attack_multiplier
    def get_defense(self):
        return self.defense_multiplier

warrior = character_class("Warrior", 2.0, 2.0, 1.0, "Sword")
cleric = character_class("Cleric", 1.0, 3.0, 1.0, "Holy Icon")
mage = character_class("Mage", 1.0, 2.0, 2.0, "Magic Staff")
ranger = character_class("Ranger", 1.0, 2.0, 2.0, "Bow")

class Character:
    def __init__(self, name, character_class):
        self.name = name
        self.character_class = character_class
        self.max_health = 100 * character_class.get_health()
        self.health = self.max_health
        self.attack = 10 * character_class.get_attack()
        self.defense = 10 * character_class.get_defense()
        self.weapon = character_class.weapon
        self.alive = True
    def get_name(self):
        return self.name
    def get_class(self):
        return self.character_class
    def get_health(self):
        return self.health
    def get_attack(self):
        return self.attack
    def get_defense(self):
        return self.defense
    def get_weapon(self):
        return self.weapon
    def get_alive(self):
        return self.alive
    def set_alive(self, alive):
        self.alive = alive
        self.health = self.max_health
    def level_up(self):
        self.max_health *= 1.5

```

```

        self.health = self.max_health
        self.attack *= 1.5
        self.defense *= 1.5
    def perform_attack(self, target):
        crit = random.randint(0, 1)
        if crit == 1:
            damage = self.get_attack()
        else:
            damage = self.get_attack() - target.get_defense()
        if damage < 0:
            damage = 0
        target.health -= damage
        if target.get_health() <= 0:
            target.alive = False
            target.announce_death()
    def announce_death(self):
        print(self.get_name() + " has died!, don't worry, your can revive them if
you survive the round!")

class Enemy:
    def __init__(self, name, health, attack, defense, weapon):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense
        self.alive = True
    def get_name(self):
        return self.name
    def get_health(self):
        return self.health
    def get_attack(self):
        return self.attack
    def get_defense(self):
        return self.defense
    def get_alive(self):
        return self.alive
    def perform_attack(self, target):
        crit = random.randint(0, 1)
        if crit == 1:

```

```

        damage = self.get_attack()
    else:
        damage = self.get_attack() - target.get_defense()
    if damage < 0:
        damage = 0
    target.health -= damage
    if target.get_health() <= 0:
        target.alive = False
        target.announce_death()
    def announce_death(self):
        print(self.get_name() + " has died!")

# create 6 goblins
goblin1 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin2 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin3 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin4 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin5 = Enemy("Goblin", 50, 10, 5, "Dagger")
goblin6 = Enemy("Goblin", 50, 10, 5, "Dagger")

goblin = Enemy("Goblin", 50, 10, 5, "Dagger")
ork = Enemy("Ork", 100, 20, 10, "Club")
dragon = Enemy("Dragon", 200, 30, 20, "Fire Breath")
troll = Enemy("Troll", 150, 25, 15, "Club")
skeleton = Enemy("Skeleton", 75, 15, 10, "Dagger")
zombie = Enemy("Zombie", 100, 20, 10, "Club")
giant = Enemy("Giant", 200, 30, 20, "Club")
golem = Enemy("Golem", 150, 25, 15, "Club")



class Party:
    def __init__(self):
        self.party = CircularLinkedList()
        self.defeated = False
    def add(self, character):
        self.party.append(character)
    def get_party(self):
        return self.party
    def get_defeated(self):

```

```

        return self.defeated
    def check_defeated(self):
        for i in range(self.party.size):
            if self.party.get_current().get_alive():
                return
            self.party.next()
        self.defeated = True

round_1 = CircularLinkedList()
round_1.append(goblin1)
round_1.append(goblin2)
round_1.append(goblin3)
round_1.append(goblin4)
round_1.append(goblin5)
round_1.append(goblin6)

round_2 = CircularLinkedList()
round_2.append(Enemy("Ork", 100, 20, 10, "Club"))

round_3 = CircularLinkedList()
round_3.append(Enemy("Dragon", 200, 30, 20, "Fire Breath"))

# have each of the 4 players select their class and name with a limit of one of each class
heroes = Party()

for i in range(4):
    print("Player " + str(i+1) + " please select your class:")
    print("1. Warrior")
    print("2. Cleric")
    print("3. Mage")
    print("4. Ranger")
    class_choice = int(input())
    if class_choice == 1:
        print("You have chosen the Warrior class")
        print("Please enter your name:")

```

```

name = input()
player = Character(name, warrior)
heroes.add(player)

elif class_choice == 2:
    print("You have chosen the Cleric class")
    print("Please enter your name:")
    name = input()
    player = Character(name, cleric)
    heroes.add(player)

elif class_choice == 3:
    print("You have chosen the Mage class")
    print("Please enter your name:")
    name = input()
    player = Character(name, mage)
    heroes.add(player)

elif class_choice == 4:
    print("You have chosen the Ranger class")
    print("Please enter your name:")
    name = input()
    player = Character(name, ranger)
    heroes.add(player)

# now that the party is set up, we can start the game
print("Welcome to the game!")
# draw a title screen that says "Data Structure Quest" using ASCII art

print("You are a party of 4 adventurers, and you have been tasked with defeating")
print("the evil dragon that has been terrorizing the land.")
print("You will be fighting a series of enemies, and each time you finish off a")
print("group of enemies, you will level up and gain more health, attack, and defense.")
print("Round 1: Goblins!")

print("You have encountered a group of goblins! They are weak, but they are in")
print("large numbers. You must defeat them all to move on to the next round.")
print("You have 4 characters in your party. Each character will take a turn")
print("attacking the goblins. You can choose which Enemy you want to attack, but choose")
print("carefully because the goblins will attack after you.")

# round 1
while not heroes.get_defeated():

```

```

if round_1.size == 0:
    break
heroes.get_party().set_current()
for i in range(heroes.get_party().size):
    # print the current character's name and health
    print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
    # print out the goblins' health, name and index
    round_1.set_current()
    for i in range(round_1.size):
        print("Goblin " + str(i+1) + " has " +
str(round_1.get_current().get_health()) + " health.")
    round_1.next()
    # ask the user which goblin they want to attack
    print("Which goblin do you want to attack?")
    goblin_choice = int(input())
    # attack the goblin

heroes.get_party().get_current().perform_attack(round_1.get(goblin_choice-1))
    # check if the goblin is dead
    if round_1.get(goblin_choice-1).get_health() <= 0:
        # remove the goblin from the list
        round_1.delete_at(goblin_choice-1)
    # check if the goblin list is empty
    if round_1.size == 0:
        break

    heroes.get_party().next()
# now that all the characters have attacked, the goblins get to attack
if round_1.size == 0:
    break
round_1.set_current()
for i in range(round_1.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():

```

```

# attack the character

round_1.get_current().perform_attack(heroes.get_party().get(random_character))
    have_attacked = True
if heroes.check_defeated():
    break
round_1.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the goblins! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

    print("Round 2: Orks!")
    print("You have encountered a group of orks! They are stronger than the
goblins, but they are fewer in number. You must defeat them all to move on to the
next round.")
    # round 2
    while not heroes.get_defeated():
        if round_2.size == 0:
            break
        heroes.get_party().set_current()
        for i in range(heroes.get_party().size):
            # print the current character's name and health
            print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
            # print out the orks' health, name and index
            round_2.set_current()
            for i in range(round_2.size):
                print("Ork " + str(i+1) + " has " +
str(round_2.get_current().get_health()) + " health.")
            round_2.next()
            # ask the user which ork they want to attack
            print("Which ork do you want to attack?")
            ork_choice = int(input())
            # attack the ork

```

```

heroes.get_party().get_current().perform_attack(round_2.get(ork_choice-1))

    # check if the ork is dead
    if round_2.get(ork_choice-1).get_health() <= 0:
        # remove the ork from the List
        round_2.delete_at(ork_choice-1)

    # check if the ork List is empty
    if round_2.size == 0:
        break

    heroes.get_party().next()

# now that all the characters have attacked, the orks get to attack
if round_2.size == 0:
    break

round_2.set_current()
for i in range(round_2.size):
    have_attacked = False
    while have_attacked == False:
        # select a random character to attack
        random_character = random.randint(0, heroes.get_party().size-1)
        # check if the character is alive
        if heroes.get_party().get(random_character).get_alive():
            # attack the character

round_2.get_current().perform_attack(heroes.get_party().get(random_character))
    have_attacked = True
    if heroes.check_defeated():
        break
    round_2.next()

if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the orks! You have leveled up!")
    for i in range(heroes.get_party().size):
        heroes.get_party().get_current().level_up()

    print("Round 3: Defeating the Dragon!")
    print("You have encountered the dragon! It is the most powerful enemy you have ever faced. You must defeat it to win the game.")

```

```

# round 3
while not heroes.get_defeated():
    if round_3.size == 0:
        break
    heroes.get_party().set_current()
    round_3.set_current()
    for i in range(heroes.get_party().size):
        # print the current character's name and health
        print(heroes.get_party().get_current().get_name() + " has " +
str(heroes.get_party().get_current().get_health()) + " health.")
        # print out the dragon's health
        print("The dragon has " + str(round_3.get_current().get_health()))
+ " health.")

        # ask the user if they want to attack the dragon
        print("Do you want to attack the dragon? (y/n)")
        attack_choice = input()
        if attack_choice == "y":
            # attack the dragon

heroes.get_party().get_current().perform_attack(round_3.get_current())
        # check if the dragon is dead
        if round_3.get_current().get_health() <= 0:
            # remove the dragon from the list
            round_3.delete_at(0)
            # check if the dragon list is empty
            if round_3.size == 0:
                break

            heroes.get_party().next()
# now that all the characters have attacked, the dragon gets to
attack
        if round_3.size == 0:
            break
        have_attacked = False
        while have_attacked == False:
            # select a random character to attack
            random_character = random.randint(0, heroes.get_party().size-1)
            # check if the character is alive
            if heroes.get_party().get(random_character).get_alive():

```

```

# attack the character

round_3.get_current().perform_attack(heroes.get_party().get(random_character))
    have_attacked = True
if heroes.check_defeated():
    break
round_3.next()
if heroes.get_defeated():
    print("You have been defeated! Game over!")
else:
    print("You have defeated the dragon! You have won the game!")

```

6.1 Mark Maker

The first program that we are going to construct is a tool used to store the test results of students and provide information about class averages and the details of a student's records.

The purpose of this program is to simplify and digitize the process of storing testing records and make information more readily accessible to students.

The code below details a singly linked list and will be used throughout the following exercises:

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self): # initialize the linked list
        self.head = None
        self.size = 0

    def append(self, data): # add to the end of the list
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.size += 1
            return
        last_node = self.head
        while last_node.next:

```

```

        last_node = last_node.next
    last_node.next = new_node
    self.size += 1

def prepend(self, data): # add to the beginning of the list
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
    self.size += 1

def get(self, index): # get the data at a specific index
    cur_node = self.head
    count = 0
    while cur_node:
        if count == index:
            return cur_node.data
        cur_node = cur_node.next
        count += 1
    return None

def index(self, data): # get the index of a specific value
    cur_node = self.head
    count = 0
    while cur_node:
        if cur_node.data == data:
            return count
        cur_node = cur_node.next
        count += 1
    return -1

def insert_at(self, index, data): # insert a new node at a
specific index
    if index == 0:
        self.prepend(data)
        return
    new_node = Node(data)
    count = 0
    cur_node = self.head
    while cur_node:
        if count == index - 1:
            new_node.next = cur_node.next
            cur_node.next = new_node
            self.size += 1
        return

```

```

        cur_node = cur_node.next
        count += 1

def delete(self, data): # delete a node with a specific value
    cur_node = self.head
    if cur_node and cur_node.data == data:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    prev = None
    while cur_node and cur_node.data != data:
        prev = cur_node
        cur_node = cur_node.next
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1

def delete_at(self, index): # delete a node at a specific index
    cur_node = self.head
    if index == 0:
        self.head = cur_node.next
        cur_node = None
        self.size -= 1
        return
    count = 0
    prev = None
    while cur_node and count != index:
        prev = cur_node
        cur_node = cur_node.next
        count += 1
    if cur_node is None:
        return
    prev.next = cur_node.next
    cur_node = None
    self.size -= 1
def get_size(self):
    return self.size

```

Exercises 1 - 2 (Coding - Grade Class)

Exercise 1

To begin, create the class **Grade** as a node that will store an inputted **name** and **score**. This will be used to represent the individual test results for students.

Given code

```
# Your code here
```

Expected input

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score
```

Exercise 2

Now create get methods for the **Grade** class to get the name and score. This will improve the use of classes and data structures, allowing our methods to be better structured and have better readability.

Given code

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
    # Your code here  
  
    # Your code here
```

Expected input

```
class Grade:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
    def get_name(self):  
        return self.name  
  
    def get_score(self):  
        return self.score
```

Exercises 3 - 7 (Coding - Student Class)

Exercise 3

Create a class **Student** with input **name** that will store the name of a student and a linked list called **grades**.

Given code

```
# Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()
```

Exercise 4

We now need a method to add grades to a student's profile. Create a method **add_grade()** for the **Student** class that will add a grade object to the end of the grades list.

Given code

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        # Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)
```

Exercise 5

To get an idea of the overall quality of a student's work we need to create a method **get_average()** that will determine the average of all of the grades stored for a student and return them.

Given code

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)
```

```
# Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.grades.size):  
            total += self.grades.get(i).score  
        return total / self.grades.size
```

Exercise 6

To provide a visual representation of all of the grades, create a method **print_grades()** for **Student** class that prints all of the grades in the form: \t Name of subject Score.

Given code

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.grades.size):  
            total += self.grades.get(i).score  
        return total / self.grades.size
```

```
# Your code here
```

Expected input

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.grades = LinkedList()  
  
    def add_grade(self, grade):  
        self.grades.append(grade)  
  
    def get_average(self):  
        total = 0
```

```

        for i in range(self.grades.size):
            total += self.grades.get(i).score
        return total / self.grades.size

    def print_grades(self):
        for i in range(self.grades.size):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

```

Exercise 7

Now create the method **get_grades()** to return the grades linked list in the **Student** class and a **get_name()** to return the name of the student.

Given code

```

class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size

    def print_grades(self):
        for i in range(self.grades.size()):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

# Your code here

# Your code here

```

Expected input

```

class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):

```

```

        total += self.grades.get(i).score
    return total / self.grades.size

    def print_grades(self):
        for i in range(self.grades.size):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

    def get_grades(self):
        return self.grades

    def get_name(self):
        return self.name

```

Exercises 8 - 14 (Coding - YearGroup Class)

Exercise 8

We now have a student class that stores grades for each student but we need a way of organizing a group of students. Create a class **YearGroup** that takes **year** as an input and stores year and students, where students are represented as a linked list.

Given code

```
# Your code here
```

Expected input

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()
```

Exercise 9

Create a method called **add_student()** to add a student to the end of the list.

Given code

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

# Your code here
```

Expected input

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()
```

```
def add_student(self, student):
    self.students.append(student)
```

Exercise 10

While we can check how an individual student is performing, we need to be able to check this against their peers. Write a `get_average()` method for the `YearGroup` class.

Given code

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

# Your code here
```

Expected input

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size
```

Exercise 11

We need a quick way to print out the details of all of the students in the year group. Create a `print_students()` method which prints the student's name and their grades.

Given code

```
class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
```

```

        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size

# Your code here

```

Expected input

```

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

```

Exercise 12

Create a method **get_student()** that will return a student with a given **name** or None if the student does not exist.

Given code

```

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

```

```
# Your code here
```

Expected input

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size()):  
            total += self.students.get(i).get_average()  
        return total / self.students.size()  
  
    def print_students(self):  
        for i in range(self.students.size()):  
            print(self.students.get(i).name)  
            self.students.get(i).print_grades()  
  
    def get_student(self, name):  
        for i in range(self.students.size()):  
            if self.students.get(i).name == name:  
                return self.students.get(i)  
        return None
```

Exercise 13

The `get_student()` method is useful for methods retrieving a certain student's details; for more complex methods we need to return the full list object. Create the method `get_students()` to implement this.

Given code

```
class YearGroup:  
    def __init__(self, year):  
        self.year = year  
        self.students = LinkedList()  
  
    def add_student(self, student):  
        self.students.append(student)  
  
    def get_average(self):  
        total = 0  
        for i in range(self.students.size()):  
            total += self.students.get(i).get_average()  
        return total / self.students.size()
```

```

def print_students(self):
    for i in range(self.students.size()):
        print(self.students.get(i).name)
        self.students.get(i).print_grades()

def get_student(self, name):
    for i in range(self.students.size()):
        if self.students.get(i).name == name:
            return self.students.get(i)
    return None

# Your code here

```

Expected input

```

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    def get_students(self):
        return self.students

```

Exercise 14

Create the method `get_size()` for the linked list to return the size of the list.

Given code

```

class YearGroup:
    def __init__(self, year):
        self.year = year

```

```

        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    def get_students(self):
        return self.students

# Your code here

```

Expected input

```

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)

```

```

        return None

    def get_students(self):
        return self.students

def get_size(self):
    return self.students.size

```

Exercises 15 - 30 (Coding - LinkedList Class)

The following exercises will utilize all three classes you just created (Grade, Student, and YearGroup). Also, the Node and LinkedList classes are still being used but try to use the methods we made in the other classes before using the LinkedList methods if possible. The three classes you created are below:

```

class Grade:
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_name(self):
        return self.name

    def get_score(self):
        return self.score

class Student:
    def __init__(self, name):
        self.name = name
        self.grades = LinkedList()

    def add_grade(self, grade):
        self.grades.append(grade)

    def get_average(self):
        total = 0
        for i in range(self.grades.size()):
            total += self.grades.get(i).score
        return total / self.grades.size()

    def print_grades(self):
        for i in range(self.grades.size()):
            print("\t" + self.grades.get(i).name,
self.grades.get(i).score)

    def get_grades(self):
        return self.grades

```

```

def get_name(self):
    return self.name

class YearGroup:
    def __init__(self, year):
        self.year = year
        self.students = LinkedList()

    def add_student(self, student):
        self.students.append(student)

    def get_average(self):
        total = 0
        for i in range(self.students.size()):
            total += self.students.get(i).get_average()
        return total / self.students.size()

    def print_students(self):
        for i in range(self.students.size()):
            print(self.students.get(i).name)
            self.students.get(i).print_grades()

    def get_student(self, name):
        for i in range(self.students.size()):
            if self.students.get(i).name == name:
                return self.students.get(i)
        return None

    def get_students(self):
        return self.students

    def get_size(self):
        return self.students.size()

```

Exercise 15

Create a YearGoup object **year_11** to store the grades for the year 11 students.

Expected input

`year_11 = YearGroup(11)`

Exercise 16

Now create 30 students with names set to "Student " + their number 0-29 and add them to the year group.

Given code

`year_11 = YearGroup(11)`

```
# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))
```

Exercise 17

For each student add a grade in "Mathematics" and give them a score out of 100, randomize the score using the imported random.randint(lowest, highest) method.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))
```

```
# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))
```

Exercise 18

Now do the same thing for the subjects "English", "Science", "History" and "Geography".

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

# Your code here
```

Expected input

```
year_11 = YearGroup(11)
```

```

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

```

Exercise 19

Now print out the students' scores.

Given code

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

# Your code here

```

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):

```

```

year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.print_students()

```

Expected output (the grades are different due to them being random, but if you have five grades for each student named from Student 0 to Student 29 then it is correct):

Student 0

Mathematics 70
 English 46
 Science 97
 History 38
 Geography 59

Student 1

Mathematics 15
 English 97
 Science 38
 History 37
 Geography 7

.....

Exercise 20

Let's find out how the first student (index 0) is performing by printing the grade average of the year group and then printing the average grade of the first student.

Given code

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",

```

```
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))
```

Your code here

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

print(year_11.get_average())
print(year_11.students.get(0).get_average())
```

Expected output (the output might vary due to the use of randint())

```
48.07333333333345
58.2
```

Exercise 21

Check if "Student 0" has a grade named "Mathematics" and print the score.

Hint: Loop through the grades of "Student 0" and use get_name() to check if it equals "Mathematics".

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
```

```

year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.print_students()

print(year_11.get_average())
print(year_11.students.get(0).get_average())

# Your code here

```

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

```

```

for i in range(year_11.get_student("Student
0").get_grades().get_size()):
    if year_11.get_student("Student
0").get_grades().get(i).get_name() == "Mathematics":
        print(year_11.get_student("Student
0").get_grades().get(i).get_score())

```

Expected output (the output might vary due to the use of randint())

72

Exercise 22

Calculate the average grade for all students for their grade named "Mathematics" and print the result. We've initialized two variables called 'total' and 'count' equal to 0 to help, where 'total' will be the sum of all "Mathematics" grades and 'count' is the number of grades for "Mathematics".

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

total = 0
count = 0
# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
```

```

random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

total = 0
count = 0
for i in range(year_11.get_students().get_size()):
    for j in
range(year_11.get_students().get(i).get_grades().get_size()):
        if
year_11.get_students().get(i).get_grades().get(j).get_name() ==
"Mathematics":
            total +=
year_11.get_students().get(i).get_grades().get(j).get_score()
            count += 1
print(total / count)

```

Expected output (the output might vary due to the use of randint())

50.33333333333336

Exercise 23

The student named "Student 18" is moving to another school. Delete them from the year group.

Given code

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

# Your code here

```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)
```

Exercise 24

The best 5 students in the year have been selected to be added to a new class. Find the top 5 students, add them to a new linked list called 'top_5' and delete them from the year group.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))
```

Exercise 25

Print out the names and grades of the top 5 students from the new list ‘top_5’.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
```

```

        year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
        year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
        year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
        year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

# Your code here

```

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0

```

```

highest_student = None
for j in range(year_11.get_students().get_size()):
    if year_11.get_students().get(j).get_average() > highest:
        highest = year_11.get_students().get(j).get_average()
        highest_student = year_11.get_students().get(j)
top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

for i in range(top_5.get_size()):
    print(top_5.get(i).get_name())
    top_5.get(i).print_grades()

```

Expected output (the output might vary due to the use of randint())

Student 0
 Mathematics 57
 English 81
 Science 90
 History 95
 Geography 55

Student 27
 Mathematics 71
 English 81
 Science 79
 History 93
 Geography 36

Student 14
 Mathematics 89
 English 68
 Science 33
 History 86
 Geography 76

Student 9
 Mathematics 73
 English 65
 Science 80
 History 79
 Geography 47

Student 20
 Mathematics 79
 English 44
 Science 90
 History 53
 Geography 71

Exercise 26

The school administrator wants to determine the best results for each subject, they want to

print out the results that are greater than 90. This should be done in the form:
Student Name + " has a grade greater than 90 in " + Subject Name. Do this just for the
year_11 students (not the top 5) list.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))
```

```

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

for i in range(year_11.get_students().get_size()):
    for j in
range(year_11.get_students().get(i).get_grades().get_size()):
        if
year_11.get_students().get(i).get_grades().get(j).get_score() > 90:
            print(year_11.get_students().get(i).get_name() + " has a
grade greater than 90 in " +
year_11.get_students().get(i).get_grades().get(j).get_name())

```

Expected output (the output might vary due to the use of randint())

Student 0 has a grade greater than 90 in Science
 Student 2 has a grade greater than 90 in Mathematics
 Student 3 has a grade greater than 90 in English
 Student 4 has a grade greater than 90 in English
 Student 6 has a grade greater than 90 in Mathematics
 Student 7 has a grade greater than 90 in English
 Student 13 has a grade greater than 90 in Mathematics
 Student 17 has a grade greater than 90 in History
 Student 19 has a grade greater than 90 in History
 Student 21 has a grade greater than 90 in History
 Student 24 has a grade greater than 90 in Geography

Exercise 27

The administrator was pleased with your work and now wants the same done for the top 5 students.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

for i in range(year_11.get_students().get_size()):
    for j in
range(year_11.get_students().get(i).get_grades().get_size()):
        if
year_11.get_students().get(i).get_grades().get(j).get_score() > 90:
            print(year_11.get_students().get(i).get_name() + " has
a grade greater than 90 in " +
year_11.get_students().get(i).get_grades().get(j).get_name())

print("\nTop 5:")
# Your code here
```

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

for i in range(year_11.get_students().get_size()):
    for j in
range(year_11.get_students().get(i).get_grades().get_size()):
        if
year_11.get_students().get(i).get_grades().get(j).get_score() > 90:
            print(year_11.get_students().get(i).get_name() + " has
a grade greater than 90 in " +
year_11.get_students().get(i).get_grades().get(j).get_name())

print("\nTop 5:")
for i in range(top_5.get_size()):
    for j in range(top_5.get(i).get_grades().get_size()):
        if top_5.get(i).get_grades().get(j).get_score() > 90:
            print(top_5.get(i).get_name() + " has a grade greater
than 90 in " + top_5.get(i).get_grades().get(j).get_name())

```

Expected output (the output might vary due to the use of randint())

Student 7 has a grade greater than 90 in Mathematics
Student 7 has a grade greater than 90 in History
Student 10 has a grade greater than 90 in English
Student 13 has a grade greater than 90 in Mathematics
Student 14 has a grade greater than 90 in English
Student 15 has a grade greater than 90 in Mathematics
Student 16 has a grade greater than 90 in English
Student 24 has a grade greater than 90 in Mathematics
Student 26 has a grade greater than 90 in Geography

Top 5:

Student 11 has a grade greater than 90 in Mathematics
Student 11 has a grade greater than 90 in English
Student 11 has a grade greater than 90 in History
Student 4 has a grade greater than 90 in Mathematics
Student 9 has a grade greater than 90 in History

Exercise 28

The administrator wants to determine where students need the most support and which subjects need to be emphasized. Determine the average score for each subject for the year group linked list only and print it in the form:

Grade name: Grade average

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
```

```

highest_student = None
for j in range(year_11.get_students().get_size()):
    if year_11.get_students().get(j).get_average() > highest:
        highest = year_11.get_students().get(j).get_average()
        highest_student = year_11.get_students().get(j)
top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

```

Your code here

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

for i in
range(year_11.get_students().get(0).get_grades().get_size()):
    total = 0
    count = 0

```

```

for j in range(year_11.get_students().get_size()):
    total += year_11.get_students().get(j).get_grades().get(i).get_score()
    count += 1

print(year_11.get_students().get(0).get_grades().get(i).get_name() +
": " + str(total / count))

```

Expected output (the output might vary due to the use of randint())

Mathematics: 42.75
 English: 53.666666666666664
 Science: 48.625
 History: 50.541666666666664
 Geography: 44.083333333333336

Exercise 29

Now generate the course averages for the top 5 students linked list.

Given code

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)

```

```

    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highe
st_student))

for i in range(top_5.get_size()):
    print(top_5.get(i).get_name())
    top_5.get(i).print_grades()

for i in range(year_11.get_students().get_size()):
    for j in
range(year_11.get_students().get(i).get_grades().get_size()):
        if
year_11.get_students().get(i).get_grades().get(j).get_score() > 90:
            print(year_11.get_students().get(i).get_name() + " has
a grade greater than 90 in " +
year_11.get_students().get(i).get_grades().get(j).get_name())

for i in range(top_5.get_size()):
    for j in range(top_5.get(i).get_grades().get_size()):
        if top_5.get(i).get_grades().get(j).get_score() > 90:
            print(top_5.get(i).get_name() + " has a grade greater
than 90 in " + top_5.get(i).get_grades().get(j).get_name())

for i in
range(year_11.get_students().get(0).get_grades().get_size()):
    total = 0
    count = 0
    for j in range(year_11.get_students().get_size()):
        total +=
year_11.get_students().get(j).get_grades().get(i).get_score()
        count += 1

print(year_11.get_students().get(0).get_grades().get(i).get_name()
+ ":" + str(total / count))

print("\nTop 5:")
# Your code here

```

Expected input

```

year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",

```

```

random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highe
st_student))

for i in
range(year_11.get_students().get(0).get_grades().get_size()):
    total = 0
    count = 0
    for j in range(year_11.get_students().get_size()):
        total +=
year_11.get_students().get(j).get_grades().get(i).get_score()
        count += 1

print(year_11.get_students().get(0).get_grades().get(i).get_name()
+ ":" + str(total / count))

print("\nTop 5:")
for i in range(top_5.get(0).get_grades().get_size()):
    total = 0
    count = 0
    for j in range(top_5.get_size()):
        total += top_5.get(j).get_grades().get(i).get_score()
        count += 1
    print(top_5.get(0).get_grades().get(i).get_name() + ":" + +
str(total / count))

```

Expected output (the output might vary due to the use of randint())

Mathematics: 44.20833333333333

English: 52.125

Science: 48.66666666666666

History: 35.45833333333336

Geography: 44.916666666666664

Top 5:

Mathematics: 93.4

English: 74.6

Science: 51.4

History: 52.4

Geography: 64.6

Exercise 30

Now to finally finish the system, create an input system that allows the user to enter a student name and a subject name and then print out the grade for that student in that subject if they exist in the year 11 linked list only.

Given code

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highe
st_student))
```

```
# Your code here
```

Expected input

```
year_11 = YearGroup(11)

for i in range(30): # create 30 students
    year_11.add_student(Student("Student " + str(i)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("Mathematics",
random.randint(0, 100)))

for i in range(30):
    year_11.students.get(i).add_grade(Grade("English",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Science",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("History",
random.randint(0, 100)))
    year_11.students.get(i).add_grade(Grade("Geography",
random.randint(0, 100)))

year_11.get_students().delete_at(18)

top_5 = LinkedList()
for i in range(5):
    highest = 0
    highest_student = None
    for j in range(year_11.get_students().get_size()):
        if year_11.get_students().get(j).get_average() > highest:
            highest = year_11.get_students().get(j).get_average()
            highest_student = year_11.get_students().get(j)
    top_5.append(highest_student)

year_11.get_students().delete_at(year_11.get_students().index(highest_student))

student_name = input("Enter a student name: ")
subject_name = input("Enter a subject name: ")
for i in range(year_11.get_students().get_size()):
    if year_11.get_students().get(i).get_name() == student_name:
        for j in
range(year_11.get_students().get(i).get_grades().get_size()):
            if
year_11.get_students().get(i).get_grades().get(j).get_name() ==
subject_name:

print(year_11.get_students().get(i).get_grades().get(j).get_score())
```

Expected output (the output might vary due to the use of randint())

Enter a student name: ***Student 1 (User input)***
Enter a subject name: ***Mathematics (User input)***

74