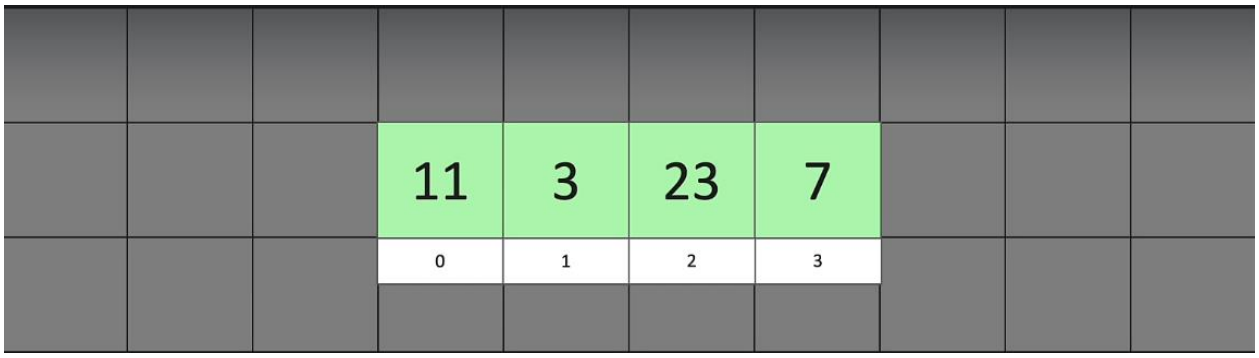
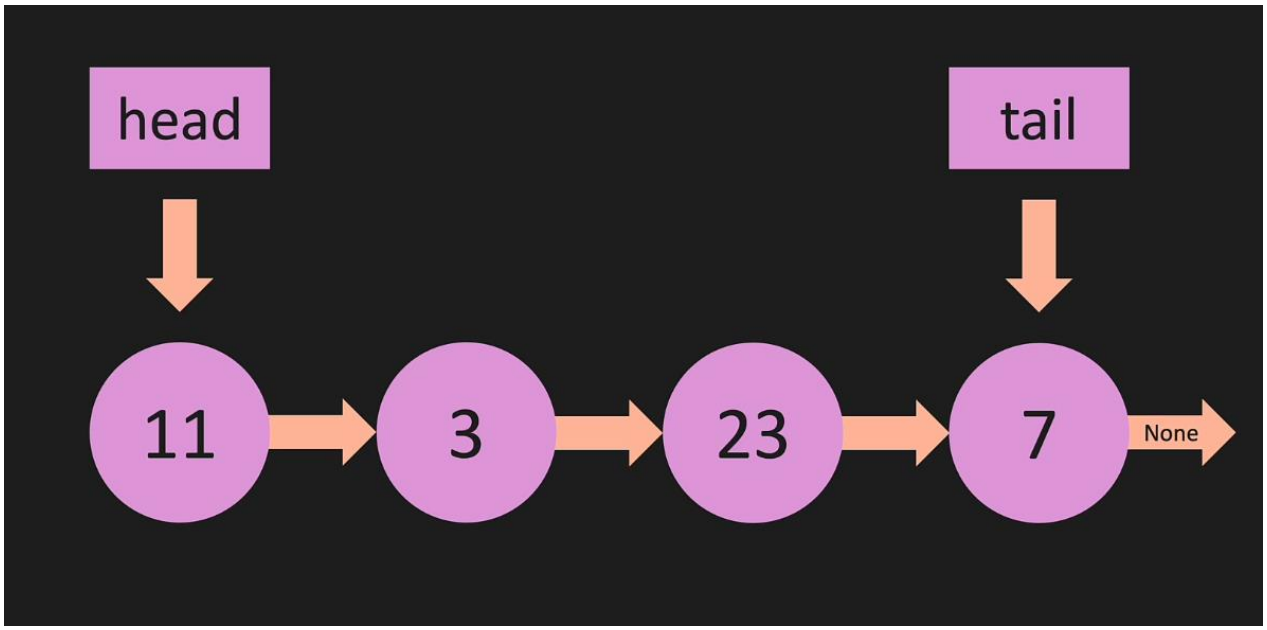
List

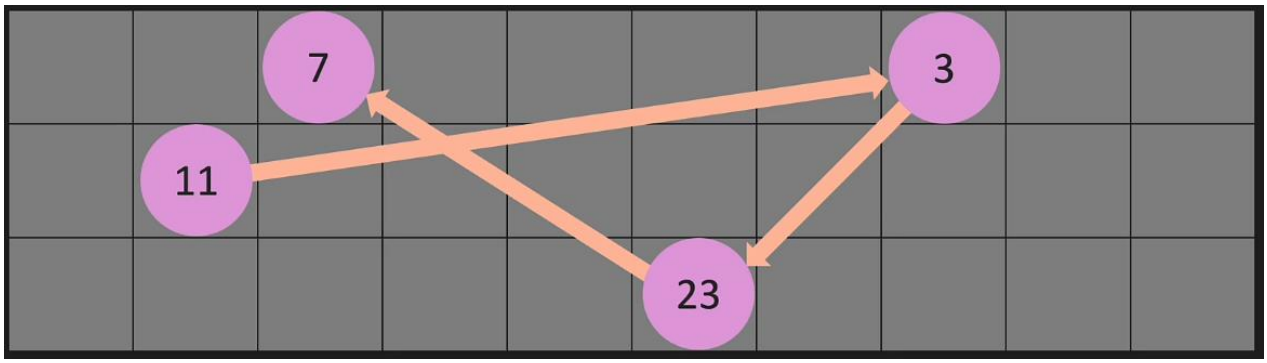
Contiguous space in memory (all elements are placed one after the other in memory), and therefore can be indexed



Linked List

Non-Contiguous space in memory, as they can be accessed via pointers even though they are placed separately.





Big O – Linked List vs List

	Linked Lists	Lists
Append	O(1)	O(1)
Pop	O(n)	O(1)
Prepend	O(1)	O(n)
Pop First	O(1)	O(n)
Insert	O(n)	O(n)
Remove	O(n)	O(n)
Lookup by Index	O(n)	O(1)
Lookup by Value	O(n)	O(n)

Linked List

Node = value + pointers

Set of Nodes make up a linked list

Code to initialize a linked list

```
1. class Node:
2.     def __init__(self, value):
3.         self.value = value
4.         self.next = None
5.
6. class LinkedList:
7.     def __init__(self, value):
8.         new_node = Node(value)
9.         self.head = new_node
10.        self.tail = new_node
11.        self.length = 1
```

This code defines two classes: **Node** and **LinkedList**.

The **Node** class represents a single node in a singly linked list, and the **LinkedList** class represents the entire singly linked list.

class Node: This line defines the **Node** class.

1. **def __init__(self, value):**: This is the constructor for the **Node** class. It is called when you create a new instance of the **Node** class.
2. **self.value = value**: This line sets the **value** attribute of the **Node** instance to the value passed as an argument.
3. **self.next = None**: This line sets the **next** attribute of the **Node** instance to **None**.
   The **next** attribute will be used to store a reference to the next node in the linked list.

class LinkedList: This line defines the **LinkedList** class.

4. **def __init__(self, value):**: This is the constructor for the **LinkedList** class. It is called when you create a new instance of the **LinkedList** class.
5. **new_node = Node(value)**: This line creates a new instance of the **Node** class with the given value, creating the first node in the linked list.
6. **self.head = new_node**: This line sets the **head** attribute of the **LinkedList** instance to the new node. The **head** attribute represents the first node in the linked list.
7. **self.tail = new_node**: This line sets the **tail** attribute of the **LinkedList** instance to the new node. The **tail** attribute represents the last node in the linked list. Since there is only one node in the list at this point, both the **head** and **tail** point to the same node.
8. **self.length = 1**: This line sets the **length** attribute of the **LinkedList** instance to 1, indicating that there is currently one node in the linked list.

Code to print a linked list

```
1.  def print_list(self):
2.      temp = self.head
3.      while temp is not None:
4.          print(temp.value)
5.          temp = temp.next
```

This code defines a method called **print_list** that prints the elements of a singly linked list, one element per line.

The method has the following logic:

1. Set a temporary pointer **temp** to the head of the list. This pointer is used to traverse the list from the beginning to the end.
2. Start a while loop that continues as long as **temp** is not None, i.e., until it has reached the end of the list.
3. Inside the loop, print the value of the current node (the node that **temp** points to) using the **print(temp.value)** statement. This will output the value of the node, followed by a newline character, ensuring that each element is printed on a separate line.
4. Move the temporary pointer **temp** to the next node in the list using the statement **temp = temp.next**. This advances the traversal to the next node in the list.

The method iterates through the entire linked list, printing the value of each node on a separate line, and stops when it reaches the end of the list (when **temp** is None).

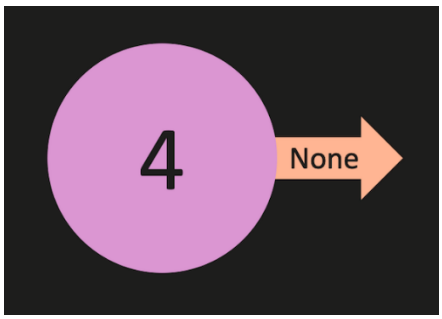
Code to Append to a linked list

```
1.  def append(self, value):
2.      new_node = Node(value)
3.      if self.head is None:
4.          self.head = new_node
5.          self.tail = new_node
6.      else:
7.          self.tail.next = new_node
8.          self.tail = new_node
9.      self.length += 1
```

Create a new node with the given value:

new_node = Node(value)

The line above creates one of these:



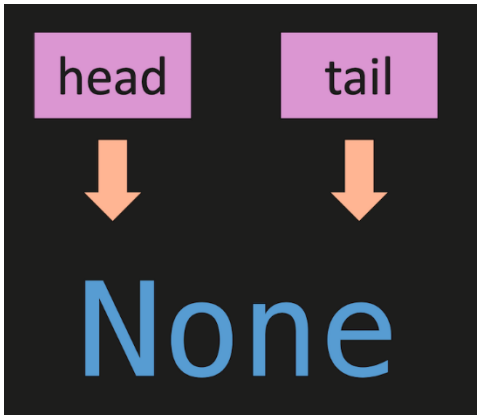
Check if the linked list is empty:

if self.head is None:

You could also do it this way:

if self.lenghth == 0:

Either of the lines above will test to see if the LL is empty:

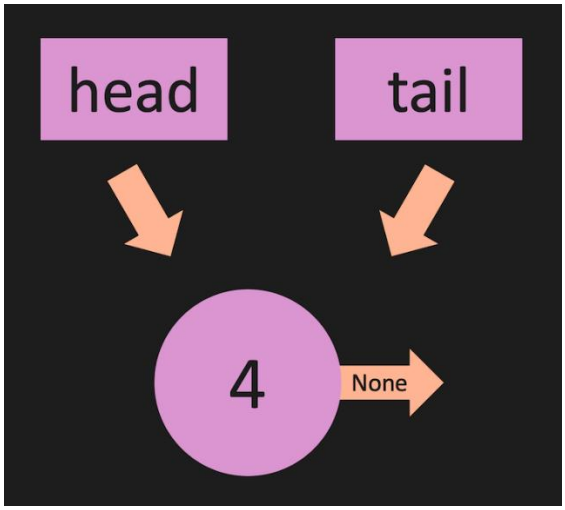


If the LL is empty, set the head and tail to point at new_node:

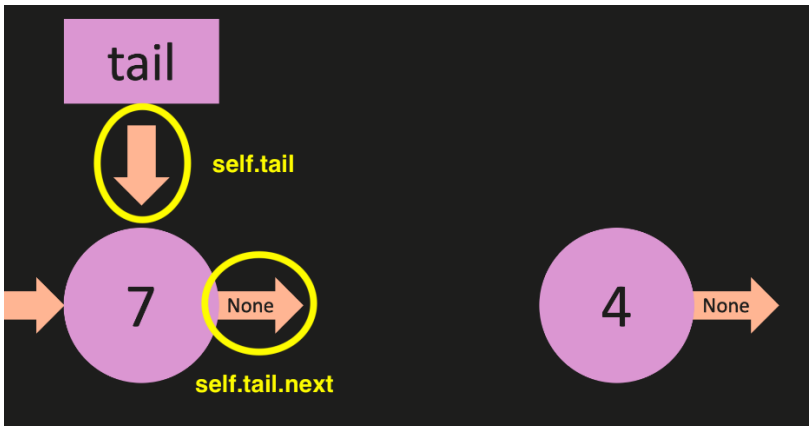
self.head = new_node

self.tail = new_node

The lines above will point both head and tail at the new_node:



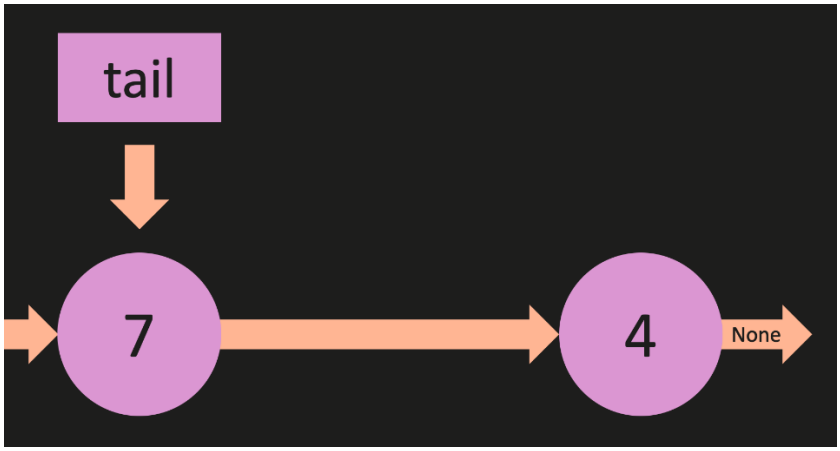
If the LL has nodes it might look something like the image below. Both **self.tail.next** and **self.tail** need to point to **new_node**:



First, we will update the next attribute of the node tail is pointing to, to point to new_node:

self.tail.next = new_node

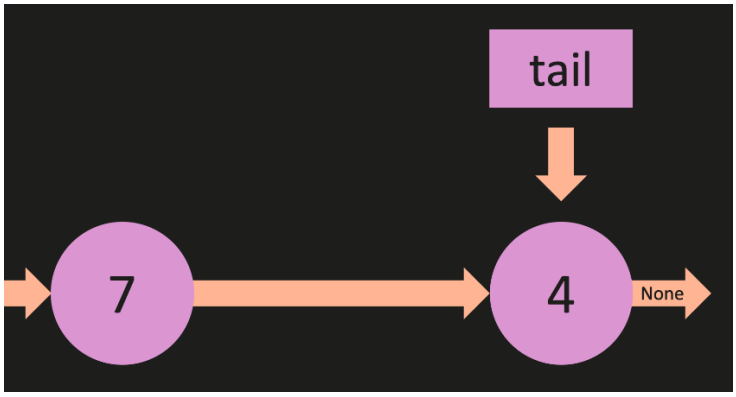
The line above will do this:



Then we will set tail to point to new_node:

self.tail = new_node

The line above will do this:



Increment the length of the linked list by 1:

self.length += 1

Big O:

- **O(1)** - **Constant Time**
    - No matter how large the linked list is, the number of operations taken to execute **append** remains constant
    - Constant time is another name for O(1)

Pop from a linked list

```
1.  def pop(self):
2.      if self.length == 0:
3.          return None
4.      temp = self.head
5.      pre = self.head
6.      while(temp.next):
7.          pre = temp
8.          temp = temp.next
9.      self.tail = pre
10.     self.tail.next = None
11.     self.length -= 1
12.     if self.length == 0:
13.         self.head = None
14.         self.tail = None
15.     return temp
```

Alternative Solution:

```
1.  def pop(self):
2.      if self.length == 0:
3.          return None
4.
```

```
5.      if self.length == 1:
6.          temp = self.head
7.          self.head = None
8.          self.tail = None
9.          self.length -= 1
10.         return temp
11.     else:
12.         temp = self.head
13.         while temp.next.next is not None:
14.             temp = temp.next
15.         last_node = temp.next
16.         temp.next = None
17.         self.tail = temp
18.         self.length -= 1
19.         return last_node
```

This code defines the **pop** method for the LinkedList class, which removes the last element (tail) from the linked list and returns it. Here is an explanation of each part of the code:

1. **if self.length == 0:** checks if the linked list is empty. If it is, the method returns **None** since there is nothing to pop.
2. **temp = self.head** and **pre = self.head** initialize two variables, **temp** and **pre**, both pointing to the head of the linked list.
3. **while(temp.next):** is a loop that continues as long as the **next** attribute of **temp** is not **None**. In other words, it iterates through the linked list until it reaches the last node.
4. Inside the loop, **pre = temp** updates the **pre** variable to be the current node before moving to the next one, and **temp = temp.next** moves the **temp** variable to the next node in the list.
5. After the loop, **self.tail = pre** updates the **tail** attribute of the LinkedList to the second-to-last node in the list, which is now stored in the **pre** variable.
6. **self.tail.next = None** removes the link between the new tail and the last node by setting the **next** attribute of the new tail to **None**.
7. **self.length -= 1** decrements the length of the linked list by 1, reflecting the removal of the last node.

8. **if self.length == 0:** checks if the list is now empty after the pop operation. If it is, both the **head** and **tail** attributes of the LinkedList are set to **None**, indicating that the list is empty.
9. **return temp** returns the removed node (previously the tail of the linked list) to the caller.

Big O:

- **O(n)** - n is the number of nodes in the linked list
    - When we say that the time complexity of a linked list operation is O(n), we mean that the execution time of the operation grows linearly with the size of the linked list. In other words, as the number of elements in the linked list increases, the time taken to perform the operation increases at the same rate.
    - An algorithm with a single loop that iterates through all n items in the worst case has a time complexity of O(n)
    - This is what lets us know this is O(n):

```
1. while(temp.next):
2.     pre = temp
3.     temp = temp.next
```

Prepend to a Linked List

```
1.  def prepend(self, value):
2.      new_node = Node(value)
3.      if self.length == 0:
4.          self.head = new_node
5.          self.tail = new_node
6.      else:
7.          new_node.next = self.head
8.          self.head = new_node
9.      self.length += 1
10.     return True
```

This code defines a **prepend** method for a LinkedList class. The purpose of this method is to add a new node with the given value to the beginning of the linked list, updating the head attribute and the length attribute accordingly. Here is an explanation of each part of the code:

1. **def prepend(self, value):**: This line defines the **prepend** method, which takes a value as its argument.
2. **new_node = Node(value)**: This line creates a new instance of the **Node** class with the given value, representing the new node that will be added to the beginning of the list.
3. **if self.length == 0:**: This condition checks if the linked list is empty (length is 0).
4. **self.head = new_node** and **self.tail = new_node**: If the list is empty, these lines set both the **head** and **tail** attributes of the LinkedList instance to the new node, since it will be the only node in the list.
5. **else:**: This block executes if the list is not empty (length is greater than 0).
6. **new_node.next = self.head**: This line sets the **next** attribute of the new node to the current head of the list. This creates a link between the new node and the previous first node in the list.
7. **self.head = new_node**: This line updates the **head** attribute of the LinkedList instance to the new node, making the new node the new head of the list.
8. **self.length += 1**: This line increments the **length** attribute of the LinkedList instance by 1, reflecting the addition of the new node.
9. **return True**: This line returns **True**, indicating that the operation was successful.

Big O:

- **O(1)** - **Constant Time**
    - No matter how large the linked list is, the number of operations taken to execute **prepend** remains constant
    - Constant time is another name for O(1)

Pop First from the Linked List

```
1.  def pop_first(self):
2.      if self.length == 0:
3.          return None
4.      temp = self.head
5.      self.head = self.head.next
6.      temp.next = None
7.      self.length -= 1
8.      if self.length == 0:
```

9. self.tail = None
10. return temp

This code defines a **pop_first** method for a LinkedList class. The purpose of this method is to remove the first node (the head) from the linked list, update the head attribute and the length attribute accordingly, and return the removed node. Here is an explanation of each part of the code:

1. **if self.length == 0:**: This condition checks if the linked list is empty (length is 0).
2. **return None**: If the list is empty, the method returns None, as there is no node to remove.
3. **temp = self.head**: This line saves a reference to the current head node in the variable **temp** before updating the head attribute.
4. **self.head = self.head.next**: This line updates the head attribute to point to the second node in the list, which is the next node after the current head.
5. **temp.next = None**: This line disconnects the removed node (**temp**) from the list by setting its **next** attribute to None.
6. **self.length -= 1**: This line decreases the length attribute of the LinkedList instance by 1 to reflect the removal of the node.
7. **if self.length == 0:**: This condition checks if the list becomes empty after removing the node.
8. **self.tail = None**: If the list is empty, the method sets the tail attribute of the LinkedList to None.
9. **return temp**: This line returns the removed node (**temp**) to the caller.

Big O:

- **O(1)** - **Constant Time**
    - No matter how large the linked list is, the number of operations taken to execute pop_first remains constant
    - Constant time is another name for O(1)

Get in LL

Make sure the if statement is like this:
if index < 0 or index >= self.length

Not like this:
if index < 0 or index >= self.length-1

Or this:
if index < 0 or index > self.length

Consider this Linked List:

1 -> 2 -> 3 -> 4

The only valid indexes are 0-3.

You cannot get a node at the index of 4 or any index greater than 4 (greater than or equal to the length).

Solution:

```
1.  def get(self, index):
2.      if index < 0 or index >= self.length:
3.          return None
4.      temp = self.head
5.      for _ in range(index):
6.          temp = temp.next
7.      return temp
```

This code defines a **get** method for a LinkedList class. The purpose of this method is to return the node at the specified index in the linked list. If the index is out of bounds (less than 0 or greater than or equal to the length of the list), the method returns None. Here is an explanation of each part of the code:

1. **if index < 0 or index >= self.length:**: This condition checks if the given index is out of bounds, i.e., it is either negative or greater than or equal to the length of the list.
2. **return None**: If the index is out of bounds, the method returns None, as there is no valid node at the specified index.
3. **temp = self.head**: This line initializes a temporary variable **temp** and sets it to the head of the list, to start traversing the list from the beginning.
4. **for _ in range(index):**: This loop iterates through the list **index** times, which helps traverse the list until the specified index.
5. **temp = temp.next**: Inside the loop, this line updates the **temp** variable to the next node in the list, moving the traversal one step forward in each iteration.
6. **return temp**: After the loop, the **temp** variable points to the node at the specified index in the list. This line returns that node.

Big O:

- **O(n)** - n is the number of nodes in the linked list
    - When we say that the time complexity of a linked list operation is O(n), we mean that the execution time of the operation grows linearly with the size of the linked list. In other words, as the number of elements in the linked list increases, the time taken to perform the operation increases at the same rate.
    - An algorithm with a single loop that iterates through all n items in the worst case has a time complexity of O(n)
    - This is what lets us know this is O(n):

```
1. for _ in range(index):
2.     temp = temp.next
```

Set in LL

```
1.  def set_value(self, index, value):
2.      temp = self.get(index)
3.      if temp:
4.          temp.value = value
5.          return True
6.      return False
```

This code defines a **set_value** method for a LinkedList class.

The purpose of this method is to update the value of the node at the specified index in the linked list.

If the index is out of bounds, the method returns False.

If the value is successfully updated, the method returns True.

Here is an explanation of each part of the code:

1. **temp = self.get(index)**: This line calls the **get** method with the given index to find the node at the specified index in the list. The result is stored in the **temp** variable. If the index is out of bounds, **temp** will be None.
2. **if temp:**: This condition checks if a valid node was found at the specified index (i.e., **temp** is not None).
3. **temp.value = value**: If a valid node was found, this line updates the value of the node with the given value.

4. **return True**: After updating the node's value, the method returns True, indicating that the value was updated successfully.
5. **return False**: If the condition in step 2 is not met (i.e., the index is out of bounds), the method returns False, indicating that the value was not updated.

Insert in LL

Make sure the first if statement is like this:
if index < 0 or index > self.length:

Not like this:
if index < 0 or index > self.length-1:

Or this:
if index < 0 or index >= self.length:

Consider this Linked List:

1 -> 2 -> 3 -> 4

The indexes are 0-3.

If you inserted an item at the end (append) it would be added at the index of 4 (which is equal to the length).

Any index of 5 or greater would be invalid (greater than the length).

```
1.  def insert(self, index, value):
2.      if index < 0 or index > self.length:
3.          return False
4.      if index == 0:
5.          return self.prepend(value)
6.      if index == self.length:
7.          return self.append(value)
8.      new_node = Node(value)
9.      temp = self.get(index - 1)
10.     new_node.next = temp.next
11.     temp.next = new_node
```

12. self.length += 1
13. return True

This code implements the **insert** method of a singly linked list.

The method takes an **index** and a **value** as input arguments and inserts a new node with the given value at the specified index in the linked list.

The method returns **True** if the node is successfully inserted and **False** if the given index is out of bounds.

Here's the explanation of the code:

1. Check if the given index is less than 0 or greater than the length of the list. If so, return **False** as the index is out of bounds.
2. If the index is 0, it means the new node should be inserted at the beginning of the list. In this case, call the **prepend** method with the given value and return the result.
3. If the index is equal to the length of the list, it means the new node should be inserted at the end of the list. In this case, call the **append** method with the given value and return the result.
4. Create a new node with the given value.
5. Call the **get** method with the (index - 1) as the argument, to get the node just before the insertion point. Store the result in the variable **temp**.
6. Set the **next** attribute of the new node to the **next** attribute of **temp**. This makes the new node point to the node that comes after the insertion point.
7. Set the **next** attribute of **temp** to the new node, making the node just before the insertion point point to the new node.
8. Increment the length of the list, as a new node has been added.
9. Return **True**, indicating that the node was successfully inserted into the list.

Big O:

- **O(n)**
    - n is the number of nodes in the linked list
    - When we say that the time complexity of a linked list operation is O(n), we mean that the execution time of the operation grows linearly with the size of the linked list. In other words, as the number of elements in the linked list increases, the time taken to perform the operation increases at the same rate.
    - An algorithm with a single loop that iterates through all n items in the worst case has a time complexity of O(n)
    - insert uses the get method, which is O(n), to iterate through the linked list

---

Remove from LL

Make sure the first if statement is like this:
if index < 0 or index >= self.length:

Not like this:
if index < 0 or index >= self.length-1:

Or this:
if index < 0 or index > self.length:

Consider this Linked List:

1 -> 2 -> 3 -> 4

The only valid indexes are 0-3.

You cannot remove a node at the index of 4 or any index greater than 4 (greater than or equal to the length).

```
1.  def remove(self, index):
2.      if index < 0 or index >= self.length:
3.          return None
4.      if index == 0:
5.          return self.pop_first()
6.      if index == self.length - 1:
7.          return self.pop()
8.      pre = self.get(index - 1)
9.      temp = pre.next
10.     pre.next = temp.next
11.     temp.next = None
12.     self.length -= 1
13.     return temp
```

This code defines a **remove** method for a LinkedList class.

The method takes an integer **index** as its parameter and removes the node at the specified index from the linked list.

The method returns the removed node or None if the index is out of bounds.

1. The method first checks if the given **index** is out of bounds by comparing it to the LinkedList's **length** attribute. If it is out of bounds, the method returns None.
2. If the **index** is 0 (the first node in the list), the method calls the **pop_first()** method to remove and return the first node.
3. If the **index** is equal to **self.length - 1** (the last node in the list), the method calls the **pop()** method to remove and return the last node.
4. If the index is within bounds and not the first or last node, the method calls the **get()** method with **index - 1** to find the previous node (named **pre**).
5. The method sets a temporary variable **temp** to **pre.next**, which is the node to be removed.
6. The method then updates **pre.next** to point to **temp.next**, effectively skipping the node to be removed and connecting the previous node to the node after the removed node.
7. The method sets **temp.next** to None, disconnecting the removed node from the list.
8. The LinkedList's **length** attribute is decremented by 1 to reflect the removal of a node.
9. Finally, the method returns the removed node (**temp**).

Big O:

- **O(n)**
    - n is the number of nodes in the linked list
    - When we say that the time complexity of a linked list operation is O(n), we mean that the execution time of the operation grows linearly with the size of the linked list. In other words, as the number of elements in the linked list increases, the time taken to perform the operation increases at the same rate.
    - An algorithm with a single loop that iterates through all n items in the worst case has a time complexity of O(n)
    - **remove** uses the **get** method, which is O(n), to iterate through the linked list

Reverse the LL

```
1.  def reverse(self):
2.      temp = self.head
3.      self.head = self.tail
4.      self.tail = temp
5.      after = temp.next
6.      before = None
7.      for _ in range(self.length):
8.      # Instead of the for loop you could use:
9.      # while temp is not None:
10.     # -- or --
11.     # while temp:
12.         after = temp.next
13.         temp.next = before
14.         before = temp
15.         temp = after
```

This code is a method that reverses a singly linked list in-place. Here's an explanation of each line:

1. **temp = self.head**: A temporary variable **temp** is pointed to the first node in the list.
2. **self.head = self.tail**: The head variable is pointed to the last node.
3. **self.tail = temp**: The tail is pointed to the first node (stored in **temp**).
4. **after = temp.next**: The **after** variable is initialized to store the next node in the list after the current node (**temp**).
5. **before = None**: The **before** variable is initialized to store the previous node in the list. It is initially set to **None** since there is no previous node for the initial head (now the tail).

The loop iterates through the list, updating the next pointers of each node to reverse their order:

1. **for _ in range(self.length):** A loop that iterates through the list based on its length. a. **after = temp.next**: Store the next node in the list after the current node (**temp**) in the **after** variable. b. **temp.next = before**: Update the next pointer of the current node (**temp**) to point to the previous node in the list (**before**). c. **before = temp**: Update the previous node (**before**) to be

the current node (**temp**) for the next iteration. d. **temp = after**: Move the current node (**temp**) to the next node in the list (**after**), as stored in the **after** variable.

After the loop finishes, the linked list is reversed with the head and tail pointers updated accordingly.

Big O:

- **O(n)**
    - n is the number of nodes in the linked list
    - When we say that the time complexity of a linked list operation is O(n), we mean that the execution time of the operation grows linearly with the size of the linked list. In other words, as the number of elements in the linked list increases, the time taken to perform the operation increases at the same rate.
    - An algorithm with a single loop that iterates through all n items in the worst case has a time complexity of O(n)
    - This is what lets us know this is O(n):

```
1. for _ in range(self.length):
2.     after = temp.next
3.     temp.next = before
4.     before = temp
5.     temp = after
```